Formal Languages and Their Applications

— Lectures Given in the Autumn of 2020 —

Prof. Dr. Karl Stroetmann

August 4, 2020

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

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

The last time I gave this lecture in 2015, I had used *Java* as the programming language. Currently, I am switching to using *Python* instead. Therefore, these lecture notes are subject to continuous change. 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`.

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.

Some parts of these lecture notes are currently written in the German language, while other parts are written in English. As time permits, I hope to eventually translate everything into the English language. As I am currently rewriting parts of these lecture notes, these notes will undoubtedly contain their fair amount of typos and other errors. If you spot an error, I would like you to either send an email to

karl.stroetmann@dhbw-mannheim.de

or to contact me via discord. Alternatively, you are also welcome to clone my github repository and send a pull request.

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:

1. $\Sigma = \{0, 1\}$ is an alphabet that can be used to represent binary numbers.
2. $\Sigma = \{\mathtt{a}, \cdots, \mathtt{z}, \mathtt{A}, \cdots, \mathtt{Z}\}$ is the alphabet used for the English language.
3. 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 $\{\mathtt{A}, \cdots, \mathtt{Z}\}$. ◇

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 bracket symbols and without separating comma symbols. 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 $\varepsilon =$ "". 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

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

   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 square brackets 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, $w_1 =$ "01" and $w_2 =$ "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 is a *precisely defined* subset. The idea is to have some kind of unambiguous definition of the subset L that makes up the language. We 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 most important in computer science.

Examples:

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

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

   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 *value*(w) by induction on the length of w. We call *value*(w) the interpretation of w. The idea is that *value*(w) computes the number represented by the string w:

(a) *value*$(0) = 0$, *value*$(1) = 1$,

(b) $|w| > 0 \rightarrow$ *value*$(w0) = 2 \cdot$ *value*(w),

(c) $|w| > 0 \rightarrow$ *value*$(w1) = 2 \cdot$ *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:

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

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 and the theorem of Nerode.

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 success of the programming language *Perl* was largely due to its efficiency in dealing with regular expressions. Today, all modern high-level languages, e.g. *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.

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 $n \in \mathbb{N}_0$. The n-th power of L is written as L^n and is definition by induction on n.

B.C.: $n = 0$:

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

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

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

1. $L^0 = \{\varepsilon\}$,
2. $L^1 = \{\varepsilon\} \cdot \{\mathtt{ab}, \mathtt{ba}\} = \{\mathtt{ab}, \mathtt{ba}\}$,
3. $L^2 = \{\mathtt{ab}, \mathtt{ba}\} \cdot \{\mathtt{ab}, \mathtt{ba}\} = \{\mathtt{abab}, \mathtt{abba}, \mathtt{baab}, \mathtt{baba}\}$. ◇

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 $n \in \mathbb{N}_0$:

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

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 such that $|s| > 0$.

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 $\mathtt{RegExp}_\Sigma$. This set 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. $\emptyset \in \mathtt{RegExp}_\Sigma$

   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 $\emptyset \notin \Sigma$.

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

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 s 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 my 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 "$\cdot$" 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

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

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.

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:

1. $r_1 + r_2 \doteq r_2 + r_1$

   This equation is true because the union of sets is commutative:

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

2. $(r_1 + r_2) + r_3 \doteq r_1 + (r_2 + r_3)$

   This equation is true because the union of sets is associative.

3. $(r_1 \cdot r_2)\cdot r_3 \doteq r_1 \cdot (r_2 \cdot r_3)$

   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.

4. $\emptyset \cdot r \doteq r \cdot \emptyset \doteq \emptyset$

5. $\varepsilon \cdot r \doteq r \cdot \varepsilon \doteq r$

6. $\emptyset + r \doteq r + \emptyset \doteq r$
7. $(r_1 + r_2) \cdot r_3 \doteq r_1 \cdot r_3 + r_2 \cdot r_3$
8. $r_1 \cdot (r_2 + r_3) \doteq r_1 \cdot r_2 + r_1 \cdot r_3$
9. $r + r \doteq r$, because
$$L(r + r) = L(r) \cup L(r) = L(r).$$
10. $(r^*)^* \doteq r^*$
We have
$$L(r^*) = \bigcup_{n \in \mathbb{N}} L(r)^n$$
and that implies $L(r) \subseteq L(r^*)$. If we replace r by r^*, we see that
$$L(r^*) \subseteq L\big((r^*)^*\big)$$
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 $L\big((r^*)^*\big) \subseteq L(r^*)$.
11. $\emptyset^* \doteq \varepsilon$
12. $\varepsilon^* \doteq \varepsilon$
13. $r^* \doteq \varepsilon + r^* \cdot r$
14. $r^* \doteq (\varepsilon + r)^*$

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 practise. To this end we discuss the tool PLY , which can generate both *scanners* and *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 belong together logically. 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`", or "`case`".
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, "`"hallo"`" 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, tabulators, line breaks, and carriage returns are white space symbols.

```
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.

To make things more precise, Figure 3.1 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:

```
[ "/* Hello World program */", "#", "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.

3.1 The Structure of a Ply Scanner Specification

In this section we introduce Python Lex-Yacc 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 14 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. Its main purpose 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 *JFlex*. 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

Although many interesting syntactical constructions can be described through regular expressions, often the regular expressions that are needed can get quite unwieldy. A case in point is the regular expression that describes multi line comments in the language *Java*. In languages like *Java*, `C`, or `C++`, a multi line comment has the form

`/*` $\cdots$ `*/`

If we had to describe multi line comments with a regular expression that does use non-greedy versions of the quantifiers, then we would have to use a regular expression that looks similar to the following regular expression:

```
\/\*([^*]|\*+[^*/])*\*+\/
```
(3.1)

[1] You know nothing, John Slow.

First, the expression is quite difficult to read. One reason is that the operator symbols "/" and "*" have to be escaped with a backslash character. The other problem is that the mechanics of this regular expression is quite involved. Let us discuss the different parts of the regular expression given above:

1. `\/\*`

   This part specifies the string "/*".

2. `([^*]|\*+[^*/])*`

   This regular expression specifies text that is surrounded by the opening string "/*" and the closing string "*/". This text must not contain the substring "*/" because otherwise we would misinterpret a line of the form

   ```
   /* first */ ++n; /* second */
   ```

   since we would interpret the command "++n;" as a part of the comment.

   The first part of the regular expression above is the expression "[^*]". This part recognizes any character that is different from the character "*". As long as there is no "*", the text can not contain the substring "*/". However, the problem is that the character "*" is legal inside a multi line comment. However, this character must not be followed by the character "/". Hence, the alternative "*+[^*/]" specifies those strings that might contain any number of "*" characters but that have to be followed by a character that is both different from the character "/" and from the character "*".

   Hence, the expression "[^*]|*+[^*/]" specifies a character that is different from "*", or a sequence of "*" characters that are followed by a character different from "/". As these sequence can occur any number of times, the expression is enclosed in parentheses and decorated with the quantifier "*".

3. `\*+\/`

   This regular expression specifies the end of the multi line comment. The end can consist of any positive number of "*" characters that are followed by the character "/". If we would just use the regular expression "*\/", then we would not be able to identify comments of the form

   ```
   /*** blah ***/
   ```

   correctly, since the second part of the regular expression discussed previously does only allow sequences of the character "*" that are followed by a character that is different from both "*" and "/".

```
1  /**
2    remove C comments from a file
3  */
4
5  %%
6
7  %class Decomment
8  %standalone
9  %unicode
10
11 %%
12
13 \/\*([^*]|\*+[^*/])*\*+\/ { /* skip multi  line comments */ }
14 \/\/.*                    { /* skip single line comments */ }
```

Figure 3.4: Entfernung von Kommentaren aus einem C-Programm.

Figure 3.4 on page 20 shows a *JFlex* program that can be used to remove all comments from a `C` program. This program will remove both single line comments of the form

"// ..."

as well as multi line comments of the form

"/* ... */.

At this point, you might ask yourself "What happens to those parts of the program that are not comments?". These parts are dealt with by the default rule of *JFlex*: Every character of the input that is not matched by any of the *JFlex* rules is just echoed, i.e. it is printed to the stream standard out. Hence, the *JFlex* program shown above will remove the comments from a program and it will print the rest of the program unchanged.

Although this program is quite concise, the logic of the regular expression is very involved and, because of that, the program is difficult to understand and to maintain. We have to options to simplify this program.

1. We could use the upto operator. If we were to use *JFlex*, this is by far the best option. However, not every scanner generator supports this operator. For example, if we would use *Flex* instead of *JFlex*, we would not have the upto operator available.
2. Alternatively, we can use scanner states. We will discuss this option using a second example that is given in the following subsection.

3.4.1 Removal of Html Tags

In this subsection 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.

The *JFlex* program shown in Figure 3.5 works as follows:

1. First, the head of an HTML file is removed. The head of an HTML file is enclosed in the tags "<head>" and "</head>".
2. Furthermore, the *JavaScript* that is part of the HTML file is also removed.
3. Finally, all HTML tags are removed.

All this is done via the use of so called scanner states. In the example, there are two scanner states. They are declared via the keyword "%xstate".

We proceed to discuss the details of the *JFlex* program shown in Figure 3.5.

1. Line 10 declares the scanner states `header` and `script`. The keyword "%xstate" tells us that these states are exclusive scanner states. We will discuss the difference between exclusive and inclusive states later. The general syntax of state declaration is as follows:
   (a) A state declaration is either started with the keyword "%xstate" or the keyword "%state". The keyword "%xstate" declares exclusive states, while "%state" declares inclusive states. The difference between these two kinds of states is discussed later.
   (b) After the keyword specifying the states as either inclusive or exclusive we have a list of the names of the states. These names are separated by blanks.
2. In line 13 we recognize the string "<head>". In this case, we execute the action "yybegin(header)". This action switches the state of the scanner from the default state "YYINITIAL" to the state "header". As we had declared the state `header` to be an exclusive state, the scanner is only able to execute those rules that are marked with the prefix "<header>". If the state had been declared as an inclusive state, then all rules that are not prefixed with a state would also be applicable.

   We find rules for the state "header" in the lines 26 and 27.
3. Line 14 switches to the state "script" if the scanner encounters an opening `script` tag.
4. Line 15 reads all remaining tags. Since the action is empty, these tags are removed.

```
1  package Converter;
2
3  %%
4
5  %class Html2Txt
6  %standalone
7  %line
8  %unicode
9
10 %xstate header script
11 %%
12
13 "<head>"            { yybegin(header);         }
14 "<script"[^>]+">"   { yybegin(script);         }
15 "<"[^>]+">"         { /* skip html tags */     }
16 \R+                 { System.out.print("\n");  }
17                { System.out.print(" ");   }
18 &auml;              { System.out.print("\"a");   }
19 &ouml;              { System.out.print("\"o");   }
20 &uuml;              { System.out.print("\"u");   }
21 &Auml;              { System.out.print("\"A");   }
22 &Ouml;              { System.out.print("\"O");   }
23 &Uuml;              { System.out.print("\"U");   }
24 &szlig;             { System.out.print("{\ss}");   }
25
26 <header>"</head>"   { yybegin(YYINITIAL);      }
27 <header>.|\R        { /* skip anything else */ }
28
29 <script>"</script>" { yybegin(YYINITIAL);      }
30 <script>.|\R        { /* skip anything else */ }
```

Figure 3.5: Transformation einer HTML-Datei in eine reine Text-Datei

5. Line 16 replaces the string " " by a blank.

6. The following lines replaces the HTML representation of strange German characters with the corresponding characters.

7. Line 26 is declared to be a rule for the state "header". Hence, this rule is only active if the scanner is in the state "header". This rule searches for the closing tag "</head>". If this tag is found, the scanner changes back into the default state YYINITIAL. In the default state, the scanner uses only those rules that are not prefixed with a state.

8. Line 27 contains another rule that is only used in the state "header". This rule reads an arbitrary character, which is not processed further and hence is simply discarded.

9. The lines 29 and 30 contain similar rules that are used in the state "script".

Exercise 2: Some programming languages allow nested comments. Create a *JFlex* program that is able to remove nested comments of the form

```
/* ··· */
```

from a given program. Furthermore, assume that the program might contains strings that are contained in double quotes. Now if either the character sequence "/*" or the character sequence "*/" occurs inside a string, then these character sequences should not be interpreted as comment delimiters. ◇

The next exercise is meant to improve the fun factor of my lecture.

Exercise 3: The purpose of this 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 3.6 on page 22 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 3.7 on page 23. 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

JFlex/LaTeX2HTML.

```
\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 3.6: An example LaTeX input file.

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.

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.

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 3.7: Output produced by the LaTeX file shown in Figure 3.6

(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 contains 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>
```

In order to cut the time necessary to complete this exercise, I have provided a JAVA class `HtmlWriter` that contains a number of useful static methods. This file is available at

JFlex/LaTeX2HTML/HtmlWriter.java.

Furthermore, the `github` directory `JFlex/LaTeX2HTML` already contains a `Makefile`. Additionally, the directory contains the file

JFlex/LaTeX2HTML/input.html

which shows the result of converting the input file "`input.tex`" into HTML. Next, there is a file `State.java` containing an enumeration that I have found useful for my solution of the problem.

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 else 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. The github directory `JFlex/LaTeX2HTML` already contains both the interface `Stack.java` as well as the class `ArrayStack.java` implementing this interface. ◇

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 FSM. 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 every 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.

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 check whether this string is an element of some language that we are interested in. 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 us render these 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 processes 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.

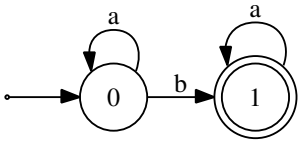


Figure 4.1: An FSM to recognize the language $L(\mathtt{a}^* \cdot \mathtt{b} \cdot \mathtt{a}^*)$.

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 pointing do it.

   If the FSM is in the state 0 and reads the character "a", then the FSM stays in 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.

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 F have the following properties:

1. Q is the finite set of states.
2. Σ is the input alphabet . Therefore, Σ is the 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.
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 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 F does not read any character, it cannot change its state.

I.S. $w = cv$ where $c \in \Sigma$ and $v \in \Sigma^*$ and $|v| = n$: 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 4: 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

$$\delta : Q \times \Sigma \to Q.$$

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 $\dagger \notin Q$. We call † the dead state.

1. $\widehat{Q} := Q \cup \{\dagger\}$,

   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) $\delta(q, c) = \Omega \rightarrow \widehat{\delta}(q, c) = \dagger$ 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 $\dagger$.

(c) $\widehat{\delta}(\dagger, c) = \dagger \quad$ 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 $\dagger$ and remains in this state regardless of the rest of the input string. As the dead state $\dagger$ is not an accepting state, the languages accepted by F and $\widehat{F}$ are identical. □

Exercise 5: Define an FSM that accepts the language specified by the regular expression

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

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 last three characters are "aaa".

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

State 1: "aab".

State 2: "aba".

State 3: "abb".

State 4: "bab".

State 5: "bba".

State 6: "bbb".

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

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

[1] What is dead may never die.

The remaining transitions are found in a similar way. Figure 4.2 on page 30 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". For the purpose of the language described this is the only thing that matters and therefor we can as well assume that in the start state the last three characters all have the value "a". Hence we can use the state 0 as the start state of our FSM.

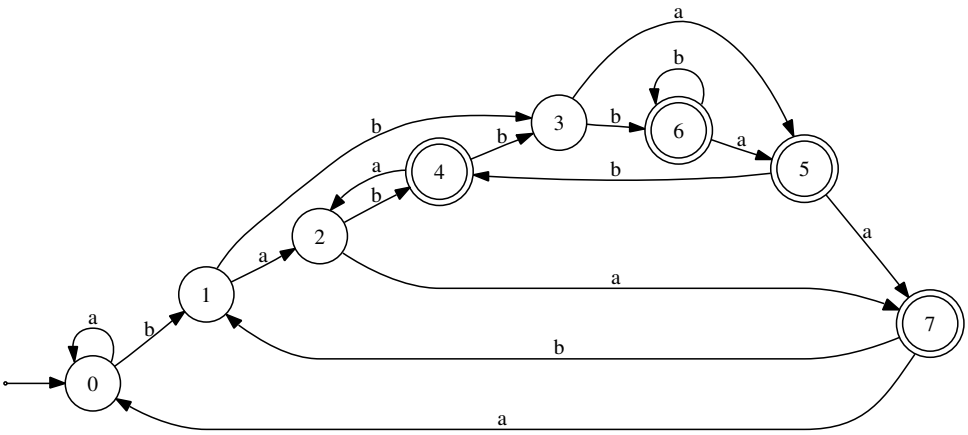


Figure 4.2: An FSM accepting $L\big((\mathtt{a}+\mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a}+\mathtt{b}) \cdot (\mathtt{a}+\mathtt{b})\big)$.

Bemerkung: 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

http://www.jflap.org.

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 is permitted to *choose* its next state from a given set of states.

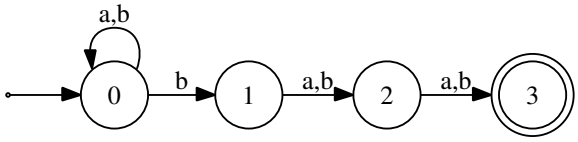


Figure 4.3: A non-deterministic finite state machine to recognize $L\big((\mathtt{a}+\mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a}+\mathtt{b}) \cdot (\mathtt{a}+\mathtt{b})\big)$.

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. The story is similar in state 2: The FSM switches to state 2 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 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

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

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 $\delta(q, \varepsilon)$ 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. $F \subseteq Q$ 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 30 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. $\delta = \big\{\langle 0, \mathtt{a}\rangle \mapsto \{0\}, \langle 0, \mathtt{b}\rangle \mapsto \{0, 1\}, \langle 0, \varepsilon\rangle \mapsto \{\}, \langle 1, \mathtt{a}\rangle \mapsto \{2\}, \langle 1, \mathtt{b}\rangle \mapsto \{2\}, \langle 1, \varepsilon\rangle \mapsto \{\},$
   $\langle 2, \mathtt{a}\rangle \mapsto \{3\}, \langle 2, \mathtt{b}\rangle \mapsto \{3\}, \langle 2, \varepsilon\rangle \mapsto \{\}\big\}$.

   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. ◇

A configuration of a non-deterministic FSM is a pair

$$\langle q, s \rangle$$

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

$$\langle q_1, cs \rangle \rightsquigarrow \langle q_2, s \rangle \quad \text{iff} \quad q_1 \overset{c}{\mapsto} q_2.$$

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

$$\langle q_1, s \rangle \rightsquigarrow \langle q_2, s \rangle \quad \text{iff} \quad q_1 \overset{\varepsilon}{\mapsto} q_2.$$

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 F 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":

$$L(F) = \big\{w \in \{\mathtt{a},\mathtt{b}\} \;\big|\; |w| \geq 3 \wedge w\big[|w|-2\big] = \mathtt{b}\big\}$$ ◇

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 6: 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". ◇

4.3 Equivalence of Deterministic and Non-Deterministic FSMs

In this section we show how a non-deterministic FSM

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

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

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

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

In order to present the construction of $\mathit{det}(A)$ 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 A the function

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

computes the set $\mathit{ec}(q)$ of all those states that the FSM A 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.

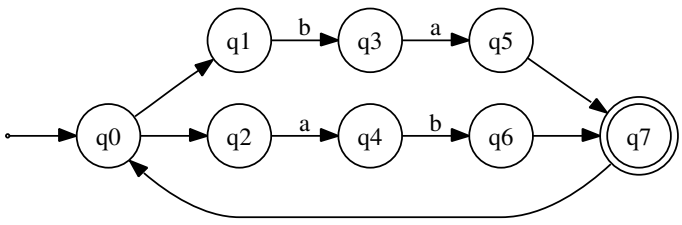


Figure 4.4: A non-deterministic FSM with ε-transitions.

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

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

$$\delta^* : Q \times \Sigma \to 2^Q.$$

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

$$\delta^*(q_1, c) := \bigcup \big\{ ec(q_2) \;\big|\; q_2 \in \delta(q_1, c) \big\}.$$

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 $ec(q_2)$.

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

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

1. $\delta^*(q_0, \mathtt{a}) = \{\}$,

   because in state q_0 there is no transition on reading the character a. Note that in our definition of the function δ^* the ε-transitions are done only after the character has been read.

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

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

   because when the letter 'b' 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 und q_2 are reachable by ε-transitions. ◇

The function δ^* maps a state into a set of states. Since the FSM $det(A)$ uses sets of states of the FSM A as its states we need a function that maps sets of states of the FSM A into sets of states. Hence we generalize the function δ^* 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 A 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:

$$\Delta(M, c) := \bigcup \big\{ \delta^*(q, c) \;\big|\; q \in M \big\}.$$

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 *det*(A) is computed from the non-deterministic FSM $A := \langle Q, \Sigma, \delta, q_0, F \rangle$. We define:

$$\textit{det}(A) = \langle 2^Q, \Sigma, \Delta, \textit{ec}(q_0), \widehat{F} \rangle \text{ where the components of this tuple are given as follows:}$$

1. The set of states of *det*(A) 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 A to *det*(A). After all, the deterministic FSM *det*(A) has to recognize the same language as the non-deterministic FSM A.

3. The function Δ that has been defined previously specified how the set of states change when a character is read.

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

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

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

Exercise 7: Transform the non-deterministic FSM A that is shown in Figure 4.3 on page 30 into the deterministic FSM *det*(A). ◇

Solution: We start by computing the set of states.

1. As we have *ec*$(0) = \{0\}$, the start state of *det*(A) is the set containing 0.

   $$S_0 := \textit{ec}(0) = \{0\}.$$

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 $\mathit{det}(A)$ can reach. For a better overview let us summarize the definitions of the individual states of the deterministic automaton:

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

Therefore the set $\widehat{Q}$ of the deterministic FSM $\mathit{det}(A)$ 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

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

Therefore we have no found the deterministic FSM $\mathit{det}(A)$. We have

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

This FSM is shown in Figure 4.5 on page 37.

We realize that this deterministic FSM $\mathit{det}(A)$ has 8 different states. The non-deterministic FSM A has 4 different states $Q = \{0,1,2,3\}$. Hence the power set 2^Q has 16 elements. Why then has the FSM $\mathit{det}(A)$ 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 FSM A 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, Thus, as sets that do not contain 0 are not needed as states of the deterministic FSM $\mathit{det}(A)$.

Exercise 8: Transform the non-deterministic FSM that is shown in Figure 4.4 on page 33 into an equivalence deterministic FSM $\mathit{det}(A)$. ◇

4.3.1 Implementing the Conversion of NFA to DFA

It is straightforward to implement the theory developed so far. Figure 4.6 on page 38 shows a *Python* program that converts a given non-deterministic FSM into a deterministic FSM. We discuss this program line by line.

1. In line 1 we define a function called `fixpoint`. This function takes two parameters:
   (a) `S0` is a set of states.

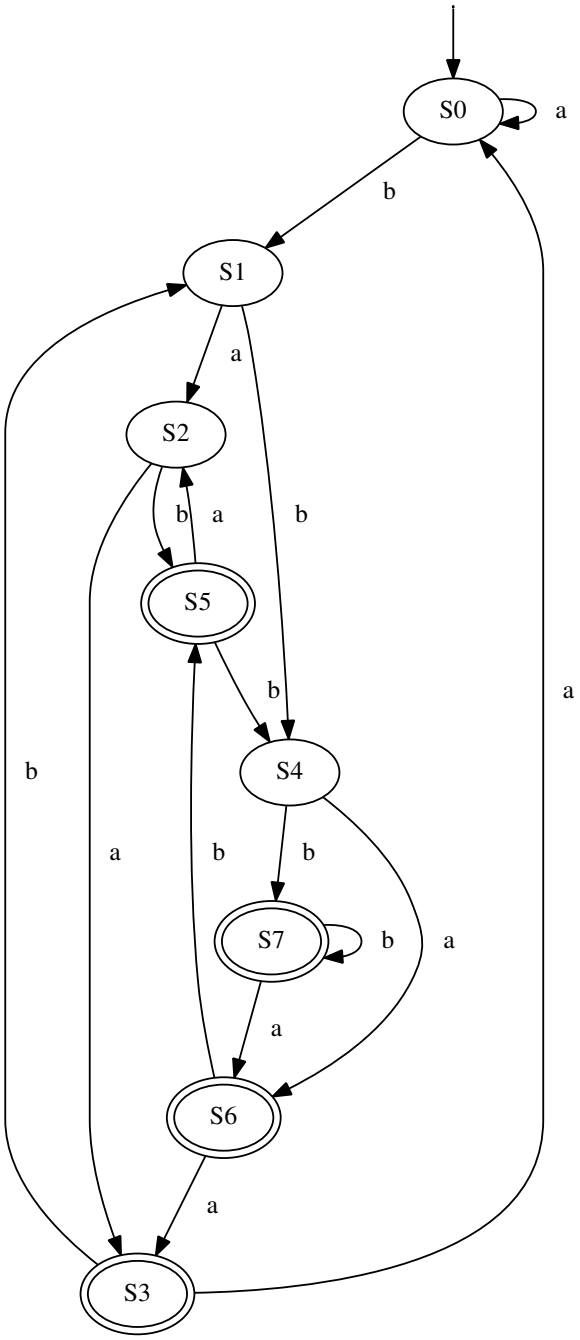


Figure 4.5: The deterministic FSM *det*(A).

(b) `f` is a function that takes a state as input and returns a set of states as its result.

The expression `fixpoint`$(\texttt{S0}, f)$ calculates the smallest set of states S such that

$$\texttt{S0} \subseteq S \quad \text{and} \quad f(S) \subseteq S$$

holds, where $f(S)$ is defined as

$$f(S) := \bigcup\{f(x) \mid x \in S\}.$$

```
def fixpoint(S0, f):
    Result = S0.copy()
    while True:
        NewElements = { x for M in Result
                          for x in f(M)
                      }
        if NewElements.issubset(Result):
            return Result
        Result |= NewElements

def epsClosure(s, delta):
    Result = fixpoint({s}, lambda q: delta.get((q, ''), set()))
    return frozenset(Result)

def deltaStar(s, c, delta):
    return { p for q in delta.get((s, c), set())
               for p in epsClosure(q, delta)
           }
def capitalDelta(M, c, delta):
    Result = { s for q in M
                 for s in deltaStar(q, c, delta)
             }
    return frozenset(Result)

def nfa2dfa(nfa):
    states, sigma, delta, q0, final = nfa
    newStart   = epsClosure(q0, delta)
    nextStates = lambda m: { capitalDelta(m, c, delta) for c in sigma }
    newStates  = fixpoint({newStart}, nextStates)
    newDelta   = { (m, c): capitalDelta(m, c, delta) for m in newStates
                                                     for c in sigma
                 }
    newFinal   = { m for m in newStates
                     if  m & final != set()
                 }
    return newStates, sigma, newDelta, newStart, newFinal
```

Figure 4.6: A program to convert a non-deterministic FSM into a deterministic FSM.

Hence, `fixpoint`$(\texttt{S0}, f)$ is a set S such that the image of S under f together with S0 is again S. The value of S is computed by a so called *fixpoint iteration*. The idea is to define a sequence $(S_n)_{n\in\mathbb{N}}$ of set S_n such that this sequence converges to the set S that is sought. The elements S_n of this sequence are defined by induction on $n \in \mathbb{N}$.

(a) Base Case: $n = 0$.
S_0 is taken to be the first argument of the function `fixpoint`, i.e. we have $S_0 := \texttt{S0}$.

(b) Induction Case: $n \mapsto n + 1$.
Define

$$S_{n+1} := S_n \cup \bigcup\{f(x) \mid x \in S_n\}.$$

Remember that $f(x)$ is really a set of states and therefore $\{f(x) \mid x \in S_n\}$ is a set of set of states. Taking the union of these sets reduces this set of sets of states to a mere set of states. These states are

combined with the states already in S_n. Therefore,

$$S_n \subseteq S_{n+1} \quad \text{for all } n \in \mathbb{N}.$$

Since the set of all states is finite, the sequence $(S_n)_{n\in\mathbb{N}}$ cannot grow indefinitely. Therefore, there will be a natural number $m \in \mathbb{N}$ such that

$$S_{m+1} = S_m.$$

At this point, the sequence $(S_n)_{n\in\mathbb{N}}$ has converged. Then, we can define $S := S_m$ and with this definition we have both

$$f(S) \subseteq S \quad \text{and} \quad \texttt{S0} \subseteq S.$$

2. The function `epsClosure` computes the ε-closure of a given state `s`. The transition function `delta` has to be provided as a second argument. In this function we assume that `delta` is represented as a dictionary. For example, the transition function δ of the non-deterministic finite state shown in Figure 4.4 can be represented as the following dictionary:

```
delta45 := { (0, '' ): {1, 2},
             (1, 'b'): {3},
             (2, 'a'): {4},
             (3, 'a'): {5},
             (4, 'b'): {6},
             (5, '' ): {7},
             (6, '' ): {7},
             (7, '' ): {0}
           };
```

The idea of the computation is to use a fixpoint iteration that starts with the set $\{\texttt{s}\}$ and adds successively all those states that can be reached by ε transitions. The second argument for `fixpoint` is a function that maps a given state q into the set $\delta(q, \varepsilon)$. Hence, we have

$$\texttt{epsilonClosure}(\texttt{s}, \texttt{delta}) = \mathit{ec}(\texttt{s}).$$

3. The function `deltaStar` takes a state s of the non-deterministic finite automaton `nfa` and computes all states that can be reached from the state s when the character c is read. This function satisfies the specification

$$\delta^*(s, c) := \bigcup \big\{ \mathit{ec}(q) \;\big|\; q \in \delta(s, c) \big\}.$$

4. The function `capitalDelta` takes as arguments a set `M` of states of the deterministic automaton and a character `c` and computes $\Delta(M, c)$, which is defined as

$$\Delta(M, c) := \bigcup \big\{ \delta^*(q, c) \;\big|\; q \in M \big\}.$$

5. The function `nfa2dfa` takes as input a non-deterministic finite automaton `nfa`. It takes a non-deterministic FSM $A := \langle Q, \Sigma, \delta, q_0, F \rangle$ and computes

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

where the components of this tuple are given as follows:

   (a) The set of states of $\texttt{det}(A)$ is the set of all subsets of Q and therefore it is equal to the power set 2^Q. Instead of computing the power set, the implementation iteratively computes all possible sets of states that the non-deterministic FSM A could be in.

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

   (c) The function Δ that has been defined previously specified how the set of states change when a character is read.

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

(e) The set of accepting states $\widehat{F}$ is the set of those subsets of Q that contain an accepting state of the FSM Q:

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

4.4 From Regular Expressions to Deterministic Finite State Machines

In this section we show how we can construct a non-deterministic FSM $A(r)$ that accepts the same language as a given 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 $\mathit{accept}\big(A(r)\big)$. Furthermore, there are no transitions from 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

   $$A(\emptyset) = \langle\{q_0, q_1\}, \Sigma, \{\}, q_0, \{q_1\}\rangle.$$

   Note that this FSM has no transitions at all.

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

   Figure 4.7 shows the FSM $A(\emptyset)$. It is obvious that we have $L\big(A(\emptyset)\big) = \{\}$.

2. The FSM $A(\varepsilon)$ is defined as

   $$A(\varepsilon) = \langle\{q_0, q_1\}, \Sigma, \{\langle q_0, \varepsilon\rangle \mapsto q_1\}, q_0, \{q_1\}\rangle.$$

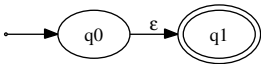
   

   Figure 4.8: The FSM $A(\varepsilon)$.

   Figure 4.8 shows the FSM ε). We have that $L\big(A(\emptyset)\big) = \{\texttt{''}\}$, i.e. the automaton only accepts the empty string.

3. For a letter $c \in \Sigma$ the FSM $A(c)$ is defined as

   $$A(c) = \langle\{q_0, q_1\}, \Sigma, \{\langle q_0, c\rangle \mapsto q_1\}, q_0, \{q_1\}\rangle.$$

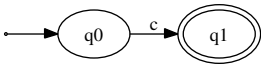
   

   Figure 4.9: The FSM $A(c)$.

   Figure 4.9 shows $A(c)$. We have that $L\big(A(c)\big) = \{c\}$, i.e. the automaton 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) $A(r_1) = \langle Q_1, \Sigma, \delta_1, q_1, \{q_2\}\rangle$,

   (b) $A(r_2) = \langle Q_2, \Sigma, \delta_2, q_3, \{q_4\}\rangle$,

   (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:

   $$A(r_1 \cdot r_2) := \langle Q_1 \cup Q_2, \Sigma, \{\langle q_2, \varepsilon\rangle \mapsto q_3\} \cup \delta_1 \cup \delta_2, q_1, \{q_4\}\rangle$$

   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:

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

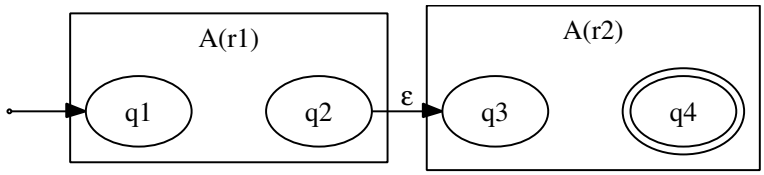


Figure 4.10: The FSM $A(r_1 \cdot r_2)$.

   Figure 4.10 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) $A(r_1) = \langle Q_1, \Sigma, \delta_1, q_1, \{q_3\}\rangle$,

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

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

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

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

   Figure 4.11 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 $A(r_2)$.

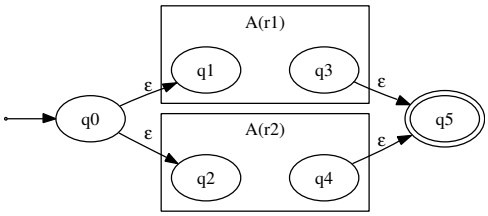


Figure 4.11: The FSM $A(r_1 + r_2)$.

(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:

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

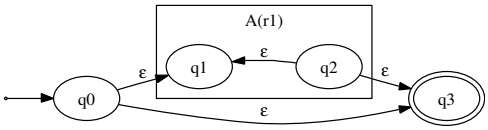


Figure 4.12: The FSM $A(r^*)$.

Figure 4.12 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!

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

Exercise 9: 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 10: Construct a non-deterministic FSM for the regular expression

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

4.5 Übersetzung eines EA in einen regulären Ausdruck

Wir runden die Theorie ab, indem wir zeigen, dass sich zu jedem deterministischen endlichen Automaten A ein regulärer Ausdruck $r(A)$ angeben lässt, der dieselbe Sprache spezifiziert, die von dem Automaten A akzeptiert wird, für den also

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

gilt. Der Automat A habe die Form

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

Für jedes Paar von Zuständen $\langle p_1, p_2 \rangle \in Q \times Q$ definieren wir einen regulären Ausdruck $r(p_1, p_2)$. Die Idee bei dieser Definition ist, dass der reguläre Ausdruck $r(p_1, p_2)$ alle die Strings w spezifiziert, die den Automaten A von dem Zustand p_1 in den Zustand p_2 überführen, formal gilt:

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

Die Definition der regulären Ausdrücke erfolgt über einen Trick: Wir definieren für $k = 0, \cdots, n+1$ reguläre Ausdrücke $r^{(k)}(p_1, p_2)$. Diese reguläre Ausdrücke beschreiben gerade die Strings, die den Automaten A von dem Zustand p_1 in den Zustand p_2 überführen, ohne dass dabei zwischendurch ein Zustand aus der Menge

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

besucht wird. Die Menge Q_k enthält also nur die Zustände, deren Index größer oder gleich k ist. Formal definieren wir dazu die dreistellige Relation

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

Für zwei Zustände $p, q \in Q$ und einen String w soll

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

genau dann gelten, wenn der Automat A von dem Zustand p beim Lesen des Wortes w in den Zustand q übergeht, ohne dabei zwischendurch in einen Zustand aus der Menge Q_k zu wechseln. Mit "zwischendurch" ist hier gemeint, dass die Zustände p und q sehr wohl in der Menge Q_k liegen können, nur die Zwischenzustände dürfen nicht in Q_k liegen. Die formale Definition der Relation $p \overset{w}{\mapsto}_k q$ erfolgt durch Induktion nach der Länge des Wortes w:

I.A.: $|w| \leq 1$. Im Induktions-Anfang haben wir zwei Fälle:

(a) $p \overset{\varepsilon}{\mapsto}_k p$,
denn mit dem leeren Wort kann von p aus nur der Zustand p erreicht werden.

(b) $\delta(p, c) = q \;\Rightarrow\; p \overset{c}{\mapsto}_k q$,
denn wenn der Automat beim Lesen des Buchstabens c von dem Zustand p direkt in den Zustand q übergeht, dann werden zwischendurch keine Zustände aus Q_k besucht, denn es werden überhaupt keine Zustände zwischendurch besucht.

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

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

Wenn der Automat A von dem Zustand p durch Lesen des Buchstabens c in einen Zustand $q \notin Q_k$ übergeht und wenn der Automat dann von diesem Zustand q beim Lesen von v in den Zustand r übergehen kann, ohne dabei Zustände aus Q_k zu benutzen, dann geht der Automat beim Lesen von cv aus dem Zustand p in den Zustand r über, ohne zwischendurch in Zustände aus Q_k zu wechseln.

Damit können wir nun für alle $k = 0, \cdots, n+1$ die regulären Ausdrücke $r^{(k)}(p_1, p_2)$ definieren. Wir werden diese regulären Ausdrücke so definieren, dass hinterher

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

gilt. Die Definition der regulären Ausdrücke $r^{(k)}(p_1, p_2)$ erfolgt durch eine Induktion nach k.

I.A.: $k = 0$.

Dann gilt $Q_0 = Q$, die Menge Q_0 enthält also alle Zustände und damit dürfen wir, wenn wir vom Zustand p_1 in den Zustand p_2 übergehen, zwischendurch überhaupt keine Zustände besuchen.

Wir betrachten zunächst den Fall $p_1 \neq p_2$. Dann kann $p_1 \overset{w}{\mapsto}_0 p_2$ nur dann gelten, wenn w aus einem einzigen Buchstaben besteht. Es sei

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

die Menge aller Buchstaben, die den Zustand p_1 in den Zustand p_2 überführen. Falls diese Menge nicht leer ist, setzen wir

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

Ist die obige Menge leer, so gibt es keinen direkten Übergang von p_1 nach p_2 und wir setzen

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

Wir betrachten jetzt den Fall $p_1 = p_2$. Definieren wir wieder

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

als die Menge aller Buchstaben, die den Zustand p_1 in sich selbst überführen, so können wir in dem Fall, dass diese Menge nicht leer ist,

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

setzen. Ist die obige Menge leer, so gibt es nur den Übergang mit dem leeren Wort von p_1 nach p_1 und wir setzen

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

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

Bei dem Übergang von $r^{(k)}(p_1, p_2)$ zu $r^{(k+1)}(p_1, p_2)$ dürfen wir zusätzlich den Zustand q_k benutzen, denn q_k ist das einzige Element der Menge Q_k, das nicht in der Menge Q_{k+1} enthalten ist. Wird ein String w gelesen, der den Zustand p_1 in den Zustand p_2 überführt, ohne dabei zwischendurch in einen Zustand aus der Menge Q_{k+1} zu wechseln, so gibt es zwei Möglichkeiten:

(a) Es gilt bereits $p_1 \overset{w}{\mapsto}_k p_2$.

(b) Der String w kann so in mehrere Teile $w_1 s_1 \cdots s_l w_2$ aufgeteilt werden, dass gilt

- $p_1 \overset{w_1}{\mapsto}_k q_k$,
  von dem Zustand p_1 gelangt der Automat also beim Lesen von w_1 zunächst in den Zustand q_k, wobei zwischendurch der Zustand q_k nicht benutzt wird.
- $q_k \overset{s_i}{\mapsto}_k q_k$ für alle $i = \{1, \cdots, l\}$,
  von dem Zustand q_k wechselt der Automat beim Lesen der Teilstrings s_i wieder in den Zustand q_k.
- $q_k \overset{w_2}{\mapsto}_k p_2$,
  schließlich wechselt der Automat von dem Zustand q_k in den Zustand p_2, wobei der Rest w_2 gelesen wird.

Daher definieren wir

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

Dieser Ausdruck kann wie folgt gelesen werden: Um von p_1 nach p_2 zu kommen, ohne dabei zwischendurch Zustände aus Q_{k+1} zu benutzen, kann der Automat entweder direkt von p_1 nach p_2 gelangen, ohne zwischendurch Zustände aus Q_k zu benutzen, was dem Ausdruck $r^{(k)}(p_1, p_2)$ entspricht, oder aber der Automat wechselt von p_1 ein erstes Mal in den Zustand q_k, was den Ausdruck $r^{(k)}(p_1, q_k)$ erklärt, wechselt dann beliebig oft von q_k nach q_k, was den Ausdruck $\left(r^{(k)}(q_k, q_k)\right)^*$ erklärt und wechselt schließlich von q_k in den Zustand p_2, wofür der Ausdruck $r^{(k)}(q_k, p_2)$ steht.

Nun haben wir alles Material zusammen, um die Ausdrücke $r(p_1, p_2)$ definieren zu können. Wir setzen

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

Dieser reguläre Ausdruck beschreibt die Wörter, die den Automaten von dem Zustand p_1 in den Zustand p_2 überführen, ohne dass der Automat dabei in einen Zustand der Menge Q_{n+1} wechselt. Nun gilt aber

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

die Menge ist also leer! Folglich werden durch den regulären Ausdruck $r^{(n+1)}(p_1, p_2)$ überhaupt keine Zustände ausgeschlossen: Der Ausdruck beschreibt also genau die Strings, die den Zustand p_1 in den Zustand p_2 überführen, es gilt also

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

Um nun einen regulären Ausdruck konstruieren zu können, der die Sprache des Automaten A beschreibt, schreiben wir die Menge F der akzeptierenden Zustände von A als

$$F = \{t_1, \cdots, t_m\}$$

und definieren den regulären Ausdruck $r(A)$ als

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

Dieser Ausdruck beschreibt genau die Strings, die den Automaten A aus dem Start-Zustand in einen der akzeptierenden Zustände überführen. □

Damit sehen wir jetzt, dass die Konzepte "*deterministischer endlicher Automat*" und "*regulärer Ausdruck*" äquivalent sind.

1. Jeder deterministische endliche Automat kann in einen äquivalenten regulären Ausdruck übersetzt werden.
2. Jeder reguläre Ausdruck kann in einen äquivalenten nicht-deterministischen endlichen Automaten transformiert werden.
3. Ein nicht-deterministischer endlicher Automat lässt sich durch die Teilmengen-Konstruktion in einen endlichen Automaten überführen.

Aufgabe 11: Konstruieren Sie für den in Abbildung 4.1 gezeigten endlichen Automaten einen äquivalenten regulären Ausdruck.

Lösung: Der Automat hat die Zustände 0 und 1. Wir berechnen zunächst die regulären Ausdrücke $r^{(k)}(i, j)$ für alle $i, j \in \{0, 1\}$ der Reihe nach für die Werte $k = 0$, 1 und 2:

1. Für $k = 0$ finden wir:
   (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. Für $k = 1$ haben wir:
   (a) Für $r^{(1)}(0, 0)$ finden wir:

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

   wobei wir im letzten Schritt die für reguläre Ausdrücke allgemeingültige Umformung

$$\begin{aligned} r + r \cdot r^* \cdot r &= r \cdot (\varepsilon + r^* \cdot r) \\ &= r \cdot r^* \end{aligned}$$

   verwendet haben. Setzen wir für $r^{(0)}(0, 0)$ den oben gefundenen Ausdruck $\mathsf{a} + \varepsilon$ ein, so erhalten wir

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

Wegen $(\mathtt{a} + \varepsilon) \cdot (\mathtt{a} + \varepsilon)^* = \mathtt{a}^*$ haben wir insgesamt

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

(b) Für $r^{(1)}(0,1)$ finden wir:

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

(c) Für $r^{(1)}(1,0)$ finden wir:

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

(d) Für $r^{(1)}(1,1)$ finden wir

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

3. Für $k = 2$ müssen wir lediglich den regulären Ausdruck $r^{(2)}(0,1)$ berechnen. Es gilt

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

Da der Zustand 0 der Start-Zustand und der Zustand 1 der einzige akzeptierende Zustand ist, können wir nun den regulären Ausdruck $r(A)$ angeben:

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

Dieses Ergebnis, das wir mühevoll abgeleitet haben, hätten wir auch durch einen einfachen Blick auf den Automaten erhalten können, aber die oben gezeigte Rechnung formalisiert das, was der geübte Betrachter unmittelbar erkennt und der beschriebene Algorithmus hat den Vorteil, dass er sich implementieren lässt. □

Aufgabe 12: Konstruieren Sie für den in Abbildung 4.13 auf Seite 46 gezeigten endlichen Automaten einen regulären Ausdruck, der dieselbe Sprache beschreibt, die von dem abgebildeten Automaten erkannt wird.

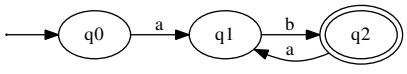


Figure 4.13: Ein deterministischer endlicher Automat.

```
1  dfa2RegExp := procedure(dfa) {
2      [states, sigma, delta, q0, accepting] := dfa;
3      return regexpSum({ rpq(q0, p, sigma, delta, states) : p in accepting });
4  };
5  regexpSum := procedure(s) {
6      match (s) {
7          case {}:
8               return 0;
9          case { r }:
10              return r;
11         case { r | rs }:
12              return Or(r, regexpSum(rs));
13     }
14 };
15 rpq := procedure(p1, p2, sigma, delta, allowed) {
16     match (allowed) {
17         case {}:
18              allChars := { c : c in sigma | delta(p1, c) == p2 };
19              sum := regexpSum(allChars);
20              if (p1 == p2) {
21                  if (allChars == {}) {
22                      return "";
23                  } else {
24                      return Or("", sum);
25                  }
26              } else {
27                  return sum;
28              }
29         case { qk | restAllowed }:
30              rkp1p2 := rpq(p1, p2, sigma, delta, restAllowed);
31              rkp1qk := rpq(p1, qk, sigma, delta, restAllowed);
32              rkqkqk := rpq(qk, qk, sigma, delta, restAllowed);
33              rkqkp2 := rpq(qk, p2, sigma, delta, restAllowed);
34              return Or(rkp1p2, Cat(Cat(rkp1qk, Star(rkqkqk)), rkqkp2));
35     }
36 };
```

Figure 4.14: Converting a DFA into a regular expression.

4.5.1 Implementing the Conversion of FSMs into Regular Expressions

Figure 4.14 on page 47 shows how to implement the conversion of a finite state machine into a regular expression. We discuss the details of this implementation next.

1. The function `dfa2RegExp` takes a deterministic finite state machine `dfa` as input. For every accepting state p of the given `dfa`, it calculates the regular expression $r(q_0, p)$, which describes those strings that transform the finite state machine from the start state q_0 into the state p. If $\{p_1, \cdots, p_k\}$ is the set of all accepting states of `dfa`, then the regular expression

   $$r(q_0, p_1) + \cdots + r(q_0, p_k)$$

   describes the language accepted by `dfa`. This regular expression is computed in line 3 via the two functions `regexpSum` and `rpq`.

2. The function `regexpSum` takes as input a set s of regular expressions. If s has the form

   $$\{r_1, \cdots, r_k\},$$

   then the regular expression

   $$r_1 + \cdots + r_k$$

   is returned. There are two special cases:

   (a) If s is empty, then the regular expression $\emptyset$ is returned.

   (b) If s contains just one element, that is if s has the form $s = \{r\}$, then r is returned.

3. The function `rpq` is called with 5 arguments:

   (a) The first two arguments p_1 and p_2 are states of a finite state machine. The idea is that the call

   ```
   rpq(p1, p2, sigma, delta, allowed)
   ```

   computes the regular expression $r(\texttt{p1}, \texttt{p2})$, which is the expression that describes the set of those strings s that take the finite state machine from the state `p1` to the state `p2`.

   (b) `sigma` is the alphabet of the finite state machine.

   (c) `delta` is the transition function of the finite state machine. In this program, we have chosen to represent `delta` as a SetlX `procedure` rather than as a binary relation.

   (d) `allowed` is the set of all states that are allowed as intermediate states in the transition of the FSM from the state p_1 into the state p_2. This set corresponds to the set $\{q_0, q_1, \cdots, q_n\} - Q_k$ in the definition of the regular expression $r^{(k)}(p_1, p_2)$, i.e. the set `allowed` is the complement of the set Q_k with respect to the set of all states.

   The function `rpq` is defined by recursion on its last argument.

   (a) The base case of the recursion is the case where the set `allowed` is empty. Then the regular expression returned by the function `rpq` must only specify those strings that take the FSM from the state p_1 to the state p_2 without visiting any other state in between. In line 18, the function computes the set of all characters c that take the FSM from the state p_1 into the state p_2 directly, i.e. that satisfy

   $$\delta(p_1, c) = p_2.$$

   If this set is given as $\{c_1, \cdots, c_k\}$, then the variable `sum` gets the value

   $$c_1 + \cdots + c_k.$$

   Of course, if the set $\{c_1, \cdots, c_k\}$ is empty, the sum $c_1 + \cdots + c_k$ has to be interpreted as the regular expression $\emptyset$. Now there are two cases:

   i. $p_1 = p_2$. In this case, the empty string ε transforms the state p_1 into p_2 and therefore in this case the result returned is

   $$c_1 + \cdots + c_k + \varepsilon.$$

   ii. $p_1 \neq p_2$. Then the result is given as

   $$c_1 + \cdots + c_k.$$

   The function `regexpSum` is used to compute the sum $c_1 + \cdots + c_k$. It takes care to return $\emptyset$ if the set $\{c_1, \cdots, c_k\}$ is empty.

   (b) In the recursive case, we arbitrarily pick a state q_k from the set `allowed` of states that may be used to move from state p_1 to p_2. The state is removed from the set `allowed` to produce the set `restAllowed`. Then, the regular expression returned has the form

   $$r(p_1, p_2) + r(p_1, q_k) \cdot \big(r(q_k, q_k)\big)^* \cdot r(q_k, p_2).$$

   Here, $r(p_1, p_2)$ is computed recursively as the regular expression that takes the FSM from the state p_1 to the state p_2 using only the states from the set `restAllowed`. The regular expressions $r(p_1, q_k)$, $r(q_k, q_k)$,

and $r(q_k, p_2)$ are defined in a similar way. The reasoning for returning this result is as follows: In order to get from state p_1 to state p_2, there are two possibilities:

- If we do not need to visit the state q_k when transforming the FSM from state p_1 into state p_2, then the regular expression

  $$r(p_1, p_2)$$

  already describes the transition.
- If we do need to visit the state q_k, then we move from p_1 to p_2 in three steps:
  - We move from p_1 to q_k.
  - Next, we can move from q_k to q_k as many times as we wish.
  - Finally, we move from q_k to p_2.

  These three steps are summarized by the regular expression

  $$r(p_1, q_k) \cdot \big(r(q_k, q_k)\big)^* \cdot r(q_k, p_2).$$

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]: We have proven in this chapter that for every regular expression r there is a non-deterministic finite state machine n such that n recognizes the same set of strings that is specified by r. Next, we have seen that every non-deterministic finite state machine n can be transformed into a deterministic finite state machine f which accepts the same language as f. Finally, we have seen that given a finite state machine f there is an algorithm to construct a regular expression r such that r describes the language recognized by f. Hence the notions of *finite state machines* and *regular expressions* are equivalent.

Chapter 5

Minimierung endlicher Automaten*

In diesem Abschnitt zeigen wir ein Verfahren, mit dem die Anzahl der Zustände eines deterministischen endlichen Automaten

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

minimiert werden kann. Ohne Beschränkung der Allgemeinheit wollen wir dabei voraussetzen, dass der Automat A vollständig ist: Wir nehmen also an, dass der Ausdruck $\delta(q, c)$ für jeden Zustand $q \in Q$ und jeden Buchstaben $c \in \Sigma$ als Ergebnis einen Zustand aus Q liefert. Wir suchen dann einen deterministischen endlichen Automaten

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

der dieselbe Sprache akzeptiert wie der Automat A, für den also

$$L(A^-) = L(A)$$

gilt und für den die Anzahl der Zustände der Menge Q^- minimal ist. Um diese Konstruktion durchführen zu können, müssen wir etwas ausholen. Zunächst erweitern wir die Funktion

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

zu einer Funktion $\hat{\delta}$, die als zweites Argument nicht nur einen Buchstaben sondern auch einen String akzeptiert:

$$\hat{\delta} : Q \times \Sigma^* \to Q.$$

Der Funktions-Aufruf $\hat{\delta}(q, s)$ soll den Zustand p berechnen, in den der Automat A gelangt, wenn der Automat im Zustand q den String s liest. Die Definition von $\hat{\delta}(q, s)$ erfolgt durch Induktion über die Länge des Strings s:

I.A.: $\hat{\delta}(q, \varepsilon) = q$,

I.S.: $\hat{\delta}(q, cs) = \hat{\delta}\big(\delta(q, c), s\big)$, falls $c \in \Sigma$ und $s \in \Sigma^*$.

Da die Funktion $\hat{\delta}$ eine Verallgemeinerung der Funktion δ ist, werden wir in der Notation nicht zwischen δ und $\hat{\delta}$ unterscheiden und einfach nur δ schreiben.

Offensichtlich können wir in einem endlichen Automaten $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ alle die Zustände $p \in Q$ entfernen, die vom Start-Zustand aus nicht *erreichbar* sind. Dabei heißt ein Zustand p *erreichbar* genau dann, wenn es einen String $w \in \Sigma^*$ gibt, so dass

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

gilt. Wir wollen im Folgenden daher voraussetzen, dass alle Zustände des betrachteten endlichen Automaten vom Start-Zustand aus erreichbar sind.

Im Allgemeinen können wir einen Automaten dadurch minimieren, dass wir bestimmte Zustände identifizieren. Betrachten wir beispielsweise den in Abbildung 5.1 gezeigten Automaten, so können wir dort die Zustände q_1 und q_2 sowie q_3 und q_4 identifizieren, ohne dass sich dadurch die Sprache des Automaten ändert. Die zentrale Idee bei der Minimierung eines Automaten besteht darin, dass wir uns überlegen, welche Zustände wir auf keinen Fall identifizieren dürfen und einfach alle anderen Zustände als äquivalent betrachten.

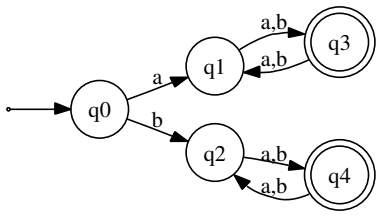


Figure 5.1: Ein endlicher Automat mit äquivalenten Zuständen.

Definition 12 (Separable States) Assume $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ 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. $\delta(p_1, s) \in F$ and $\delta(p_2, s) \notin F$ or
2. $\delta(p_1, s) \notin F$ and $\delta(p_2, s) \in F$

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 are not equivalent. We define an equivalence relation $\sim$ on the set Q of all states by setting

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

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 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.

Es bleibt die Frage zu klären, wie wir feststellen können, welche Zustände unterscheidbar sind. Eine Möglichkeit besteht darin, eine Menge V von Paaren von Zustände anzulegen. Wir fügen das Paar $\langle p, q \rangle$ in die Menge V ein, wenn wir erkannt haben, dass p und q unterscheidbar sind. Wir erkennen p und q als unterscheidbar, wenn es einen Buchstaben $c \in \Sigma$ und zwei Zustände s und t gibt, so dass

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

gilt. Diese Idee liefert einen Algorithmus, der aus zwei Schritten besteht:

1. Zunächst initialisieren wir V mit alle den Paaren $\langle p, q \rangle$, für die entweder p ein akzeptierender Zustand und q kein akzeptierender Zustand ist, oder umgekehrt q ein akzeptierender Zustand und p kein akzeptierender Zustand ist, denn ein akzeptierender Zustand kann durch den leeren String ε von einem nicht-akzeptierenden Zustand unterschieden werden:

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

2. Solange wir ein neues Paar $\langle p, q \rangle \in Q \times Q$ finden, für dass es einen Buchstaben c gibt, so dass die Zustäne $\delta(p, c)$ und $\delta(q, c)$ bereits unterscheidbar sind, fügen wir dieses Paar zur Menge V hinzu:

```
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⟩ };
   }
}
```

Haben wir alle Paare $\langle p, q \rangle$ von unterscheidbaren Zuständen gefunden, so können wir anschließend alle Zustände p und q identifizieren, die nicht unterscheidbar sind, für die also $\langle p, q \rangle \notin V$ gilt. Es lässt sich zeigen, dass der so konstruierte Automat tatsächlich minimal ist.

Beispiel: Wir betrachten den in Abbildung 5.1 gezeigten endlichen Automaten und wenden den oben skizzierten Algorithmus auf diesen Automaten an. Wir bedienen uns dazu einer Tabelle, deren Spalten und Zeilen mit den verschiedenen Zuständen durchnummeriert sind. Wenn wir im ersten Schritt erkannt haben, dass die Zustände i und j unterscheidbar sind, so fügen wir in dieser Tabelle in der i-ten Zeile und der j-ten Spalte eine 1 ein. Da mit den Zuständen i und j auch die Zustände j und i unterscheidbar sind, fügen wir außerdem in der j-ten Zeile und der i-ten Spalte ebenfalls eine 1 ein.

1. Im ersten Schritt erkennen wir, dass die beiden akzeptierenden Zustände q_3 und q_4 von allen nicht-akzeptierenden Zuständen unterscheidbar sind. Also sind die Paare $\langle q_0, q_3\rangle$, $\langle q_0, q_4\rangle$, $\langle q_1, q_3\rangle$, $\langle q_1, q_4\rangle$, $\langle q_2, q_3\rangle$ und $\langle q_2, q_4\rangle$ unterscheidbar. Damit hat die Tabelle nun die folgende Gestalt:

	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. Als nächstes erkennen wir, dass die Zustände q_0 und q_1 unterscheidbar sind, denn es gilt

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

   Genauso sehen wir, dass die Zustände 0 und 2 unterscheidbar sind, denn es gilt

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

   Da wir im zweiten Schritt nun gefunden haben, dass $q_0 \not\sim q_1$ und $q_0 \not\sim q_2$ gilt, tragen wir in der Tabelle an den entsprechenden Stellen eine 2 ein. Damit hat die Tabelle jetzt die folgende Gestalt:

	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. Nun finden wir keine weiteren Paare von unterscheidbaren Zuständen mehr, denn wenn wir das Paar $\langle q_1, q_2\rangle$ betrachten, sehen wir

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

   aber da die Zustände 3 und 4 bisher nicht unterscheidbar sind, liefert dies kein neues unterscheidbares Paar. Genausowenig liefert

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

   ein neues unterscheidbares Paar. Jetzt bleiben noch die beiden Zustände q_3 und q_4. Hier finden wir

   $$\delta(q_3, c) = q_1 \quad \text{und} \quad \delta(q_4, c) = q_2 \quad \text{für alle } c \in \{\mathsf{a}, \mathsf{b}\}$$

   und da die Zustände q_1 und q_2 bisher nicht als unterscheidbar bekannt sind, haben wir keine neuen unterscheidbaren Zustände gefunden. Damit können wir die äquivalenten Zustände aus der Tabelle ablesen, es gilt:

   (a) $q_1 \sim q_2$

   (b) $q_3 \sim q_4$

   Abbildung 5.2 zeigt den entsprechenden reduzierten endlichen Automaten.

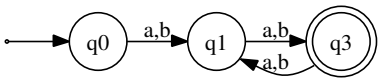


Figure 5.2: Der reduzierte endliche Automat.

Aufgabe 13: Konstruieren Sie den minimalen deterministischen endlichen Automaten, der die Sprache $L(a \cdot (b \cdot a)^*)$ erkennt. Gehen Sie dazu in folgenden Schritten vor:

(a) Berechnen Sie einen nicht-deterministischen endlichen Automaten, der diese Sprache erkennt.

(b) Transformieren Sie diesen Automaten in einen deterministischen Automaten.

(c) Minimieren Sie die Zahl der Zustände dieses Automaten mit dem oben angegebenen Algorithmus.

5.1 Implementing the Minimization of Finite Automata in SetlX

Figure 5.3 on page 54 shows the function `minimize` that takes a determininistic finite state machine `fa` as input. The function eliminates all states from `fa` that are not reachable from the start state and then tries to minimize equivalent states as discussed in the previous section. It returns a finite state machine that accepts the same language as `fa` and that, furthermore, is guaranteed to have as few states as possible. The implementation works as discussed below.

1. First, all states *reachable* from the start state q_0 are computed. Here, a state p is *reachable* from the state q_0 iff there is a string s such that $\delta(q_0, s) = p$. The computation of the reachable states is done via a fixpoint computation:

   (a) Since $\delta(q_0, \varepsilon) = q_0$, the state q_0 is reachable from q_0 and therefore we initialize the set `reachable` with the start state q_0.

   (b) The remaining reachable states are found by a fixpoint iteration. Given a state q, the function

   ```
   q |=> { delta(q, c) :  c in sigma }
   ```

   computes the set of all states reachable from q by reading some letter $c \in \Sigma$.

   Using the second order function `fixpoint` discussed in the previous chapter in Figure 4.6 on page 38, the set of all states reachable from q_0 is then found by iterating the function

   ```
   q |=> { delta(q, c) :  c in sigma }
   ```

   until no new states are found.

2. Next, we try to find all pairs of states that are *separable*. Remember, a pair $\langle p, q \rangle$ is called *separable* if there is a string s such that either $\delta(p, s)$ is accepting while $\delta(q, s)$ is not accepting, or $\delta(p, s)$ is not accepting while $\delta(q, s)$ is accepting.

   (a) Initially, we know that a pair $\langle p, q \rangle$ is separable if either p is a member of the set `accepting` of accepting states while q is not a member of `accepting` or it is the other way around: $p \notin$ `accepting` and $q \in$ `accepting`.

   Therefore, the set `separable` is initialzed as the set

   $$(\texttt{states} \backslash \texttt{accepting}) \times \texttt{accepting} \cup \texttt{accepting} \times (\texttt{states} \backslash \texttt{accepting}).$$

   This expression is coded in SetlX in line 4. Note that the set difference $a \backslash b$ of two sets a and b is written as a-b in SetlX, while the *cartesian product* $a \times b$ of a and b is written as a><b. Remember that the cartesian product of two sets $a \times b$ is defined as

```
1  minimize := procedure(fa) {
2      [states, sigma, delta, q0, accepting] := fa;
3      states    := fixpoint( {q0}, q |=> { delta[q, c] : c in sigma });
4      separable := (states-accepting) >< accepting + accepting >< (states-accepting);
5      moreSep   :=
6          procedure(knownSep) {
7              return { [q1, q2] : [q1, q2] in states >< states
8                     | exists (c in sigma | [delta[q1, c], delta[q2, c]] == knownSep)
9                     };
10         };
11     allSeparable := fixpoint(separable, moreSep);
12     equivalent   := states >< states - allSeparable;
13     equivClasses := { { p : p in states | [p, q] in equivalent }: q in states };
14     newQ0        := arb({ m : m in equivClasses | q0 in m });
15     newAccept    := { m : m in equivClasses | arb(m) in accepting };
16     newDelta     := {};
17     for (q in states, c in sigma) {
18         p := delta[q, c];
19         if (p != om) {
20             classOfP := findEquiv(p, equivClasses);
21             classOfQ := findEquiv(q, equivClasses);
22             newDelta += { [[classOfQ, c], classOfP] };
23         }
24     }
25     return [equivClasses, sigma, newDelta, newQ0, newAccept];
26 };
27 findEquiv := procedure(p, eqClasses) {
28     return first({ cl : cl in eqClasses | p in cl });
29 };
```

Figure 5.3: A procedure to minimize a finite state machine.

$$a \times b := \{\langle x, y\rangle \mid x \in a \wedge y \in b\}.$$

(b) Next, if the states $\delta(q_1, c)$ and $\delta(q_2, c)$ are already known to be separable, then the states q_1 and q_2 are also separable. The reasoning is as follows: As $\delta(q_1, c)$ and $\delta(q_2, c)$ are separable, there is a string s such that

$$\delta(\delta(q_1, c), s) \text{ is accepting} \quad \text{while} \quad \delta(\delta(q_2, c), s) \text{ is not accepting}$$

or the other way arround. As we have $\delta(\delta(q_1, c), s) = \delta(q_1, cs)$ and $\delta(\delta(q_2, c), s) = \delta(q_2, cs)$ this can be rewritten as

$$\delta(q_1, cs) \text{ is accepting} \quad \text{while} \quad \delta(q_2, cs) \text{ is not accepting}$$

or the other way around. By the definition of two states being separable this implies that q_1 and q_2 are separable. Hence, the set of all pairs of separable states can be found by a fixpoint iteration. The procedure `moreSep` defined in line 5 takes a pair of states `knownSep` that are already known to be separable. For any pair of states $\langle q_1, q_2\rangle$ and any character $c \in \Sigma$ it then checks whether the pair

$$\langle \delta(q_1, c), \delta(q_2, c)\rangle$$

equals the pair `knownSep`, because then $\langle q_1, q_2\rangle$ is separable, too. Using this function, the set of all separable pairs can then be computed via a straightforward fixpoint iteration in line 11.

I have to admit that the current implementation of the separable states could be done more efficiently.

However, this would have complicated the program and for the purpose of the examples presented in this lecture the efficiency is sufficient.

3. Next, we *identify* those states that are *equivalent*: Two states p and q are called *equivalent* if and only if they are not separable.

   There are several way to identify equivalent states. The easiest way is to compute the associated *equivalence classes*, where an equivalence class contains all thoses states that are equivalent to each other.

   (a) Therefore, the set of states of the minimized finite state machine is the set of equivalence classes of states of the given finite state machine `fa`. For example, if the set of states of `fa` is

   $$\{\ q_0,\ q_1,\ q_2,\ q_3,\ q_4,\ q_5\ \}$$

   and the state q_1 is equivalent to q_2 and, furthermore, the states q_3, q_4, and q_5 are pairwise equivalent, then the set of equivalence classes is given as

   $$\{\ \{q_0\},\ \{q_1,\ q_2\},\ \{q_3,\ q_4,\ q_5\}\ \}.$$

   This set of equivalence classes is the new set of states.

   (b) Of course, the new start state is the set of equivalent states that contains the start state q_0 of the given finite state machine `fa`. In line 14 we collect the set of all equivalence classes that contain `q0`. Of course, there can be just one equivalence class. Hence we can extract this one class using the function `arb`.

   (c) A set of equivalent states is accepting if any of its member states is an accepting state. Of course, if a set of equivalent states contains an accepting state, then the other states in this equivalence class have to be accepting also, since otherwise these states would be separable from the accepting state and therefore could not be equivalent.

4. In order to compute the new state transition function, we have to construct a function that takes a set of equivalent states of the old finite state machine `fa` and turns this set into a new set of states that are again equivalent. This is done by taking all states `q` and all characters `c` in `sigma` and computing

   $$\mathtt{p} = \delta(\mathtt{q}, \mathtt{c}).$$

   Now, if `classOfQ` is the equivalence class containing `q` and likewise `classOfP` is the equivalence class containing `p`, then we have

   $$\mathtt{classOfP} = \mathtt{newDelta}(\mathtt{classOfQ}, \mathtt{c}).$$

   Here, `newDelta` is the transition function of the minimized finite state machine.

5.2 The Theorem of Nerode

A language is called a *regular* language iff there is a finite state machine A recognizing the language. We have already seen that a language is regular iff it is accepted by a finite state machine. In this section we discuss a theorem that can be used to prove that a given language is not a regular language. The main idea is to extend the notion of *separability* from states to strings.

Definition 13 (separable) Given an alphabet Σ and a formal language $L \subseteq \Sigma^*$, a pair of strings $\langle s, t\rangle \in \Sigma^* \times \Sigma^*$ is called *separable with respect to* L iff there is a string $w \in \Sigma^*$ such that

$$(sw \in L \wedge tw \notin L) \vee (sw \notin L \wedge tw \in L).$$

In this case, w is the *witness of the separability* of $\langle s, t\rangle$. If the pair $\langle s, t\rangle$ is separable with respect to L and if the language L is obvious from the context, then in order to shorten our notation, we call s and t separable. □

Example: Take $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and define L as the language of all strings of the form $a^n b^n$ where n is a natuaral number, i. e. define

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

Then the strings $s := \mathtt{aaab}$ and $t := \mathtt{bba}$ are separable and $w := \mathtt{bb}$ is a witness of separability because

$$sw = \mathtt{aaabbb} \in L \quad \text{but} \quad tw = \mathtt{bbabb} \notin L. \qquad \diamond$$

Exercise 14: Assume Σ is an alphabet and $L \subseteq \Sigma^*$ is a formal language. Define a relation $\sim_L$ on Σ^* as follows:

$$s_1 \sim_L s_2 \quad \text{iff} \quad s_1 \text{ and } s_2 \text{ are } \underline{\text{not}} \text{ separable with respect to } L.$$

Prove that the relation $\sim_L$ is an equivalence relation! $\diamond$

The following theorem has been proven by Anil Nerode in 1958 [Ner58]. It can be used to show that certain languages are not regular.

Theorem 14 (Nerode) If $L \subseteq \Sigma^*$ is a formal language and $S = \{s_1, \cdots, s_n\} \subseteq \Sigma^*$ is a set of strings that are pairwise separable with respect to L and, furthermore, A is a determininistic finite state machine recognizing the language L, then A has at least n different states.

Proof: Assume $A = \langle Q, \Sigma, \delta, q_0, F\rangle$ is a DFA accepting the language L, i. e. $L(A) = L$. Define states

$$q_i := \delta(q_0, s_i) \quad \text{for all } i \in \{1, \cdots, n\}.$$

The claim is that all these states are pairwise different, that is we have

$$\forall i, j \in \{1, \cdots, n\} : i \neq j \rightarrow q_i \neq q_j.$$

Therefore, assume that we have $i \neq j$. By our assumption, the strings s_i and s_j are separable. Then there is a witness $w \in \Sigma^*$ such that

$$(s_i w \in L \wedge s_j w \notin L) \vee (s_i w \notin L \wedge s_j w \in L).$$

These are two cases which we consider separately.

1. $s_i w \in L \wedge s_j w \notin L$.

   Since the DFA A accepts the language L, we know that

   $$\delta(q_0, s_i w) \in F \ \wedge \ \delta(q_0, s_j w) \notin F.$$

   On the other hand, we have

   $$\delta(q_0, s_i w) = \delta(\delta(q_0, s_i), w) = \delta(q_i, w) \quad \text{and} \quad \delta(q_0, s_j w) = \delta(\delta(q_0, s_j), w) = \delta(q_j, w).$$

   From this we can conclude

   $$\delta(q_i, w) \in F \ \wedge \ \delta(q_j, w) \notin F.$$

   This is only possible if $q_i \neq q_j$.

2. $s_i w \notin L \wedge s_j w \in L$.

   If we exchange the roles of i and j, this case is reduced to the previous case and we can again conclude that $q_i \neq q_j$.

We have just shown that $q_i \neq q_j$ as long as $i \neq j$. Therefore the states $\{q_1, \cdots, q_n\}$ are all pairwise different and the finite state machine A needs to have at least n different states. $\square$

Corollary 15 If $L \subseteq \Sigma^*$ is a formal language such that for every natural number $n \in \mathbb{N}$ there is a set of states $\{s_1, \cdots, s_n\}$ that are pairwise separable with respect to L, then L is not regular.

Proof: The proof of this claim is indirect. Assume that L is regular. Then there is a finite state machine

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

such that $L = L(F)$. Define $n := \mathit{card}(Q) + 1$ where $\mathit{card}(Q)$ denotes the number of elements of Q. By our assumption, there is a set of strings $\{s_1, \cdots, s_n\}$ that are pairwise separable. By the theorem of Nerode the finite state machine F must then have at least $n = \mathsf{card}(Q) + 1$ states, contradicting the fact that F has only $\mathit{card}(Q)$ states. □

Example: We prove that the language

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

is not regular. The proof is done by contradiction. Assume L is regular. Then there is a DFA

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

that recognizes L. Next, pick an arbitrary natural number n and consider the following set of strings:

$$S := \{\mathsf{a}^1, \mathsf{a}^2, \cdots, \mathsf{a}^n\}.$$

S contains n strings and we claim that these strings are all pairwise separable. In order to see this, take $i, j \in \{1, \cdots, n\}$ such that $i \neq j$ and consider the strings a^i and a^j. The witness b^i separates these strings because

$$\mathsf{a}^i\mathsf{b}^i \in L \quad \text{but} \quad \mathsf{a}^j\mathsf{b}^i \notin L$$

since $j \neq i$. Since n was arbitrary, the corollary to the theorem of Nerode now shows that the language L can not be a regular language. □

Exercise 15: Prove that the language

$$\{\mathsf{a}^{k^2} \mid k \in \mathbb{N}\}$$

is not regular. ◇

Hint: Use the fact that the gap between the two successive square numbers k^2 and $(k+1)^2$ has size $2 \cdot k + 1$.

Exercise 16: Prove that the language

$$\{\mathsf{a}^p \mid p \in \mathbb{N} \wedge p \text{ is prime}\}$$

is not regular.

Hint: It is known that the number of primes is infinite and that there are gaps of arbitrary size between the prime numbers, so given an arbitrary natural number k, there is a pair of primes $\langle p_1, p_2 \rangle$ such

$$p_1 + k < p_2$$

and none of the natural numbers between p_1 and p_2 is prime. ◇

Chapter 6

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:

1. The union $L_1 \cup L_2$ of two regular languages L_1 and L_2 is a regular language.
2. The intersection $L_1 \cap L_2$ of two regular languages L_1 und L_2 is a regular language.
3. The complement $\Sigma^* \backslash L$ of a regular language L is a regular language.

As an application of these closure properties we 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 the language

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

is not regular.

6.1 Abschluss-Eigenschaften regulärer Sprachen

In diesem Abschnitt zeigen wir, dass reguläre Sprachen unter den Boole'schen Operationen *Vereinigung*, *Durchschnitt* und *Komplement* abgeschlossen sind. Wir beginnen mit der Vereinigung.

Satz 16 Sind L_1 und L_2 reguläre Sprachen, so ist auch die Vereinigung $L_1 \cup L_2$ eine reguläre Sprache.

Beweis: Da L_1 und L_2 reguläre Sprachen sind, gibt es reguläre Ausdrücke r_1 und r_2, so dass

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

gilt. Wir definieren $r := r_1 + r_2$. Offenbar gilt

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

Damit ist klar, dass auch $L_1 \cup L_2$ eine reguläre Sprache ist. □

Satz 17 Sind L_1 und L_2 reguläre Sprachen, so ist auch der Durchschnitt $L_1 \cap L_2$ eine reguläre Sprache.

Beweis: Während der letzte Satz unmittelbar aus der Definition der regulären Ausdrücke gefolgert werden kann, müssen wir nun etwas weiter ausholen. Im vorletzten Kapitel haben wir gesehen, dass es zu jedem regulären Ausdruck r einen äquivalenten deterministischen endlichen Automaten A gibt, der die durch r spezifizierte Sprache akzeptiert und wir können außerdem annehmen, dass dieser Automat vollständig ist. Es seien r_1 und r_2 reguläre Ausdrücke, die die Sprachen L_1 und L_2 spezifizieren:

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

Dann konstruieren wir zunächst zwei vollständige deterministische endliche Automaten A_1 und A_2, die diese Sprachen akzeptieren, es gilt also

$$L(A_1) = L_1 \quad \text{und} \quad L(A_2) = L_2.$$

Wir werden für die Sprache $L_1 \cap L_2$ einen Automaten A bauen, der diese Sprache akzeptiert. Da es zu jedem Automaten auch einen regulären Ausdruck gibt, der die Sprache beschreibt, die von dem Automaten akzeptiert wird, haben wir damit dann gezeigt, dass die Sprache $L_1 \cap L_2$ regulär ist. Als Baumaterial für den Automaten A, der die Sprache $L_1 \cap L_2$ akzeptiert, verwenden wir natürlich die Automaten A_1 und A_2. Wir nehmen an, dass

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

gilt und definieren A als ein verallgemeinertes kartesisches Produkt der Automaten A_1 und A_2:

$$A := \langle Q_1 \times Q_2, \Sigma, \delta, \langle q_1, q_2 \rangle, F_1 \times F_2 \rangle,$$

wobei die Zustands-Übergangs-Funktion

$$\delta : (Q_1 \times Q_2) \times \Sigma \to Q_1 \times Q_2$$

durch die Gleichung

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

definiert wird. Der so definierte endliche Automat A simuliert gleichzeitig die beiden Automaten A_1 und A_2 indem er parallel berechnet, in welchem Zustand die Automaten A_1 und A_2 jeweils sind. Damit das möglich ist, bestehen die Zustände von A aus Paaren $\langle p_1, p_2 \rangle$, so dass p_1 ein Zustand von A_1 und p_2 ein Zustand von A_2 ist und die Funktion δ berechnet den Nachfolgezustand zu $\langle p_1, p_2 \rangle$, indem gleichzeitig die Nachfolgezustände von p_1 und p_2 berechnet werden. Ein String wird genau dann akzeptiert, wenn sowohl der Automat A_1 als auch der Automat A_2 einen akzeptierenden Zustand erreicht haben. Daher wird die Menge der akzeptierenden Zustände wie folgt definiert:

$$F := \big\{ \langle p_1, p_2 \rangle \in Q_1 \times Q_2 \mid p_1 \in F_1 \wedge p_2 \in F_2 \big\} = F_1 \times F_2.$$

Damit gilt für alle $s \in \Sigma^*$:

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

Also haben wir nachgewiesen, dass

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

gilt und das war zu zeigen. □

Bemerkung: Prinzipiell wäre es möglich, für reguläre Ausdrücke eine Funktion

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

zu definieren, so dass für den Ausdruck $r_1 \wedge r_2$ die Beziehung

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

gilt: Zunächst berechnen wir zu r_1 und r_2 äquivalente nicht-deterministische endliche Automaten, überführen diese Automaten dann in einen vollständigen deterministischen Automaten, bilden wie oben gezeigt das kartesische Produkt dieser Automaten und gewinnen schließlich aus diesem Automaten einen regulären Ausdruck zurück. Der so gewonnene reguläre Ausdruck wäre allerdings so groß, dass diese Funktion in der Praxis nicht implementiert wird, denn bei der Überführung eines nicht-deterministischen in einen deterministischen Automaten kann der Automat stark anwachsen und der reguläre Ausdruck, der sich aus einem Automaten ergibt, kann schon bei verhältnismäßig kleinen Automaten sehr unübersichtlich werden.

Satz 18 Ist L eine reguläre Sprache über dem Alphabet Σ, so ist auch das Komplement von L, die Sprache $\Sigma^* \backslash L$ eine reguläre Sprache.

Beweis: Wir gehen ähnlich vor wie beim Beweis des letzten Satzes und nehmen an, dass ein vollständiger deterministischer endlicher Automat A gegeben ist, der die Sprache L akzeptiert:

$$L = L(A).$$

Wir konstruieren einen Automaten $\hat{A}$, der ein Wort w genau dann akzeptiert, wenn A dieses Wort nicht akzeptiert. Dazu ist es lediglich erforderlich das Komplement der Menge der akzeptierenden Zustände von A zu bilden. Sei also

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

Dann definieren wir

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

Offenbar gilt

$$\begin{array}{ll} & w \in L(\hat{A}) \\ \Leftrightarrow & \delta(q_0, w) \in Q \backslash F \\ \Leftrightarrow & \delta(q_0, w) \notin F \\ \Leftrightarrow & w \notin L(A) \end{array}$$

und daraus folgt die Behauptung. □

Korollar 19 Sind L_1 und L_2 reguläre Sprachen, so ist auch die Mengen-Differenz $L_1 \backslash L_2$ eine reguläre Sprache.

Beweis: Es sei Σ das Alphabet, das den Sprachen L_1 und L_2 zu Grunde liegt. Dann gilt

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

denn ein Wort w ist genau dann in $L_1 \backslash L_2$, wenn w einerseits in L_1 und andererseits im Komplement von L_2 liegt. Nach dem letzten Satz wissen wir, dass mit L_2 auch das Komplement $\Sigma^* \backslash L_2$ regulär ist. Da der Durchschnitt zweier regulärer Sprachen wieder regulär ist, ist damit auch $L_1 \backslash L_2$ regulär. □

Insgesamt haben wir jetzt gezeigt, dass reguläre Sprachen unter den Boole'schen Mengen-Operationen abgeschlossen sind.

Exercise 17: 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 $s = \mathtt{abc}$, then $s^R = \mathtt{cba}$. 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. ◇

6.2 Erkennung leerer Sprachen

In diesem Abschnitt untersuchen wir für einen gegeben deterministischen endlichen Automaten

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

die Frage, ob die von A erkannte Sprache leer ist, ob also $L(A) = \{\}$ gilt. Dazu fassen wir den endlichen Automaten als einen Graphen auf: Die Knoten dieses Graphen sind die Zustände von A und zwischen zwei Zuständen q_1 und q_2 gibt es genau dann eine Kante, die q_1 mit q_2 verbindet, wenn es einen Buchstaben $c \in \Sigma$ gibt, so dass $\delta(q_1, c) = q_2$ gilt. Die Sprache $L(A)$ ist genau dann leer, wenn es in diesem Graphen keinen Pfad gibt, der von dem Start-Zustand q_0 ausgeht und in einem akzeptierenden Zustand endet, wenn also die akzeptierenden Zustände von dem Start-Zustand aus nicht erreichbar sind.

Daher berechnen wir zur Beantwortung der Frage, ob $L(A)$ leer ist, die Menge R der von dem Start-Zustand q_0 erreichbaren Zustände. Diese Berechnung kann am einfachsten iterativ erfolgen:

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

   Dieser Schritt wird solange wiederholt, bis wir der Menge R keine neuen Zustände mehr hinzufügen können.

Die Sprache $L(A)$ ist genau dann leer, wenn keiner der akzeptierenden Zustände erreichbar ist, mit anderen Worten haben wir

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

Damit haben wir einen Algorithmus zur Beantwortung der Frage, ob $L(A) = \{\}$ ist: Wir bilden die Menge alle vom Start-Zustand q_0 erreichbaren Zustände und überprüfen dann, ob diese Menge einen akzeptierenden Zustand enthält.

Bemerkung: Ist die reguläre Sprache L nicht durch einen endlichen Automaten A, sondern durch einen regulären Ausdruck r gegeben, so lässt sich durch einen einfachen rekursiven Algorithmus, der dem Aufbau des regulären Ausdrucks folgt, entscheiden, ob $L(r)$ leer ist.

1. $L(\emptyset) = \{\}$.
2. $L(\varepsilon) \neq \{\}$.
3. $L(c) \neq \{\}$ für alle $c \in \Sigma$.
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. $L(r^*) \neq \{\}$.

6.3 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 20 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 $L(r_1) = L(r_2)$. First, we observe that $L(r_1)$ and $L(r_2)$ are the same sets 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 19 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_{12} and F_{21} such that

$$L(r_1)\backslash L(r_2) = L(F_{12}) \quad \text{and} \quad L(r_2)\backslash L(r_1) = L(F_{21})$$

holds. Hence we have

$$r_1 \doteq r_2 \;\Leftrightarrow\; L(F_{12}) = \{\} \wedge L(F_{21}) = \{\}$$

and according to Section 6.2 this question is decidable by checking whether any of the accepting states of F_{12} or F_{21} are reachable from the start state. □

6.4 Implementing an Equivalence Checker

Figure 6.1 on page 63 shows how to implement the algorithm sketched in the previous section. The details are discussed below.

1. The function `regExpEquiv` is called with three arguments:
   (a) `r1` and `r2` are regular expressions that have to be compared. These regular expressions are represented as terms in the same way as in Figure **??** in the definition of the function `regexp2NFA`.
   (b) `sigma` is the alphabet.

   The implementation of `regExpEquiv` is straightforward:
   (a) `r1` and `r2` are converted into deterministic finite state machines `fsm1` and `fsm2`, respectively.
   (b) Next, we construct the finite state machines `r1MinusR2` and `r2MinusR1`.
   `r1MinusR2` accepts all strings that are accepted by `fsm1` but are rejected by `fsm2`, while `r2MinusR1` accepts all strings that are accepted by `fsm2` but are rejected by `fsm1`. Therefore
   $$L(\texttt{r1MinusR2}) = L(r_1)\backslash L(r_2) \quad \text{and} \quad L(\texttt{r1MinusR2}) = L(r_1)\backslash L(r_2).$$
   (c) The regular expressions `r1` and `r2` are equivalent iff
   $$L(r_1) \subseteq L(r_2) \quad \text{and} \quad L(r_2) \subseteq L(r_1)$$
   holds. This is the case if and only if
   $$L(r_1)\backslash L(r_2) = \{\} \quad \text{and} \quad L(r_2)\backslash L(r_1) = \{\}$$
   and these conditions are checked using the function `isEmpty`.
2. The function `regexp2DFA` takes a regular expression r together with an alphabet `sigma` and constructs a deterministic finite state machine that accepts the language described by r. This is done by first converting r into a non-deterministic finite state machine `nfa` via the function `regexp2NFA`. The non-deterministic finite state machine `nfa` is then converted into a deterministic finite state machine with the help of the function `nfa2dfa`. However, there is one minor problem to solve: The function `toNFA` returns a finite state machine that has exactly one accepting state. For this reason this function does return a 5-tuple of the form

```
regExpEquiv := procedure(r1, r2, sigma) {
    fsm1      := regexp2DFA(r1, sigma);
    fsm2      := regexp2DFA(r2, sigma);
    r1MinusR2 := fsmComplement(fsm1, fsm2);
    r2MinusR1 := fsmComplement(fsm2, fsm1);
    return isEmpty(r1MinusR2) && isEmpty(r2MinusR1);
};
regexp2DFA := procedure(r, sigma) {
    converter := regexp2NFA(sigma);
    [states, sigma, delta, q0, qf] := converter.toNFA(r);
    return nfa2dfa([states, sigma, delta, q0, { qf }]);
};
fsmComplement := procedure(f1, f2) {
    [states1, sigma, delta1, q1, a1] := f1;
    [states2,     _, delta2, q2, a2] := f2;
    states := states1 >< states2;
    delta  := {};
    for ([q1, q2] in states, c in sigma) {
        delta[[q1, q2], c] := [delta1[q1, c], delta2[q2, c]];
    }
    return [states, sigma, delta, [q1, q2], a1 >< (states2 - a2)];
};
isEmpty := procedure(fsm) {
    [states, sigma, delta, q0, accepting] := fsm;
    reachable := fixpoint({q0}, q |=> { delta[q, c] : c in sigma });
    return reachable * accepting == {};
};
```

Figure 6.1: An algorithm to compare regular expressions.

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

where q_f is the single accepting state. However, the function `nfa2dfa` expects its argument to be a 5-tuple of the form

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

where A is the set of accepting states. Therefore, we had to extract the components of NFA returned by `toNFA` and turn the single accepting state q_f into the set $\{q_f\}$ in order to call the function `nfa2dfa`.

The function `toNFA` has already been shown in Figure **??** on page **??**, while the function `nfa2dfa` is shown in Figure 4.6 on page 38.

3. The function `fsmComplement` has two arguments f_1 and f_2. These arguments are deterministic finite state machines. The function returns a new finite state machine F that accepts the language

$$L(f_1) \backslash L(f_2).$$

The finite state machine F simulates the two finite state machines f_1 and f_2 in parallel. Therefore, the states of F are pairs of the form $\langle p_1, p_2 \rangle$ where p_1 is a state of f_1 while p_2 is a state of f_2. The transition function δ of F is a composition of the transition function δ_1 of f_1 and δ_2 of f_2 that is defined as follows:

$$\delta\big(\langle q_1, q_2 \rangle, c\big) := \big\langle \delta_1(q_1, c),\, \delta_2(q_2, c) \big\rangle.$$

A state $\langle p_1, p_2 \rangle$ is an accepting state of F iff p_1 is an accepting state of f_1 but p_2 is not an accepting state of f_2.

4. The input of the function `isEmpty` is a deterministic finite state machine `fsm`. The function checks whether the language accepted by `fsm` is the empty set. To this end, it computes the set of all states that are reachable from the start state. If any of these states is accepting, then the language of `fsm` is not empty. The computation of the reachable states is done via a fixpoint iteration. The function `fixpoint` that is used here has already been discussed in Figure 4.6 on page 38.

6.5 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 know as the *pumping lemma for regular languages*.

Theorem 21 (Pumping Lemma for Regular Languages)
Assume L is a regular language. Then there exists a 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. $\forall h \in \mathbb{N}_0 : uv^h w \in L$.

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

$$\exists n \in \mathbb{N} : \forall s \in L : \Big(|s| \geq n \rightarrow \exists u, v, w \in \Sigma^* : s = uvw \wedge v \neq \varepsilon \wedge |uv| \leq n \wedge \forall h \in \mathbb{N}_0 : uv^h w \in L\Big).$$

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 $|s| \geq n$. Then there are $m := |w|$ 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 $k, l \in \{0, \cdots, n\}$ 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
$$q_0 \overset{u}{\longmapsto} q_k. \tag{6.1}$$
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
$$q_k \overset{v}{\longmapsto} q_k. \tag{6.2}$$
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:
$$q_k \overset{w}{\longmapsto} q_m. \tag{6.3}$$

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

$$q_k \overset{v^h}{\longmapsto} q_k \quad \text{for all } h \in \mathbb{N}_0. \tag{6.4}$$

Combining the equations (6.1), (6.3), and (6.4) 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 $h \in \mathbb{N}_0$. □

Proposition 22 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:

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

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 such that every $s \in L$ that satisfies $|s| \geq n$ can be written as

$$s = uvw$$

such that

$$|uv| \leq n, \quad v \neq \varepsilon, \quad \text{and} \quad \forall h \in \mathbb{N}_0 : uv^hw \in L$$

holds. Let us define the string s as

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

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

$$\mathsf{a}^n\mathsf{b}^n = uvw, \quad |uv| \leq n, \quad v \neq \varepsilon, \quad \text{and} \quad \forall h \in \mathbb{N}_0 : uv^hw \in L.$$

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

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

If we take the formula $\forall h \in \mathbb{N}_0 : uv^hw \in L$ and set $h := 0$, we conclude that

$$uw \in L. \tag{6.5}$$

In order to facilitate our argument, we define the function

$$\mathit{count} : \Sigma^* \times \Sigma \to \mathbb{N}_0.$$

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

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

On one hand we have:

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

But on the other hand we have

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

Therefore, we have

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

and this shows that the string uw is not a member of the language L because for all strings in L the same 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. □

Bemerkung: The previous proposition shows that the expressive power of regular languages is quite weak. We could easily adapt the previous proposition to show that the language

$$\left\{(^n)^n \mid n \in \mathbb{N}\right\}$$

is not regular. Hence, regular expressions are unable to check even such simple questions as to whether the parentheses in an expressions 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 18: The language L_{square} is the set of all strings of the form $\mathtt{a}^n$ where n is a square, we have

$$L_{\text{square}} = \left\{a^m \mid \exists k \in \mathbb{N}_0 : m = k^2\right\}$$

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

Hint: When looking for a counter example, you should try to set $h := 2$.

Lösung: Wir führen den Beweis indirekt und nehmen an, dass L_{square} regulär wäre. Nach dem Pumping-Lemma gibt es dann eine positive natürliche Zahl n (dies war die Anzahl der Zustände des deterministischen Automaten, der die Sprache erkennt), so dass sich jeder String $s \in L_{\text{square}}$ mit $|s| \geq n$ in drei Teilstrings u, v und w aufspalten lässt, so dass gilt:

1. $s = uvw$,
2. $|uv| \leq n$,
3. $v \neq \varepsilon$,
4. $\forall h \in \mathbb{N}_0 : uv^hw \in L_{\text{square}}$.

Wir betrachten nun den String $s = a^{n^2}$. Für die Länge dieses Strings gilt offenbar

$$|s| = \left|a^{n^2}\right| = n^2 \geq n.$$

Also gibt es eine Aufspaltung von s der Form $s = uvw$ mit den oben angegebenen Eigenschaften. Da a der einzige Buchstabe ist, der in s vorkommt, können in u, v und w auch keine anderen Buchstaben vorkommen und es muss natürliche Zahlen x, y und z geben, so dass

$$u = a^x,\ v = a^y \text{ und } w = a^z$$

gilt. Wir untersuchen, welche Konzequenzen sich daraus für die Zahlen x, y und z ergeben.

1. Die Zerlegung $s = uvw$ schreibt sich als $a^{n^2} = a^x a^y a^z$ und daraus folgt
$$n^2 = x + y + z. \tag{6.6}$$
2. Die Ungleichung $|uv| \leq n$ ist jetzt äquivalent zu $x + y \leq n$, woraus
$$y \leq n \tag{6.7}$$
folgt.
3. Die Bedingung $v \neq \varepsilon$ liefert
$$y > 0. \tag{6.8}$$
4. Aus der Formel $\forall h \in \mathbb{N}_0 : uv^h w \in L_{\text{square}}$ folgt
$$\forall h \in \mathbb{N}_0 : a^x a^{y \cdot h} a^z \in L_{\text{square}}. \tag{6.9}$$

Die letzte Gleichung muss insbesondere auch für den Wert $h = 2$ gelten:

$$a^x a^{y \cdot 2} a^z \in L_{\text{square}}.$$

Nach Definition der Sprache L_{square} gibt es dann eine natürliche Zahl k, so dass gilt

$$x + 2 \cdot y + z = k^2 \tag{6.10}$$

Addieren wir in Gleichung (6.6) auf beiden Seiten y, so erhalten wir insgesamt

$$n^2 + y = x + 2 \cdot y + z = k^2.$$

Wegen $y > 0$ folgt daraus

$$n < k. \tag{6.11}$$

Andererseits haben wir

$$\begin{array}{rcll} k^2 & = & x + 2 \cdot y + z & \text{nach Gleichung (6.10)} \\ & = & x + y + z + y & \text{elementare Umformung} \\ & \leq & x + y + z + n & \text{nach Ungleichung (6.7)} \\ & = & n^2 + n & \text{nach Gleichung (6.6)} \\ & < & n^2 + 2 \cdot n + 1 & \text{da } n + 1 > 0 \\ & = & (n+1)^2 & \text{elementare Umformung} \end{array}$$

Damit haben wir insgesamt $k^2 < (n+1)^2$ gezeigt und das impliziert

$$k < n + 1. \tag{6.12}$$

Zusammen mit Ungleichung (6.11) liefert Ungleichung (6.12) nun die Ungleichungs-Kette

$$n < k < n + 1.$$

Da andererseits k eine natürliche Zahl sein muss und n ebenfalls eine natürliche Zahl ist, haben wir jetzt einen Widerspruch, denn zwischen n und $n + 1$ gibt es keine natürliche Zahl. □

Exercise 19*: The language L is defined as

$$L := \{\mathsf{a}^m\mathsf{b}^n\mathsf{c}^n \mid m, n \in \mathbb{N}\} \cup \{\mathsf{b}^m\mathsf{c}^n \mid m, n \in \mathbb{N}\}.$$

(a) Prove that L is not regular.

(b) Prove that L satisfies the pumping lemma. ◇

Exercise 20: Define $\Sigma := \{\mathsf{a}, \mathsf{b}\}$. Prove that the language

$$L := \left\{\mathsf{a}^p \mid p \text{ is a prime number}\right\}$$

is not regular. ◇

Beweis: Wir führen den Beweis indirekt und nehmen an, L wäre regulär. Nach dem Pumping-Lemma gibt es dann eine Zahl n, so dass es für alle Strings $s \in L$, deren Länge größer-gleich n ist, eine Zerlegung

$$s = uvw$$

mit den folgenden drei Eigenschaften gibt:

1. $v \neq \varepsilon$,
2. $|uv| \leq n$ und
3. $\forall h \in \mathbb{N}_0 : uv^hw \in L$.

Wir wählen nun eine Primzahl p, die größer-gleich $n + 2$ ist und setzen $s = \mathsf{a}^p$. Dann gilt $|s| = p \geq n$ und die Voraussetzung des Pumping-Lemmas ist erfüllt. Wir finden also eine Zerlegung von a^p der Form

$$\mathsf{a}^p = uvw$$

mit den oben angegebenen Eigenschaften. Aufgrund der Gleichung $s = uvw$ können die Teilstrings u, v und w nur aus dem Buchstaben "a" bestehen. Also gibt es natürliche Zahlen x, y, und z so dass gilt:

$$u = \mathsf{a}^x, \quad v = \mathsf{a}^y \quad \text{und} \quad w = \mathsf{a}^z.$$

Für x, y und z gilt dann Folgendes:

1. $x + y + z = p$,
2. $y \neq 0$,
3. $x + y \leq n$,
4. $\forall h \in \mathbb{N}_0 : x + h \cdot y + z \in \mathbb{P}$.

Setzen wir in der letzten Gleichung für h den Wert $(x + z)$ ein, so erhalten wir

$$x + (x + z) \cdot y + z \in P.$$

Wegen $x + (x + z) \cdot y + z = (x + z) \cdot (1 + y)$ hätten wir dann

$$(x + z) \cdot (1 + y) \in \mathbb{P}.$$

Das kann aber nicht sein, denn wegen $y > 0$ ist der Faktor $1 + y$ von 1 verschieden und wegen $x + y \leq n$ und $x + y + z = p$ und $p \geq n + 2$ wissen wir, dass $z \geq 2$ ist, so dass auch der Faktor $(x + z)$ von 1 verschieden ist. Damit kann das Produkt $(x + z) \cdot (1 + y)$ aber keine Primzahl mehr sein und wir haben einen Widerspruch zu der Annahme, dass L regulär ist. □

Aufgabe 21: Die Sprache L_{power} beinhaltet alle Wörter der Form a^n für die n eine Zweier-Potenz ist, es gilt also

$$L_{\text{power}} = \left\{\mathsf{a}^{2^k} \mid k \in \mathbb{N}_0\right\}$$

Zeigen Sie mit Hilfe des Pumping-Lemmas, dass die Sprache L_{power} keine reguläre Sprache ist. ◇

Aufgabe 22: Für zwei natürliche Zahlen k und l bezeichne der Ausdruck $\texttt{ggt}(k, l)$ den größten gemeinsamen Teiler von k und l. Wir definieren die Sprache L_{ggt} als

$$L_{\text{ggt}} := \left\{ \mathtt{a}^k\mathtt{b}^l \mid k \in \mathbb{N} \wedge l \in \mathbb{N} \wedge \texttt{ggt}(k, l) = 1 \right\}.$$

Zeigen Sie, dass die Sprache L_{ggt} keine reguläre Sprache ist.

Hinweis: Nutzen Sie aus, dass reguläre Sprachen unter Komplement-Bildung abgeschlossen sind. ◇

Historical Remark The pumping lemma is a special case of a general theorem that has been proved by Bar-Hillel, Perles and Shamir in 1961 [BHPS61].

Chapter 7

Context-Free Languages

In this chapter we present the notion of *context-free languages*. This concept is much more powerful than the notion of regular languages. The syntax of most modern programming languages can be described via context-free languages. 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*. Usually, a parser builds 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 the chapter by introducing some of the less complex algorithms that are available for parsing a string into a parse tree.

7.1 Kontextfreie Grammatiken

Kontextfreie Sprachen dienen zur Beschreibung von Programmier-Sprachen, insofern handelt es sich bei den kontextfreien Sprachen genau wie bei den regulären Sprachen ebenfalls um formale Sprachen. Allerdings wollen wir später beim Einlesen eines Programms nicht nur entscheiden, ob das Programm korrekt ist, sondern wir wollen darüber hinaus den Programm-Text *strukturieren*. Den Vorgang des *Strukturierens* bezeichnen wir auch als *parsen* und das Programm, das diese Strukturierung vornimmt, wird als *Parser* bezeichnet. Als Eingabe erhält ein Parser üblicherweise nicht den Text eines Programms, sondern stattdessen eine Folge sogenannter *Terminale*, die auch als *Token* bezeichnet werden. Diese Token werden von einem Scanner erzeugt, der mit Hilfe regulärer Ausdrücke den Programmtext in einzelne Wörter aufspaltet, die wir in diesem Zusammenhang als Token bezeichnen. Beispielsweise spaltet der Scanner des `C`-Compilers ein `C`-Programm in die folgenden Token auf:

- Operator-Symbole wie "`+`", "`+=`", "`<`", "`<=`" etc.,
- Klammer-Symbole wie "`(`", "`[`", "`{`" oder die schließenden Klammern "`)`", "`]`", "`}`",
- vordefinierte Schlüsselwörter wie "`if`", "`while`", "`typedef`", "`struct`", etc.,
- Variablen- und Funktions-Namen wie "`x`", "`y`", "`printf`", etc.,
- Namen für Typen wie "`int`", "`char`" oder auch benutzerdefinierte Typnamen,
- Literale zur Bezeichnung von Konstanten, wie "`1.23`", "`"hallo"`" oder "`'c'`"
- Kommentare,
- *White-Space-Zeichen*, (Leerzeichen, Tabulatoren, Zeilenumbrüche).

Der Parser erhält vom Scanner eine Folge solcher Token und hat die Aufgabe, daraus einen sogenannten *Syntax-Baum* zu konstruieren. Dazu bedient sich der Parser einer *Grammatik*, die mit Hilfe von *Grammatik-Regeln* angibt, wie die Eingabe zu strukturieren ist. Betrachten wir als Beispiel das Parsen arithmetischer Ausdrücke. Die Menge *ArithExpr*

der arithmetischen Ausdrücke können wir induktiv definieren. Um die Struktur arithmetischer Ausdrücke korrekt wiedergeben zu können, definieren wir gleichzeitig die Mengen *Product* und *Factor*. Die Menge *Product* enthält arithmetische Ausdrücke, die Produkte und Quotienten darstellen und die Menge *Factor* enthält einzelne Faktoren. Die Definition dieser zusätzlichen Mengen ist notwendig, um später die Präzedenzen der Operatoren korrekt darstellen zu können. Die Grundbausteine der arithmetischen Ausdrücke sind Variablen, Zahlen, die Operator-Symbole "+", "–", "∗", "/", und die Klammer-Symbole "(" und ")". Aufbauend auf diesen Symbolen verläuft die induktive Definition der Mengen *Factor*, *Product* und *ArithExpr* wie folgt:

1. Jede Zahlenkonstante ist ein Faktor:

   $$C \in \mathit{Number} \Rightarrow C \in \mathit{Factor}.$$

2. Jede Variable ist ein Faktor:

   $$V \in \mathit{Variable} \Rightarrow V \in \mathit{Factor}.$$

3. Ist A ein arithmetischer Ausdruck und schließen wir diesen Ausdruck in Klammern ein, so erhalten wir einen Ausdruck, den wir als Faktor benützen können:

   $$A \in \mathit{ArithExpr} \Rightarrow \text{ "(" } A \text{ ")" } \in \mathit{Factor}.$$

   Ein Wort zur Notation: Während in der obigen Formel A eine Meta-Variable ist, die für einen beliebigen arithmetischen Ausdruck steht, sind die Strings "(" und ")" wörtlich zu interpretieren und deshalb in Gänsefüßchen eingeschlossen. Die Gänsefüßchen sind natürlich nicht Teil des arithmetischen Ausdrucks sondern dienen lediglich der Notation.

4. Ist F ein Faktor, so ist F gleichzeitig auch ein Produkt:

   $$F \in \mathit{Factor} \Rightarrow F \in \mathit{Product}.$$

5. Ist P ein Produkt und ist F ein Faktor, so sind die Strings P "∗" F und P "/" F ebenfalls Produkte:

   $$P \in \mathit{Product} \wedge F \in \mathit{Factor} \Rightarrow P \text{"*"} F \in \mathit{Product} \ \wedge \ P \text{"/"} F \in \mathit{Product}.$$

6. Jedes Produkt ist gleichzeitig auch ein arithmetischer Ausdruck

   $$P \in \mathit{Product} \Rightarrow P \in \mathit{ArithExpr}.$$

7. Ist A ein arithmetischer Ausdruck und ist P ein Produkt, so sind auch die Strings A "+" P und A "–" P arithmetische Ausdrücke:

   $$A \in \mathit{ArithExpr} \wedge P \in \mathit{Product} \Rightarrow A \text{"+"} P \in \mathit{ArithExpr} \ \wedge \ A \text{"–"} P \in \mathit{ArithExpr}.$$

Die oben angegebenen Regeln definieren die Mengen *Factor*, *Product* und *ArithExpr* durch wechselseitige Rekursion. Diese Definition können wir in Form von sogenannten *Grammatik-Regeln* wesentlich kompakter schreiben:

$$\begin{array}{rcl}
\mathit{arithExpr} & \to & \mathit{arithExpr} \text{ "+" } \mathit{product} \\
\mathit{arithExpr} & \to & \mathit{arithExpr} \text{ "–" } \mathit{product} \\
\mathit{arithExpr} & \to & \mathit{product} \\
\mathit{product} & \to & \mathit{product} \text{ "*" } \mathit{factor} \\
\mathit{product} & \to & \mathit{product} \text{ "/" } \mathit{factor} \\
\mathit{product} & \to & \mathit{factor} \\
\mathit{factor} & \to & \text{ "(" } \mathit{arithExpr} \text{ ")"} \\
\mathit{factor} & \to & \textsc{Variable} \\
\mathit{factor} & \to & \textsc{Number}
\end{array}$$

Die Ausdrücke auf der linken Seite einer Grammatik-Regel bezeichnen wir als *syntaktische Variablen* oder auch als *Nicht-Terminale*. Wir werden syntaktische Variablen üblicherweise klein schreiben, denn das ist die Konvention bei den Parser-Generatoren ANTLR und *JavaCup*, die wir später vorstellen werden. In der Literatur ist es allerdings oft

anders herum. Dort werden die syntaktischen Variablen groß und die *Terminale* klein geschrieben. Gelegentlich wird eine syntaktische Variable auch als eine *syntaktische Kategorie* bezeichnet.

In dem Beispiel sind *arithExpr*, *product* und *factor* die syntaktischen Variablen. Die restlichen Ausdrücke, in unserem Fall also NUMBER, VARIABLE und die Zeichen "+", "-", "*", "/", "(" und ")" bezeichnen wir als *Terminale* oder auch *Token*. Dies sind also genau die Zeichen, die nicht auf der linken Seite einer Grammatik-Regel stehen. Bei den Nicht-Terminalen gibt es zwei Arten:

1. Operator-Symbole und Trennzeichen wie beispielsweise "/" und "(".

   Solche Nicht-Terminalen stehen für sich selbst.

2. Token wie NUMBER oder VARIABLE ist zusätzlich ein Wert zugeordnet. Im Falle von NUMBER ist dies eine Zahl, im Falle von VARIABLE ist dies ein String, der den Namen der Variablen wiedergibt. Diese Art von Token werden wir zur besseren Unterscheidung von den Variablen immer mit Großbuchstaben schreiben. Damit folgen wir der Praxis von Parser-Generatoren wie ANTLR und *JavaCup*.

Üblicherweise werden Grammatik-Regeln in einer kompakteren Notation als der oben vorgestellten wiedergegeben, indem alle Regeln für ein Nicht-Terminal zusammengefasst werden. Für unser Beispiel sieht das dann wie folgt aus:

$$\begin{array}{rcl} \textit{arithExpr} & \to & \textit{arithExpr}\ \text{"+"}\ \textit{product} \mid \textit{arithExpr}\ \text{"-"}\ \textit{product} \mid \textit{product} \\ \textit{product} & \to & \textit{product}\ \text{"*"}\ \textit{factor} \mid \textit{product}\ \text{"/"}\ \textit{factor} \mid \textit{factor} \\ \textit{factor} & \to & \text{"("}\ \textit{arithExpr}\ \text{")"} \mid \textsc{Number} \mid \textsc{Variable} \end{array}$$

Hier werden also die einzelnen Alternativen einer Regel durch das Metazeichen "|" getrennt. Nach dem obigen Beispiel geben wir jetzt die formale Definition für den Begriff der kontextfreien Grammatik.

Definition 23 (Kontextfreie Grammatik) Eine *kontextfreie Grammatik* G ist ein 4-Tupel

$$G = \langle V, T, R, S \rangle,$$

so dass folgendes gilt:

1. V ist eine Menge von Namen, die wir als *syntaktische Variablen* oder auch *Nicht-Terminale* bezeichnen.

   In dem obigen Beispiel gilt

   $$V = \{\textit{arithExpr}, \textit{product}, \textit{factor}\}.$$

2. T ist eine Menge von Namen, die wir als *Terminale* bezeichnen. Die Mengen T und V sind disjunkt, es gilt also

   $$T \cap V = \emptyset.$$

   In dem obigen Beispiel gilt

   $$T = \{\textsc{Number}, \textsc{Variable}, \text{"+"}, \text{"-"}, \text{"*"}, \text{"/"}, \text{"("}, \text{")"}\}$$

3. R ist die Menge der Regeln. Formal ist eine Regel ein Paar $\langle A, \alpha \rangle$:

   (a) Die erste Komponente dieses Paares ist eine syntaktische Variable:

   $$A \in V.$$

   (b) Die zweite Komponente ist ein String, der aus syntaktischen Variablen und Terminalen aufgebaut ist:

   $$\alpha \in (V \cup T)^*.$$

   Insgesamt gilt für die Menge der Regeln R damit

   $$R \subseteq V \times (V \cup T)^*$$

   Ist $\langle x, \alpha \rangle$ eine Regel, so schreiben wir diese Regel als

   $$x \to \alpha.$$

Beispielsweise haben wir oben die erste Regel als

$$\textit{arithExpr} \rightarrow \textit{arithExpr}\ \text{“+”}\ \textit{product}$$

geschrieben. Formal steht diese Regel für das Paar

$$\big\langle \textit{arithExpr}, [\textit{arithExpr}, \text{“+”}, \textit{product}] \big\rangle.$$

4. S ist ein Element der Menge V, das wir als das *Start-Symbol* bezeichnen. In dem obigen Beispiel ist *arithExpr* das Start-Symbol. □

7.1.1 Ableitungen

Als nächstes wollen wir festlegen, welche Sprache durch eine gegebene Grammatik G definiert wird. Dazu definieren wir zunächst den Begriff eines *Ableitungs-Schrittes*. Es sei

1. $G = \langle V, T, R, S \rangle$ eine Grammatik,
2. $a \in V$ eine syntaktische Variable,
3. $\alpha a \beta \in (V \cup T)^*$ ein String aus Terminalen und syntaktischen Variablen, der die Variable a enthält,
4. $(a \rightarrow \gamma) \in R$ eine Regel.

Dann kann der String $\alpha a \beta$ durch einen Ableitungs-Schritt in den String $\alpha \gamma \beta$ überführt werden, wir ersetzen also ein Auftreten der syntaktische Variable a durch die rechte Seite der Regel $a \rightarrow \gamma$. Diesen Ableitungs-Schritt schreiben wir als

$$\alpha a \beta \Rightarrow_G \alpha \gamma \beta.$$

Geht die verwendete Grammatik G aus dem Zusammenhang klar hervor, so wird der Index G weggelassen und wir schreiben kürzer $\Rightarrow$ an Stelle von $\Rightarrow_G$. Der transitive und reflexive Abschluss der Relation $\Rightarrow_G$ wird mit $\Rightarrow_G^*$ bezeichnet. Wollen wir ausdrücken, dass die Ableitung des Strings w aus dem Nicht-Terminal a aus n Ableitungs-Schritten besteht, so schreiben wir

$$a \Rightarrow^n w.$$

Wir geben ein Beispiel:

$$\begin{array}{rcl}
\textit{arithExpr} & \Rightarrow & \textit{arithExpr}\ \text{“+”}\ \textit{product} \\
& \Rightarrow & \textit{product}\ \text{“+”}\ \textit{product} \\
& \Rightarrow & \textit{product}\ \text{“*”}\ \textit{factor}\ \text{“+”}\ \textit{product} \\
& \Rightarrow & \textit{factor}\ \text{“*”}\ \textit{factor}\ \text{“+”}\ \textit{product} \\
& \Rightarrow & \textsc{Number}\ \text{“*”}\ \textit{factor}\ \text{“+”}\ \textit{product} \\
& \Rightarrow & \textsc{Number}\ \text{“*”}\ \textsc{Number}\ \text{“+”}\ \textit{product} \\
& \Rightarrow & \textsc{Number}\ \text{“*”}\ \textsc{Number}\ \text{“+”}\ \textit{factor} \\
& \Rightarrow & \textsc{Number}\ \text{“*”}\ \textsc{Number}\ \text{“+”}\ \textsc{Number}
\end{array}$$

Damit haben wir also gezeigt, dass

$$\textit{arithExpr} \Rightarrow^* \textsc{Number}\ \text{“*”}\ \textsc{Number}\ \text{“+”}\ \textsc{Number}$$

oder genauer

$$\textit{arithExpr} \Rightarrow^8 \textsc{Number}\ \text{“*”}\ \textsc{Number}\ \text{“+”}\ \textsc{Number}$$

gilt. Ersetzen wir hier das Terminal $\textsc{Number}$ durch verschiedene Zahlen, so haben wir damit beispielsweise gezeigt, dass der String

$$2 * 3 + 4$$

ein arithmetischer Ausdruck ist. Allgemein definieren wir die durch eine Grammatik G definierte Sprache $L(G)$ als

die Menge aller Strings, die einerseits nur aus Terminalen bestehen und die sich andererseits aus dem Start-Symbol S der Grammatik ableiten lassen:

$$L(G) := \{w \in T^* \mid S \Rightarrow^* w\}.$$

Beispiel: Die Sprache

$$L = \{(^n)^n \mid n \in \mathbb{N}_0\}$$

wird von der Grammatik

$$G = \langle \{s\}, \{\text{“(”}, \text{“)”}\}, R, s \rangle$$

erzeugt, wobei die Regeln R wie folgt gegeben sind:

$$s \to \text{“(”}\ s\ \text{“)”} \mid \varepsilon.$$

Beweis: Wir zeigen zunächst, dass sich jedes Wort $w \in L$ aus dem Start-Symbol s ableiten lässt:

$$w \in L \implies s \Rightarrow^* w.$$

Es sei also $w_n = (^n)^n$. Wir zeigen durch Induktion über $n \in \mathbb{N}_0$, dass $w_n \in L(G)$ ist.

I.A.: $n = 0$.

Es gilt $w_0 = \varepsilon$. Offenbar haben wir

$$s \Rightarrow \varepsilon,$$

denn die Grammatik enthält die Regel $s \to \varepsilon$. Also gilt $w_0 \in L(G)$.

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

Der String w_{n+1} hat die Form $w_{n+1} =$ “(” w_n “)” , wobei der String w_n natürlich ebenfalls in L liegt. Also gibt es nach I.V. eine Ableitung von w_n:

$$s \Rightarrow^* w_n.$$

Insgesamt haben wir dann die Ableitung

$$s \Rightarrow \text{“(”}\ s\ \text{“)”} \Rightarrow^* \text{“(”}\ w_n\ \text{“)”} = w_{n+1}.$$

Also gilt $w_{n+1} \in L(G)$.

Als nächstes zeigen wir, dass jedes Wort w, das sich aus s ableiten lässt, ein Element der Sprache L ist. Wir führen den Beweis durch Induktion über die Anzahl $n \in \mathbb{N}$ der Ableitungs-Schritte:

I.A.: $n = 1$.

Die einzige Ableitung eines aus Terminalen aufgebauten Strings, die nur aus einem Schritt besteht, ist

$$s \Rightarrow \varepsilon.$$

Folglich muss $w = \varepsilon$ gelten und wegen $\varepsilon = (^0)^0 \in L$ haben wir $w \in L$.

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

Wenn die Ableitung aus mehr als einem Schritt besteht, dann muss die Ableitung die folgende Form haben:

$$s \Rightarrow \text{“(”}\ s\ \text{“)”} \Rightarrow^n w$$

Daraus folgt

$$w = \text{“(”}\ v\ \text{“)”} \ \wedge\ s \Rightarrow^n v.$$

Nach I.V. gilt dann $v \in L$. Damit gibt es $k \in \mathbb{N}_0$ mit $v = (^k)^k$. Also haben wir

$$w = \text{“(”}\ v\ \text{“)”} = ((^k)^k) = (^{k+1})^{k+1} \in L.$$ □

Aufgabe 23: Wir definieren für $w \in \Sigma^*$ und $c \in \Sigma$ die Funktion

$\mathit{count}(w, c)$,

die zählt, wie oft der Buchstabe c in dem Wort w vorkommt, durch Induktion über w.

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

Wir setzen

$$\mathit{count}(\varepsilon, c) := 0.$$

I.S.: $w = dv$ mit $d \in \Sigma$ und $v \in \Sigma^*$.

Dann wird $\mathit{count}(dv, c)$ durch eine Fall-Unterscheidung definiert:

$$\mathit{count}(dv, c) := \begin{cases} \mathit{count}(v, c) + 1 & \text{falls } c = d; \\ \mathit{count}(v, c) & \text{falls } c \neq d. \end{cases}$$ ◇

Wir setzen nun $\Sigma = \{\text{“A”}, \text{“B”}\}$ und definieren die Sprache L als die Menge der Wörter $w \in \Sigma^*$, in denen die Buchstaben “A” und “B” mit der selben Häufigkeit vorkommen:

$$L := \{w \in \Sigma^* \mid \mathit{count}(w, \text{“A”}) = \mathit{count}(w, \text{“B”})\}$$

Geben Sie eine Grammatik G an, so dass $L = L(G)$ gilt und beweisen Sie Ihre Behauptung! ◇

Exercise 24: 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. ◇

Aufgabe 25*: Es sei $\Sigma := \{\text{“A”}, \text{“B”}\}$. Wir definieren die Menge L als die Menge der Strings s, die sich nicht in der Form $s = ww$ schreiben lassen:

$$L = \{s \in \Sigma^* \mid \neg(\exists w \in \Sigma^* : s = ww)\}.$$

Geben Sie eine kontextfreie Grammatik G an, die diese Sprache erzeugt. ◇

Lösung: Die Lösung dieser Aufgabe ist so umfangreich, dass wir unsere Überlegungen in vier Teile aufspalten.

Vorüberlegung I: String-Notationen
Für einen String s bezeichnen wir mit $s[i]$ den i-ten Buchstaben und mit $s[i:j]$ den Teilstring, der sich vom i-ten Buchstaben bis zum j-ten Buchstaben einschließlich erstreckt. Bei der Nummerierung beginnen wir mit 1. Dann gilt

1. $|s[i:j]| = j - i + 1$

   Von der Notwendigkeit, hier eine 1 zu addieren, können wir uns dadurch überzeugen, wenn wir den Fall $i = j$ betrachten, denn $s[i:i]$ ist der Teilstring, der nur aus dem i-ten Buchstaben besteht und der hat natürlich die Länge 1.

2. $s[i:j][k] = s[i + k - 1]$.

   Dass in diesem Fall 1 subtrahiert werden muss, sehen Sie, wenn Sie den Fall $k = 1$ betrachten, denn der erste Buchstabe des Teilstrings $s[i:j]$ ist natürlich der i-te Buchstabe von s.

3. Hat ein Wort $s \in \Sigma^*$ eine ungerade Länge, gilt also

$$|s| = 2 \cdot n + 1 \quad \text{für ein } n \in \mathbb{N}_0,$$

so liegt der Buchstabe $s[n+1]$ in der Mitte von s. Um dies einzusehen, betrachten wir die Teilstrings $s[1:n]$ und $s[n+2:2 \cdot n+1]$, die links und rechts von $s[n+1]$ liegen:

$$\underbrace{s[1] \cdots s[n]}_{s[1:n]}\, s[n+1]\, \underbrace{s[n+2] \cdots s[2 \cdot n+1]}_{s[n+2:2 \cdot n+1]}$$

Offenbar sind diese Teilstrings gleich lang, denn wir haben

$$|s[1:n]| = n \quad \text{und} \quad |s[n+2:2 \cdot n+1]| = 2 \cdot n + 1 - (n+2) + 1 = n.$$

Also liegt der Buchstabe $s[n+1]$ tatsächlich in der Mitte von s.

Für einen String s ungerader Länge definieren wir $\hat{s}$ als den Buchstaben, der in der Mitte von s liegt:

$$\hat{s} := s[n+1] \quad \text{falls } |s| = 2 \cdot n + 1.$$

Vorüberlegung II: Zunächst ist klar, dass alle Strings deren Längen ungerade sind, in der Sprache L liegen, denn jeder String der Form $s = ww$ hat offenbar die Länge

$$|s| = |w| + |w| = 2 \cdot |w|$$

und das ist eine gerade Zahl.

Gilt nun $s \in L$ mit $|s| = 2 \cdot n$, so lässt sich s in zwei Teile u und v gleicher Länge zerlegen:

$$s = uv \quad \text{mit} \quad u = s[1\,{:}\,n], \quad v = s[n+1\,{:}\,2 \cdot n] \quad \text{und} \quad u \neq v.$$

Aus der Ungleichung $u \neq v$ folgt, dass es mindestens einen Index $k \in \{1, \cdots, n\}$ gibt, so dass sich die Strings u und v an diesem Index unterscheiden:

$$u[k] \neq v[k].$$

Der Trick besteht jetzt darin, den String s in zwei Teilstrings x und y aufzuteilen, von denen der eine Teilstring in der Mitte den Buchstaben $u[k]$ enthält, während der andere Teilstring in der Mitte den Buchstaben $v[k]$ enthält. Wir definieren

$$x := s[1\,{:}\,2 \cdot k - 1] \quad \text{und} \quad y := s[2 \cdot k\,{:}\,2 \cdot n].$$

Für die Längen von x und y folgt daraus

$$|x| = 2 \cdot k - 1 \quad \text{und} \quad |y| = 2 \cdot (n-k) + 1.$$

Dann gilt einerseits

$$x[k] = s[k] = u[k]$$

und andererseits haben wir

$$\begin{aligned} y[n-k+1] &= s[2 \cdot k\,{:}\,2 \cdot n][n-k+1] \\ &= s[2 \cdot k + (n-k+1) - 1] \\ &= s[n+k] \\ &= s[n+1\,{:}\,2 \cdot n][k] \\ &= v[k] \end{aligned}$$

Die beiden Buchstaben $u[k]$ und $v[k]$, die dafür verantwortlich sind, dass u und v verschieden sind, befinden sich also genau in der Mitte der Strings x und y.

Bemerkung: Wir haben soeben Folgendes gezeigt: Falls $s \in L$ mit $|s| = 2 \cdot n$ ist, so lässt sich s so in zwei Strings x und y aufspalten, dass die Buchstaben, die jeweils in der Mitte von x und y liegen, unterschiedlich sind:

$$s \in L \wedge |s| = 2 \cdot n \rightarrow \exists x, y \in \Sigma^* : \big(s = xy \wedge \hat{x} \neq \hat{y}\big).$$

Vorüberlegung III: Wir überlegen uns nun, dass auch die Umkehrung des in der letzten Bemerkung angegebenen

Zusammenhangs gilt: Sind $x, y \in \Sigma^*$ mit ungerader Länge und gilt $\hat{x} \neq \hat{y}$, so liegt der String xy in der Sprache L:

$$x, y \in \Sigma^* \wedge |x| = 2 \cdot m + 1 \wedge |y| = 2 \cdot n + 1 \wedge \hat{x} \neq \hat{y} \rightarrow xy \in L. \qquad (*)$$

Beweis: Wir definieren s als die Konkatenation von x und y, also $s := xy$. Für die Länge von s gilt dann

$$|s| = 2 \cdot (m + n + 1).$$

Wir werden zeigen, dass

$$s[m+1] \neq s[(m+n+1) + (m+1)]$$

gilt. Spalten wir s in zwei gleich lange Teile u und v auf, definieren also

$$u := s[1 : m+n+1] \quad \text{und} \quad v := s[m+n+2 : 2 \cdot (m+n+1)],$$

so werden wir gleich sehen, dass

$$u[m+1] = s[m+1] \neq s[(m+n+1) + (m+1)] = v[m+1],$$

gilt, woraus $u \neq v$ und damit $s = uv \in L$ folgt.

Es bleibt der Nachweis von $s[m+1] \neq s[(m+n+1) + (m+1)]$ zu erledigen:

$$\begin{array}{lcll} s[(m+n+1)+m+1] & = & (xy)[(m+n+1)+m+1] & \text{wegen } s = xy \\ & = & y[n+1] & \text{denn } |x| = 2 \cdot m + 1 \\ & = & \hat{y} & \text{denn } |y| = 2 \cdot n + 1 \\ & \neq & \hat{x} & \\ & = & x[m+1] & \text{denn } |x| = 2 \cdot m + 1 \\ & = & s[m+1] & \text{wegen } s = xy. \end{array}$$

Damit ist der Beweis der Behauptung $(*)$ abgeschlossen.

Aufstellen der Grammatik: Fassen wir die letzten beiden Vorüberlegungen zusammen, so stellen wir fest, dass die Sprache L aus genau den Wörtern besteht, die entweder eine ungerade Länge haben, oder die aus Paaren von Strings ungerader Länge bestehen, die in der Mitte unterschiedliche Buchstaben haben:

$$\begin{array}{lcl} L & = & \{ s \in \Sigma^* \mid |s| \,\%\, 2 = 1 \} \\ & \cup & \{ s \in \Sigma^* \mid \exists x, y \in \Sigma^* : s = xy \wedge |x| \,\%\, 2 = 1 \wedge |y| \,\%\, 2 = 1 \wedge \hat{x} \neq \hat{y} \} \end{array}$$

Damit lässt sich die Menge L durch die folgende Grammatik beschreiben

$$G = \langle \{s, a, b, x, u\}, \{\text{“A”}, \text{“B”}\}, R, s \rangle,$$

wobei die Menge der Regeln wie folgt gegeben ist:

$$\begin{array}{lcl} s & \rightarrow & u \mid ab \mid ba \\ a & \rightarrow & \text{“A”} \mid xax \\ b & \rightarrow & \text{“B”} \mid xbx \\ u & \rightarrow & x \mid uxx \\ x & \rightarrow & \text{“A”} \mid \text{“B”} \end{array}$$

Wir diskutieren die verschiedenen syntaktischen Variablen.

1. $L(x) = \{\text{“A”}, \text{“B”}\}$.
2. $L(u) = \{w \in \Sigma^* \mid \ |w| \,\%\, 2 = 1\}$,
   denn ein String ungerader Länge hat entweder die Länge 1 oder er kann aus einem String ungerader Länge durch Anfügen zweier Buchstaben erzeugt werden.
3. $L(a) = \{w \in \Sigma^* \mid \ \exists k \in \mathbb{N} : |w| = 2 \cdot k - 1 \wedge w[k] = \text{“A”}\}$,

denn wenn wir an einen String, bei dem der Buchstabe "A" in der Mitte steht, vorne und hinten jeweils einen Buchstaben anfügen, erhalten wir wieder einen String, in dessen Mitte der Buchstabe "A" steht

4. $L(b) = \{w \in \Sigma^* \mid \exists k \in \mathbb{N} : |w| = 2 \cdot k - 1 \wedge w[k] = \text{"B"}\}$,

   denn die Variable b ist analog zur Variablen a definiert worden. Der einzige Unterschied ist der, dass nun der Buchstabe B in der Mitte liegt.

5. $$\begin{aligned} L(s) &= \big\{w \in \Sigma^* \big| \; |w| \,\%\, 2 = 1\big\} \\ &\quad \cup \big\{w \in \Sigma^* \big| \; \exists x, y \in \Sigma^* : w = xy \wedge |x| \,\%\, 2 = 1 \wedge |y| \,\%\, 2 = 1 \wedge \hat{x} \neq \hat{y}\big\} \\ &= \big\{w \in \Sigma^* \mid \neg(\exists v \in \Sigma^* : w = vv)\big\} \end{aligned}$$

   denn wir haben oben argumentiert, dass alle Strings der Sprache L entweder eine ungerade Länge haben oder in zwei Teile ungerader Länge zerlegt werden können, so dass in der Mitte dieser Teile verschiedene Buchstaben stehen: Entweder steht im ersten Teil ein "A" und im zweiten Teil steht ein "B" oder es ist umgekehrt.

Um die obigen Behauptungen formal zu beweisen müssten wir nun einerseits noch durch eine Induktion nach der Länge der Herleitung zeigen, dass die von den Grammatik-Symbolen erzeugten Strings tatsächlich in den oben angegebenen Mengen liegen. Andererseits müssten wir für die oben angegebenen Mengen zeigen, dass sich jeder String der jeweiligen Menge auch tatsächlich mit den angegebenen Grammatik-Regeln erzeugen lässt. Dieser Nachweis würde dann durch Induktion über die Länge der einzelnen Strings geführt werden. Da diese Nachweise einfach sind und keine Überraschungen mehr bieten, verzichten wir hier darauf. □

Bemerkung: Wir werden später sehen, dass das Komplement der in der letzten Aufgabe definierten Sprache L, also die Sprache

$$L^{\mathsf{c}} := \Sigma^* \backslash L = \big\{ww \mid w \in \Sigma^*\big\}$$

keine kontextfreie Sprache ist. Damit sehen wir dann, dass die Menge der kontextfreien Sprachen nicht unter Komplementbildung abgeschlossen ist. ◇

7.1.2 Parse-Bäume

Mit Hilfe einer Grammatik G können wir nicht nur erkennen, ob ein gegebener String s ein Element der von der Grammatik erzeugten Sprache $L(G)$ ist, wir können den String auch *strukturieren* indem wir einen *Parse-Baum* aufbauen. Ist eine Grammatik

$$G = \langle V, T, R, S\rangle$$

gegeben, so ist ein Parse-Baum für diese Grammatik ein Baum, der den folgenden Bedingungen genügt:

1. Jeder innere Knoten (also jeder Knoten, der kein Blatt ist), ist mit einer Variablen beschriftet.
2. Jedes Blatt ist mit einem Terminal oder mit einer Variablen beschriftet.
3. Falls ein Blatt mit einer Variablen a beschriftet ist, dann enthält die Grammatik eine Regel der Form

   $$a \to \varepsilon.$$

4. Ist ein innerer Knoten mit einer Variablen a beschriftet und sind die Kinder dieses Knotens mit den Symbolen X_1, X_2, $\cdots$, X_n beschriftet, so enthält die Grammatik G eine Regel der Form

   $$a \to X_1 X_2 \cdots X_n.$$

Die Blätter des Parse-Baums ergeben dann, wenn wir sie von links nach rechts lesen, ein Wort, das von der Grammatik G abgeleitet wird. Abbildung 7.1 zeigt einen Parse-Baum für das Wort "2*3+4", der mit der oben angegebenen Grammatik für arithmetische Ausdrücke abgeleitet worden ist.

Da Bäume der in Abbildung 7.1 gezeigten Art sehr schnell zu groß werden, vereinfachen wir diese Bäume mit Hilfe der folgenden Regeln:

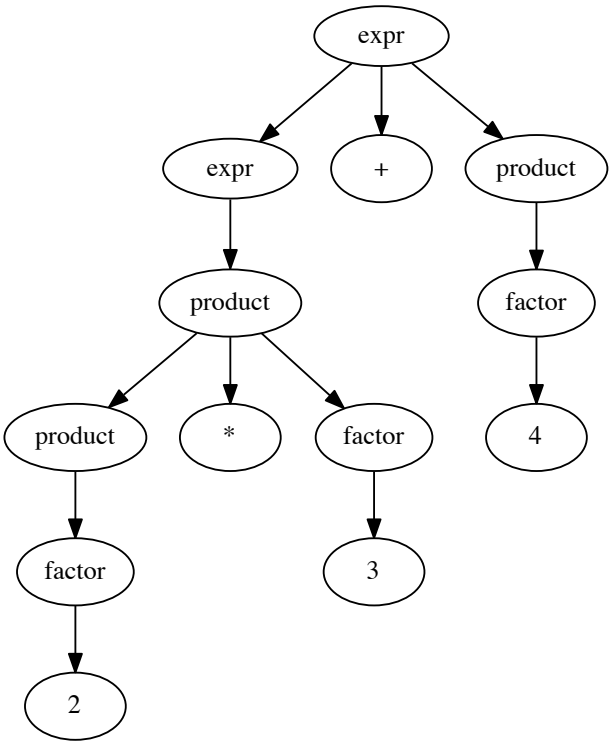


Figure 7.1: Ein Parse-Baum für den String "2*3+4".

1. Ist n ein innerer Knoten, der mit der Variablen A beschriftet ist und gibt es unter den Kindern dieses Knotens genau ein Kind, dass mit einem Terminal o beschriftet ist, so entfernen wir dieses Kind und beschriften den Knoten n stattdessen mit dem Terminal o.

2. Hat ein innerer Knoten nur ein Kind, so ersetzen wir diesen Knoten durch sein Kind.

Den Baum, den wir auf diese Weise erhalten, nennen wir den *abstrakten Syntax-Baum*. Abbildung 7.2 zeigt den abstrakten Syntax-Baum den wir erhalten, wenn wir den in Abbildung 7.1 gezeigten Parse-Baum nach diesen Regeln vereinfachen. Die in diesem Baum gespeicherte Struktur ist genau das, was wir brauchen um den arithmetischen Ausdruck "2*3+4" auszuwerten, denn der Baum zeigt uns, in welcher Reihenfolge die Operatoren ausgewertet werden müssen.

7.1.3 Mehrdeutige Grammatiken

Die zu Anfang des Abschnitts 7.1 angegebene Grammatik zur Beschreibung arithmetischer Ausdrücke erscheint durch ihre Unterscheidung der syntaktischen Kategorien *arithExpr*, *product* und *factor* unnötig kompliziert. Wir stellen eine einfachere Grammatik G vor, welche dieselbe Sprache beschreibt:

$$G = \big\langle \{\mathit{expr}\}, \{\textsc{Number}, \textsc{Variable}, \text{"+"}, \text{"-"}, \text{"*"}, \text{"/"}, \text{"("}, \text{")"}\}, R, \mathit{expr} \big\rangle,$$

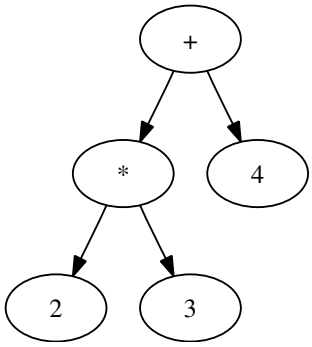


Figure 7.2: Ein abstrakter Syntax-Baum für den String "2*3+4".

wobei die Regeln R wie folgt gegeben sind:

$$
\begin{array}{rcl}
\textit{expr} & \to & \textit{expr}\ \text{“+”}\ \textit{expr} \\
& | & \textit{expr}\ \text{“−”}\ \textit{expr} \\
& | & \textit{expr}\ \text{“*”}\ \textit{expr} \\
& | & \textit{expr}\ \text{“/”}\ \textit{expr} \\
& | & \text{“(”}\ \textit{expr}\ \text{“)”} \\
& | & \textsc{Number} \\
& | & \textsc{Variable}
\end{array}
$$

Um zu zeigen, dass der String "2*3+4" in der von dieser Sprache erzeugten Grammatik liegt, geben wir die folgende Ableitung an:

$$
\begin{array}{rcl}
\textit{expr} & \Rightarrow & \textit{expr}\ \text{“+”}\ \textit{expr} \\
& \Rightarrow & \textit{expr}\ \text{“*”}\ \textit{expr}\ \text{“+”}\ \textit{expr} \\
& \Rightarrow & 2\ \text{“*”}\ \textit{expr}\ \text{“+”}\ \textit{expr} \\
& \Rightarrow & 2\ \text{“*”}\ 3\ \text{“+”}\ \textit{expr} \\
& \Rightarrow & 2\ \text{“*”}\ 3\ \text{“+”}\ 4
\end{array}
$$

Diese Ableitung entspricht dem abstrakten Syntax-Baum, der in Abbildung 7.2 gezeigt ist. Es gibt aber noch eine andere Ableitung des Strings "2*3+4" mit dieser Grammatik:

$$
\begin{array}{rcl}
\textit{expr} & \Rightarrow & \textit{expr}\ \text{“*”}\ \textit{expr} \\
& \Rightarrow & \textit{expr}\ \text{“*”}\ \textit{expr}\ \text{“+”}\ \textit{expr} \\
& \Rightarrow & 2\ \text{“*”}\ \textit{expr}\ \text{“+”}\ \textit{expr} \\
& \Rightarrow & 2\ \text{“*”}\ 3\ \text{“+”}\ \textit{expr} \\
& \Rightarrow & 2\ \text{“*”}\ 3\ \text{“+”}\ 4
\end{array}
$$

Dieser Ableitung entspricht der abstrakte Syntax-Baum, der in Abbildung 7.3 gezeigt ist. Bei dieser Ableitung wird der String "2*3+4" offenbar als Produkt aufgefasst, was der Konvention widerspricht, dass der Operator "*" stärker bindet als der Operator "+". Würden wir den String anhand des letzten Syntax-Baums auswerten, würden wir offenbar ein falsches Ergebnis bekommen! Die Ursache dieses Problems ist die Tatsache, dass die zuletzt angegebene Grammatik mehrdeutig ist. Eine solche Grammatik ist zum Parsen ungeeignet. Leider ist die Frage, ob eine gegebene Grammatik mehrdeutig ist, im Allgemeinen nicht entscheidbar: Es lässt sich zeigen, dass diese Frage zum *Postschen Korrespondenz-Problem* äquivalent ist. Da das Postsche Korrespondenz-Problem als unlösbar nachgewiesen wurde, ist auch die Frage, ob eine Grammatik mehrdeutig ist, unlösbar. Ein Beweis dieser Behauptungen findet sich beispielsweise in dem Buch von Hopcroft, Motwani und Ullman [HMU06].

Beispiel: Es sei $\Sigma = \{\text{“A”}, \text{“B”}\}$. Die Sprache L enthalte alle die Wörter aus Σ^*, bei denen die Buchstaben "A" and "B" mit der gleichen Häufigkeit auftreten, es gilt also

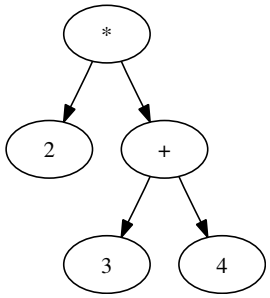


Figure 7.3: Ein anderer abstrakter Syntax-Baum für den String "2*3+4".

$$L = \big\{w \in \Sigma^* \mid \mathit{count}(w, \text{“A”}) = \mathit{count}(w, \text{“B”})\big\}.$$

Dann wird die Sprache L durch die kontextfreie Grammatik $G_1 = \langle\{s\}, \Sigma, R_1, s\rangle$ beschrieben, deren Regeln wie folgt gegeben sind:

$$s \;\rightarrow\; \text{“A”}\ s\ \text{“B”}\ s \;\mid\; \text{“B”}\ s\ \text{“A”}\ s \;\mid\; \varepsilon$$

Der Grund ist, dass ein String $w \in L$ entweder mit einem "A" oder mit einem "B" beginnt. Im ersten Fall muss es zu diesem "A" ein korrespondierendes "B" geben, denn die Anzahl der Auftreten von "A" und "B" sind gleich. Fassen wir den Buchstaben "A" wie eine öffnende Klammer auf und interpretieren den Buchstaben "B" als die zu "A" korrespondierende schließende Klammer, so ist klar, dass der String, der zwischen diesen beiden Auftreten von "A" und "B" liegt, ebenfalls gleich viele Auftreten von "A" wie von "B" hat. Genauso muss dies dann für den Rest des Strings gelten, der nach dem "B" folgt. Diese Überlegung erklärt die Regel

$$s \;\rightarrow\; \text{“A”}\ s\ \text{“B”}\ s$$

Die Regel

$$s \;\rightarrow\; \text{“B”}\ s\ \text{“A”}\ s$$

lässt sich in analoger Weise erklären, wenn wir den Buchstaben "B" als öffnende Klammer und "A" als schließende Klammer interpretieren.

Diese Grammatik ist allerdings mehrdeutig: Betrachten wir beispielsweise den String "abab", so stellen wir fest, dass sich dieser prinzipiell auf zwei Arten ableiten lässt:

$$\begin{array}{lll} s & \Rightarrow & \text{“A”}\ s\ \text{“B”}\ s \\ & \Rightarrow & \text{“A”}\ \ \text{“B”}\ s \\ & \Rightarrow & \text{“A”}\ \ \text{“B”}\ \ \text{“A”}\ s\ \text{“B”}\ s \\ & \Rightarrow & \text{“A”}\ \ \text{“B”}\ \ \text{“A”}\ \ \text{“B”}\ s \\ & \Rightarrow & \text{“A”}\ \ \text{“B”}\ \ \text{“A”}\ \ \text{“B”} \end{array}$$

Eine andere Ableitung desselben Strings ergibt sich, wenn wir im zweiten Ableitungs-Schritt nicht das erste s durch ε ersetzen sondern stattdessen das zweite s durch ε ersetzen:

$$\begin{array}{lll} s & \Rightarrow & \text{“A”}\ s\ \text{“B”}\ s \\ & \Rightarrow & \text{“A”}\ s\ \text{“B”} \\ & \Rightarrow & \text{“A”}\ \ \text{“B”}\ s\ \text{“A”}\ s\ \text{“B”} \\ & \Rightarrow & \text{“A”}\ \ \text{“B”}\ \ \text{“A”}\ s\ \text{“B”} \\ & \Rightarrow & \text{“A”}\ \ \text{“B”}\ \ \text{“A”}\ \ \text{“B”} \end{array}$$

Abbildung 7.4 zeigt die Parse-Bäume, die sich aus den beiden Ableitungen ergeben. Wir können erkennen, dass die Struktur dieser Bäume unterschiedlich ist: Im ersten Fall gehört das erste "A" zu dem ersten "B", im zweiten Fall gehört das erste "A" zu dem letzten "B".

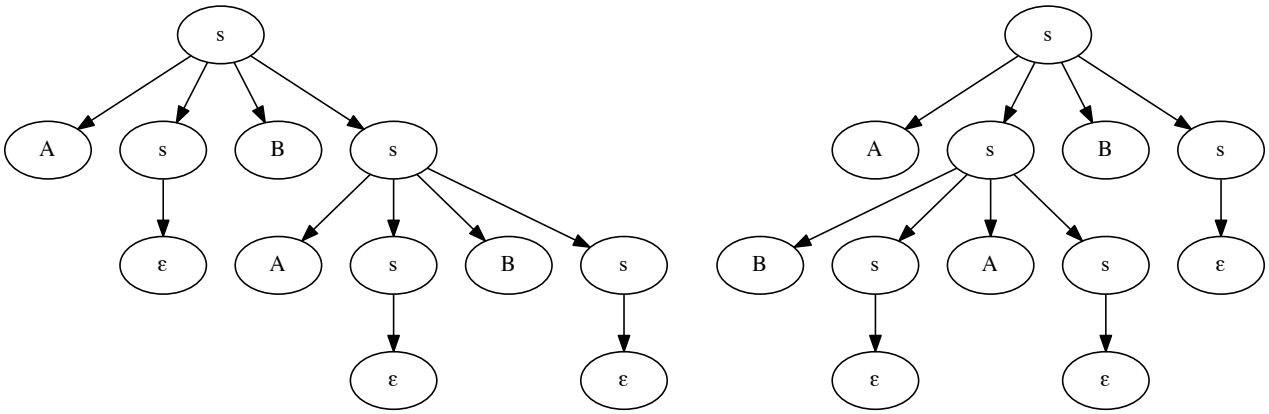


Figure 7.4: Zwei strukturell verschiedene Parse-Bäume für den String "ABAB".

Wir definieren nun eine kontextfreie Grammatik $G_2 = \langle \{s, u, v, x, y\}, \Sigma, R_2, s \rangle$, deren Regeln wie folgt gegeben sind:

$$
\begin{array}{lcl}
s & \to & us \mid vs \mid \varepsilon \\
u & \to & \text{“A”}\ x\ \text{“B”} \\
v & \to & \text{“B”}\ y\ \text{“A”} \\
x & \to & ux \mid \varepsilon \\
y & \to & vy \mid \varepsilon
\end{array}
$$

Um die Sprachen, die von den einzelnen Variablen erzeugt werden, klarer beschreiben zu können, definieren wir für zwei Strings σ und ω die Relation $\sigma \preceq \omega$ (lese: σ ist ein Präfix von ω) wie folgt:

$$\sigma \preceq \omega \quad \stackrel{\text{def}}{\iff} \exists \tau \in \Sigma^* : \sigma\tau = \omega$$

Sodann bemerken wir, dass von den syntaktischen Variablen x und y die folgenden Sprachen erzeugt werden:

$$L(x) = \big\{\omega \in \Sigma^* \mid \omega \in L \ \wedge\ \forall \sigma \preceq \omega : \mathit{count}(\sigma, \text{“A”}) \geq \mathit{count}(\sigma, \text{“B”})\big\} \quad \text{und}$$

$$L(y) = \big\{\omega \in \Sigma^* \mid \omega \in L \ \wedge\ \forall \sigma \preceq \omega : \mathit{count}(\sigma, \text{“A”}) \leq \mathit{count}(\sigma, \text{“B”})\big\}.$$

Ist $w \in L(x)$, so gibt es zu jedem Auftreten des Buchstabens "B" in dem String w ein dazu korrespondierendes Auftreten des Buchstabens "A", das dem Auftreten des Buchstabens "B" vorangeht. Würden wir den Buchstaben "A" durch eine öffnende Klammer und den Buchstaben "B" durch eine schließende Klammer ersetzen, so wird also niemals eine Klammer geschlossen, die nicht vorher geöffnet wurde. Damit ist klar, dass in einem String der Form

$$\text{“A”}\ w\ \text{“B”} \quad \text{mit } w \in L(x)$$

das zu dem ersten "A" korrespondierende "B" nur das letzte "B" sein kann. Analog können wir sehen, dass in einem String der Form

$$\text{“B”}\ w\ \text{“A”} \quad \text{mit } w \in L(Y)$$

das zu dem ersten "B" korrespondierende "A" nur das letzte "A" sein kann.

Ein String der Sprache L fängt nun entweder mit "A" oder mit "B" an. Im ersten Fall interpretieren wir das "A" als öffnende Klammer und das "B" als schließende Klammer und suchen nun das "B", das dem "A" am Anfang des Strings zugeordnet ist. Der String, der mit dem "A" anfängt und dem "B" endet, liegt in der Sprache $L(u)$. Auf dieses "B" kann dann noch ein weiterer Teilstring folgen, der gleich viele "A"s und "B"s enthält. Ein solcher Teilstring liegt offensichtlich ebenfalls in der Sprache L und kann daher von s mittels der Regel

$$s \to us$$

erzeugt werden. Im zweiten Fall fängt der String mit einem "B" an. Dieser Fall ist analog zum ersten Fall. □

In dem obigen Beispiel hatten wir Glück und konnten eine Grammatik finden, mit der sich die Sprache eindeutig parsen lässt. Es gibt allerdings auch kontextfreie Sprachen, die inhärent mehrdeutig sind: Es lässt sich beispielsweise zeigen, dass für das Alphabet $\Sigma = \{\text{"A"}, \text{"B"}, \text{"C"}, \text{"D"}\}$ die Sprache

$$L = \left\{ \mathtt{A}^m\mathtt{B}^m\mathtt{C}^n\mathtt{D}^n \mid m, n \in \mathbb{N} \right\} \cup \left\{ \mathtt{A}^m\mathtt{B}^n\mathtt{C}^n\mathtt{D}^m \mid m, n \in \mathbb{N} \right\}$$

kontextfrei ist, aber jede Grammatik G mit der Eigenschaft $L = L(G)$ ist notwendigerweise mehrdeutig. Das Problem ist, dass für gewisse große Zahlen $n \in \mathbb{N}$ ein String der Form

$$\mathtt{A}^n\mathtt{B}^n\mathtt{C}^n\mathtt{D}^n$$

immer zwei strukturell verschiedene Parse-Bäume besitzen muss. Ein Beweis dieser Behaupung findet sich in der ersten Auflage des Buchs von Hopcroft und Ullman auf Seite 100 [HU79].

7.2 Top-Down-Parser

In diesem Abschnitt stellen wir ein Verfahren vor, mit dem sich eine ganze Reihe von Grammatiken bequem parsen lassen. Die Grundidee ist einfach: Um einen String w mit Hilfe einer Grammatik-Regel der Form

$$a \to X_1 X_2 \cdots X_n$$

zu parsen, versuchen wir, zunächst ein X_1 zu parsen. Dabei zerlegen wir den String w in die Form $w = w_1 r_1$ so, dass $w_1 \in L(X_1)$ gilt. Dann versuchen wir, in dem Rest-String r_1 ein X_2 zu parsen und zerlegen dabei r_1 so, dass $r_1 = w_2 r_2$ mit $w_2 \in L(X_2)$ gilt. Setzen wir diesen Prozess fort, so haben wir zum Schluss den String w in

$$w = w_1 w_2 \cdots w_n \quad \text{mit } w_i \in L(X_i) \text{ für alle } i = 1, \cdots, n$$

aufgespaltet. Leider funktioniert dieses Verfahren dann nicht, wenn die Grammatik *links-rekursiv* ist, das heißt, dass eine Regel die Form

$$a \to a\beta$$

hat, denn dann würden wir um ein a zu parsen sofort wieder rekursiv versuchen, ein a zu parsen und wären damit in einer Endlos-Schleife. Es gibt mehrere Möglichkeiten, um mit diesem Problem umzugehen:

1. Wir können die Grammatik so umschreiben, dass sie danach nicht mehr links-rekursiv ist.
2. Wir können versuchen, den String *rückwärts* zu parsen, d.h. bei einer Regel der Form

   $$a \to X_1 X_2 \cdots X_n$$

   versuchen wir als erstes, ein X_n am Ende eines zu parsenden Strings w zu entdecken und arbeiten dann den String w von hinten nach vorne ab.
3. Die einfachste Lösung erhalten wir, wenn wir uns klar machen, dass kontextfreie Grammatiken nicht unbedingt die bequemste Art darstellen, eine Sprache zu beschreiben. Wir werden daher den Begriff der *erweiterten Backus-Naur-Form*-Grammatik (abgekürzt EBNF-Grammatik) einführen. Hierbei handelt es sich um eine Verallgemeinerung des Begriffs der kontextfreien Grammatik. Theoretisch ist die Ausdrucksstärke der EBNF-Grammatiken dieselbe wie die Ausdrucksstärke der kontextfreien Grammatiken. In der Praxis zeigt sich aber, dass die Konstruktion von Top-Down-Parsern für EBNF-Grammatiken einfacher ist, weil dort die Links-Rekursion durch eine Iteration ersetzt werden kann.

Im Rahmen dieses Kapitels werden wir alle oben genannten Verfahren anhand der Grammatik für arithmetische Ausdrücke ausführlich diskutieren.

7.2.1 Umschreiben der Grammatik*

In der folgenden Grammatik ist a eine syntaktische Variable und die griechischen Buchstaben β und γ stehen für irgendwelche Strings, die aus syntaktischen Variablen und Tokens bestehen. Wird die syntaktische Variable a durch

die beiden Regeln

$$\begin{array}{lcl} a & \to & a\beta \\ & \mid & \gamma \end{array}$$

definiert, so hat eine Ableitung von a, bei der zunächst immer die syntaktische Variable a ersetzt wird, die Form

$$a \Rightarrow a\beta \Rightarrow a\beta\beta \Rightarrow a\beta\beta\beta \Rightarrow \cdots \Rightarrow a\beta^n \Rightarrow \gamma\beta^n.$$

Damit sehen wir, dass die durch die syntaktische Variable a beschriebene Sprache $L(a)$ aus allen den Strings besteht, die sich aus dem Ausdruck $\gamma\beta^n$ ableiten lassen:

$$L(a) = \bigl\{w \in \Sigma^* \mid \exists n \in \mathbb{N}_0 : \gamma\beta^n \Rightarrow^* w\bigr\}.$$

Diese Sprache kann offenbar auch durch die folgenden Regeln für a beschrieben werden:

$$\begin{array}{lcl} a & \to & \gamma b \\ b & \to & \beta b \\ & \mid & \varepsilon \end{array}$$

Hier haben wir die Hilfs-Variable b eingeführt. Die Ableitungen, die von dem Nicht-Terminal b ausgehen, haben die Form

$$b \Rightarrow \beta b \Rightarrow \beta\beta b \Rightarrow \cdots \Rightarrow \beta^n b \Rightarrow \beta^n.$$

Folglich beschreibt das Nicht-Terminal b die Sprache

$$L(b) = \bigl\{w \in \Sigma \mid \exists n \in \mathbb{N}_0 : \beta^n \Rightarrow w\bigr\}.$$

Damit ist klar, dass auch mit der oben angegeben Grammatik

$$L(a) = \bigl\{w \in \Sigma^* \mid \exists n \in \mathbb{N}_0 : \gamma\beta^n \Rightarrow^* w\bigr\}$$

gilt. Um die Links-Rekursion aus der in Abbildung 7.5 auf Seite 85 gezeigten Grammatik zu entfernen, müssen wir das obige Beispiel verallgemeinern. Wir betrachten jetzt den allgemeinen Fall und nehmen an, dass ein Nicht-Terminal a durch Regeln der Form

$$\begin{array}{lcl} a & \to & a\beta_1 \\ & \mid & a\beta_2 \\ & \vdots & \vdots \\ & \mid & a\beta_k \\ & \mid & \gamma_1 \\ & \vdots & \vdots \\ & \mid & \gamma_l \end{array}$$

beschrieben wird. Wir können diesen Fall durch Einführung zweier Hilfs-Variablen b und c auf den ersten Fall zurückführen:

$$\begin{array}{lcl} a & \to & ab \mid c \\ b & \to & \beta_1 \mid \cdots \mid \beta_k \\ c & \to & \gamma_1 \mid \cdots \mid \gamma_l \end{array}$$

Dann können wir die Grammatik umschreiben, indem wir eine neue Hilfs-Variable, nennen wir sie l für Liste, einführen und erhalten

$$\begin{array}{lcl} a & \to & c\ l \\ l & \to & b\ l \mid \varepsilon. \end{array}$$

Die Hilfs-Variablen b und c können nun wieder eliminiert werden und dann bekommen wir die folgende Grammatik:

$$
\begin{array}{lcl}
a & \to & \gamma_1\, l \mid \gamma_2\, l \mid \cdots \mid \gamma_l\, l \\
l & \to & \beta_1\, l \mid \beta_2\, l \mid \cdots \mid \beta_k\, l \mid \varepsilon
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{expr} & \to & \mathit{expr}\ \text{“+”}\ \mathit{product} \\
& \mid & \mathit{expr}\ \text{“−”}\ \mathit{product} \\
& \mid & \mathit{product} \\
\mathit{product} & \to & \mathit{product}\ \text{“∗”}\ \mathit{factor} \\
& \mid & \mathit{product}\ \text{“/”}\ \mathit{factor} \\
& \mid & \mathit{factor} \\
\mathit{factor} & \to & \text{“(”}\ \mathit{expr}\ \text{“)”} \\
& \mid & \textsc{Number}
\end{array}
$$

Figure 7.5: Links-rekursive Grammatik für arithmetische Ausdrücke.

Wenden wir dieses Verfahren auf die in Abbildung 7.5 gezeigte Grammatik für arithmetische Ausdrücke an, so erhalten wir die in Abbildung 7.6 gezeigte Grammatik.

$$
\begin{array}{rcl}
\mathit{expr} & \to & \mathit{product}\ \ \mathit{exprRest} \\
\mathit{exprRest} & \to & \text{“+”}\ \mathit{product}\ \ \mathit{exprRest} \\
& \mid & \text{“−”}\ \mathit{product}\ \ \mathit{exprRest} \\
& \mid & \varepsilon \\
\mathit{product} & \to & \mathit{factor}\ \ \mathit{productRest} \\
\mathit{productRest} & \to & \text{“∗”}\ \mathit{factor}\ \ \mathit{productRest} \\
& \mid & \text{“/”}\ \mathit{factor}\ \ \mathit{productRest} \\
& \mid & \varepsilon \\
\mathit{factor} & \to & \text{“(”}\ \mathit{expr}\ \text{“)”} \\
& \mid & \textsc{Number}
\end{array}
$$

Figure 7.6: Grammatik für arithmetische Ausdrücke ohne Links-Rekursion.

Die Variablen *exprRest* und *productRest* können wie folgt interpretiert werden:

1. *exprRest* beschreibt eine Liste der Form

   $$\mathit{op}\ \mathit{product} \cdots \mathit{op}\ \mathit{product},$$

   wobei $\mathit{op} \in \{$ “+” , “−” $\}$ gilt.

2. *productRest* beschreibt eine Liste der Form

   $$\mathit{op}\ \mathit{factor} \cdots \mathit{op}\ \mathit{factor},$$

   wobei $\mathit{op} \in \{$ “∗” , “/” $\}$ gilt.

Aufgabe 26:

(a) Die folgende Grammatik beschreibt reguläre Ausdrücke:

$$
\begin{array}{lcl}
\textit{regExp} & \to & \textit{regExp}\ \text{“+”}\ \textit{regExp} \\
& | & \textit{regExp}\ \ \textit{regExp} \\
& | & \textit{regExp}\ \text{“*”} \\
& | & \text{“(”}\ \textit{regExp}\ \text{“)”} \\
& | & \textsc{Letter}
\end{array}
$$

Diese Grammatik verwendet nur die syntaktische Variable $\{\textit{regExp}\}$ und die folgenden Terminale

“+”, “*”, “(”, “)”, Letter.

Da die Grammatik mehrdeutig ist, ist diese Grammatik zum Parsen ungeeignet. Transformieren Sie diese Grammatik in eine eindeutige Grammatik, bei welcher der Postfix-Operator “*” stärker bindet als die Konkatenation zweier regulärer Ausdrücke, während der Operator “+” schwächer bindet als die Konkatenation. Orientieren Sie sich dabei an der Grammatik für arithmetische Ausdrücke und führen Sie geeignete neue syntaktische Variablen ein.

(b) Entfernen Sie die Links-Rekursion aus der in Teil (a) dieser Aufgabe erstellten Grammatik. ◇

$$
\begin{array}{lcl}
s & \to & a\ \text{“X”} \\
& | & \text{“Y”} \\
a & \to & b\ \text{“Y”} \\
& | & \text{“x”} \\
b & \to & s\ \text{“Z”}
\end{array}
$$

Figure 7.7: Wechselseitige Links-Rekursion.

Bei den bisher diskutierten Beispielen war die Links-Rekursion der Grammatik unmittelbar anzusehen. Es gibt allerdings Fälle, in denen die Links-Rekursion ihren Ursprung in *wechselseitiger Rekursion* hat. Abbildung 7.7 auf Seite 86 zeigt ein solches Beispiel: Bei der dort angegebenen Grammatik erstreckt sich die Links-Rekursion über drei Stufen: Ein s kann mit einem a beginnen, das mit einem b beginnen kann, welches dann wieder mit einem s beginnt. Eine Links-Rekursion der Form

$$a \to a\beta$$

bezeichnen wir als *unmittelbare Links-Rekursion*, jede kompliziertere Form der Links-Rekursion wird als *allgemeine Links-Rekursion* bezeichnet. Um allgemeine Links-Rekursion zu eliminieren, gehen wir wie folgt vor:

1. Zunächst nummerieren wir die syntaktischen Variablen der Grammatik in willkürlicher Weise durch. Im Folgenden seien die Variablen mit $a_1, \cdots, a_n$ bezeichnet. Durch diese Nummerierung wird implizit eine Ordnung $\succ$ auf den syntaktischen Variablen definiert, wir setzen

$$a_1 \succ a_2 \succ \cdots \succ a_i \succ a_{i+1} \succ \cdots \succ a_n.$$

2. Das Ziel ist nun, die Grammatik so umzuschreiben, dass für jede Grammatik-Regel der Form

   $a \to b\gamma$ die Ungleichung $a \succ b$

   gilt. Dieses Ziel wird durch den folgenden Algorithmus erreicht:

```
for (i = 1; i <= n; ++i) {
   for (j = 1; j < i; ++j) {
      forall (a_i → a_j γ) ∈ R {
         let a_j → δ_1 | ··· | δ_k be all a_j-productions
         replace a_i → a_j γ by a_i → δ_1 γ | ··· | δ_k γ
      }
   }
   eliminate immediate left recursion for variable a_i
}
```

Um den Algorithmus zu verstehen, führen wir zunächst einen neuen Begriff ein: Falls die Grammatik eine Regel der Form

$$a \to b\gamma$$

enthält, wobei b eine syntaktische Variable ist, dann sagen wir, dass a unmittelbar von b abhängt. Die Idee bei dem oben angegebenen Algorithmus ist nun, dass nach dem i-ten Durchlaufen der äußeren `for`-Schleife die Variablen a_1, $\cdots$, a_i nur noch unmittelbar von solchen Variablen abhängen, die in der Aufzählung a_1, $\cdots$, a_n auf diese Variablen folgen. Formal gilt nach dem i-ten Durchlauf der äußeren `for`-Schleife für alle Indizes $k \in \{1, \cdots, i\}$: Falls die Grammatik eine Regel der Form

$$a_k \to a_l\beta$$

enthält, dann muss $l > k$ gelten. Es ist leicht zu sehen, dass diese Invariante tatsächlich gilt: Vor dem i-ten Durchlauf gilt die Invariante für die Indizes der Menge $\{1, \cdots, i-1\}$. Die Variable a_i selber kann dann noch unmittelbar von den Variablen a_1, $\cdots$, a_n abhängen. In der inneren `for`-Schleife erreichen wir, dass nacheinander die unmittelbaren Abhängigkeiten von a_1, $\cdots$, a_{i-1} aufgelöst werden. Anschließend kann a_i höchstens noch unmittelbar von a_i abhängen. Um diese Abhängigkeit gegebenenfalls aufzulösen, führen wir die bereits früher diskutierte Transformation zur Elimination unmittelbarer Links-Rekursion durch. Anschließend hängt a_i höchstens noch unmittelbar von a_{i+1}, $\cdots$, a_n ab. Läuft die Schleife bis zum Ende durch, ist damit die Links-Rekursion vollständig eliminiert, denn dann kann jede Variable nur von solchen Variablen abhängen, die in der Aufzählung $a_1 \cdots, a_n$ hinter ihr stehen. Folglich ist kein Zyklus der Form

$$a_i \Rightarrow a_j\beta \Rightarrow \cdots \Rightarrow a_i\gamma$$

mehr möglich.

Beispiel: Wir demonstrieren das Verfahren an der in Abbildung 7.7 gezeigten Grammatik. Dazu ordnen wir zunächst die Variablen in der Form

$$s, a, b$$

an, mit der Notation des oben angegebenen Algorithmus gilt also $a_1 := s$, $a_2 := a$ und $a_3 := b$.

1. $i = 1$: Da $\{1, \cdots, i-1\} = \{\}$ gilt, wird in diesem Fall die innere `for`-Schleife nicht ausgeführt. Wir müssen lediglich die unmittelbare Links-Rekursion in der Variablen s entfernen. Da die Grammatik aber für s keine unmittelbare Links-Rekursion enthält, ist in diesem Fall nichts zu tun.

2. $i = 2$: In diesem Schritt müssen wir in der inneren `for`-Schleife sicherstellen, dass a nicht unmittelbar von s abhängt. da a in der gegebenen Grammatik nicht unmittelbar von s abhängt, ist bei der inneren `for`-Schleife wieder nichts zu tun.

   Weiter müssen wir die unmittelbare Rekursion aus allen Regeln für a eliminieren. Da es für die Variable a keine unmittelbare Rekursion gibt, ist an dieser Stelle ebenfalls nichts zu tun.

3. $i = 3$: In diesem Fall kommen für die innere `for`-Schleife zwei Werte von j in Frage, die wir nacheinander behandeln müssen:

   (a) $j = 1$: Hier müssen wir sicherstellen, dass b nicht unmittelbar von s abhängt. Bei der Regel

   $$b \to s \text{ “Z”}$$

   ist dies aber der Fall. Wir ersetzen daher das s auf der rechten Seite dieser Regel durch die beiden rechten Seiten der Regeln für s und erhalten für b nun die Regeln

   $$b \to a \text{ “X” “Z”} \quad \text{und} \quad b \to \text{ “Y” “Z”}.$$

   (b) $j = 2$: Nun müssen wir sicherstellen, dass b nicht unmittelbar von a abhängt. Bei der Regel

   $$b \to a \text{ “X” “Z”}$$

   ist dies aber der Fall. Wir ersetzen daher das a auf der rechten Seite dieser Regel durch die beiden rechten Seiten der Regeln für a und erhalten für b nun insgesamt die folgenden Regeln:

   $$b \to b \text{ “Y” “X” “Z”}, \quad b \to \text{ “X” “X” “Z”} \quad \text{und} \quad b \to \text{ “Y” “Z”}.$$

   Diese Regeln enthalten nun nur noch ummittelbare Links-Rekursion, die wir mit dem früher beschriebenen Verfahren eliminieren. Wir erhalten dann für b die Regeln

   $$b \to \text{ “X” “X” “Z”} \; l \quad \text{und} \quad b \to \text{“Y” “Z”} \; l,$$

   wobei die neu eingeführte Variable l durch die Regeln

   $$l \to \text{ “Y” “X” “Z”} \; l \quad \text{und} \quad l \to \varepsilon$$

   definiert ist.

Abbildung 7.8 zeigt die resultierende Grammatik.

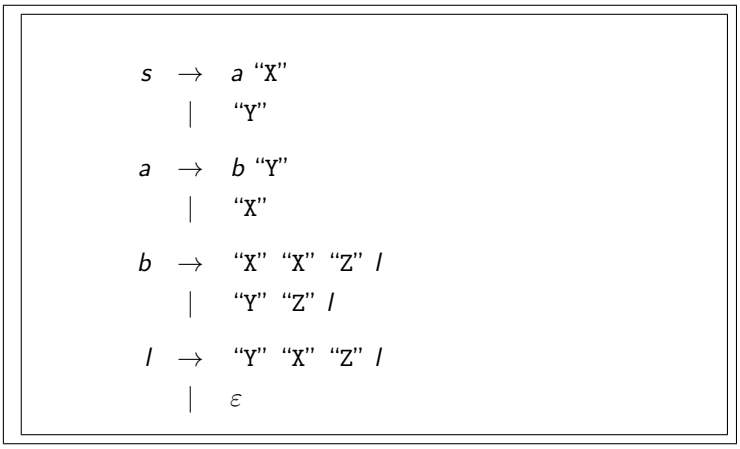


Figure 7.8: Grammatik ohne Links-Rekursion.

Das letzte Beispiel zeigt, dass sich eine Grammatik durch die Elimination indirekter Links-Rekursion stark aufblähen kann. Zwar sind viele Grammatiken links-rekursiv, aber in der Regel handelt es sich dabei um direkte Links-Rekursion. Wechselseitige Links-Rekursion ist in den Grammatiken, die Ihnen in der Praxis begegnen werden, ein eher seltenes Phänomen.

7.2.2 Implementing a Top Down Parser in SetlX

Now we are ready to implement a parser for recognizing arithmetic expressions. Figure 7.9 on page 89 shows an implementation of a recursive decent parser in SetlX.

```
myParse := procedure(s) {
     [result, rl] := parseExpr(tokenizeString(s));
     assert(rl == [], "Parse Error: could not parse $tl$");
     return result;
};
parseExpr := procedure(tl) {
    [product, rl] := parseProduct(tl);
    return parseExprRest(product, rl);
};
parseExprRest := procedure(sum, tl) {
    match (tl) {
        case ["+" | rl] : [product, ql] := parseProduct(rl);
                          return parseExprRest(sum + product, ql);
        case ["-" | rl] : [product, ql] := parseProduct(rl);
                          return parseExprRest(sum - product, ql);
        default:          return [sum, tl];
    }
};
parseProduct := procedure(tl) {
    [factor, rl] := parseFactor(tl);
    return parseProductRest(factor, rl);
};
parseProductRest := procedure(product, tl) {
    match (tl) {
        case ["*" | rl] : [factor, ql] := parseFactor(rl);
                          return parseProductRest(product * factor, ql);
        case ["/" | rl] : [factor, ql] := parseFactor(rl);
                          return parseProductRest(product / factor, ql);
        default:          return [product, tl];
    }
};
parseFactor := procedure(tl) {
    match (tl) {
        case ["(" | rl] : [expr, ql] := parseExpr(rl);
                          assert(ql[1] == ")", "Parse Error");
                          return [expr, ql[2..]];
        default : return [tl[1], tl[2..]];
    }
};
tokenizeString := procedure(s) {
    tokenList := [];
    scan (s) {
        regex '0|[1-9][0-9]*' as [ number   ]: tokenList += [ int(number) ];
        regex '[-+*/()]'      as [ operator ]: tokenList += [ operator    ];
        regex '[ \t\v\n]+'                   : // skip
    }
    return tokenList;
};
```

Figure 7.9: A top down parser for arithmetic expressions.

1. The main function is `myParse`[1]. This function takes a string s representing an arithmetic expression. This string is tokenized using the function `tokenizeString`. The function `tokenizeString` turns a string into a list of tokens. For example, the expression

   ```
   tokenizeString("(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.

2. The function `parseExpr` implements the grammar rule

   expr $\rightarrow$ *product exprRest*.

   It takes a token list `tl` as input. It will return a pair of the form

   `[`v`, rl]`,

   where v is the value of the arithmetic expression that has been parsed, while `rl` 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.

3. The function `parseExprRest` takes a number and a list of tokens. It implements the following grammar rules:

   $$\begin{array}{lcl} \textit{exprRest} & \rightarrow & \text{“+”}\ \textit{product}\ \textit{exprRest} \\ & | & \text{“−”}\ \textit{product}\ \textit{exprRest} \\ & | & \varepsilon \end{array}$$

[1] We had to name the function `myParse` instead of `parse` as SetlX already implements a function with the name `parse`. This function parses strings as SetlX expressions. The function `parse` returns a term representing the abstract syntax tree corresponding to the parsed expression.

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

$$\textit{exprRest} \rightarrow \varepsilon$$

is used. Therefore, in that case we have not consumed any tokens and hence the input arguments are already the result.

4. The function `parseProduct` implements the rule

   $$\textit{product} \rightarrow \textit{factor}\ \textit{exprRest}.$$

   The implementation is similar to the implementation of *parseExpr*.

5. The function `parseProductRest` implements the rules

   $$\begin{array}{lcl} \textit{productRest} & \rightarrow & \text{"*"}\ \textit{factor}\ \textit{productRest} \\ & | & \text{"/"}\ \textit{factor}\ \textit{productRest} \\ & | & \varepsilon \end{array}$$

   The implementation is similar to the implementation of *parseExprRest*.

6. 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.

7. The last function `tokenizeString` transforms a string into a list of tokens. To this end it uses the `scan` mechanism that is already built into SetlX. For example, in line 43 it is checked whether the next part of the input string is matched by the regular expression `0|[1-9][0-9]*`. If this is the case, the matching part is choped off the string and converted into a number which is then added to the list of tokens seen so far.

   In line 44 we recognize the operator symbols and the parenthesis. Note that we had to put the operator "−" first here since otherwise it would have been mistaken as a range operator.

   Line 45 is needed to skip white space.

The parser shown in Figure 7.9 does not contain any error handling. Appropriate error handling will be discussed once we have covered the theory of top-down parsing.

7.2.3 Implementing a Backward Recursive Decent Parser

If a grammar is left recursive but not right recursive then, instead of rewriting the grammar, we can just try to process the grammar rules backwards. Figure 7.10 on page 92 shows an recursive decent parser for arithmetic expressions that works in this way.

1. The function function `parsearithExpr` implements the following grammar rules:

   $$\textit{arithExpr} \rightarrow \textit{arithExpr}\ \text{"+"}\ \textit{product} \mid \textit{arithExpr}\ \text{"−"}\ \textit{product} \mid \textit{product}.$$

   According to these rules, an *arithExpr* always ends with a *product*. Therefore, the first thing to do is to parse a *product* from the end of the token list. This is done using the procedure `parseProduct`. Invoking this procedure consumes some of the tokens from the end of the token list `tl` and returns the list `fp` of those tokens that have not been consumed together with the `product` that has been recognized. Then, there are three cases:

```
1  parseExpr := procedure(tl) {
2      [fp, product] := parseProduct(tl);
3      [rp, op] := split(fp);
4      if (op in ["+", "-"]) {
5          [fp, expr] := parseExpr(rp);
6          match (op) {
7              case "+": return [fp, expr + product];
8              case "-": return [fp, expr - product];
9          }
10     }
11     return [fp, product];
12 };
13 parseProduct := procedure(tl) {
14     [fp, factor] := parseFactor(tl);
15     [rp, op] := split(fp);
16     if (op in ["*", "/"]) {
17         [fp, product] := parseProduct(rp);
18         match (op) {
19             case "*": return [fp, product * factor];
20             case "/": return [fp, product / factor];
21         }
22     }
23     return [fp, factor];
24 };
25 parseFactor := procedure(tl) {
26     [fp, op] := split(tl);
27     if (op == ")") {
28         [fp, expr] := parseExpr(fp);
29         [fp, op] := split(fp);
30         assert(op == "(", "parse error in $parseFactor(tl)$");
31         return [fp, expr];
32     }
33     assert(isNumber(op), "parse error in $parseFactor(tl)$");
34     return [fp, op];
35 };
36 split := procedure(l) {
37     if (#l > 0) {
38         return [l[1 .. #l-1], l[#l]];
39     }
40     return [[], ""];
41 };
```

Figure 7.10: A recursive decent parser working backwards.

(a) If the token immediately preceding the `product` is the symbol "+", then the parser tries to recognize an arithmetic expression using a recursive invocation of the procedure `parseExpr`. If this works and returns the result `expr`, then the end result is the sum `expr` + `product`, which is returned in line 7 together with the tokens that have not been consumed.

(b) If the token immediately preceding the `product` is "-", then everything works as in the first case, but instead the parser returns the difference `expr` − `product`.

(c) Otherwise, the parser has either hit an opening parenthesis or has already parsed the entire list of tokens.

In this case, the parser just returns the `product` together with the remaining tokens.

2. The function `parseProduct` tries to parse a product using the following rules:

$$\begin{array}{lcl} \textit{product} & \rightarrow & \textit{product} \text{ “}*\text{” } \textit{factor} \\ & | & \textit{product} \text{ “/” } \textit{factor} \\ & | & \textit{factor}. \end{array}$$

   This time, the parser tries to recognize a `factor` at the end of the token list `tl`. If this `factor` is preceded by either the token “`*`” or “/”, the parser tries to recognize the `product` that must be preceding this operator. In that case, depending on the operator, the parser either returns `product*factor` or `product/factor`.

   If the factor is not preceded by either “`*`” or “/” it must either be preceded by an opening parenthesis or the parser has already parsed the entire list of tokens. In this case, the parser just returns the `factor` together with the tokens that have not yet been parsed.

3. The function `parseFactor` implements the following grammar rules:

$$\begin{array}{lcl} \textit{factor} & \rightarrow & \text{“(” } \textit{arithExpr} \text{ “)”} \\ & | & \textsc{Number}. \end{array}$$

   If the token list `tl` ends with a closing parenthesis, the first of these rules has to be used to parse `tl`. Therefore in this case we parse an expression and expect an opening parenthesis before this expression.

   If the token list `tl` does not end with a closing parenthesis, we expect to find a number at the end of `tl`.

4. The method `split` is an auxilliary method that takes a list l as input. If this list is not empty, this method returns a pair: The first component of this list is the list of all elements of l but the last element, while the second component is the last element of the list l.

7.2.4 Implementing a Recursive Decent Parser that Uses an EBNF Grammar

The previous two solutions to parse an arithmetical expression were not completely satisfying: The reason is that we did not really fix the problem but rather cured the symptoms. The real problem 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 tool *JFlex*.

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 7.11 displays an EBNF grammar for arithmetical expressions. We have extended this grammar to allow for the exponentiation operator "**". In order to support this operator, we had to introduce a new syntactical variable, which we called *base*. In an arithmetical expression, *base* serves as the base of a power. The exponent can be an arbitrary *factor*. This way, an expression of the form

```
2 ** 3 ** 4    is parsed as    2 ** (3 ** 4)
```

and therefore the operator "**" is right associative. This is also the convention used in mathematics. Furthermore, we have added the function symbols "exp" and "ln" to be able to support the exponential function and the natural logarithm. The grammar also supports variables. The reason is that we want to implement a program for *symbolic differentiation*: We want to implement a function that takes a string respresenting an arithmetical expression and then does the following:

1. In the first step, the string is translated into an abstract syntax tree.
2. In the second step, this tree is differentiated symbolically with respect to a given variable.

For example, given the string

```
x * exp(x)
```

the program is supposed to compute the answer

```
1 * exp(x) + x * exp(x),
```

since we have

$$\frac{d}{dx}\big(x \cdot \exp(x)\big) = 1 \cdot \exp(x) + x \cdot \exp(x)$$

Obviously, the grammar in Figure 7.11 is more concise than the context-free grammar discussed at the beginning of this chapter. For example, the first rule clearly expresses that an arithmetical expression is a list of products that are separated by the operators "+" and "−".
Figure 7.12 shows a parser that implements this grammar.

1. The function `parseArithExpr` recognizes a `product` in line 2. The value of this `product` is stored in the variable `result` together with the list `rl` of those tokens that have not been consumed yet. If the list `rl` is not empty and the first token in this list is either the operator "+" or the operator "−", then the function `parseArithExpr` tries to recognize more products. These are added to or subtracted from the `result` computed so far in line 7 or 8. If there are no more products to be parsed, the `while` loop terminates and the function returns the `result` together with the remaining tokens.
2. The function `parseProduct` recognizes a `factor` in line 14. The value of this `factor` is stored in the variable `result` together with the list `rl` of those tokens that have not been consumed yet. If the list `rl` 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 19 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 `rl` of tokens that have not been consumed.

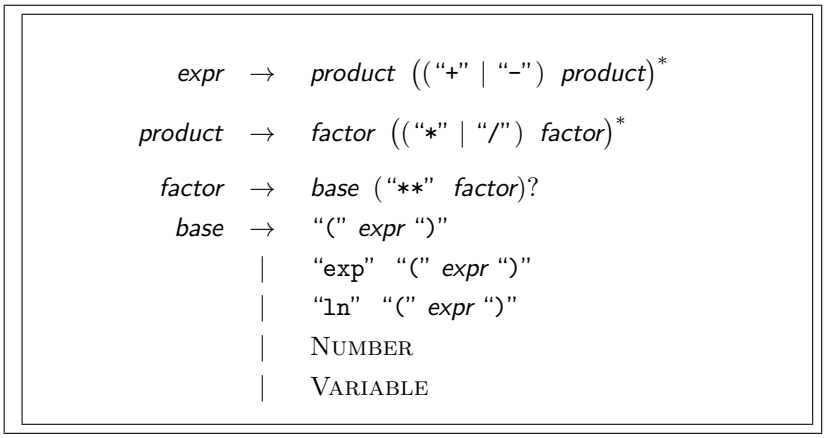


Figure 7.11: EBNF grammar for arithmetical expressions.

3. The function `parseFactor` recognizes a `factor`. In any case, a factor starts with a base. Optionally, a factor can be a power. This is the case if the base is followed by the exponentiation operator "**".

4. The function `parseBase` recognizes a call of the exponential function, a call of the natural logarithm, a parenthesized expression, a number, or a variable. Fortunately, the first token of the token list tells us which case we have.

The program in Figure 7.12 generates an abstract syntax tree. This syntax tree is represented as a term in SETLX. Note that in SETLX it is possible to use operators as functors. For example, if we have the expression

$$s + t$$

and at least one of the arguments s or t is a term, then $s + t$ is a term, too. This enables us to write

```
result := result + arg;
```

in line 7 of Figure 7.12. Finally, Figure 7.13 shows the implementation of the function `diff` that can be used for symbolic differentiation. The argument t of this function is supposed to be a term, the second argument x is interpreted as the name of a variable. For example, line 7 and 8 of Figure 7.13 implement the product rule. We have

$$\frac{d}{dx}(u \cdot v) = \frac{d\,u}{dx} \cdot v + u \cdot \frac{d\,v}{dx}.$$

The right hand side of this equation is returned in line 8: `t1` corresponds to u and `t2` corresponds to v. The other rules for differentiation are implemented in a similar way.

Historical Notes The language ALGOL [Bac59, NBB+60] was the first programming language with a syntax that was based on an EBNF grammar.

```
parseArithExpr := procedure(tl) {
    [result, rl] := parseProduct(tl);
    while (#rl > 1 && rl[1] in ["+", "-"]) {
        op := rl[1];
        [arg, rl] := parseProduct(rl[2..]);
        match (op) {
            case "+": result := result + arg;
            case "-": result := result - arg;
        }
    }
    return [result, rl];
};
parseProduct := procedure(tl) {
    [result, rl] := parseFactor(tl);
    while (#rl > 1 && rl[1] in ["*", "/"]) {
        op := rl[1];
        [arg, rl] := parseFactor(rl[2..]);
        match (op) {
            case "*": result := result * arg;
            case "/": result := result / arg;
        }
    }
    return [result, rl];
};
parseFactor := procedure(tl) {
    [atom, rl] := parseBase(tl);
    match (rl) {
        case [ "**" | ql ]: [factor, rl] := parseFactor(ql);
                            return [atom ** factor, rl];
        default:            return [atom, rl];
    }
};
parseBase := procedure(tl) {
    match (tl) {
        case [ "exp", "(" | rl ]: [expr, ql] := parseArithExpr(rl);
                                  assert(ql[1] == ")", "Parse Error");
                                  return [ Exp(expr), ql[2..]];
        case [ "ln", "(" | rl ]:  [expr, ql] := parseArithExpr(rl);
                                  assert(ql[1] == ")", "Parse Error");
                                  return [ Ln(expr), ql[2..]];
        case ["(" | rl ]:         [expr, ql] := parseArithExpr(rl);
                                  assert(ql[1] == ")", "Parse Error");
                                  return [expr, ql[2..]];
        case [ Number(n) | rl ]:  return [Number(n), rl];
        case [ Var(v) | rl ]:     return [Var(v), rl];
        default:                  abort("parse error in parseBase($tl$)");
    }
};
```

Figure 7.12: A parser for the grammar in Figure 7.11.

```
diff := procedure(t, x) {
    match (t) {
        case t1 + t2 :
             return diff(t1, x) + diff(t2, x);
        case t1 - t2 :
             return diff(t1, x) - diff(t2, x);
        case t1 * t2 :
             return diff(t1, x) * t2 + t1 * diff(t2, x);
        case t1 / t2 :
             return ( diff(t1, x) * t2 - t1 * diff(t2, x) ) / (t2 * t2);
        case f ** Number(n):
            return Number(n) * diff(f, x) * f ** Number(n-1);
        case f ** g :
             return diff( Exp(g * Ln(f)), x);
        case Ln(a) :
             return diff(a, x) / a;
        case Exp(a) :
             return diff(a, x) * Exp(a);
        case Var(x) : // x is defined above as second argument to diff
             return 1;
        case Var(y) : // y is undefined, therefore matches any other variable
             return 0;
        case Number(n):
             return 0;
        default:
            abort("error in diff($t$, $x$)");
    }
};
```

Figure 7.13: A function for symbolic differentiation.

Chapter 8

Introducing ANTLR

If the 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 tool 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 and most powerful is ANTLR [Par12, PHF14]. The name is short for *another tool for language recognition*[1]. ANTLR can be downloaded at

`http://www.antlr.org`.

In this lecture we will use ANTLR version 4.8. The tool ANTLR is written in *Java* and therefore its main target language is *Java*. However, ANTLR can alsobe 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 lack of time we will only discuss the most important features of ANTLR. 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.

8.1 A Parser for Arithmetic Expressions

We start with a parser for arithmetic expressions. The structure of arithmetic expressions can be described by the grammar that is shown in Figure 7.5 on page 85. If we use ANTLR we can implement this grammar as shown in Figure 8.1. 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`. The name of the file that contains this grammar is created by appending the file extension "`.g4`". Therefore this grammar has to be stored in a file with the name "`Expr.g4`".

2. Line 3 gives the production 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 production are separated by a colon. ANTLR terminates every production with the character "`;`

3. Line 6–9 give the productions 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 productions for the non-terminals *product* and *factor*.

[1] 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 13.

```
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 8.1: 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,[2] 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" 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 range and characters inside a range must never be quoted.

7. Line 21 defines the terminal WS, where the name 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[3], 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 8.1 is stored in the file Expr.g4 we can generate a parser by using the following command:

```
java -jar /usr/local/lib/antlr-4.8-complete.jar -Dlanguage=Python3 Expr.g4
```

[2] It is a convention that the names of non-terminals consist of only upper case letters, but this is not required.

[3] Unfortunately, *Python* is an exception to this rule. Therefore, parsing *Python* is considerably harder than parsing languages like C or *Java*.

Of course, this only works if the file `antlr-4.8-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 8.2 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      stream      = antlr4.CommonTokenStream(lexer)
10     parser      = ExprParser(stream)
11     parser.start()
```

Figure 8.2: 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-4.8-complete.jar -Dlanguage=Java Expr.g4
```

The generated *Java* files have to be compiled with the following command:

```
javac -cp .:/usr/local/lib/antlr-4.8-complete.jar *.java
```

After that we can generate a parse tree using the command

```
java -cp .:/usr/local/lib/antlr-4.8-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 as shown in Figure 8.3.

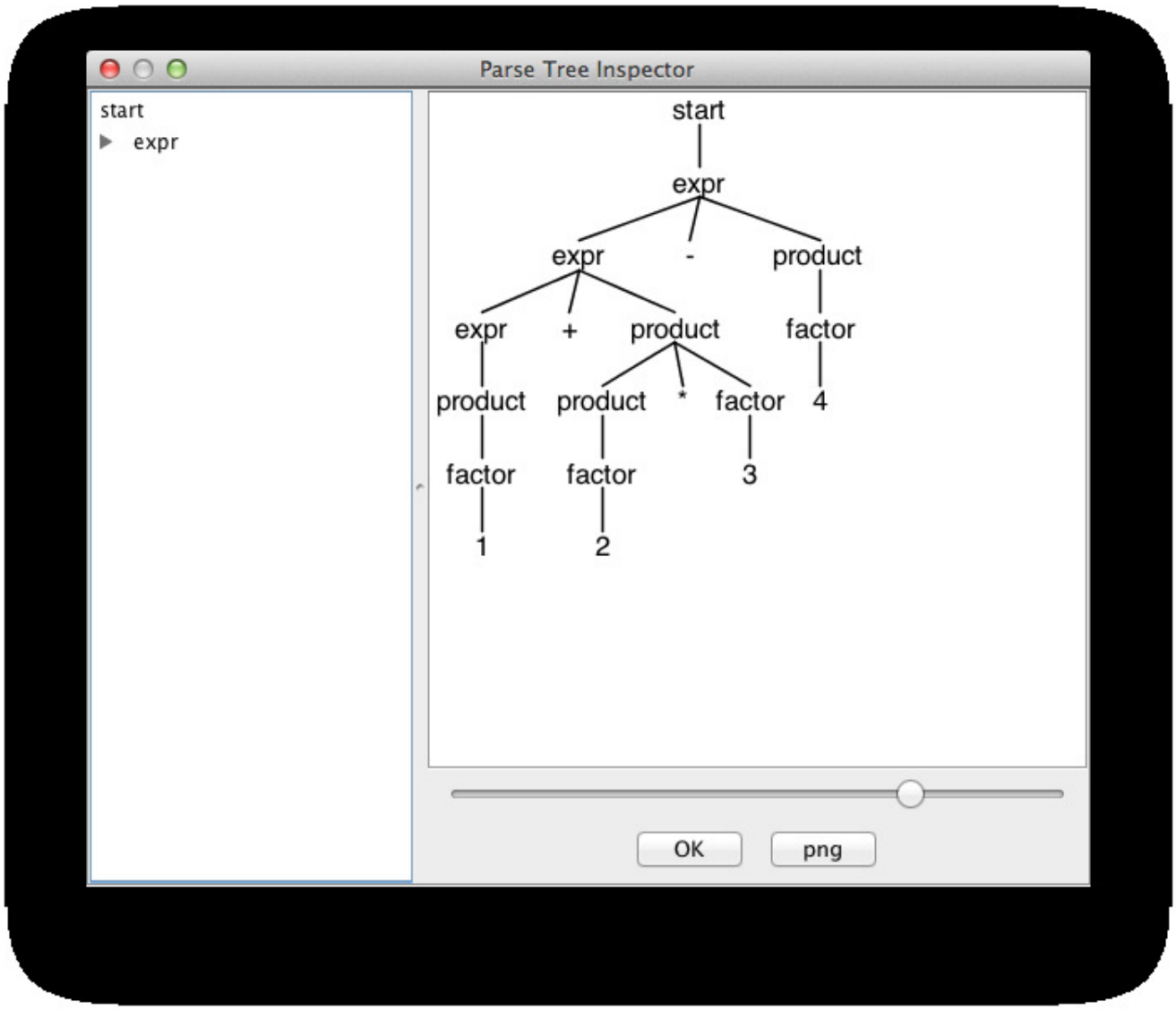


Figure 8.3: Parse tree for the string "1 + 2 * 3 - 4".

8.2 Evaluation of Arithmetical Expressions

The last example isn't very exciting as the arithmetical expressions that the parser has read 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 small symbolic calculator.
2. Secondly, we show how this grammar can be extended with actions so that the expressions can be evaluated.

Figure 8.4 presents the grammar. In comparison with grammar for arithmetical expressions that was shown in Figure 8.1 there are the following changes:

1. Das 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.

```
grammar Program;

start: statement+ ;

statement
    : IDENTIFIER ':=' expr ';'
    | expr ';'
    ;

expr: expr '+' product
    | expr '-' product
    | product
    ;

product
    : product '*' factor
    | product '/' factor
    | factor
    ;

factor
    : '(' expr ')'
    | FLOAT
    | IDENTIFIER
    | 'sqrt' '(' expr ')'
    ;

IDENTIFIER: [a-zA-Z][a-zA-Z0-9_]*;
FLOAT     : '0'([.][0-9]+)?
          | [1-9][0-9]*([.][0-9]+)?
          ;
WS        : [ \t\n\r] -> skip;
```

Figure 8.4: A grammar for a symbolic calculator.

2. A `statement` is either an assignment or an expression.
3. The rules for expressions and products are the same as previously.
4. The rule for the non-terminal `factor` has changed.
   (a) We can still put expressions in parenthesis.
   (b) Instead of integers the grammar now supports floating point numbers. These are recognized by the terminal `FLOAT`.
   (c) Expressions can now contain variables. These are recognized by the terminal `IDENTIFIER`.
   (d) Furthermore, expressions are allowed to contain calls of the function `sqrt`, which is supposed to compute the square root of a given number.
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 8.4. Figure 8.5 shows how an interpreter can be realized using ANTLR.

```
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 8.5: Ein Interpreter zur Auswertung von Ausdrücken.

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.

2. If the parser recognizes an assignment of the form

   IDENTIFIER := *expr*;

   it has to evaluate the expression *expr* and store 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 by 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 "e=expr" assigns the result of evaluating *product* to the variable p. The action code adds these variables and assigns the sum to the variables `result`. Note that all these variable names have to prefixed by a dollar symbol.

6. Line 28 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.

```
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 8.6: A program to utilize the generated parser.

Next, we need a driver program to call the parser that ANTLR generates from the grammar shown in Figure 8.5. Figure 8.6 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.

(c) Line 8 reads a line of input. As long as there is still 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.

(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.

8.3 Generating Abstract Syntax Trees with Antlr

The evaluation of arithmetical was relatively easy as it is possible to evaluate an arithmetical expression directly via semantic actions that are embedded in the grammar. If the problems are 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 8.7 shows the grammar that we are going to use for symbolic differentiation.

expr	→	*expr* "+" *product*
	|	*expr* "-" *product*
	|	*product*
product	→	*product* "*" *factor*
product	|	*product* "/" *factor*
	|	*factor*
factor	→	"(" *expr* ")"
	|	"exp" "(" *expr* ")"
	|	"log" "(" *expr* ")"
	|	Variable
	|	Number

Figure 8.7: A grammar for arithmetical expressions with exponential function and logarithm.

8.3.1 Implementing the Parser

Figure 8.8 shows how the grammar from Figure 8.7 is implemented with ANTLR.

```
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 = $v.text           }
20     | n=NUM                {$result = int($n.text)      }
21     ;
22
23 VAR : [a-zA-Z][a-zA-Z0-9]*;
24 NUM : '0'|[1-9][0-9]*;
25 WS  : [ \t\n\r] -> skip;
```

Figure 8.8: ANTLR implementation of the grammar form Figure 8.7.

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 `$e.result`, while the syntax tree for the product is stored in the variable `$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 19 we can get the actual text that is matched by the terminal `VAR` by writing `$v.text`.

5. In line 20 we have to convert the string recognized by the terminal `NUM` 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 8.9. 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 e[0] == '+':
3          f , g  = e[1:]
4          fs, gs = diff(f), diff(g)
5          return ('+', fs, gs)
6      if e[0] == '-':
7          f , g  = e[1:]
8          fs, gs = diff(f), diff(g)
9          return ('-', fs, gs)
10     if e[0] == '*':
11         f , g  = e[1:]
12         fs, gs = diff(f), diff(g)
13         return ('+', ('*', fs, g), ('*', f, gs))
14     if e[0] == '/':
15         f , g  = e[1:]
16         fs, gs = diff(f), diff(g)
17         return ('/', ('-', ('*', fs, g), ('*', f, gs)), ('*', g, g))
18     if e[0] == 'ln':
19         f  = e[1]
20         fs = diff(f)
21         return ('/', fs, f)
22     if e[0] == 'exp':
23         f  = e[1]
24         fs = diff(f)
25         return ('*', fs, e)
26     if e == 'x':
27         return '1'
28     return 0
```

Figure 8.9: A function to compute the symbolic differentiation of a given expression.

Finally, Figure 8.10 shows how the parser can be invoked. Invoking the parser in line 10 creates an abstract syntax tree that is stored in the variable `term` in line 10. This abstract syntax tree is then used as input to the function `diff`.

```
def main():
    parser        = DifferentiatorParser(None)
    parser.Values = {}
    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 8.10: A driver program for the grammar shown in Figure 8.8

Exercise 27: 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:

1. 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.

2. 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.

3. 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`".

Chapter 9

LL(k)-Sprachen

In diesem Kapitel werden wir die Theorie vorstellen, die Top-Down Parser-Generatoren wie beispielsweise ANTLR zu Grunde liegt. Es handelt sich dabei um die Theorie der LL(k)-Sprachen. Dabei steht das erste L dafür, dass der Parser die Eingabe von links nach rechts parst, das zweite L steht dafür, dass der Parser versucht, eine Links-Ableitung des zu parsenden Wortes zu berechnen. Eine Links-Ableitung ist dabei eine Ableitung, bei der immer die linkeste Variable ersetzt wird. Die Zahl k in LL(k) bedeutet, dass der Parser anhand der nächsten k Token entscheidet, welche Regel verwendet wird. Falls beispielsweise $k = 1$ ist, wird also nur das nächste Token zur Entscheidung herangezogen. Dieses Token bezeichnen wir dann als *Lookahead-Token*. Wir betrachten zunächst den Fall $k = 1$. Bevor wir die Theorie der LL(1)-Parser darstellen, machen wir auf zwei Probleme aufmerksam, die wir bei der Erstellung von Top-Down-Parsern lösen müssen.

1. Das erste Problem ist die Links-Rekursion, die beispielsweise in den folgenden Regeln zur Beschreibung arithmetischer Ausdrücke auftritt:

   $$\begin{array}{lcl} \textit{expr} & \to & \textit{expr}\ \text{“+”}\ \textit{product} \\ & | & \textit{expr}\ \text{“–”}\ \textit{product} \\ & | & \textit{product} \end{array}$$

   Wir hatten im Abschnitt 7.2.1 gezeigt, wie die Links-Rekursion aus einer Grammatik eliminiert werden kann.

   Zusätzlich haben wir gesehen, dass es bei der Verwendung von EBNF-Grammatiken oft in natürlicher Weise möglich ist, die Links-Rekursion durch die Verwendung der Postfix-Operatoren “∗” und “+” zu vermeiden. Beispielsweise können wir die obigen Regeln für arithmetische Ausdrücke zu der EBNF-Regel

   $$\textit{expr} \to \textit{product}\ \big(\big(\ \text{“+”} \mid \text{“–”}\ \big)\ \textit{product}\big)^*$$

   umschreiben.

2. Das zweite Problem erkennen wir, wenn wir die folgende Grammatik-Regel für Gleichungen und Ungleichungen betrachten:

   $$\begin{array}{lcl} \textit{boolExpr} & \to & \textit{expr}\ \text{“==”}\ \textit{expr} \\ & | & \textit{expr}\ \text{“<”}\ \ \textit{expr} \end{array}$$

   Diese Grammatik-Regeln sind nicht links-rekursiv, aber für einen Top-Down-Parser, der mit nur einem Token Look-Ahead auskommen soll, ist die Frage, welche der beiden Regeln zum Parsen verwendet werden soll, offenbar nicht zu beantworten. Wir stellen gleich ein Verfahren vor, mit dem sich Grammatiken so transformieren lassen, dass dieses Problem verschwindet.

9.1 Links-Faktorisierung

Ist a ein Nicht-Terminal und gibt es zwei verschiedene Regeln, mit denen a abgeleitet werden kann, beispielsweise

$$a \to \beta \quad \text{und} \quad a \to \gamma,$$

so muss es bei der Verwendung eines LL(1)-Parsers möglich sein, anhand des Look-Ahead-Tokens zu erkennen, welche Regel benutzt werden soll. In der Praxis gibt es häufig Situationen, wo diese Voraussetzung nicht erfüllt ist. Wir haben oben bereits ein solches Beispiel gesehen. Um das Beispiel zu vervollständigen, benötigen wir noch Regeln zur Ableitung von *expr*. Abbildung 9.1 zeigt eine vollständige Grammatik, mit der Gleichungen und Ungleichungen beschrieben werden können, die aus arithmetischen Ausdrücken aufgebaut sind.

boolExpr	→	*expr* "==" *expr*
	|	*expr* "!=" *expr*
	|	*expr* "<=" *expr*
	|	*expr* ">=" *expr*
	|	*expr* ">" *expr*
	|	*expr* "<" *expr*
expr	→	*product exprRest*
exprRest	→	"+" *product exprRest*
	|	"−" *product exprRest*
	|	ε
product	→	*factor productRest*
productRest	→	"*" *factor productRest*
	|	"/" *factor productRest*
	|	ε
factor	→	"(" *expr* ")"
	|	NUMBER
	|	IDENTIFIER

Figure 9.1: Grammatik ohne Links-Rekursion für Gleichungen und Ungleichungen.

Es ist nicht möglich einen LL(1)-Parser zu implementieren, der Boole'sche Ausdrücke mit Hilfe dieser Regeln erkennen kann, denn alle Regeln für *boolExpr* beginnen mit *expr*. Wir können die Grammatik aber durch *Links-Faktorisierung* (Englisch: *left factoring*) so umschreiben, dass ein Token als Look-Ahead ausreicht, indem wir den Teil aus den beiden Grammatik-Regeln ausklammern, der am Anfang der beiden Regeln identisch ist. In dem obigen Beispiel führen wir dann für den verbleibenden Rest das neue Nicht-Terminal *boolExprRest* ein und erhalten so die Regeln

$$\begin{array}{rcl} \textit{boolExpr} & \to & \textit{expr}\ \ \textit{boolExprRest} \\ \textit{boolExprRest} & \to & \text{"=="}\ \textit{expr} \mid \text{"!="}\ \textit{expr} \mid \text{"<="}\ \textit{expr} \\ & \mid & \text{">="}\ \textit{expr} \mid \text{"<"}\ \textit{expr} \mid \text{">"}\ \textit{expr}. \end{array}$$

Mit diesen Regeln reicht ein Token als Look-Ahead aus, denn die verschiedenen Alternativen für *boolExprRest* unterscheiden sich in dem ersten Token des Rumpfs der Regel. Verwenden wir statt einer einfachen Grammatik eine EBNF-Grammatik, so lassen sich die obigen Regeln kürzer und klarer in der Form

$$\textit{boolExpr} \to \textit{expr}\ (\ \text{"=="} \mid \text{"!="} \mid \text{"<="} \mid \text{">="} \mid \text{"<"}\ )\ \textit{expr}$$

darstellen.

Um den allgemeinen Fall der Links-Faktorisierung diskutieren zu können, nehmen wir an, dass *a* ein Nicht-Terminal ist, das durch insgesamt $m + n$ Regeln definiert wird, wobei der Rumpf der ersten m Regeln immer mit α anfängt, wobei α ein String aus Terminalen und syntaktischen Variablen ist. Die Regeln haben also die folgende Form:

$$\begin{array}{lcl} a & \to & \alpha\ \beta_1 \\ & | & \alpha\ \beta_2 \\ & \vdots & \vdots \\ & | & \alpha\ \beta_m \\ & | & \gamma_1 \\ & \vdots & \vdots \\ & | & \gamma_n \end{array}$$

Bei dieser Darstellung sei vorausgesetzt, dass die Strings $\beta_1, \cdots, \beta_m$ keinen Präfix haben, der allen β_i gemeinsam ist und dass α kein Präfix einer der Strings $\gamma_1, \cdots, \gamma_n$ ist. Bei der Links-Faktorisierung dieser Regeln klammern wir einerseits den gemeinsamen Präfix α aus und führen andererseits eine neue syntaktische Variable *b* ein, die den auf α folgenden Rest bezeichnet. Wir erhalten dann die folgenden Regeln:

$$\begin{array}{lcl} a & \to & \alpha\ b \\ & | & \gamma_1 \\ & \vdots & \vdots \\ & | & \gamma_n \end{array} \qquad\qquad \begin{array}{lcl} b & \to & \beta_1 \\ & | & \beta_2 \\ & \vdots & \vdots \\ & | & \beta_m \end{array}$$

Um alle gemeinsamen Präfixe auszuklammern muss dieses Verfahren unter Umständen mehrfach durchgeführt werden. Die nächste Aufgabe gibt dafür ein Beispiel.

Aufgabe 28: Geben Sie eine Links-Faktorisierung für die folgenden Grammatik-Regeln an.

$$\begin{array}{lcl} a & \to & \text{“A”}\ \ \text{“B”}\ u\ \text{“D”} \\ & | & \text{“A”}\ v\ \text{“B”}\ \ \text{“D”} \\ & | & \text{“A”}\ \ \text{“B”}\ w \\ & | & \text{“X”}\ u \\ & | & \text{“X”}\ v \end{array}$$

Lösung: Zunächst eliminieren wir das gemeinsame Präfix “A” und führen dazu die neue syntaktische Variable *b* ein. Wir erhalten:

$$\begin{array}{lcl} a & \to & \text{“A”}\ b \\ & | & \text{“X”}\ u \\ & | & \text{“X”}\ v \\ \\ b & \to & \text{“B”}\ u\ \text{“D”} \\ & | & v\ \text{“B”}\ \ \text{“D”} \\ & | & \text{“B”}\ w \end{array}$$

Nun eliminieren wir das Präfix “X” aus beiden letzten Regeln für *a*. Wir führen dazu die neue syntaktische Variable

c ein. Dann erhalten wir:

$$\begin{array}{lcl} a & \to & \text{“A”}\ b \\ & | & \text{“X”}\ c \\ c & \to & u \\ & | & v \\ b & \to & \text{“B”}\ u\ \text{“D”} \\ & | & v\ \text{“B”}\ \ \text{“D”} \\ & | & \text{“B”}\ w \end{array}$$

Als letztes eliminieren wir das Präfix “B”, das in zwei der Regeln für die syntaktische Variable b auftritt. Wir nennen die neu eingeführte Variable d und erhalten:

$$\begin{array}{lcl} a & \to & \text{“A”}\ b \\ & | & \text{“X”}\ c \\ c & \to & u \\ & | & v \\ b & \to & \text{“B”}\ d \\ & | & v\ \text{“B”}\ \ \text{“D”} \\ d & \to & u\ \text{“D”} \\ & | & w \end{array}$$

□

Remark: The parser generator ANTLR performs left-factorisation automatically.

9.2 First und Follow

Nicht für jede links-faktorisierte Grammatik lässt sich ein LL(1)-Parser bauen. Betrachten wir die folgenden Regeln:

$$\begin{array}{lcl} a & \to & b \mid c \\ b & \to & \text{“A”}\ u \\ c & \to & \text{“A”}\ v \end{array}$$

Will der Parser ein a parsen und ist das nächste Token ein “A”, so ist nicht klar, ob der Parser als nächstes die Regel

$$a \to b \quad \text{oder} \quad a \to c$$

verwenden soll. Für die obige Grammatik lässt sich daher kein LL(1)-Parser implementieren. Zur Entscheidung, ob sich für eine gegebene Grammatik ein LL(1)-Parser implementieren lässt, benötigen wir die Funktionen *First*() und *Follow*(), die wir gleich definieren werden. Um diese Funktionen implementieren zu können, definieren wir vorher den Begriff einer ε-erzeugenden syntaktischen Variablen.

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:

1. Bei der in Abbildung 9.1 auf Seite 110 gezeigten Grammatik sind offenbar die Variablen *exprRest* und *productRest* ε-erzeugend.
2. Wir betrachten nun ein weniger offensichtliches Beispiel. Die Grammatik G enthalte die folgenden Regeln:

   $$S \to a\ b\ c$$
   $$a \to \text{“X”}\ b \mid a\ \text{“Y”} \mid b\ c$$
   $$b \to \text{“X”}\ b \mid a\ \text{“Y”} \mid c\ c$$
   $$c \to a\ b\ c \mid \varepsilon$$

   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 $\mathit{First}(a) := \{\}$ 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 9.1 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

   $$\mathit{factor} \to \text{“(”}\ \mathit{expr}\ \text{“)”} \mid \textsc{Number} \mid \textsc{Identifier},$$

   dass jeder von *Factor* abgeleitete String entweder mit einer öffnenden Klammer, einer Zahl oder einem Bezeichner beginnt:

   $$\mathit{First}(\mathit{factor}) = \{\ \text{“(”}\ , \textsc{Number}, \textsc{Identifier}\ \}.$$

2. Analog folgt aus den Regeln

   $$\mathit{productRest} \to \text{“*”}\ \mathit{factor}\ \mathit{productRest} \mid \text{“/”}\ \mathit{factor}\ \mathit{productRest} \mid \varepsilon,$$

dass ein *productRest* entweder mit dem Zeichen "*" oder "/" beginnt:

$$\mathit{First}(\mathit{productRest}) = \{\ \text{“*”}\ ,\ \text{“/”}\ \}$$

3. Die Regel für die Variable *product* lautet

   $$\mathit{product} \to \mathit{factor}\ \ \mathit{productRest}.$$

   Da die Variable *factor* nicht ε erzeugend ist, sehen wir, dass die Menge $\mathit{First}(\mathit{product})$ mit der Menge $\mathit{First}(\mathit{factor})$ übereinstimmt:

   $$\mathit{First}(\mathit{product}) = \{\ \text{“(”}\ , \textsc{Number}, \textsc{Identifier}\ \}.$$

4. Aus den Regeln

   $$\mathit{exprRest} \to\ \text{“+”}\ \mathit{product}\ \ \mathit{exprRest} \mid\ \text{“–”}\ \mathit{product}\ \ \mathit{exprRest} \mid \varepsilon$$

   können wir $\mathit{First}(\mathit{exprRest})$ wie folgt berechnen:

   $$\mathit{First}(\mathit{exprRest}) = \{\ \text{“+”}\ ,\ \text{“–”}\ \}.$$

5. Weiter folgt aus der Regel

   $$\mathit{expr} \to \mathit{product}\ \ \mathit{exprRest}$$

   und der Tatsache, dass *product* nicht ε-erzeugend ist, dass die Menge $\mathit{First}(\mathit{expr})$ mit der Menge $\mathit{First}(\mathit{product})$ übereinstimmt:

   $$\mathit{First}(\mathit{expr}) = \{\ \text{“(”}\ , \textsc{Number}, \textsc{Identifier}\ \}.$$

6. Schließlich folgt aus den Regeln für die syntaktische Variable *boolExpr* sowie der Tatsache, dass die syntaktische Variable *expr* nicht ε-erzeugend ist, dass $\mathit{First}(\mathit{boolExpr})$ mit $\mathit{First}(\mathit{expr})$ identisch ist:

   $$\mathit{First}(\mathit{boolExpr}) = \{\ \text{“(”}\ , \textsc{Number}, \textsc{Identifier}\ \}.$$

Since we have computed the sets $\mathit{First}(a)$ 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:

$$\widehat{S} \to S\ \text{“\$”}\ .$$

Das Terminal "$" steht hierbei für das Ende der Eingabe (Eof, *end of file*). Weiter definieren wir

$$\widehat{T} := T \cup \{\ \text{“\$”}\ \}.$$

Die so veränderte Grammatik bezeichnen wir als die *augmentierte* Grammatik. Dann definieren wir $\mathit{Follow}(a)$ als die Menge aller der Token t, die in einer Ableitung auf a folgen können:

$$\mathit{Follow}(a) := \{t \in \widehat{T} \mid \exists \beta, \gamma \in (V \cup \widehat{T})^* : \widehat{S} \Rightarrow^* \beta a t \gamma\}.$$

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 $\mathit{Follow}(a)$. ◇

Beispiel: Wir untersuchen wieder die in Abbildung 9.1 gezeigte Grammatik für arithmetische Ausdrücke.

1. Aufgrund der neu hinzugefügten Regel

   $$\widehat{S} \to \mathit{boolExpr}\ \text{“\$”}$$

   muss die Menge $\mathit{Follow}(\mathit{boolExpr})$ das Zeichen "$" enthalten. Da die syntaktische Variable *boolExpr* sonst nirgends in der Grammatik vorkommt, haben wir

   $$\mathit{Follow}(\mathit{boolExpr}) = \{\ \text{“\$”}\ \}.$$

2. Die Grammatik-Regeln für die syntaktische Variable *boolExpr* zeigen uns zunächst, dass die Menge *Follow*(*expr*) die Zeichen "==", "!=", "<=", ">=", "<" und ">" enthält. Da *expr* auch am Ende dieser Regeln steht, folgt weiter, dass alle Elemente aus *Follow*(*boolExpr*) auch auf *expr* folgen können, wir haben also auch

   "$" $\in$ *Follow*(*expr*).

   Aufgrund der Regel

   factor $\rightarrow$ "(" *expr* ")"

   muss die Menge *Follow*(*expr*) außerdem das Zeichen ")" enthalten. Also haben wir insgesamt

   Follow(*expr*) = { "==" , "!=" , ">=" , "<=" , ">" , "<" , "$" , ")" }.

3. Aufgrund der Regel

   expr $\rightarrow$ *product 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

   Follow(*exprRest*) = { "==" , "!=" , ">=" , "<=" , ">" , "<" , "$" , ")" }.

4. Die Regeln

   exprRest $\rightarrow$ "+" *product exprRest* | "−" *product 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

   Follow(*product*) = { "+" , "−" , "==" , "!=" , ">=" , "<=" , ">" , "<" , "$" , ")" }.

5. Die Regel

   product $\rightarrow$ *factor 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

   Follow(*productRest*) = { "+" , "−" , "==" , "!=" , ">=" , "<=" , ">" , "<" , "$" , ")" }.

6. Die Regeln

   productRest $\rightarrow$ "*" *factor productRest* | "/" *factor 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

   Follow(*factor*) = { "*" , "/" , "+" , "−" , "==" , "!=" , ">=" , "<=" , ">" , "<" , "$" , ")" }.

□

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

$$a \rightarrow Y_1 Y_2 \cdots Y_k$$

eine Grammatik-Regel. Dann bestehen zwischen dem Prädikat `nullable`() und den beiden Funktionen *First*() und *Follow*() die folgenden Beziehungen:

1. $\forall t \in T : \neg$ *nullable*(t).
2. $k = 0 \Rightarrow$ *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 LL(1)-Grammatiken

Wir können nun die Frage beantworten, für welche Grammatiken ein Top-Down-Parser erzeugt werden kann, der immer mit einem Token Lookahead auskommt.

Definition 27 (LL(1)-Grammatik) Eine Grammatik G ist eine *LL(1)-Grammatik* genau dann, wenn für jede syntaktische Variable a, für die es in der Grammatik G zwei verschiedene Regeln

$$a \to \alpha \quad \text{und} \quad a \to \beta$$

gibt, die folgenden Bedingungen erfüllt sind:

1. $\neg(\alpha \Rightarrow^* \varepsilon \wedge \beta \Rightarrow^* \varepsilon)$.
   Die Rümpfe zweier verschiedener Regeln derselben Variablen dürfen nicht beide das leere Wort ableiten.
2. $\mathit{First}(\alpha) \cap \mathit{First}(\beta) = \{\}$.
   Die Ableitungen der Rümpfe zweier verschiedener Regeln derselben Variablen dürfen nicht mit demselben Token beginnen.
3. $(\beta \Rightarrow^* \varepsilon) \to \mathit{First}(\alpha) \cap \mathit{Follow}(a) = \{\}$.
   Wenn β den leeren String ableitet, dann müssen die Mengen *First*(α) und *Follow*(a) disjunkt sein. ◇

Wir diskutieren nun die Idee, die hinter der obigen Definition steht.

1. Falls das leere Wort sowohl über die Regel
   $$a \to \alpha \quad \text{als auch über} \quad a \to \beta$$
   ableitbar wäre, so wissen wir nicht, welche Regel wir anwenden sollen, wenn wir ein a ableiten sollen und das nächste Eingabe-Token ein Element der Menge *Follow*(a) ist.

2. Um ein *a* zu parsen und zwischen den beiden Regeln für *a* unterscheiden zu können, verwenden wir das folgende Rezept:

   Parsen wir ein *a* und ist das Lookahead-Token ein Element der Menge *First*(α), so verwenden wir die Regel

   $$a \to \alpha.$$

   Analog verwenden wir die Regel

   $$a \to \beta,$$

   wenn das Lookahead-Token ein Element der Menge *First*(β) ist.

   Dieses Rezept funktioniert natürlich nur, wenn die Mengen *First*(α) und *First*(β) disjunkt sind.

3. Das obige Rezept um ein *a* zu parsen muss in dem Fall, dass β das leere Wort ableitet, wie folgt erweitert werden.

   Gilt $\beta \Rightarrow^* \varepsilon$ und ist das Lookahead-Token ein Element der Menge *Follow*(*a*), so verwenden wir die Regel

   $$a \to \beta.$$

   Damit diese Regel nicht im Widerspruch zu den unter Punkt 2. genannten Regeln steht, benötigen wir die Bedingungen

   $$(\beta \Rightarrow^* \varepsilon) \to \mathit{First}(\alpha) \cap \mathit{Follow}(a) = \{\}.$$

Insgesamt versuchen wir also dann mit einer Regel $a \to \alpha$ zu reduzieren, wenn eine der beiden folgenden Bedingungen erfüllt ist. In diesen Bedingungen bezeichnet *lat* das Lookahead-Token.

1. $\mathit{lat} \in \mathit{First}(\alpha)$ oder
2. $\alpha \Rightarrow^* \varepsilon$ und $\mathit{lat} \in \mathit{Follow}(\alpha)$.

Bemerkung: Falls eine Grammatik G links-rekursiv ist, dann kann G keine LL(1)-Grammatik sein.

9.3.1 Berechnung der Parse-Tabelle

Nach diesen Vorbereitungen können wir nun zu einer LL(1)-Grammatik die *Parse-Tabelle* berechnen. Für eine Grammatik $G = \langle V, T, R, S \rangle$ ist die Parse-Tabelle

$$\mathit{parseTable} : V \times T \to 2^R,$$

eine Funktion, so dass der Aufruf *parseTable*(a, t) einer syntaktischen Variable *a* und einem Token t die Menge aller Regeln der Form

$$a \to \alpha$$

zuordnet, die bei einer Ableitung von *a* in Frage kommen, wenn das nächste zu lesenden Token den Wert t hat. Diese Funktion genügt den folgenden beiden Bedingungen:

1. Ist $a \to \alpha$ eine Regel der Grammatik und ist t ein Token aus der Menge *First*(α), dann ist diese Regel ein Element der Menge *parseTable*(a, t):

   $$(a \to \alpha) \in R \;\wedge\; t \in \mathit{First}(\alpha) \;\Rightarrow\; (a \to \alpha) \in \mathit{parseTable}(a, t).$$

2. Ist $a \to \alpha$ eine Regel der Grammatik, wobei α ε-erzeugend ist, und ist t ein Token aus der Menge *Follow*(*a*), dann ist diese Regel ein Element der Menge *parseTable*(a, t):

   $$(a \to \alpha) \in R \;\wedge\; \alpha \Rightarrow^* \varepsilon \;\wedge\; t \in \mathit{Follow}(a) \;\Rightarrow\; (a \to \alpha) \in \mathit{parseTable}(a, t).$$

Eine Grammatik ist genau dann eine $LL(1)$-Grammatik, wenn die Menge *parseTable*(a, t) für jede syntaktische Variable *a* und jedes Token t maximal eine Regel enthält:

$$G \text{ ist LL(1)} \quad \text{g.d.w.} \quad \forall a \in V : \forall t \in T : \mathit{card}\big(\mathit{parseTable}(a, t)\big) \leq 1.$$

Falls die Menge *parseTable*(a, t) leer ist, so heißt dies einfach, dass wir beim Parsen von *a* nicht auf das Token t stoßen können. Parsen wir also ein *a* und sehen als erstes Zeichen das Token t, so muss ein Syntax-Fehler vorliegen.

9.4 LL(k)-Grammatiken

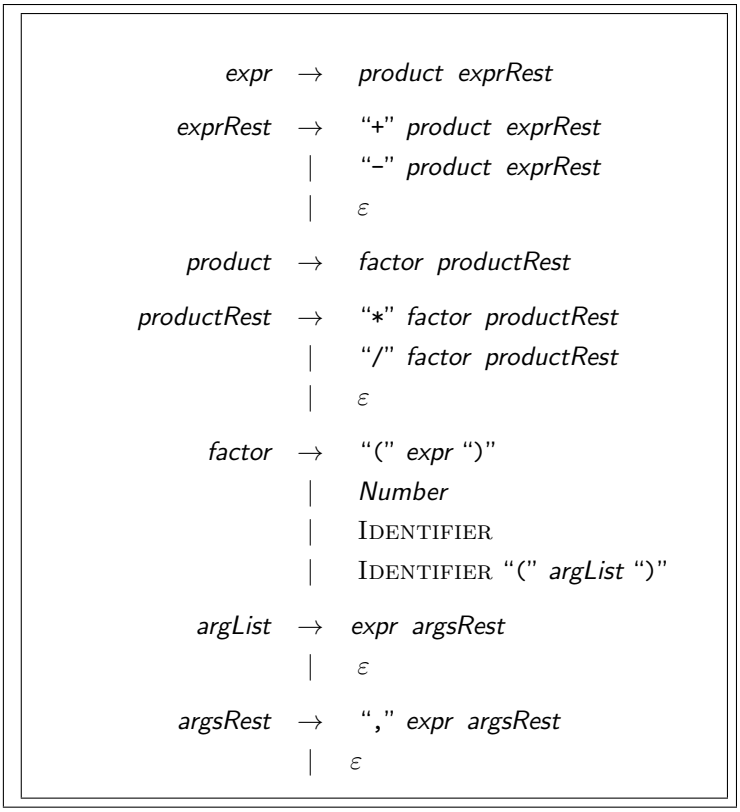


Figure 9.2: Arithmetische Ausdrücke mit Funktions-Aufrufen.

Viele interessante Grammatiken sind keine $LL(1)$-Grammatiken. Abbildung 9.2 zeigt ein Beispiel. Bei dieser Grammatik werden für arithmetische Ausdrücke auch Funktionsaufrufe der Form

$$f(a_1, \cdots, a_n)$$

zugelassen. Dabei ist f ein Funktionszeichen, das syntaktisch nicht von einem Identifier zu unterscheiden ist. Dadurch gibt es zwischen den beiden Regeln

factor → IDENTIFIER und *factor* → IDENTIFIER "(" *argList* ")"

einen Konflikt: Soll ein *factor* geparst werden und ist das nächste zu lesende Zeichen ein IDENTIFIER, so ist nicht klar, welche der beiden Regeln angewendet werden soll. Es gibt hier zwei mögliche Lösungen: Einerseits könnten wir die Grammatik durch eine Links-Faktorisierung umschreiben. Andererseits ist es auch möglich, das Problem dadurch zu lösen, dass wir bei der Entscheidung, welche der Grammatik-Regeln verwendet werden soll, zusätzlich das zweite Zeichen berücksichtigen: Handelt es sich dabei um das Zeichen "(" , so ist offenbar die Regel

factor → IDENTIFIER "(" *argList* ")"

heranzuziehen, andernfalls muss die Regel

factor → IDENTIFIER

verwendet werden.

Im allgemeinen Fall kann das Verfahren so erweitert werden, dass k Token bei der Entscheidung, welche Regel zu verwenden ist, als Lookahead herangezogen werden. Wir skizzieren die Grundzüge dieser Theorie. Als erstes verallgemeinern wir die Definition der Funktion *First*().

Definition 28 (First(k, α)) Es sei $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik. Wir definieren eine Funktion

$$\mathit{First} : \mathbb{N} \times (V \cup T)^* \to 2^{T^*},$$

so dass $\mathit{First}(k, \alpha)$ für eine natürliche Zahl k und einen String α, der aus Terminalen und syntaktischen Variablen besteht, die Menge der Token-Strings berechnet, die höchstens die Länge k haben und die Präfix eines von α abgeleiteten Strings sind. Formal lautet die Definition:

$$\mathit{First}(k, \alpha) := \big\{x \in T^* \mid \exists y \in T^* : \alpha \Rightarrow^* xy \wedge |x| = k\big\} \cup \big\{x \in T^* \mid \alpha \Rightarrow^* x \wedge |x| < k\big\}. \quad \diamond$$

Beispiel: Abbildung 9.3 zeigt eine Grammatik für arithmetische Ausdrücke. Berechnen wir für die in dieser Grammatik auftretenden Variablen v die Mengen $\mathit{First}(2, v)$, so erhalten wir beispielsweise:

1. $\mathit{First}(2, \mathit{expr}) = \{$ ID, "(" ID, "((", ID"+", ID"-", ID"*", ID"/" $\}$,
2. $\mathit{First}(2, \mathit{product}) = \{$ ID, "(" ID, "((", ID"*", ID"/" $\}$,
3. $\mathit{First}(2, \mathit{factor}) = \{$ ID, "(" ID, "((" $\}$.

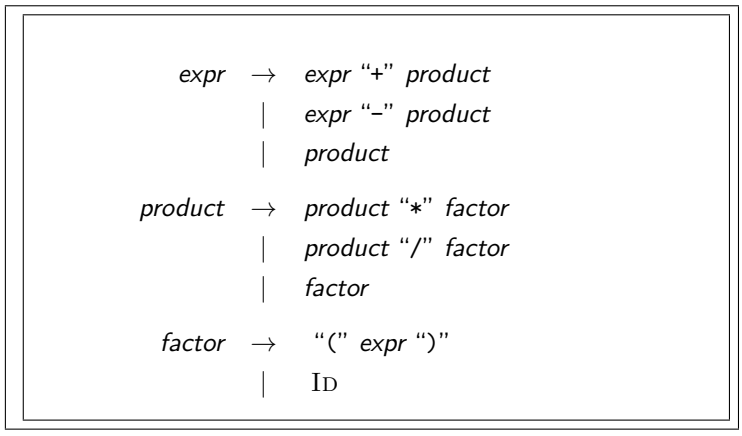


Figure 9.3: Eine Beispiel-Grammatik für arithmetische Ausdrücke.

Definition 29 (Follow$(k, \mathbf{a})$) Es sei $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik. Wir definieren eine Funktion

$$\mathit{Follow} : \mathbb{N} \times V \to 2^{T^*},$$

so dass $\mathit{Follow}(k, a)$ für eine natürliche Zahl k und eine syntaktische Variable a die Menge der Token-Strings berechnet, die höchstens die Länge k haben und in einer Ableitung, die vom Start-Symbol S ausgeht, auf a folgen können. Formal lautet die Definition:

$$\mathit{Follow}(k, a) := \big\{x \in T^* \mid \exists \alpha, \gamma \in (T \cup V)^* : S \Rightarrow^* \alpha a \gamma \wedge x \in \mathit{First}(k, \gamma)\big\}. \quad \diamond$$

Beispiel: Setzen wir das letzte Beispiel sinngemäß fort, so erhalten wir:

1. $\mathit{Follow}(2, \mathit{expr}) = \{ \varepsilon,$ ")", "))", ")*", ")+", ")-", ")/", "+"ID, "-"ID, "+(", "-(" $\}$,
2. $\mathit{Follow}(2, \mathit{product}) = \{ \varepsilon,$ ")", "))", ")+", ")-", ")*", ")/", "+"ID, "-"ID, "*"ID, "/"ID, "+(", "-(", "*(", "/(" $\}$,
3. $\mathit{Follow}(2, \mathit{factor}) = \{ \varepsilon,$ ")", "))", ")+", ")-", ")*", ")/", "+"ID, "-"ID, "*"ID, "/"ID, "+(", "-(", "*(", "/(" $\}$.

Definition 30 (Starke $LL(k)$-Grammatik) Eine kontextfreie Grammatik $G = \langle V, T, R, S \rangle$ ist eine *starke* $LL(k)$-*Grammatik* genau dann, wenn für je zwei verschiedene Grammatik-Regeln

$$a \to \beta \quad \text{und} \quad a \to \gamma$$

aus der Menge R die Bedingung

$$\forall \sigma, \tau \in \mathit{Follow}(k, a) : \mathit{First}(k, \beta\sigma) \cap \mathit{First}(k, \gamma\tau) = \{\}$$

erfüllt ist. ◇

Erklärung: Um die obige Definition zu verstehen, nehmen wir an, wir wollten ein a parsen. Wenn wir einen $LL(k)$-Parser bauen wollen, dürfen wir die nächsten k Symbole der Eingabe lesen und müssen entscheiden, welche der Regeln von a in Frage kommen. Diese k Symbole können das Resultat der Ableitung von β oder γ sein. Wenn die von β oder γ abgeleiteten Strings kürzer als k sind, so kann es sich aber auch schon um Token handeln, die in einer Ableitung auf a folgen, die also Elemente der Menge $\mathit{Follow}(k, a)$ sind. Für eine Regel

$$a \to \beta$$

und einen String $\sigma \in \mathit{Follow}(k, a)$ enthält die Menge $\mathit{First}(k, \beta\sigma)$ alle die Strings der Länge $\leq k$, die in einer Ableitung von a, welche die Regel $a \to \beta$ benutzt, folgen können. Sind diese Mengen für verschiedene Regeln disjunkt, so lässt sich anhand der k folgenden Token entscheiden, welche der Regeln angewendet werden muss.

Bemerkung: In der theoretischen Informatik gibt es neben dem Begriff der *starken* $LL(k)$-Grammatik auch noch den Begriff der (einfachen) $LL(k)$-Grammatik. Bei einer solchen $LL(k)$-Grammatik dürfen bei der Auswahl der Regel nicht nur die nächsten k Eingabe-Token berücksichtigt werden, sondern zusätzlich kann der Parser alle bisher gelesenen Token mit zu Rate ziehen. Dadurch kann in bestimmten Fällen zu gegebener Variable und gegebenem Lookahead auch dann noch eine Regel ausgewählt werden, wenn das Kriterium der starken $LL(k)$-Grammatik nicht erfüllt ist. Da dieser Begriff wesentlich komplexer ist als der Begriff der starken $LL(k)$-Grammatik, verzichten wir auf eine formale Darstellung des allgemeineren Begriffs. Die dem allgemeineren Begriff zu Grunde liegende Theorie ist sehr ausführlich in [AU72] dargestellt.

9.4.1 Berechnung von First() und Follow()

In diesem Abschnitt zeigen wir, wir die Funktionen $\mathit{First}(k, \alpha)$ und $\mathit{Follow}(k, a)$ berechnet werden können. Dazu benötigen wir verschiedene Hilfsfunktionen, die wir vorab definieren.

1. Die Funktion $\mathit{prefix}(k, w)$ berechnet für eine natürliche Zahl k und einen String w den Präfix von w mit der Länge k. Ist die Länge von w kleiner oder gleich k, so wird w zurück gegeben:

   $$\mathit{prefix}(k, w) = \begin{cases} w[1\,{:}\,k] & \text{falls } k < |w|; \\ w & \text{sonst.} \end{cases}$$

   Hier bezeichnet die Notation $w[1\,{:}\,k]$ den Teilstring von w, der aus den ersten k Buchstaben von w besteht.

2. Der Operator $+_k$ verkettet zwei Strings und bildet anschließend das Päfix der Länge k:

   $$v +_k w = \mathit{prefix}(k, vw).$$

   Hier bezeichnet vw die Verkettung der Strings v und w.

3. Die Definition des Operators $+_k$ wird auf Mengen von Strings verallgemeinert:

   $$M +_k N := \{v +_k w \mid v \in M \wedge w \in N\}.$$

Beispiel: Wir haben

$$\begin{aligned} &\{\varepsilon,\ \text{“}\texttt{A}\text{”}\ ,\ \text{“}\texttt{AB}\text{”}\ ,\ \text{“}\texttt{ABC}\text{”}\ \} +_2 \{\varepsilon,\ \text{“}\texttt{X}\text{”}\ ,\ \text{“}\texttt{YX}\text{”}\ \} \\ = &\{\varepsilon,\ \text{“}\texttt{A}\text{”}\ ,\ \text{“}\texttt{AB}\text{”}\ ,\ \text{“}\texttt{X}\text{”}\ ,\ \text{“}\texttt{YX}\text{”}\ ,\ \text{“}\texttt{AX}\text{”}\ ,\ \text{“}\texttt{AY}\text{”}\ \}. \end{aligned}$$ □

Die Berechnung von $\mathit{First}(k, \alpha)$ für $\alpha \in (V \cup T)^*$ wird auf die Berechnung von $\mathit{First}(k, X)$ mit $X \in V \cup T$ zurück geführt, denn es gilt

$$\mathit{First}(k, X_1 X_2 \cdots X_n) = \mathit{First}(k, X_1) +_k \mathit{First}(k, X_2) +_k \cdots +_k \mathit{First}(k, X_n).$$

Für ein Terminal $t \in T$ gilt offenbar

$$\mathit{First}(k, t) = \{t\}.$$

Die Berechnung der Menge $\mathit{First}(k, a)$ für eine syntaktische Variable $a \in V$ erfolgt iterativ über folgenden Fixpunkt-Algorithmus:

1. Zunächst werden alle Mengen $\mathit{First}(k, a)$ mit der leeren Menge initialisiert:

   $$\mathit{First}(k, a) := \{\}.$$

2. Anschließend wird für jede Grammatik-Regel der Form

   $$a \to \beta$$

   die Menge $\mathit{First}(k, a)$ wie folgt erweitert:

   $$\mathit{First}(k, a) := \mathit{First}(k, a) \cup \mathit{First}(k, \beta).$$

3. Der zweite Schritt wird in einer Schleife solange durchgeführt, bis sich keine der Mengen $\mathit{First}(k, a)$ mehr durch die Hinzunahme von $\mathit{First}(k, \beta)$ ändert.

Sind die Mengen $\mathit{First}(k, a)$ berechnet, so können wir anschließend die Mengen $\mathit{Follow}(k, a)$ für alle syntaktischen Variablen berechnen. Auch die Berechnung der Mengen $\mathit{Follow}(k, a)$ ist iterativ. Sie erfolgt nach dem folgenden Schema:

1. Zunächst werden alle Mengen $\mathit{Follow}(k, a)$ mit der leeren Menge initialisiert:

   $$\mathit{Follow}(k, a) = \{\}.$$

   Anschließend setzen wir für das Start-Symbol S der Grammatik

   $$\mathit{Follow}(k, S) = \{\$\}.$$

   Hier steht "\$" für das Ende der Eingabe. Die Idee ist, dass hinter dem Start-Symbol keine weitere Eingabe mehr kommen kann. Beachten Sie, dass in diesem Fall der String "\$" nicht aus k Zeichen besteht, sondern nur aus einem Zeichen.

2. Für jede Grammatik-Regel der Form

   $$a \to Y_1 Y_2 \cdots Y_l,$$

   für die Y_l eine syntaktische Variable ist, erweitern wir die Menge $\mathit{Follow}(k, Y_l)$ wie folgt:

   $$\mathit{Follow}(k, Y_l) := \mathit{Follow}(k, Y_l) \cup \mathit{Follow}(k, a),$$

   denn alles, was auf ein a folgen kann, kann auch auf ein Y_l folgen.

3. Für jede Grammatik-Regel der Form

   $$a \to Y_1 Y_2 \cdots Y_i Y_{i+1} \cdots Y_l,$$

   und jeden Index $i \in \{1, \cdots, l-1\}$, für den Y_i eine syntaktische Variable ist, erweitern wir die Menge $\mathit{Follow}(k, Y_i)$ wie folgt:

   $$\mathit{Follow}(k, Y_i) := \mathit{Follow}(k, Y_i) \cup \big(\mathit{First}(k, Y_{i+1} \cdots Y_l) +_k \mathit{Follow}(k, a)\big).$$

   Der Grund, warum wir hier noch die Menge $\mathit{Follow}(k, a)$ anhängen ist der, dass die Strings aus der Menge $\mathit{First}(k, Y_{i+1} \cdots Y_l)$ eventuell kürzer als k sind. In diesem Fall müssen noch die Präfixe von $\mathit{Follow}(k, a)$ angehängt werden.

   Bemerkung: Beachten Sie, dass der zweite Schritt ein Spezialfall des dritten Schritts ist, denn wenn wir im dritten Schritt $i := l$ setzen, dann ist der String $Y_{i+1} \cdots Y_l$ leer und somit enthält die Menge $\mathit{First}(k, Y_{i+1} \cdots Y_l)$ dann nur den leeren String ε, so dass der Ausdruck

   $$\big(\mathit{First}(k, Y_{i+1} \cdots Y_l) +_k \mathit{Follow}(k, a)\big) \quad \text{zu} \quad \mathit{Follow}(k, a)\big)$$

   vereinfacht werden kann. Bei der Implementierung werden wir daher nur den dritten Schritt umsetzen.

4. Der zweite und der dritte Schritt werden in einer Schleife solange durchgeführt, bis sich keine der Mengen $\mathit{Follow}(k, a)$ mehr ändert.

9.4.2 Implementation in SetlX

Figure 9.4 on page 122 shows an implementation of the function `First` in SetlX. We proceed to discuss the implementation line by line.

```
1  computeFirst := procedure(k, rules, variables) {
2      first := initializeMap(variables);
3      change := true;
4      while (change) {
5          change := false;
6          for ([a, body] in rules) {
7              new := firstList(k, body, first);
8              if (!(new <= first[a])) {
9                  change := true;
10                 first[a] += new;
11             }
12         }
13     }
14     return first;
15 };
```

Figure 9.4: The function `computeFirst`.

1. The first parameter k is the number of lookahead tokens, while the second parameter `rules` is the set of all grammar rules. Here, a rule of the form

   $$a \to \beta$$

   is represented in SetlX as a pair of the form

   `[ a, `β` ]`.

   Finally. the parameter `variables` is the set of all syntactical variables.

2. The function `computeFirst`$(k, \texttt{rules}, \texttt{variables})$ is supposed to compute $\mathit{First}(k, a)$ for all syntactical variables a. This is done by creating a binary relation `first` in line 2. At the end of the computation, the relation `first` will contain a pair $[a, \mathit{First}(k, a)]$ for every syntactical variable a. Therefore, the variable `first` codes the function *First*.

   Since the function `first` is computed via a fixpoint iteration, `first`$[a]$ is initialized to the empty set for all syntactical variables a. This is done by the function `initializeMap` in line 2. The function `initializeMap` is shown in Figure 9.7 in line 1.

3. The computation of `First` is done in the `while`-loop that extends from line 4 to line 13. This `while`-loop is controlled by the Boolean flag `change`. This variable is set to false at the beginning of the loop in line 5. If we ever find a variable `a` in line 6 such that `first[a]` gets incremented in line 10, then the variable `changed` is changed to `true` so that the iteration can keep going. On the other hand, if we don't find any new strings that have to be added to `first[a]` for any variable `a`, then we have successfully computed the function *First* for all variables and the fixpoint iteration can be stopped.

4. Given a rule of the form

   $$a \to \beta$$

   we know that

   $$\mathit{First}(k, \beta) \subseteq \mathit{First}(a)$$

and therefore *First*(k, β) has to be added to *First*(a). In our implementation, *First*(k, β) is computed by the function `firstList` that is shown in Figure 9.5 on page 123.

```
1  firstList := procedure(k, alpha, first) {
2      match (alpha) {
3          case []:
4               return { [] };
5          case [ Var(v) | r ]:
6               firstV := first[v];
7               firstR := firstList(k, r, first);
8               return unionK(firstV, firstR, k);
9          case [ Token(t) | r ]:
10              firstR := firstList(k, r, first);
11              return unionK({ [t] }, firstR, k);
12     }
13 };
```

Figure 9.5: The implementation of `firstList`.

Figure 9.5 shows the implementation of the function `firstList`.

1. The first parameter k is the number of lookahead tokens, the second parameter `alpha` is a list of variables and terminals, and the last parameter `first` is a binary relation coding the function *First*(k, a) for all syntactical variables *a*.

2. If `alpha` has the form $[Y_1, \cdots, Y_n]$, then the formula to compute *First*$(k, \texttt{alpha})$ is

   $$\mathit{First}(k, [Y_1, \cdots, Y_n]) = \mathit{First}(k, Y_1) +_k \mathit{First}(k, Y_2) +_k \cdots +_k \mathit{First}(k, Y_n).$$

   This sum is computed recursively. The operator $+_k$ is implemented via the function `unionK` that is shown in Figure 9.7.

Figure 9.6 on page 124 shows the implementation of the function `Follow`.

1. The parameter k is the number of lookahead tokens, `rules` is the set of grammar rules, `s` is the start symbol of the grammar, `first` is a binary relation representing the function *First*, and `variables` is the set of all syntactical variables of the grammar.

2. Like the function `First`, the function `Follow` is also implemented as a binary relation. This relation is stored in the variable `follow`. At the beginning of the computation, for all syntactical variables *a* the set `follow`[*a*] is initialized as an empty set via the function `initializeMap`. Additionally, the end-of-file symbol "$" is added into the follow set of the start symbol `s`.

3. If we have a grammar rule of the form

   $$a \to Y_1 Y_2 \cdots Y_i Y_{i+1} \cdots Y_l,$$

   and if, furthermore, Y_i is a syntactical variable, then we have to extend the set $\texttt{follow}(Y_i)$ as follows:

   $$\texttt{follow}(Y_i) \texttt{ += } \texttt{First}(k, [Y_{i+1}, \cdots, Y_l)]) +_k \texttt{follow}(a).$$

   This rule is implemented in line 11 and 12. The expression $\texttt{First}(k, [Y_{i+1}, \cdots, Y_l)])$ is computed via the function `firstList`.

4. The variable `change` controls the fixpoint iteration: The `while`-loop keeps going as long as there is a syntactical variable *a* such that `follow`[*a*] has changed.

```
computeFollow := procedure(k, rules, s, first, variables) {
    follow := initializeMap(variables);
    follow[s] := { [ "\$" ] };
    change := true;
    while (change) {
        change := false;
        for ([a, body] in rules) {
            for (i in [1 .. #body]) {
                match (body[i]) {
                    case Var(yi):
                        tail := firstList(k, body[i+1 ..], first);
                        new  := unionK(tail, follow[a], k);
                        if (!(new <= follow[yi])) {
                            change := true;
                            follow[yi] += new;
                        }
                }
            }
        }
    }
    return follow;
};
```

Figure 9.6: The function `computeFollow`.

```
initializeMap := procedure(variables) {
    return { [a, {}] : a in variables };
};
prefixK := procedure(s, k) {
    if (#s <= k) {
        return s;
    }
    return s[1..k];
};
addK := procedure(u, v, k) {
    return prefixK(u + v, k);
};
unionK := procedure(s, t, k) {
    return { addK(u, v, k) : u in s, v in t };
};
```

Figure 9.7: Some auxiliary functions.

Finally, Figure 9.7 on page 124 shows the implementation of some auxiliary functions.

1. The function `initializeMap` takes a set of variables and creates a relation that assigns the empty set to all of these variables.
2. The function `prefixK` takes a list s and computes the prefix of s that has a length of k. If the length of s is at most k, then s is returned unchanged.
3. The function $\texttt{addK}(u, v, k)$ computes $u +_k v$ for two lists u and v.

4. The function `unionK`(s, t, k) computes the set $s +_k t$ for two sets of lists s and t.

Chapter 10

Interpreter

```
1  grammar Pure;
2
3  program  : statement+
4           ;
5  statement: VAR ':=' expr ';'
6           | VAR ':=' 'read' '(' ')' ';'
7           | 'print' '(' expr ')' ';'
8           | 'if'    '(' boolExpr ')' '{' statement* '}'
9           | 'while' '(' boolExpr ')' '{' statement* '}'
10          ;
11 boolExpr : expr '==' expr
12          | expr '<'  expr
13          ;
14 expr     : expr '+' product
15          | expr '-' product
16          | product
17          ;
18 product  : product '*' factor
19          | product '/' factor
20          | factor
21          ;
22 factor   : '(' 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\v\n\r]   -> skip;
```

Figure 10.1: ANTLR-Grammatik für eine einfache Programmier-Sprache.

In diesem Kapitels erstellen wir mit Hilfe des Parser-Generators ANTLR einen Interpreter für eine einfache Programmiersprache. Erfreulicherweise akzeptiert ANTLR seit der Version `4.0` auch Grammatiken, die einfache Links-Rekursion enthalten. Auch eine Links-Faktorisierung der Grammatik ist nicht mehr notwendig. Abbildung 10.1 zeigt die ANTLR-Grammatik der Programmier-Sprache, für die wir in diesem Abschnitt einen Interpreter entwickeln. Die

Befehle dieser Sprache sind Zuweisungen, Print-Befehle, `if`-Abfragen, sowie `while`-Schleifen. Abbildung 10.2 zeigt ein Beispiel-Programm, das dieser Grammatik entspricht. Dieses Programm liest zunächst eine Zahl ein, die in der Variablen `n` gespeichert wird. Anschließend wird die Summe

$$\sum_{i=1}^{n^2} i$$

in der Variablen `s` akkumuliert und am Ende des Programms ausgegeben.

```
1  n := read();
2  s := 0;
3  i := 0;
4  while (i < n * n) {
5      i := i + 1;
6      s := s + i;
7  }
8  print(s);
```

Figure 10.2: Ein Programm zur Berechnung der Summe $\sum_{i=0}^{n^2} i$.

Um einen Interpreter für diese Sprache entwickeln zu können, benötigen wir zunächst Klassen, mit denen wir die einzelnen Befehle darstellen können. Wir beginnen mit der abstrakten Klasse `Statement`. Diese Klasse ist in Abbildung 10.3 gezeigt und dient dazu, Anweisungen unserer Programmier-Sprache darzustellen. Wir werden von der Klasse `Statement` später die Klassen `Assignment`, `Print`, `IfThen` und `While` ableiten. Die Klasse `Statement` ist selber abstrakt und enthält im Wesentlichen die Deklaration der abstrakten Methode `execute()`. Diese Methode können wir später benutzen, um einen Befehle ausführen. Zusätzlich speichern wir hier das Flag `isInteractive` als statische Variable. Mit diesem Flag steuern wir, ob der Interpreter interaktiv in einer Kommandozeile betrieben wird, oder ob im Batch-Modus eine Datei abgearbeitet werden soll. Außerdem haben wir in der Klasse `Statement` noch die statische Methode `prompt()`. Diese wird nur dann benutzt, wenn der Interpreter interaktiv von der Kommandozeile aus betrieben wird. In diesem Fall gibt die Methode den folgenden Prompt aus:

```
SL>
```

Dieser Prompt signalisiert dem Benutzer, dass der Interpreter auf eine Eingabe wartet.

```
1  public abstract class Statement {
2      static boolean isInteractive = false;
3
4      public abstract void execute();
5
6      static void prompt() {
7          if (isInteractive) {
8              System.out.print("SL> ");
9              System.out.flush();
10         }
11     }
12 }
```

Figure 10.3: Die abstrakte Klasse `Statement`

```
1  public class Assignment extends Statement {
2      Variable mLhs;
3      Expr     mRhs;
4
5      public Assignment(Variable lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public void execute() {
10         Expr.sValueTable.put(mLhs.mName, mRhs.eval());
11     }
12 }
```

Figure 10.4: Die Klasse `Assignment`.

Abbildung 10.4 zeigt die Implementierung der Klasse `Assignment`. Da diese Klasse eine Zuweisung der Form

var := *expr*

darstellt, bei der einer Variablen *var* der Wert eines arithmetischen Ausdrucks *expr* zugewiesen wird, hat diese Klasse zwei Member-Variablen um die Variable und den Ausdruck zu speichern.

1. Die erste Member-Variable ist `mLhs`. Diese Member-Variable entspricht der Variablen auf der linken Seite des Zuweisungs-Operators ":=".
2. Die zweite Member-Variable ist `mRhs`. Hier wird der arithmetische Ausdruck, der auf der rechten Seite des Zuweisungs-Operators steht, kodiert. Diese Member-Variable hat den Typ `Expr`. Hierbei handelt es sich um eine abstrakte Klasse zur Darstellung arithmetischer Ausdrücke, von der wir später konkrete Klassen ableiten. Diese Klasse besitzt eine abstrakte Methode `eval`(), mit der ein arithmetischer Ausdruck ausgewertet werden kann.

In der Klasse `Assignment` wertet die Methode `execute`() den Ausdruck, der in der Variablen `mRhs` gespeichert wird, mit Hilfe der für Objekte der Klasse `Expr` zur Verfügung stehenden Methode `eval`() aus und speichert den erhaltenen Wert in der Hashtabelle `sValueTable` unter dem Namen der Variablen ab. Es handelt sich bei dieser Tabelle um eine sogenannte *Symboltabelle*, in der die Werte der einzelnen Variablen abgelegt werden. Die Tabelle ist als statische Variable in der Klasse `Expr` definiert. Diese Klasse wird in Abbildung 10.8 auf Seite 130 gezeigt.

Abbildung 10.5 zeigt die Implementierung der Klasse `Read`. Diese Klasse stellt eine Zuweisung der Form

var `= read();`

dar. Daher besitzt diese Klasse eine Member-Variable `mVar`, in welcher das Ergebnis der Lese-Operation abgespeichert wird. Die Methode `execute`() gibt zunächst den String "> " als Eingabe-Aufforderung aus. Anschließend wird ein Objekt der in *Java* vordefinierten Klasse `Scanner` erzeugt. Diese Klasse stellt die Methode `nextDouble`() zur Verfügung, mit deren Hilfe eine Fließkomma-Zahl eingelesen werden kann. Diese wird dann in der Symboltabelle unter dem Namen der Variablen, der auf der linken Seite der Zuweisung steht, abgespeichert.

Von den übrigen Klassen zur Darstellung von Befehlen diskutieren wir noch die Klasse `While`, die in Abbildung 10.6 gezeigt wird. Diese Klasse stellt einen Befehl der Form

`while` (b) { *stmnts* }

dar, wobei b ein Boole'scher Ausdruck ist, während *stmnts* eine Liste von Befehlen ist. Der Boole'sche Ausdruck wird in der Member-Variablen `mCond` gespeichert, die Liste von Befehlen findet sich in der Member-Variablen `mStmntList`. Zur Auswertung eines solchen Befehls führen wir solange alle Befehle in der Liste `mStmntList` aus, wie die Auswertung des Boole'schen Ausdrucks b den Wert `true` ergibt.

```
1  public class Read extends Statement {
2      Variable mLhs;
3
4      public Read(Variable lhs) {
5          mLhs = lhs;
6      }
7      public void execute() {
8          System.out.print("> "); // write prompt
9          System.out.flush();
10         Scanner scanner = new Scanner(System.in);
11         Double  value   = scanner.nextDouble();
12         Expr.sValueTable.put(mLhs.mName, value);
13     }
14 }
```

Figure 10.5: Die Klasse `Read`.

```
1  import java.util.*;
2
3  public class While extends Statement {
4      BoolExpr        mCond;
5      List<Statement> mStmntList;
6
7      public While(BoolExpr cond, List<Statement> stmntList) {
8          mCond      = cond;
9          mStmntList = stmntList;
10     }
11     public void execute() {
12         while (mCond.eval()) {
13             for (Statement stmnt: mStmntList) {
14                 stmnt.execute();
15             }
16         }
17     }
18 }
```

Figure 10.6: Die Klasse `While`.

Die abstrakte Klasse `BoolExpr` dient zur Darstellung Boole'scher Ausdrücke. In unserem Fall sind das Ausdrücke der Form

$$l == r \quad \text{und} \quad l < r,$$

wobei Gleichungen durch die Klasse `Equal` dargestellt werden, während Ungleichungen durch die Klasse `LessThan` dargestelt werden, die beide von der Klasse `BoolExpr` abgeleitet sind. Abbildung 10.7 zeigt die Klassen `BoolExpr` und `Equal`. Die Klasse `BoolExpr` hat die beiden Member-Variablen `mLhs` und `mRhs` und repräsentiert die Gleichung

`mLhs == mRhs`.

Um diese Gleichung auszuwerten, werden rekursiv die linke und die rechte Seite der Gleichung, die in `mLhs` und `mRhs` gespeichert sind, ausgewertet. Anschließend wird das Ergebnis dieser Auswertung zurück gegeben. Die Klasse `LessThan` ist analog zur Klasse `Equal` aufgebaut und wird daher nicht gezeigt.

```
1  public abstract class BoolExpr {
2      public abstract Boolean eval();
3  }
4  public class Equal extends BoolExpr {
5      Expr mLhs;
6      Expr mRhs;
7
8      public Equal(Expr lhs, Expr rhs) {
9          mLhs = lhs;
10         mRhs = rhs;
11     }
12     public Boolean eval() {
13         return mLhs.eval() == mRhs.eval();
14     }
15 }
```

Figure 10.7: Die Klassen `BoolExpr` und `Equal`

```
1  import java.util.*;
2
3  public abstract class Expr {
4      public static Map<String, Double> sValueTable = new HashMap<String, Double>();
5
6      public abstract Double eval();
7  }
```

Figure 10.8: Die abstrakte Klasse `Expr`.

Schließlich haben wir noch die Klassen, die zur Repräsentation von arithmetischen Ausdrücken benötigt werden. Diese Klassen werden alle von der abstrakten Klasse `Expr` abgeleitet, die in Abbildung 10.8 gezeigt ist.

1. Die Klasse `Expr` definiert die statische Variable `sValueTable`. Diese Variable beinhaltet eine Hashtabelle, in der für jede Variable, der ein Wert zugewiesen wurde, der aktuelle Wert dieser Variablen gespeichert ist.

2. Weiter deklariert die Klasse die abstrakte Methode `eval()`, mit der ein Ausdruck ausgewertet werden kann.

Von der Klasse `Expr` werden die Klassen `Sum`, `Difference`, `Product`, `Quotient`, `MyNumber` und `Variable` abgeleitet, die wir jetzt der Reihe nach diskutieren. Abbildung 10.9 zeigt die Klasse `Sum`. Da diese Klasse eine Summe der Form

$$l + r$$

darstellt, hat diese Klasse zwei Member-Variablen `mLhs` und `mRhs` um die beiden Summanden l und r darzustellen. Die Methode `eval` wertet diese beiden Member-Variablen getrennt aus und addiert das Ergebnis. Die Klassen `Difference`, `Product` und `Quotient` sind analog zur Klasse `Sum` aufgebaut und werden daher nicht weiter diskutiert.

```
1  public class Sum extends Expr {
2      Expr mLhs;
3      Expr mRhs;
4
5      public Sum(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public Double eval() {
10         return mLhs.eval() + mRhs.eval();
11     }
12 }
```

Figure 10.9: Die Klasse `Sum`

```
1  public class Variable extends Expr {
2      String mName;
3
4      public Variable(String name) {
5          mName = name;
6      }
7      public Double eval() {
8          return sValueTable.get(mName);
9      }
10 }
```

Figure 10.10: Die Klasse `Variable`.

Abbildung 10.10 zeigt die Implementierung der Klasse `Variable`. Die Methode `eval()` wertet eine Variable dadurch aus, dass sie unter dem Namen der Variablen in der Hashtabelle `sValueTable` den zugeordneten Wert nachschlägt.

Abbildung 10.11 zeigt das Treiber-Programm, das den von ANTLR erzeugten Parser einbindet. Das Programm soll später wahlweise in der Form

`java SLInterpreter` *file*$_1$ $\cdots$ *file*$_n$

oder einfach als

`java SLInterpreter`

aufgerufen werden. Im ersten Fall bezeichnet *file*$_i$ für $i = 1, \cdots, n$ jeweils eine Datei, die ein auszuführendes Programm enthält. Im zweiten Fall, oder falls anstelle eines Dateinamens der String "–" als Argument übergeben wird, sollen die Befehle stattdessen interaktiv eingegeben werden. Die Methode *parseFile*() behandelt dabei den Fall, dass die Befehle aus einer Datei gelesen werden, während die Methode `parseInteractive()` für den Fall der interaktiven Verarbeitung zuständig ist. Die in der Methode `parseInteractive()` verwendete Klasse `InputReader` wird dazu benötigt, um von der Standardeingabe zu lesen und einen `ANTLRInputStream` zu erzeugen. Wir werden diese Klasse gleich genauer diskutieren. Sowohl die Methode `parseFile()` als auch die Methode `parseInteractive()` dienen beide nur dazu, ein Objekt der Klasse `ANTLRInputStream` zu erzeugen. Im ersten Fall wird dieses Objekt aus der zu lesenden Datei erzeugt, im zweiten Fall benutzen wir dazu den noch zu diskutierenden `InputReader`. In beiden Fällen wird schließlich die Methode `parseAndExecute()` aufgerufen, deren Aufgabe es ist, die einzelnen Befehle zu erkennen und auszuführen.

```
import org.antlr.v4.runtime.*;
import java.io.FileInputStream;
import java.io.InputStream;

import java.util.List;
import java.io.*;

public class SLInterpreter {
    public static void main(String[] args) throws Exception {
        for (String file: args) {
            if (!file.equals("-")) {
                parseFile(file);
            } else {
                parseInteractive();
            }
        }
        if (args.length == 0) {
            parseInteractive();
        }
    }
    private static void parseFile(String fileName) throws Exception {
        try {
            FileInputStream fis = new FileInputStream(fileName);
            parseAndExecute(fis);
        } catch (IOException e) {
            System.err.println("File " + fileName + " could not be read.");
        }
    }
    private static void parseInteractive() throws Exception {
        Statement.isInteractive = true;
        Statement.prompt();
        while (true) {
            InputStream stream = InputReader.getStream();
            parseAndExecute(stream);
        }
    }
    private static void parseAndExecute(InputStream stream)
        throws Exception
    {
        ANTLRInputStream  input  = new ANTLRInputStream(stream);
        SimpleLexer       lexer  = new SimpleLexer(input);
        CommonTokenStream ts     = new CommonTokenStream(lexer);
        SimpleParser      parser = new SimpleParser(ts);
        parser.program();
    }
}
```

Figure 10.11: Die Klasse `SLInterpreter`.

```
grammar Simple;

@header {
    import java.util.List;
    import java.util.ArrayList;
}

program
    : (s = statement { $s.stmnt.execute(); Statement.prompt(); })+
    ;
statement returns [Statement stmnt]
    @init {
        List<Statement> stmnts = new ArrayList<Statement>();
    }
    : v = VAR ':=' e = expr ';'
      { $stmnt = new Assignment(new Variable($v.text), $e.result); }
    | v = VAR ':=' 'read' '(' ')' ';'
      { $stmnt = new Read(new Variable($v.text)); }
    | 'print' '(' r = expr ')' ';'
      { $stmnt = new Print($r.result); }
    | 'if' '(' b = boolExpr ')' '{'
          (l = statement { stmnts.add($l.stmnt); })*
      '}'
      { $stmnt = new IfThen($b.result, stmnts); }
    | 'while' '(' b = boolExpr ')' '{'
          (l = statement { stmnts.add($l.stmnt); })*
      '}'
      { $stmnt = new While($b.result, stmnts); }
    ;
boolExpr returns [BoolExpr result]
    : l = expr '==' r = expr { $result = new Equal(   $l.result, $r.result); }
    | l = expr '<'  r = expr { $result = new LessThan($l.result, $r.result); }
    ;
expr returns [Expr result]
    : e = expr '+' p = product { $result = new Sum(       $e.result, $p.result); }
    | e = expr '-' p = product { $result = new Difference($e.result, $p.result); }
    | p = product              { $result = $p.result;                            }
    ;
product returns [Expr result]
    : p = product '*' f = factor { $result = new Product( $p.result, $f.result); }
    | p = product '/' f = factor { $result = new Quotient($p.result, $f.result); }
    | f = factor                 { $result = $f.result;                          }
    ;
factor returns [Expr result]
    : '(' expr ')' { $result = $expr.result;          }
    | v = VAR      { $result = new Variable($v.text); }
    | n = NUMBER   { $result = new MyNumber($n.text); }
    ;
```

Figure 10.12: ANTLR-Spezifikation der Grammatik.

Abbildung 10.12 zeigt die Implementierung des Parsers mit dem Werkzeug ANTLR. Der Parser liest in Zeile 8 eine nicht-leere Folge von Befehlen, die sofort nach dem Einlesen ausgeführt werden. Die übrigen Grammatik-Regeln erzeugen jeweils einen abstrakten Syntax-Baum der erkannten Eingabe. So liefert beispielsweise die Regel für die syntaktische Variable `statement` als Ergebnis ein Objekt der abstrakten Klasse `Statement` zurück. Wir diskutieren nur den Fall, dass es sich bei dem Statement um eine `while`-Schleife handelt, denn die anderen Fälle sind analog. Zunächst wird in Zeile 11 in der Variablen `stmnts` eine leere Liste von `Statements` angelegt. Diese Liste enthält später alle Befehle, die im Rumpf der `while`-Schleife stehen. Anschließend wird in Zeile 21 das Schlüsselwort "`while`" zusammen mit der Bedingung erkannt. Dann werden der Reihe nach alle Befehle, die sich im Rumpf der Schleife befinden, der Liste `stmnts` hinzugefügt. Nach dem Lesen der schließenden geschweiften Klammer wird als Rückgabewert ein Objekt der Klasse `While` erzeugt und zurückgegeben.

```
49  VAR    : [a-zA-Z][a-zA-Z_0-9]*;
50  NUMBER : '0'|[1-9][0-9]*;
51
52  MULTI_COMMENT : '/*' .*? '*/' -> channel(HIDDEN);
53  LINE_COMMENT  : '//' ~('\n')* -> channel(HIDDEN);
54  WS            : [ \t\v\n\r]   -> channel(HIDDEN);
```

Figure 10.13: ANTLR-Spezifikation der Token.

Die Spezifikation der Token ist in Abbildung 10.13 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. Beachten Sie, dass Whitespace und Kommentare an den `channel` mit den Namen `HIDDEN` weitergereicht werden. Dadurch werden sie vom Parser ignoriert, stehen aber noch für Fehlermeldungen zur Verfügung. 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.

Abbildung 10.14 zeigt die Klasse `InputReader`, die dann benutzt wird, wenn der Interpreter interaktiv betrieben wird. Hier müssen wir uns zunächst überlegen, woran der Interpreter erkennen soll, dass der Benutzer ein Kommando eingegeben hat. Ein Ansatz wäre, dass der Parser nach jedem Zeilenumbruch überprüft, ob der Benutzer ein vollständiges Kommando eingegeben hat. Dieser Ansatz scheitert aber daran, dass manche Kommandos sich über mehrere Zeilen erstrecken. Daher wartet der Interpreter darauf, dass nacheinander zwei Zeilenumbrüche eingegeben werden. Dann wird die bis dahin gelesene Eingabe in Zeile 23 in einem Feld von Bytes zusammengefasst und als Objekt der Klasse `ByteArrayInputStream` zurück gegeben.

Aufgabe 29:

(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 an. Dabei soll der Operator "!" am stärksten und der Operator "||" am schwächsten binden.

```
1  public final class InputReader {
2      private static BufferedReader br  = null;
3      private static String         EOL = "\n";
4
5      public static InputStream getStream() throws EOFException {
6          if (br == null) {
7              br = new BufferedReader(new InputStreamReader(System.in));
8          }
9          String input     = "";
10         String line      = null;
11         int    endlAdded = 0;
12         try {
13             while (true) {
14                 // line is read and returned without termination character(s)
15                 line = br.readLine();
16                 // add line termination (Unix style '\n' by default)
17                 input     += line + EOL;
18                 endlAdded += EOL.length();
19                 if (line == null) {
20                     throw new EOFException("EndOfFile");
21                 } else if (line.length() == 0 && input.length() > endlAdded) {
22                     byte[] byteArray =
23                     input.substring(0, input.length() - EOL.length()).getBytes();
24                     return new ByteArrayInputStream(byteArray);
25                 }
26             }
27         } catch (IOException e) {
28             e.printStackTrace();
29         }
30         return null;
31     }
32 }
```

Figure 10.14: Die Klasse `InputReader`.

(d) Erweitern Sie die Syntax der arithmetischen Ausdrücke so, dass auch vordefinierte mathematische Funktionen wie `exp()` oder `ln()` benutzt werden können.

Hinweis: Wenn Sie das Paket `java.lang.reflect` benutzen, kommen Sie mit einer zusätzlichen Klasse aus und können damit alle in `java.lang.Math` definierten Methoden implementieren.

(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 einer global definierten Hashtabelle zu verwalten. ◇

(f) Erweitern Sie den Interpreter so, dass die Verwendung rationaler Zahlen unterstützt wird. Das Ziel dieser Erweiterung ist eine Sprache, mit der Sie Rechnungen ohne Rundungsfehler durchführen können.

Chapter 11

Grenzen kontextfreier Sprachen

In diesem Kapitel diskutieren wir die Grenzen kontextfreier Sprachen und leiten dazu das sogenannte "*große Pumping-Lemma*" her, mit dessen Hilfe wir beispielsweise zeigen können, dass die Sprache L_{square}, die durch

$$L_{square} = \{ww \mid w \in \Sigma^*\}$$

definiert wird, für das Alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ keine kontextfreie Sprache ist. Bevor wir das Pumping-Lemma für kontextfreie Sprachen beweisen, zeigen wir, wie sich nutzlose Symbole aus einer Grammatik entfernen lassen.

11.1 Beseitigung nutzloser Symbole*

In diesem Abschnitt zeigen wir, wie wir nutzlose Symbole aus einer kontextfreien Grammatik entfernen können. Ist $G = \langle V, T, R, S\rangle$ eine kontextfreie Grammatik, so nennen wir eine syntaktische Variable $A \in V$ *nützlich*, wenn es Strings $w \in T^*$ und $\alpha, \beta \in (V \cup T)^*$ gibt, so dass

$$S \Rightarrow^* \alpha A \beta \Rightarrow^* w$$

gilt. Eine syntaktische Variable ist also genau dann nützlich, wenn diese Variable in der Herleitung eines Wortes $w \in L(G)$ verwendet werden kann. Analog heißt ein Terminal $t \in T$ *nützlich*, wenn es Wörter $w_1, w_2 \in T^*$ gibt, so dass

$$S \Rightarrow^* w_1 t w_2$$

gilt. Ein Terminal t ist also genau dann nützlich, wenn es in einem Wort $w \in L(G)$ auftritt. Variablen und Terminale, die nicht nützlich sind, bezeichnen wir als *nutzlose Symbole*.

Die Erkennung nutzloser Symbole ist eine Überprüfung, die in manchen Parser-Generatoren (beispielsweise in *Bison*[1]) eingebaut ist, weil das Auftreten nutzloser Symbole oft einen Hinweis darauf gibt, dass die Grammatik nicht die Sprache beschreibt, die intendiert ist. Insofern ist die jetzt vorgestellte Technik auch von praktischem Interesse. Wir beginnen mit zwei Definitionen.

Definition 31 (erzeugende Variable)
Eine syntaktische Variable $A \in V$ einer Grammatik $G = \langle V, T, R, S\rangle$ ist eine *erzeugende* Variable, wenn es ein Wort $w \in T^*$ gibt, so dass

$$A \Rightarrow^* w$$

gilt, aus einer erzeugende Variable lässt sich also immer mindestens ein Wort aus T^* herleiten. Die Notation $A \Rightarrow^* w$ drückt aus, dass das Wort w aus der Variablen A in endlich vielen Schritten abgeleitet werden kann. □

Offenbar ist eine syntaktische Variable, die nicht erzeugend ist, nutzlos. Die Menge aller erzeugenden Variablen einer Grammatik G kann mit Hilfe der folgenden induktiven Definition gefunden werden.

[1] *Bison* ist ein Parser-Generator für die Sprache C.

1. Enthält die Grammatik $G = \langle V, T, R, S\rangle$ eine Regel der Form

   $$A \to w \quad \text{mit } w \in T^*,$$

   so ist die Variable A offenbar erzeugend.

2. Enthält die Grammatik $G = \langle V, T, R, S\rangle$ eine Regel der Form

   $$A \to \alpha$$

   und sind alle syntaktischen Variablen, die in dem Wort α auftreten, bereits als erzeugende Variablen erkannt, so ist auch die syntaktische Variable A erzeugend.

Beispiel: Es sei $G = \langle \{S, A, B, C\}, \{\mathtt{x}\}, R, S\rangle$ und die Menge der Regeln R sei wie folgt gegeben:

$$\begin{array}{lcl} S & \to & ABC \mid A, \\ A & \to & AA \mid \mathtt{x}, \\ B & \to & AC, \\ C & \to & BA. \end{array}$$

Aufgrund der Regel

$$A \to \mathtt{x}$$

ist zunächst A erzeugend. Aufgrund der Regel

$$S \to A$$

ist dann auch S erzeugend. Die Variablen B und C sind hingegen nicht erzeugend und damit sicher nutzlos. □

Ist $G = \langle V, T, R, S\rangle$ eine Grammatik und ist E die Menge der erzeugenden Variablen, so können wir alle Variablen, die nicht erzeugend sind, einfach weglassen. Zusätzlich müssen wir natürlich auch die Regeln weglassen, in denen Variablen auftreten, die nicht erzeugend sind. Dabei ändert sich die von der Grammatik G erzeugte Sprache offenbar nicht.

Eine Variable kann gleichzeitig erzeugend und trotzdem nutzlos sein. Als einfaches Beispiel betrachten wir die Grammatik $G = \langle \{S, A, B\}, \{\mathtt{x}, \mathtt{y}\}, R, S\rangle$, deren Regeln durch

$$\begin{array}{lcl} S & \to & A\mathtt{y}, \\ A & \to & AA \mid \mathtt{x} \quad \text{und} \\ B & \to & A\mathtt{y} \end{array}$$

gegeben sind. Die erzeugenden Variable sind in diesem Falle A, B und S. Die Variable B ist trotzdem nutzlos, denn von S aus ist diese Variable gar nicht *erreichbar*. Formal definieren wir für eine Grammatik $G = \langle V, T, R, S\rangle$ eine syntaktische Variable X als *erreichbar*, wenn es Wörter $\alpha, \beta \in (V \cup T)^*$ gibt, so dass

$$S \Rightarrow^* \alpha X \beta$$

gilt. Für eine gegebene Grammatik G lässt sich die Menge der Variablen, die erreichbar sind, mit dem folgenden induktiven Algorithmus berechnen.

1. Das Start-Symbol S ist erreichbar.

2. Enthält die Grammatik G eine Regel der Form

   $$X \to \alpha$$

   und ist X erreichbar, so sind auch alle Variablen, die in α auftreten, erreichbar.

Offenbar sind Variablen, die nicht erreichbar sind, nutzlos und wir können diese Variablen, sowie alle Regeln, in denen diese Variablen auftreten, weglassen, ohne dass sich dabei die Sprache ändert. Damit haben wir jetzt ein Verfahren, um aus einer Grammatik alle nutzlosen Variablen zu entfernen.

1. Zunächst entfernen wir alle Variablen, die nicht erzeugend sind.

2. Anschließend entfernen wir alle Variablen, die nicht erreichbar sind.

Es ist wichtig zu verstehen, dass die Reihenfolge der obigen Regeln nicht umgedreht werden darf. Dazu betrachten wir die Grammatik $G = \langle \{S, A, B, C\}, \{\mathtt{x}, \mathtt{y}\}, R, S \rangle$, deren Regeln durch

$$\begin{array}{lcll} S & \to & BC \mid A, & \\ A & \to & AA \mid \mathtt{x}, & \\ B & \to & \mathtt{y} & \text{und} \\ C & \to & CC & \end{array}$$

gegeben sind. Die beiden Regeln

$$S \to BC \quad \text{und} \quad S \to A$$

zeigen, dass alle Variablen erreichbar sind. Weiter sehen wir, dass die Variablen A, B und S erzeugend sind, denn es gilt

$$A \Rightarrow^* \mathtt{x}, \quad S \Rightarrow^* \mathtt{x} \quad \text{und} \quad B \Rightarrow^* \mathtt{y}.$$

Damit sieht es zunächst so aus, als ob nur C nutzlos ist. Das stimmt aber nicht, auch die Variable B ist nutzlos, denn wenn wir die Variable C aus der Grammatik entfernen, dann wird auch die Regel

$$S \to BC$$

entfernt und damit ist dann B nicht mehr erreichbar und somit ebenfalls nutzlos.

Bemerkung: An dieser Stelle können wir uns fragen, warum es funktioniert, wenn wir erst alle Variablen entfernen, die nicht erzeugend sind und anschließend dann die nicht erreichbaren Variablen entfernen. Der Grund ist, dass eine Variable B, die nicht erreichbar ist, niemals von einer anderen Variablen A, die erreichbar ist, benötigt wird um ein Wort $w \in T^*$ abzuleiten, denn wenn die Variable B in der Ableitung

$$A \Rightarrow^* w$$

auftritt, dann folgt aus der Erreichbarkeit von A auch die Erreichbarkeit von B. Wenn also eine Variable A gleichzeitig erreichbar und erzeugend ist und wir andererseits eine Variable B als nicht erreichbar erkannt haben, dann kann B getrost entfernt werden, denn das kann an der Tatsache, dass A erzeugend ist, nichts ändern. $\diamond$

Bemerkung: Der nachfolgende Beweis des Pumping-Lemmas ist logisch von der Beseitigung der nutzlosen Symbole unabhängig. Das war in einer früheren Version dieses Skriptes anders, denn bei dem damals verwendeten Beweis war es notwendig, die Grammatik vorher in *Chomsky-Normal-Form* zu transformieren. Die Beseitigung der nutzlosen Symbole ist ein Teil dieser Transformation. Der gleich folgende Beweis setzt nicht mehr voraus, dass die Grammatik in Chomsky-Normal-Form vorliegt. Da die Technik der Beseitigung nutzloser Symbole aber auch einen praktischen Wert hat, habe ich diesen Abschnitt trotzdem beibehalten.

11.2 Parse-Bäume als Listen

Zum Beweis des Pumping-Lemmas für kontextfreie Sprachen benötigen wir eine Abschätzung, bei der wir die Länge eines Wortes w aus einer kontextfreien Sprache $L(G)$ mit der Höhe des Parse-Baums für W in Verbindung bringen. Es zeigt sich, dass es zum Nachweis dieser Abschätzung hilfreich ist, wenn wir einen Parse-Baum als Liste aller Pfade des Parse-Baums auffassen. Die Idee wird am ehesten anhand eines Beispiels klar. Abbildung 11.2 auf Seite 139 zeigt einen Parse-Baum für den String "2*3+4", der mit Hilfe der in Abbildung 11.1 gezeigten Grammatik erstellt wurde.

Fassen wir diesen Parse-Baum als Liste seiner Zweige auf, wobei jeder Zweig eine Liste von Grammatik-Symbolen ist, so erhalten wir die folgende Liste:

$$
\begin{array}{rcl}
\mathit{expr} & \to & \mathit{expr}\ \text{“+”}\ \mathit{product} \\
& | & \mathit{expr}\ \text{“−”}\ \mathit{product} \\
& | & \mathit{product} \\
\mathit{product} & \to & \mathit{product}\ \text{“*”}\ \mathit{factor} \\
& | & \mathit{product}\ \text{“/”}\ \mathit{factor} \\
& | & \mathit{factor} \\
\mathit{factor} & \to & \text{“(”}\ \mathit{expr}\ \text{“)”} \\
& | & \text{“2”} \mid \text{“3”} \mid \text{“4”}
\end{array}
$$

Figure 11.1: Grammatik für arithmetische Ausdrücke.

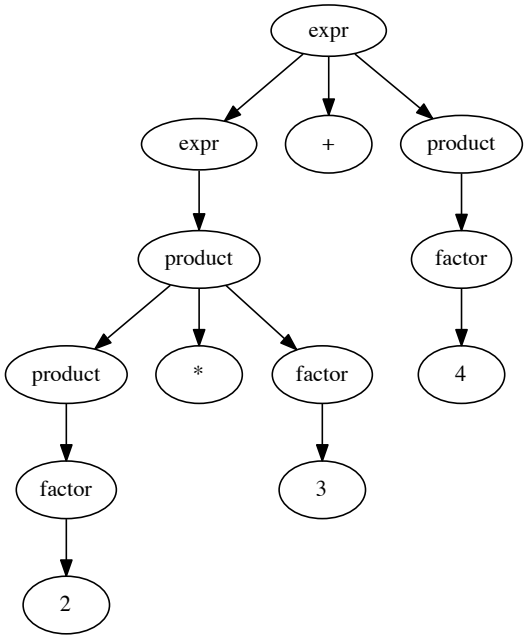


Figure 11.2: Ein Parse-Baum für den String “2*3+4”.

[[*expr*, *expr*, *product*, *product*, *factor*, “2”],
[*expr*, *expr*, *product*, “*”],
[*expr*, *expr*, *product*, *factor*, “3”],
[*expr*, “+”],
[*expr*, *product*, *factor*, “4”]
]. ◇

Die nun folgende Definition der Funktion *parseTree*() formalisiert die Berechnung des Parse-Baums.

Definition 32 (parseTree)
Ist $G = \langle V, T, R, S\rangle$ eine kontextfreie Grammatik, ist $w \in T^*$, $A \in V$ und gibt es eine Ableitung

$$A \Rightarrow^* w,$$

so definieren wir *parseTree*$(A \Rightarrow^* w)$ induktiv als Liste von Listen:

1. Falls $(A \to t_1 t_2 \cdots t_m) \in R$ mit $A \in V$ und $t_1 \cdots t_m = w$ ist, so setzen wir
   $$\textit{parseTree}(A \Rightarrow^* w) = \big[\, [A, t_1],\, [A, t_2],\, \cdots,\, [A, t_m] \,\big].$$
2. Falls $(A \to \varepsilon) \in R$ mit $A \in V$ und $w = \varepsilon$ ist, so setzen wir
   $$\textit{parseTree}(A \Rightarrow^* w) = \big[\, [A, \varepsilon] \,\big].$$
3. Falls $A \Rightarrow^* w$ gilt, weil
   $$(A \to B_1 B_2 \cdots B_m) \in R, \quad B_i \Rightarrow^* w_i \text{ für alle } i = 1, \cdots, m \quad \text{und } w = w_1 w_2 \cdots w_m$$
   gilt, so definieren wir (unter Benutzung der SetlX-Notation für die Definition von Listen)
   $$\textit{parseTree}(A \Rightarrow^* w) = \big[\, [A] + l \,:\, i \in [1, \cdots, m],\, l \in \textit{parseTree}(B_i \Rightarrow^* w_i) \,\big].$$
   Damit diese Definition auch tatsächlich alle Fälle abdeckt, müssen wir noch den Fall diskutieren, dass eines der Symbole B_i ein Terminal ist: In diesem Fall setzen wir
   $$\textit{parseTree}(B_i \Rightarrow^* B_i) := \big[[B_i]\big].$$

Wir definieren die *Breite* b einer Grammatik als die größte Anzahl von Symbolen, die auf der rechten Seite einer Grammatik-Regel der Form

$$A \to \alpha$$

auftreten.

Lemma 33 (Beschränktheits-Lemma) Die Grammatik $G = \langle V, T, R, S\rangle$ habe die Breite b. Ferner gelte

$$A \Rightarrow^* w$$

für eine syntaktische Variable $A \in V$ und ein Wort $w \in T^*$. Falls n die Länge der längsten Liste in

$$\textit{parseTree}(A \Rightarrow^* w)$$

ist, so gilt für die Länge des Wortes w die Abschätzung

$$|w| \leq b^{n-1}.$$

Beweis: Wir führen den Beweis durch Induktion nach der Länge n der längsten Liste in *parseTree*$(A \Rightarrow^* w)$.

I.A. $n = 2$: Wenn alle Listen in *parseTree*$(A \Rightarrow^* w)$ nur zwei Elemente haben, dann besteht die Ableitung aus genau einem Schritt und daher muss es eine Regel der Form

$$A \to t_1 t_2 \cdots t_m \quad \text{mit } w = t_1 t_2 \cdots t_m \text{ und } t_i \in T \text{ für alle } i = 1, \cdots, m$$

in der Grammatik G geben. Es gilt dann

$$\textit{parseTree}(A \Rightarrow^* w) = \big[\, [A, t_1],\, [A, t_2],\, \cdots,\, [A, t_m] \,\big].$$

Daraus folgt

$$|w| = |t_1 t_2 \cdots t_m| = m \leq b = b^1 = b^{2-1},$$

wobei die Ungleichung $m \leq b$ aus der Tatsache folgt, dass die Länge der Regeln der Grammatik G durch die Breite b beschränkt ist.

I.S. $n \mapsto n+1$: Da die Ableitung nun aus mehr als einem Schritt besteht, hat die Ableitung die Form

$$A \Rightarrow B_1 B_2 \cdots B_m \Rightarrow^* w_1 w_2 \cdots w_m = w.$$

Außerdem haben dann die Listen in

$$\mathit{parseTree}(B_i \Rightarrow^* w_i)$$

für alle $i = 1, \cdots, m$ höchstens die Länge n. Nach Induktions-Voraussetzung wissen wir also, dass

$$|w_i| \leq b^{n-1} \quad \text{für alle } i = 1, \cdots, m$$

gilt. Daher haben wir

$$|w| = |w_1| + \cdots + |w_m| \leq b^{n-1} + \cdots + b^{n-1} = m \cdot b^{n-1} \leq b \cdot b^{n-1} = b^n = b^{(n+1)-1}. \qquad \square$$

11.3 Das Pumping-Lemma für kontextfreie Sprachen

Satz 34 (Pumping-Lemma) Es sei L eine kontextfreie Sprache. Dann gibt es ein $n \in \mathbb{N}$, so dass jeder String $s \in L$, dessen Länge größer oder gleich n ist, in der Form

$$s = uvwxy$$

geschrieben werden kann, so dass außerdem folgendes gilt:

1. $|vwx| \leq n$,
   der mittlere Teil des Strings hat folglich eine Länge von höchstens n Buchstaben.
2. $vx \neq \varepsilon$,
   die Teilstrings v und x können also nicht beide gleichzeitig leer sein.
3. $\forall h \in \mathbb{N}_0 : uv^h wx^h y \in L$.
   die Strings v und x können beliebig oft repliziert (*gepumpt*) werden.

Beweis: Da L eine kontextfreie Sprache ist, gibt es eine kontextfreie Grammatik $G = \langle V, T, R, S \rangle$, so dass

$$L = L(G)$$

gilt. Wir nehmen an, dass die Grammatik G insgesamt k syntaktische Variablen enthält und außerdem die Breite b hat. Wir definieren

$$n := b^{k+1}.$$

Sei nun $s \in L$ mit $|s| \geq n$. Wir wählen einen Parse-Baum τ aus, der einerseits den String s ableitet und der andererseits unter allen Parse-Bäumen, die den String s aus S ableiten, die minimale Anzahl von Knoten hat. Für diesen Parse-Baum τ betrachten wir die Listen aus

$$\tau = \mathit{parseTree}(S \Rightarrow^* s).$$

Falls alle Listen hier eine Länge kleiner-gleich $k+1$ hätten, so würde aus den Beschränktheits-Lemma 33 folgen, dass

$$|s| \leq b^{(k+1)-1} = b^k < b^{k+1} = n, \quad \text{also} \quad |s| < n$$

gilt, im Widerspruch zu der Voraussetzung $|s| \geq n$. Also muss es in τ eine Liste geben, die mindestens die Länge $k+2$ hat. Wir wählen die längste Liste unter den Listen in τ aus. Diese Liste hat dann die Form

$$[A_1, \cdots, A_l, t] \quad \text{mit } A_i \in V \text{ für alle } i \in \{1, \cdots, l\}, \quad t \in T, \quad \text{sowie } l \geq k+1.$$

Wegen $l \geq k+1$ können nicht alle Variablen A_1, $\cdots$, A_l voneinander verschieden sein, denn es gibt ja nur insgesamt k verschiedene syntaktische Variablen. Wir finden daher in der Menge $\{l, l-1, \cdots, l-k\}$ zwei verschiedene Indizes

i und j, so dass die Variablen A_i und A_j gleich sind und definieren $A := A_i = A_j$. Von den beiden Indizes i und j bezeichne i den kleineren Index, es gelte also $i < j$. Die Ableitung von s aus S hat dann die folgende Form

$$S \Rightarrow^* uA_iy \Rightarrow^* uvA_jxy \Rightarrow^* uvwxy = s.$$

Insbesondere gilt also

$$S \Rightarrow^* uAy, \quad A \Rightarrow^* vAx, \quad \text{und} \quad A \Rightarrow^* w.$$

Damit haben wir aber folgendes:

1. $S \Rightarrow^* uAy \Rightarrow^* uwy$, also
$$S \Rightarrow^* uv^0wx^0y.$$
2. $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvvAxxy \Rightarrow^* uvvwxxy$, also
$$S \Rightarrow^* uv^2wx^2y.$$
3. $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uv^2Ax^2y \Rightarrow^* uv^3Ax^3y \Rightarrow^* \cdots \Rightarrow^* uv^hAx^hy \Rightarrow^* uv^hwx^hy$, also
$$uv^hwx^hy \in L \quad \text{für beliebige } h \in \mathbb{N}_0.$$

Wir müssen jetzt noch zeigen, dass $vx \neq \varepsilon$ gilt. Die Ableitung

$$A \Rightarrow^* vAx$$

ist eigentlich die Ableitung

$$A_i \Rightarrow^* vA_jx$$

und enthält daher mindestens einen Ableitungsschritt. Wir führen den Nachweis der Behauptung $vx \neq \varepsilon$ indirekt und nehmen $v = x = \varepsilon$ an. Dann würde wegen $A_i = A_j$ also

$$A_i \Rightarrow^+ A_i$$

gelten, wobei das Zeichen $^+$ an dem Pfeil $\Rightarrow$ anzeigt, dass diese Ableitung mindestens einen Schritt enthält. In diesem Fall wäre aber der Parse-Baum τ nicht minimal, denn wir könnten die Ableitungs-Schritte, die A_i in A_i überführen, einfach weglassen. Damit ist die Annahme $vx = \varepsilon$ widerlegt und es muss $vx \neq \varepsilon$ gelten.

Als letztes zeigen wir, dass die Ungleichung $|vwx| \leq n$ gilt. Wir haben

$$A = A_i \Rightarrow^* vA_jx \Rightarrow^* vwx.$$

Weil einerseits $i \in \{l-k, l-(k-1), \cdots, l\}$ gilt und andererseits die der Ableitung $A \Rightarrow^* vwx$ zugeordnete Liste aus *parseTree*$(S \Rightarrow^* w)$ von maximaler Länge war, wissen wir, dass die Länge der längsten Liste in

$$\textit{parseTree}(A \Rightarrow^* vwx)$$

kleiner-gleich $k+2$ ist. Nach dem Beschränktheits-Lemma 33 folgt damit für die Länge von vwx die Abschätzung

$$|vwx| \leq b^{(k+2)-1} = b^{k+1} = n. \qquad \square$$

Bemerkung: Das Pumping-Lemma wird in der deutschsprachigen Literatur gelegentlich als "*Schleifensatz*" bezeichnet. Es wurde 1961 von Bar-Hillel, Perles und Shamir [BHPS61] bewiesen.

11.4 Anwendungen des Pumping-Lemmas

Wir zeigen, wie mit Hilfe des Pumping-Lemmas der Nachweis erbracht werden kann, dass bestimmte Sprachen nicht kontextfrei sind.

11.4.1 Die Sprache $L = \{\mathtt{a}^k\mathtt{b}^k\mathtt{c}^k \mid k \in \mathbb{N}\}$ ist nicht kontextfrei

Wir weisen nun nach, dass die Sprache

$$L = \{\mathtt{a}^k\mathtt{b}^k\mathtt{c}^k \mid k \in \mathbb{N}\}$$

nicht kontextfrei ist. Wir führen diesen Nachweis indirekt und nehmen zunächst an, dass L kontextfrei ist. Nach dem Pumping-Lemma gibt es dann eine natürliche Zahl n, so dass jeder String $s \in L$, dessen Länge größer oder gleich n ist, sich in Teilstrings der Form

$$s = uvwxy$$

zerlegen lässt, so dass außerdem folgendes gilt:

1. $|vwx| \leq n$,
2. $vx \neq \varepsilon$,
3. $\forall i \in \mathbb{N}_0 : uv^iwx^iy \in L$.

   Insbesondere können wir hier $i = 0$ wählen und erhalten dann $uwy \in L$.

Wir definieren nun den String s wie folgt:

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

Dieser String hat die Länge $3 \cdot n \geq n$ und erfüllt also die Voraussetzung über die Länge. Damit finden wir eine Zerlegung $s = uvwxy$ mit den obigen Eigenschaften. Da der Teilstring vwx eine Länge kleiner oder gleich n hat, können in diesem String nicht gleichzeitig die Buchstaben "a" und "c" vorkommen. Wir betrachten die nach dieser Erkenntnis noch möglichen Fälle getrennt.

1. Fall: In dem String vwx kommen nur die Buchstaben "a" und "b" vor, der Buchstabe "c" kommt nicht vor:

   $$\mathit{count}(vwx, \mathtt{c}) = 0.$$

   Da $vx \neq \varepsilon$ ist, folgt

   $$\mathit{count}(vx, \mathtt{a}) + \mathit{count}(vx, \mathtt{b}) > 0.$$

   Wir nehmen zunächst an, dass $\mathit{count}(vx, \mathtt{a}) > 0$ gilt, der Fall $\mathit{count}(vx, \mathtt{b}) > 0$ ist analog zu behandeln. Dann erhalten wir einerseits

   $$\begin{array}{rcl} \mathit{count}(uwy, \mathtt{c}) & = & \mathit{count}(s, \mathtt{c}) - \mathit{count}(vx, \mathtt{c}) \\ & = & \mathit{count}(s, \mathtt{c}) - 0 \\ & = & \mathit{count}(\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n, \mathtt{c}) \\ & = & n. \end{array}$$

   Zählen wir nun die Häufigkeit, mit welcher der Buchstabe "a" in dem String uwy auftritt, so erhalten wir

   $$\begin{array}{rcl} \mathit{count}(uwy, \mathtt{a}) & = & \mathit{count}(s, \mathtt{a}) - \mathit{count}(vx, \mathtt{a}) \\ & < & \mathit{count}(s, \mathtt{a}) \\ & = & \mathit{count}(\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n, \mathtt{a}) \\ & = & n. \end{array}$$

   Damit haben wir dann aber

   $$\mathit{count}(uwy, \mathtt{a}) < \mathit{count}(uwy, \mathtt{c})$$

   und daraus folgt $uwy \notin L$, was im Widerspruch zum Pumping-Lemma steht.

2. Fall: In dem String vwx kommt der Buchstabe "a" nicht vor.

   Dieser Fall lässt sich analog zum ersten Fall behandeln. □

Aufgabe 30: Zeigen Sie, dass die Sprache

$$L = \left\{ \mathsf{a}^{k^2} \;\middle|\; k \in \mathbb{N} \right\}$$

nicht kontextfrei ist.

Hinweis: Argumentieren Sie über die Länge der betrachteten Strings. ◇

Lösung: Wir führen den Beweis indirekt und nehmen an, dass die Sprache L_{square} kontextfrei wäre. Nach dem Pumping-Lemma für kontextfreie Sprachen gibt es dann eine positive natürliche Zahl n, so dass sich jeder String $s \in L_{\text{square}}$ mit $|s| \geq n$ in fünf Teilstrings u, v, w, x, und y aufspalten lässt, so dass gilt:

1. $s = uvwxy$,
2. $|vwx| \leq n$,
3. $vx \neq \varepsilon$,
4. $\forall h \in \mathbb{N}_0 : uv^h wx^h y \in L_{\text{square}}$.

Wir betrachten nun den String $s = a^{n^2}$. Für die Länge dieses Strings gilt offenbar

$$|s| = \left|a^{n^2}\right| = n^2 \geq n.$$

Also gibt es eine Aufspaltung von s der Form $s = uvwxy$ mit den oben angegebenen Eigenschaften. Da a der einzige Buchstabe ist, der in s vorkommt, können in den Teilstrings u, v, w, x und y auch keine anderen Buchstaben vorkommen und es muss natürliche Zahlen c, d, e, f, und g geben, so dass

$$u = a^c,\ v = a^d,\ w = a^e,\ x = a^f \text{ und } y = a^g$$

gilt. Wir untersuchen, welche Konzequenzen sich daraus für die Zahlen c, d, e, f, g ergeben.

1. Die Zerlegung $s = uvwxy$ schreibt sich als $a^{n^2} = a^c a^d a^e a^f a^g$ und daraus folgt
$$n^2 = c + d + e + f + g. \tag{11.1}$$
2. Aus der Ungleichung $|vwx| \leq n$ folgt
$$d + e + f \leq n. \tag{11.2}$$
3. Die Bedingung $vx \neq \varepsilon$ liefert
$$d + f > 0. \tag{11.3}$$
4. Aus der Formel $\forall h \in \mathbb{N}_0 : uv^h wx^h y \in L_{\text{square}}$ folgt
$$\forall h \in \mathbb{N}_0 : a^c a^{d \cdot h} a^e a^{f \cdot h} a^g \in L_{\text{square}}. \tag{11.4}$$

Die letzte Gleichung muss insbesondere auch für den Wert $h = 2$ gelten:

$$a^c a^{2 \cdot d} a^e a^{2 \cdot f} a^g \in L_{\text{square}}.$$

Nach Definition der Sprache L_{square} gibt es dann eine natürliche Zahl k, so dass gilt

$$c + 2 \cdot d + e + 2 \cdot f + g = k^2. \tag{11.5}$$

Addieren wir in Gleichung (11.1) auf beiden Seiten $d + f$, so erhalten wir insgesamt

$$n^2 + d + f = c + 2 \cdot d + e + 2 \cdot f + g = k^2.$$

Wegen $d + f > 0$ folgt daraus

$$n < k. \tag{11.6}$$

Andererseits haben wir

$$\begin{array}{rcll} k^2 & = & c + 2 \cdot d + e + 2 \cdot f + g & \text{nach Gleichung (11.5)} \\ & = & (c + d + e + f + g) + (d + f) & \text{elementare Umformung} \\ & \leq & (c + d + e + f + g) + (d + e + f) & \text{denn } e \geq 0 \\ & \leq & (c + d + e + f + g) + n & \text{nach Ungleichung (11.2)} \\ & = & n^2 + n & \text{nach Gleichung (11.1)} \\ & < & n^2 + 2 \cdot n + 1 & \text{da } n + 1 > 0 \\ & = & (n + 1)^2 & \text{elementare Umformung} \end{array}$$

Damit haben wir insgesamt $k^2 < (n + 1)^2$ gezeigt und das impliziert

$$k < n + 1. \tag{11.7}$$

Zusammen mit Ungleichung (11.6) liefert Ungleichung (11.7) nun die Ungleichungs-Kette

$$n < k < n + 1.$$

Da andererseits k eine natürliche Zahl sein muss und n ebenfalls eine natürliche Zahl ist, haben wir jetzt einen Widerspruch, denn zwischen n und $n + 1$ gibt es keine natürliche Zahl. □

Aufgabe 31: Das Alphabet Σ sei durch die Festlegung $\Sigma := \{\mathsf{a}, \mathsf{b}\}$ definiert. Zeigen Sie, dass die Sprache

$$L = \left\{ tt \;\middle|\; t \in \Sigma^* \right\}$$

nicht kontextfrei ist. ◇

Lösung: Wir führen den Beweis indirekt und nehmen an, dass die Sprache L kontextfrei wäre. Nach dem Pumping-Lemma für kontextfreie Sprachen gibt es dann eine positive natürliche Zahl n, so dass sich jeder String $s \in L$ mit $|s| \geq n$ in fünf Teilstrings u, v, w, x, und y aufspalten lässt, so dass gilt:

1. $s = uvwxy$,
2. $|vwx| \leq n$,
3. $vx \neq \varepsilon$,
4. $\forall h \in \mathbb{N}_0 : uv^h wx^h y \in L$.

Wir betrachten nun den String $s := a^n b^n a^n b^n$. Definieren wir $t := a^n b^n$, so ist klar, dass $s = tt$ ist und damit gilt $s \in L$. Für die Länge von s gilt

$$|s| = \left|a^n b^n a^n b^n\right| = 4 \cdot n \geq n.$$

Also gibt es nach dem Pumping-Lemma eine Aufspaltung von s der Form $s = uvwxy$ mit den oben angegebenen Eigenschaften. Im weiteren führen wir eine Fallunterscheidung nach der Lage des Strings vwx innerhalb des gesamten Strings $s = a^n b^n a^n b^n$ durch. Dabei berücksichtigen wir, dass $|vwx| \leq n$ gilt. Zur Vereinfachung der Darstellung des Beweises vereinbaren wir für zwei Strings r und s die Schreibweise

$$r \sqsubseteq s \quad \text{g.d.w.} \quad r \text{ ist Teilstring von } s.$$

Wollen wir zusätzlich die Position einschränken, an der r innerhalb von s auftritt, so schreiben wir

$$r \sqsubseteq_{i,j} s \quad \text{g.d.w.} \quad r \sqsubseteq s[i..j].$$

Die Schreibweise $r \sqsubseteq_{i,j} s$ drückt also aus, dass der Teilstring r ab der Position i in dem String s auftritt und dass das Ende r nicht über die Position j hinausreicht.

Für die Lage des Teilstrings vwx innerhalb von $a^n b^n a^n b^n$ gibt es aufgrund der Längenbeschränkung $|vwx| \leq n$ nur drei Möglichkeiten:

1. Fall: $vwx \sqsubseteq_{1,2 \cdot n} a^n b^n a^n b^n$,

   der Teilstring vwx liegt also in der ersten Hälfte von s und ist folglich Teil von $a^n b^n$.

2. Fall: $vwx \sqsubseteq_{n+1,3 \cdot n} a^n b^n a^n b^n$,

   der Teilstring vwx liegt in der Mitte von s und ist Teil von $b^n a^n$.

3. Fall: $vwx \sqsubseteq_{2 \cdot n+1,4 \cdot n} a^n b^n a^n b^n$,

   der Teilstring vwx liegt in der zweiten Hälfte von s und ist folglich Teil von $a^n b^n$.

Wir setzen nun im Pumping-Lemma in der vierten Aussage für h den Wert 0 ein und folgern, dass der String

$$uv^0wx^0y = uwy$$

in der Sprache L liegt. Wir zeigen, dass dies zu einem Widerspruch führt und untersuchen dazu die obigen drei Fälle der Reihe nach.

1. Fall: $vwx \sqsubseteq_{1,2 \cdot n} a^n b^n a^n b^n$.

   Da vx in der ersten Hälfte von $s = a^n b^n a^n b^n$ liegt und der String uwy aus s dadurch entsteht, dass wir v und x weglassen, hat der String uwy die Form

   $$uwy = a^{n-k_1} b^{n-k_2} a^n b^n$$

   und wegen $vx \neq \varepsilon$ wissen wir, dass $k_1 + k_2 > 0$ ist. Die Mitte des Strings uwy liegt daher innerhalb des Teilstrings $a^n b^n$. Wenn wir also

   $$t't' = a^{n-k_1} b^{n-k_2} a^n b^n \quad \text{für ein } t' \in \Sigma^*$$

   hätten, so würde das erste Auftreten von t' in den Teilstring a^n hineinragen und müsste folglich mit einem a enden. Das funktioniert aber nicht, denn das zweite Auftreten von t' endet mit einem b. Also kann dieser Fall nicht eintreten.

2. Fall: $vwx \sqsubseteq_{n+1,3 \cdot n} a^n b^n a^n b^n$,

   Da vx nun innerhalb von $s = b^n a^n$ liegt und der String uwy aus s dadurch entsteht, dass wir v und x weglassen, hat der String uwy die Form

   $$uwy = a^n b^{n-k_1} a^{n-k_2} b^n$$

   Wenn $uwy \in L$ ist, müsste es also ein t' geben mit

   $$t't' = a^n b^{n-k_1} a^{n-k_2} b^n \quad \text{und } t' \in \Sigma^*.$$

   Wenn wir uns das erste Auftreten von t' in dieser Gleichung ansehen, stellen wir fest, dass t' mit dem String a^n beginnt. Betrachten wir das zweite Auftreten von t' , so sehen wir, dass t' mit dem String b^n endet. Damit hat dann aber t' mindestens die Länge $2 \cdot n$ und $t't' = uwy$ hätte mindestens die Länge $4 \cdot n$. Wegen $vx \neq \varepsilon$ ist dies nicht möglich und auch dieser Fall kann nicht eintreten.

3. Fall: $vwx \sqsubseteq_{2 \cdot n+1,4 \cdot n} a^n b^n a^n b^n$.

   Dieser Fall ist analog zum ersten Fall.

Da wir in jedem Fall einen Widerspruch erhalten haben, können wir schließen, dass die Sprache L nicht kontextfrei sein kann. □

Remark: This result shows that context-free languages are not closed under complementation, since we have already shown that the complement of L, which is the language

$$L^c = \{s \in \Sigma^* \mid \neg(\exists w \in \Sigma^* : s = ww)\},$$

is context-free. ◇

Aufgabe 32: Zeigen Sie, dass die Sprache

$$L = \left\{ \mathsf{a}^p \;\middle|\; p \in \mathbb{P} \right\}$$

nicht kontextfrei ist. Hier bezeichnet $\mathbb{P}$ die Menge der Primzahlen. ◇

Lösung: Wir führen den Beweis indirekt und nehmen an, L wäre kontextfrei. Nach dem Pumping-Lemma gibt es dann eine Zahl n, so dass es für alle Strings $s \in L$, deren Länge größer als n ist, eine Zerlegung

$$s = uvwxy$$

mit den folgenden drei Eigenschaften gibt:

1. $|vwx| \leq n$ und
2. $vx \neq \varepsilon$,
3. $\forall h \in \mathbb{N}_0 : uv^h wx^h y \in L$.

Wir wählen nun eine Primzahl p, die größer als $n+1$ ist und setzen $s = \mathsf{a}^p$. Wir wissen, dass eine solche Primzahl existieren muss, denn es gibt unendlich viele Primzahlen. Dann gilt $|s| = p > n$ und die Voraussetzung des Pumping-Lemmas ist erfüllt. Wir finden also eine Zerlegung von a^p der Form

$$\mathsf{a}^p = uvwxy$$

mit den oben angegebenen Eigenschaften. Aufgrund der Gleichung $s = uvwxy$ können die Teilstrings u, v, w, x und y nur aus dem Buchstaben "a" bestehen. Also gibt es natürliche Zahlen c, d, e, f und g so dass gilt:

$$u = \mathsf{a}^c, \quad v = \mathsf{a}^d, \quad w = \mathsf{a}^e, \quad x = \mathsf{a}^f \quad \text{und} \quad y = \mathsf{a}^g.$$

Für die Zahlen c, d, e, f und g gilt dann Folgendes:

1. $c + d + e + f + g = p$,
2. $d + f \neq 0$,
3. $d + e + f \leq n$,
4. $\forall h \in \mathbb{N}_0 : c + h \cdot d + e + h \cdot f + g \in \mathbb{P}$.

Setzen wir in der letzten Gleichung für h den Wert $(c + e + g)$ ein, so erhalten wir

$$c + (c + e + g) \cdot d + e + (c + e + g) \cdot f + g \in P.$$

Wegen

$$c + (c + e + g) \cdot d + e + (c + e + g) \cdot f + g = (c + e + g) \cdot (1 + d + f)$$

hätten wir dann

$$(c + e + g) \cdot (1 + d + f) \in \mathbb{P}.$$

Das kann aber nicht sein, denn wegen $d + f > 0$ ist der Faktor $1 + d + f$ größer als 1 und wegen

$$d + e + f \leq n, \quad c + d + e + f + g = p \quad \text{und} \quad p > n + 1$$

wissen wir, dass

$$c + e + g \geq c + g = c + d + e + f + g - (d + e + f) = p - (d + e + f) \geq p - n > 1$$

ist, so dass auch der Faktor $(c + e + g)$ ebenfalls größer als 1 ist. Damit kann das Produkt

$$(c + e + g) \cdot (1 + d + f)$$

aber keine Primzahl sein und wir haben einen Widerspruch zu der Annahme, dass L kontextfrei ist. □

Aufgabe 33: Zeigen Sie, dass die Sprache $L = \left\{ \mathsf{a}^{2^k} \mid k \in \mathbb{N} \right\}$ nicht kontextfrei ist. ◇

Aufgabe 34: Zeigen Sie, dass die Sprache $L = \left\{ \mathsf{a}^k\mathsf{b}^l\mathsf{c}^{k \cdot l} \mid k, l \in \mathbb{N} \right\}$ nicht kontextfrei ist. ◇

Aufgabe 35: Es seien L_1 und L_2 kontextfreie Sprachen. Beweisen oder widerlegen Sie, dass dann auch die Sprache $L_1 \cap L_2$ kontextfrei ist. ◇

Chapter 12

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, denn 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 hat allerdings eine Laufzeit von $\mathcal{O}(n^3)$ und ist außerdem 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 ebenfalls 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. Dies ist beispielsweise sowohl für $LL(k)$-Grammatiken als auch für $LR(1)$-Grammatiken, die wir in einem späteren Kapitel analysieren werden, der Fall. Dieses Kapitel gliedert sich in die folgenden Abschnitte.

1. Zunächst skizzieren wir die Theorie, die Earley's Algorithmus zu Grunde liegt.
2. Danach geben wir eine einfache Implementierung des Algorithmus in SETLX an.
3. Anschließend beweisen wir die Korrektheit und Vollständigkeit des Algorithmus.
4. Zum Abschluss des Kapitels untersuchen wir die Komplexität.

12.1 Der Algorithmus von Earley

Der zentrale Begriff des von Earley angegebenen Algorithmus ist der Begriff des Earley-Objekts, das wie folgt definiert ist.

Definition 35 (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

$$\langle A \to \alpha \bullet \beta, k \rangle$$

dann als ein *Earley-Objekt*, falls folgendes gilt:

1. $(A \to \alpha\beta) \in R$ und
2. $k \in \{0, 1, \cdots, n\}$. □

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 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 folgende 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 um einen String der Form $s = x_1 \cdots x_n$ zu parsen funktioniert 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:

   1. Wir vergrößern Q_i mit der Vervollständigungs-Operation so lange, bis mit dieser Operation keine neuen Earley-Objekte mehr gefunden werden können.
   2. Anschließend vergrößern wir Q_i mit Hilfe der Vorhersage-Operation. Diese Operation wird ebenfalls so lange durchgeführt, wie neue Earley-Objekte gefunden werden.
   3. 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 12.1 zeigt eine vereinfachte Grammatik für arithmetische Ausdrücke, die nur aus den Zahlen "1", "2" und "3" und den beiden Operator-Symbolen "+" und "*" aufgebaut sind. Die Menge T der Terminale dieser Grammatik ist also durch

$$T = \{ \text{“1”} \ , \ \text{“2”} \ , \ \text{“3”} \ , \ \text{“+”} \ , \ \text{“*”} \ \}$$

gegeben. Wie 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 12.1: 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:

1. $\langle \widehat{S} \to \bullet\, E, 0\rangle$,
2. $\langle E \to \bullet\, E \text{“+”} P, 0\rangle$
3. $\langle E \to \bullet\, P, 0\rangle$,
4. $\langle P \to \bullet\, P \text{“*”} F, 0\rangle$,
5. $\langle P \to \bullet\, F, 0\rangle$,
6. $\langle F \to \bullet \text{“1”}, 0\rangle$,
7. $\langle F \to \bullet \text{“2”}, 0\rangle$,
8. $\langle F \to \bullet \text{“3”}, 0\rangle$.

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 wie 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 der Vervollständigungs-Operation) für Q_1 insgesamt die folgenden Earley-Objekte durch Vervollständigung:

   1. $\langle P \to F\, \bullet, 0\rangle$,
   2. $\langle P \to P\, \bullet \text{“*”} F, 0\rangle$,
   3. $\langle E \to P\, \bullet, 0\rangle$,
   4. $\langle E \to E\, \bullet \text{“+”} P, 0\rangle$,
   5. $\langle \widehat{S} \to E\, \bullet, 0\rangle$.

   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-Objekte 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:

   1. $\langle F \to \bullet \text{“1”}, 2\rangle$,
   2. $\langle F \to \bullet \text{“2”}, 2\rangle$,
   3. $\langle F \to \bullet \text{“3”}, 2\rangle$.

   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 eine weiteres Mal die Vervollständigungs-Operation anwenden. Durch iterierte Anwendung der Vervollständigungs-Operation erhalten wir zusätzlich die folgenden Earley-Objekte:

   1. $\langle P \to P \bullet \text{“*”} F, 2\rangle$,
   2. $\langle E \to E \text{“+”} P \bullet, 0\rangle$,
   3. $\langle E \to E\ \bullet \text{“+”} P, 0\rangle$
   4. $\langle \widehat{S} \to E \bullet, 0\rangle$.

   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:

   1. $\langle F \to \bullet \text{“1”}, 4\rangle$,
   2. $\langle F \to \bullet \text{“2”}, 4\rangle$,
   3. $\langle F \to \bullet \text{“3”}, 4\rangle$.

   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:

   1. $\langle P \to P \text{“*”} F\ \bullet, 2\rangle$,
   2. $\langle E \to E \text{“+”} P\ \bullet, 0\rangle$,
   3. $\langle P \to P\ \bullet \text{“*”} F, 2\rangle$,
   4. $\langle E \to E\ \bullet \text{“+”} P, 0\rangle$

5. $\langle \widehat{S} \to E\ \bullet, 0 \rangle$.

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: Zeigen Sie, dass der String "1*2+3" in der Sprache der von der in Abbildung 12.1 angegebenen Grammatik liegt. Benutzen Sie dazu den von Earley angegebenen Algorithmus.

12.2 Implementing Earley's Algorithm in SetlX

In this section we present a simple implementation of Earley's algorithm in SetlX. Our implementation consists of three classes.

1. The class `earleyItem` is our implementation of an Earley item.
2. The class `grammar` is used to represent a grammar.
3. The class `earleyParser` implements Earley's algorithm.

We will discuss these classes in the order given above.

12.2.1 The class `earleyItem`

```
1  class earleyItem(variable, alpha, beta, index) {
2      this.mVariable := variable;
3      this.mAlpha    := alpha;
4      this.mBeta     := beta;
5      this.mIndex    := index;
6
7      isComplete := procedure() {
8          return mBeta == [];
9      };
10     sameVar := procedure(c) {
11         return #mBeta > 0 && mBeta[1] == Var(c);
12     };
13     myScan := procedure(t) {
14         return #mBeta > 0 && mBeta[1] == Token("\'$t$\'");
15     };
16     nextVar := procedure() {
17         if (#mBeta > 0) {
18             match (mBeta[1]) {
19                 case Var(c): return c;
20             }
21         }
22     };
23     moveDot := procedure() {
24         return earleyItem(mVariable, mAlpha + [ mBeta[1]], mBeta[2..], mIndex);
25     };
26 }
```

Figure 12.2: The class `earleyItem`.

Figure 12.2 on page 154 gives the implementation of the class `earleyItem` that is used to represent an Earley item of the form

$$\langle A \Rightarrow \alpha \bullet \beta, k \rangle.$$

The member variables of the class `earleyItem` represent this Earley item as follows:

1. `mVariable` codes the syntactical variable A,
2. `mAlpha` corresponds to α,
3. `mBeta` represents β, and
4. `mIndex` gives the index k.

The methods provided in this class work as follows:

1. `isComplete()` checks whether the Earley item has the form

   $$\langle A \Rightarrow \alpha \bullet \varepsilon, k \rangle.$$

   This is the case if the list `mBeta` is empty.

2. `sameVar`(C) checks whether the Earley item has the form

   $$\langle A \rightarrow \alpha \bullet C\, \beta, k \rangle,$$

   i.e. it checks, whether the variable C that is given as argument to this method is following the "•". In order to do this, we first have to check, whether there is anything following the "•", i.e. whether β is not empty. Only then can we check whether the first item of β is indeed the variable C.

3. The method `myScan`(t) checks whether the Earley item has the form

   $$\langle A \rightarrow \alpha \bullet t\, \beta, k \rangle,$$

   that is it checks whether the token t that is given as argument to this method follows the "•".

4. The method *nextVar*() first checks whether the Earley item has the form

   $$\langle A \rightarrow \alpha \bullet C\delta, k \rangle,$$

   that is it checks whether a variable follows the "•". If this is the case, the name of this variable is returned.

5. The method `moveDot()` moves the "•" over the next variable or token. Therefore, an Earley item of the form

   $$\langle A \rightarrow \alpha \bullet X\delta, k \rangle$$

   is transformed into

   $$\langle A \rightarrow \alpha X \bullet \delta, k \rangle.$$

   Here, X can be either a variable or a token. The method `moveDot` should only be called for an Earley item $\langle A \rightarrow \alpha \bullet \beta, k \rangle$ if we know that $\beta \neq \varepsilon$.

12.2.2 The class `grammar`

Next, we show how we represent a grammar. Figure 12.3 on page 156 shows the implementation of the class `grammar` that represents a context free grammar.

1. The constructor gets a list of rules. It stores these rules in the member variable `mRules`. In this list, a rule of the form

   $$A \rightarrow \beta$$

   is represented as the pair $[A, \beta]$, where β is a list of syntactical variables and tokens. In this list, a variable is represented as a term of the form `Var(`x`)`, where x is the name of the variable. A token is represented as a term of the form `Token(`t`)`, where t is the quoted string of the token.

```
1  class grammar(rules) {
2      this.mRules := rules;
3
4      startItem := procedure() {
5          return earleyItem("sHat", [], [ Var(startVar()) ], 0);
6      };
7      finishItem := procedure() {
8          return earleyItem("sHat", [ Var(startVar()) ], [], 0);
9      };
10     startVar := procedure() {
11         return mRules[1][1];
12     };
13 }
```

Figure 12.3: The class `grammar`

2. The method `startItem` creates the Earley item

   $$\langle \hat{S} \to \bullet S, 0 \rangle.$$

   Here, S is the start symbol of the grammar. The symbol $\hat{S}$ is represented by the string `sHat`.

3. The method `finishItem` creates the Earley item

   $$\langle \hat{S} \to S \bullet, 0 \rangle.$$

   Again, S is the start symbol of the grammar and $\hat{S}$ is represented as the string `sHat`.

4. The method `startVar` computes the start variable of the grammar. It is assumed that the first grammar rule in `mRules` defines the start variable of the grammar.

12.2.3 The class `earleyParser`

We are finally ready to implement Earley's algorithm. It is implemented in the class `earleyParser` shown in Figure 12.4 on page 157. The class `earleyParser` maintains three member variables:

1. `mGrammar` is the given grammar.

2. `mString` is the given string that is to be parsed.

   The parser that is implemented in the class `earleyParser` does not use a scanner but rather works directly with the characters that make up the string `mString`.

3. `mStateList` is the list $[Q_0, Q_1, \cdots, Q_n]$. Since in SetlX lists are 1-based, we have to store the set Q_0 at the index 1 in `mStateList`. Therefore, in general we have

   $$Q_i = \texttt{mStateList}[i+1].$$

Figure 12.4 shows the outline of the class `earleyParser`. The constructor of this class has two formal arguments:

1. `gr` is the grammar and

2. `w` is the string to be parsed.

The main task of the constructor is to store the given arguments in the member variables and, furthermore, to initialize the sets Q_i that are stored in the member variable `mStateList`. Initially, all Q_i are assigned the empty set. Additionally, the Earley item

$$\langle \widehat{S} \to \bullet S, 0 \rangle$$

```
class earleyParser(gr, w) {
    this.mGrammar   := gr;
    this.mString    := w;
    this.mStateList := [ {}: i in {1 .. #mString+1} ]; // Q_i == mStateList[i+1]
    this.mStateList[1] :=  mGrammar.startItem() ;

    earleyParse := procedure()  { ··· };
    complete    := procedure(i) { ··· };
    predict     := procedure(i) { ··· };
    myScan      := procedure(i) { ··· };
}
```

Figure 12.4: Outline of the class `earleyParser`.

is inserted into Q_0. This Earley item is computed by the method `startItem()` of the class `grammar` that is discussed later. The main work of the class `earleyParser` is delegated to the method `earleyParse`. Furthermore, the class `earleyParser` implements the following methods:

1. `complete` performs the completion operation on a given set Q_i,
2. `predict` implements the prediction operation on Q_i, and
3. `myScan` executes the scanning operation on Q_i.

The method `earleyParse()` is shown in Figure 12.5 on page 157. The outer `for`-loop iterates over the sets Q_0, Q_1, $\cdots$, Q_n. For a fixed index $i \in \{0, \cdots, n\}$ the method `earleyParse` will execute the completion operation for the set Q_i followed by the prediction operation. Since performing the prediction operation can have the effect of making additional completion operations possible, these two steps are iterated in an inner `do-while`-loop. Once the set Q_i can no longer be extended using either completion or prediction, the scanning operation is executed in order to initialize the set Q_{i+1}.

```
earleyParse := procedure() {
    for (i in [0 .. #mString]) {
        change := false;
        do {
            change := complete(i);
            change := change || predict(i);
        } while (change);
        myScan(i);
    }
};
```

Figure 12.5: The method *earleyParse*().

Figure 12.6 on page 158 shows the implementation of the completion operation. The argument i specifies the set Q_i that has to be completed. The method `complete` returns a `Boolean` value. The return value is `true` if the completion operation has added any new Earley items into the set Q_i. Otherwise `false` is returned. In the previous section, the completion operation has been specified as follows:

$$\langle C \to \gamma\bullet, j\rangle \in Q_i \wedge \langle A \to \beta \bullet C\delta, k\rangle \in Q_j \;\Rightarrow\; Q_i := Q_i \cup \big\{\langle A \to \beta C \bullet \delta, k\rangle\big\}$$

This formula is implemented as follows:

1. The variable i is the parameter `i` of the method `complete`.

```
1  complete := procedure(i) {
2      change := false;
3      qi     := mStateList[i+1];
4      do {
5          added := false;
6          newQi := {};
7          for (item in qi | item.isComplete()) {
8              c  := item.mVariable;
9              j  := item.mIndex;
10             qj := mStateList[j+1];
11             for (cItem in qj) {
12                 if (cItem.sameVar(c)) {
13                     moved := cItem.moveDot();
14                     newQi += { moved };
15                 }
16             }
17         }
18         if (!(newQi <= qi)) {
19             change := added := true;
20             this.mStateList[i+1] += newQi;
21             qi := mStateList[i+1];
22         }
23     } while (added);
24     return change;
25 };
```

Figure 12.6: Implementing the completion operation via the method `complete()`.

2. The variable `change` stores the return value of the method `complete`. This return value is `true` if the completion operation adds any items into the set, Q_i. Therefore, the variable `change` is initialized as `false`, since initially nothing has been added to Q_i.

3. If the completion operation is executed once and has added a new Earley item into the set Q_i it is possible that the completion operation can be applied to these new Earley items also. This is the reason that the completion operation is executed in a `do-while`-loop. This loop runs as long as new Earley items are added to Q_i. The `do-while`-loop is controlled by the variable `added`: At the start of the loop, this variable is set to `false`. Every times a new Earley item is added into the set Q_i, the variable `added` is set to `true`.

4. Inside the `do-while`-loop, the set `newQi` is intended to collect all those Earley items that are found using the completion operation.

5. The `for`-loop in line 7 iterates over all Earley items in the set `Qi` that are of the form $\langle C \to \gamma\bullet, j\rangle$, i.e. it iterates over those items that have the dot at their rightmost position.

   1. For those items, `c` is set to the variable C on the left hand side of the grammar rule, `j` is set to the index of this Earley item, and `qj` is assigned the set Q_j.
   2. Next, we have to check whether the set Q_j contains an Earley item `cItem` such that the dot is positioned in front of the variable C. In this case `cItem` is an Earley item of the form

      $$\langle A \to \beta \bullet C\delta, k\rangle.$$

      This checking of all Earley items in the set Q_j is performed in the innermost `for`-loop. Then, the method invocation `cItem.moveDot()` computes the Earley item

      $$\langle A \to \beta C \bullet \delta, k\rangle,$$

which is then added to the set `newQi`.

6. If we do indeed find new Earley items, we add these new items to the set Q_i. Furthermore, we need to maintain the variables `change` and `added`.

```
1  predict := procedure(i) {
2      change := false;
3      qi     := mStateList[i+1];
4      do {
5          added := false;
6          newQi := {};
7          for (item in qi) {
8              c := item.nextVar();
9              if (c != om) {
10                 for (rule in mGrammar.mRules | c == rule[1]) {
11                     newQi += { earleyItem(c, [], rule[2], i) };
12                 }
13             }
14         }
15         if (!(newQi <= qi)) {
16             change := added := true;
17             this.mStateList[i+1] += newQi;
18             qi := mStateList[i+1];
19         }
20     } while (added);
21     return change;
22 };
```

Figure 12.7: Implementing the prediction operation via the method `predict`.

Figure 12.7 on page 157 shows the method `predict`. This method implements the prediction operation. In the previous section, this operation had been specified via the following formula:

$$\langle A \to \beta \bullet C\delta, k\rangle \in Q_i \wedge (C \to \gamma) \in R \;\Rightarrow\; Q_i := Q_i \cup \left\{\langle C \to \bullet\gamma, i\rangle\right\}$$

We discuss the details of the implementation of `predict` next.

1. The parameter `i` specifies the set Q_i. The prediction operation is then applied to this set.

2. The method `predict` will return `true` if it has added any new Earley item to the set Q_i. Otherwise, `false` is returned. The return value is stored in the variable `change`.

3. If a new Earley item is added to Q_i because of prediction, it is well possible that the prediction operation can again be applied to this new Earley item. Therefore, in order to find all possible applications of the prediction operation, we have to loop as long as new Earley items are found. This loop is controlled by the variable `added` in much the same way as the corresponding `do-while`-loop in the method `complete`.

4. In order to perform the prediction operation, we need to loop over all Earley items in the set Q_i and we have to check, whether there are Earley items of the form

   $$\langle A \to \beta \bullet C\delta, k\rangle$$

   in Q_i. Once we have found an item of this form, we need to loop over all grammar rules and check, whether the variable on left hand side of the rule is the variable C. If this is the case, and the grammar rule has the form

$$C \to \gamma,$$

then we have to add the Earley item

$$\langle C \to \bullet\, \gamma, i \rangle$$

to the set Q_i.

```
1  myScan := procedure(i) {
2      qi := mStateList[i+1];
3      if (i < #mString) {
4          a := mString[i+1];
5          for (item in qi | item.myScan(a)) {
6              this.mStateList[i+2] += { item.moveDot() };
7          }
8      }
9  };
```

Figure 12.8: Implementing the scan operation via the method `myScan`.

Figure 12.8 gives the implementation of the scanning operation. The formula describing this operation is given as:

$$\langle A \to \beta \bullet a\gamma, k \rangle \in Q_i \wedge x_{i+1} = a \;\Rightarrow\; Q_{i+1} := Q_{i+1} \cup \big\{ \langle A \to \beta a \bullet \gamma, k \rangle \big\}.$$

We discuss how this formula is implemented in the method `myScan`. (By the way, we had to name this method `scan` instead of `myScan` since `scan` is a keyword in SetlX.)

1. The argument `i` specifies the set Q_i that is to be used as the basis of the scanning operation.

2. If the string that is to be parsed consists of n characters, then the scanning operation can only be performed as long as $i < n$. The reason is that the scanning operation reads the $(i+1)$-st character of the input string and this is not defined if $i = n$. This explains the test of the first `if`-statement.

3. Once we have found the character `a` at position $i+1$ we have to check all Earley items in the set Q_i whether they have the form

   $$\langle A \to \beta \bullet a\, \gamma, k \rangle.$$

   This is done by calling the method `myScan` of all Earley items in the set Q_i. If indeed the Earley item has the form $\langle A \to \beta \bullet a\, \gamma, k \rangle$ the Earley item

   $$\langle A \to \beta\, a \bullet \gamma, k \rangle$$

   is added to the set Q_{i+1}. This Earley item can be conveniently computed using the method `moveDot`.

Finally, Figure 12.9 on page 161 shows the sets Q_i that are computed by our implementation if we parse the input string "1+2*3" using the grammar from Figure 12.1.

12.3 Korrektheit und Vollständigkeit

In diesem Kapitel beweisen wir zwei Eigenschaften des von Earley angegebenen Algorithmus. Zunächst zeigen wir, dass immer dann, wenn wir den Algorithmus auf einen String $s = x_1 x_2 \cdots x_n$ anwenden und nach Beendigung des Algorithmus das Earley-Objekt $\langle \widehat{S} \to S \bullet, 0 \rangle$ in der Menge Q_n enthalten ist, wir schließen können, dass der String s sich von dem Start-Symbol S ableiten lässt. Diese Eigenschaft bezeichnen wir als die Korrektheit des Algorithmus. Außerdem beweisen wir, dass auch die Umkehrung gilt: Falls der String s in der von der Variablen S erzeugten Sprache liegt, dann ist das Earley-Objekt $\langle \widehat{S} \to S \bullet, 0 \rangle$ nach Beendigung des Algorithmus ein Element der Menge

```
Q0:
<expr -> (*) expr '+' prod, 0>
<expr -> (*) prod, 0>
<fact -> (*) '1', 0>
<fact -> (*) '2', 0>
<fact -> (*) '3', 0>
<prod -> (*) fact, 0>
<prod -> (*) prod '*' fact, 0>
<sHat -> (*) expr, 0>

Q1:
<fact -> '1' (*), 0>
<expr -> expr (*) '+' prod, 0>
<sHat -> expr (*), 0>
<prod -> fact (*), 0>
<expr -> prod (*), 0>
<prod -> prod (*) '*' fact, 0>

Q2:
<fact -> (*) '1', 2>
<fact -> (*) '2', 2>
<fact -> (*) '3', 2>
<prod -> (*) fact, 2>
<prod -> (*) prod '*' fact, 2>
<expr -> expr '+' (*) prod, 0>

Q3:
<fact -> '2' (*), 2>
<expr -> expr (*) '+' prod, 0>
<sHat -> expr (*), 0>
<expr -> expr '+' prod (*), 0>
<prod -> fact (*), 2>
<prod -> prod (*) '*' fact, 2>

Q4:
<fact -> (*) '1', 4>
<fact -> (*) '2', 4>
<fact -> (*) '3', 4>
<prod -> prod '*' (*) fact, 2>

Q5:
<fact -> '3' (*), 4>
<expr -> expr (*) '+' prod, 0>
<sHat -> expr (*), 0>
<expr -> expr '+' prod (*), 0>
<prod -> prod (*) '*' fact, 2>
<prod -> prod '*' fact (*), 2>
```

Figure 12.9: Output of our Earleyd parser for the input "1+2*3" and the grammar from Figure 12.1.

Q_n. Diese Eigenschaft bezeichnen wir als die Vollständigkeit des Algorithmus. Bei allen folgenden Betrachtungen

gehen wir davon aus, dass $G = \langle V, T, S, R\rangle$ die verwendete kontextfreie Grammatik bezeichnet.

Das nachfolgende Lemma wird später benötigt, um die Korrektheit zu zeigen. Es formalisiert die Idee, die der Definition eines Earley-Objekts zu Grunde liegt.

Lemma 36 Es sei $s = x_1 x_2 \cdots x_n \in T^*$ der String, auf den wir den Algorithmus von Earley anwenden. Weiter gelte

$$\langle A \to \alpha \bullet \beta, k\rangle \in Q_i \quad \text{mit } i \in \{0, \cdots, n\}.$$

Dann gilt

$$\alpha \Rightarrow^* x_{k+1} \cdots x_i.$$

Beweis: Wir führen den Beweis durch Induktion über die Anzahl l der Berechnungs-Schritte, die der Algorithmus durchgeführt hat, um $\langle A \to \alpha \bullet \beta, k\rangle \in Q_i$ nachzuweisen.

I.A.: $l = 0$. Zu Beginn enthält die Menge Q_0 nur das Earley-Objekt

$$\langle \widehat{S} \to \bullet S, 0\rangle$$

und alle anderen Mengen Q_i sind leer. Damit müssen wir die Behauptung nur für dieses eine Earley-Objekt nachweisen. Für dieses Earley-Objekt haben die Variablen A, α, β und k folgende Werte:

1. $A = \widehat{S}$,
2. $\alpha = \varepsilon$,
3. $\beta = S$,
4. $i = 0$,
5. $k = 0$.

Wir müssen dann zeigen, dass

$$\varepsilon \Rightarrow^* x_1 \cdots x_0$$

gilt. Diese Behauptung folgt aus $x_1 \cdots x_0 = \varepsilon$.

I.S.: $0, \cdots, l \mapsto l + 1$. Wir müssen eine Fallunterscheidung nach der Art der Operation durchführen, mit der das Earley-Objekt $E = \langle A \to \alpha \bullet \beta, k\rangle$ erzeugt worden ist.

1. E ist durch eine Vorhersage-Operation in Q_i eingefügt worden. Daher enthält Q_i ein Earley-Objekt der Form

   $$F = \langle A' \to \alpha' \bullet A\,\delta, k'\rangle.$$

   Dann muss die Grammatik eine Regel der Form $A \to \beta$ enthalten, mit der die Vorhersage-Operation das Earley-Objekt

   $$\langle A \to \bullet\beta', i\rangle$$

   erzeugt hat. Damit gilt also $\alpha = \varepsilon$, $\beta = \beta'$ und $k = i$. Es ist zu zeigen, dass

   $$\varepsilon \Rightarrow^* x_{i+1} \cdots x_i$$

   gilt. Wegen $x_{i+1} \cdots x_i = \varepsilon$ ist das trivial.
2. E ist durch eine Lese-Operation in Q_i eingefügt worden. Dann gibt es ein Earley-Objekt der Form $\langle A \to \alpha' \bullet x_i\beta, k\rangle \in Q_{i-1}$, es gilt $\alpha = \alpha' x_i$ und nach Induktions-Voraussetzung gilt

   $$\alpha' \Rightarrow^* x_{k+1} \cdots x_{i-1}$$

   Wir müssen zeigen, dass

   $$\alpha \Rightarrow^* x_{k+1} \cdots x_i$$

   gilt. Dies folgt aus

   $$\alpha = \alpha' x_i \Rightarrow^* x_{k+1} \cdots x_{i-1} x_i = x_{k+1} \cdots x_i.$$

3. E ist durch eine Vervollständigungs-Operation in Q_i eingefügt worden. Dann hat E die Form

   $$E = \langle A \to \alpha' C \bullet \beta, k \rangle$$

   und es gibt ein Earley-Objekt

   $$\langle C \to \delta \bullet, j \rangle \in Q_i$$

   und ein weiteres Earley-Objekt

   $$\langle A \to \alpha' \bullet C\beta, k \rangle \in Q_j.$$

   Also haben wir $\alpha = \alpha' C$. Aus $\langle A \to \alpha' \bullet C\beta, k \rangle \in Q_j$ folgt nach Induktions-Voraussetzung

   $$\alpha' \Rightarrow^* x_{k+1} \cdots x_j$$

   und aus $\langle C \to \delta \bullet, j \rangle \in Q_i$ folgt nach Induktions-Voraussetzung

   $$\delta \Rightarrow^* x_{j+1} \cdots x_i,$$

   so dass wir insgesamt die folgende Ableitung haben:

   $$\begin{array}{lcl} \alpha & = & \alpha' C \\ & \Rightarrow^* & x_{k+1} \cdots x_j C \\ & \Rightarrow & x_{k+1} \cdots x_j \delta \\ & \Rightarrow^* & x_{k+1} \cdots x_j x_{j+1} \cdots x_i \\ & = & x_{k+1} \cdots x_i \end{array}$$

   Damit ist $\alpha \Rightarrow^* x_{k+1} \cdots x_i$ nachgewiesen. □

Korollar 37 (Korrektheit)
Wenden wir den Algorithmus von Earley auf den String $s = x_1 \cdots x_n$ an und gilt $\langle \widehat{S} \to S \bullet, 0 \rangle \in Q_n$, so folgt $s \in L(G)$.

Beweis: Setzen wir $A := \widehat{S}$, $\alpha := S$, $\beta := \varepsilon$, $k := 0$ und $i := n$, so folgt das Ergebnis unmittelbar, wenn wir das letzte Lemma auf die Voraussetzung anwenden. □

Das Korollar zeigt: Produziert der Algorithmus von Earley das Earley-Objekt

$$\langle \widehat{S} \to S \bullet, 0 \rangle \in Q_n,$$

so liegt der String, auf den wir den Algorithmus angewendet haben, tatsächlich in der von der Grammatik erzeugten Sprache. Wir wollen aber auch die Umkehrung dieser Aussage nachweisen: Falls ein String s von der Grammatik erzeugt wird, so wird der Algorithmus von Earley dies auch erkennen. Dazu zeigen wir zunächst das folgende Lemma.

Lemma 38 Angenommen, wir haben die folgenden Voraussetzungen:

1. $\langle A \to \alpha \bullet \beta\gamma, k \rangle \in Q_i$,
2. $\beta \Rightarrow^* x_{i+1} \cdots x_l$.

Dann gilt auch

$$\langle A \to \alpha\beta \bullet \gamma, k \rangle \in Q_l.$$

Beweis: Wir führen den Beweis durch Induktion über die Anzahl der Ableitungs-Schritte in der Ableitung $\beta \Rightarrow^* x_{i+1} \cdots x_l$.

1. Falls die Ableitung die Länge 0 hat, gilt offenbar $\beta = x_{i+1} \cdots x_l$. Damit hat die Voraussetzung

   $$\langle A \to \alpha \bullet \beta\gamma, k \rangle \in Q_i$$

   die Form

   $$\langle A \to \alpha \bullet x_{i+1} \cdots x_l \gamma, k \rangle \in Q_i.$$

Wenden wir hier $(l-i)$ mal die Lese-Operation an, so erhalten wir

$$\langle A \to \alpha x_{i+1} \cdots x_l \bullet \gamma, k\rangle \in Q_l$$

und wegen $\beta = x_{i+1} \cdots x_l$ ist das äquivalent zu

$$\langle A \to \alpha\beta \bullet \gamma, k\rangle \in Q_l.$$

Das ist aber gerade die Behauptung.

2. Wir führen nun den Induktions-Schritt durch, wobei die Ableitung $\beta \Rightarrow^* x_{i+1} \cdots x_l$ eine Länge größer als 0 hat und nehmen an, dass die erste Regel, die bei dieser Ableitung angewendet worden ist, die Form $D \to \delta$ hat. Zur Vereinfachung nehmen wir außerdem an, dass die Ableitungs-Schritte so durchgeführt werden, dass immer die linkeste Variable ersetzt wird. Die Ableitung hat dann insgesamt die Form

$$\beta = x_{i+1} \cdots x_j D\mu \Rightarrow x_{i+1} \cdots x_j \delta\mu \Rightarrow^* x_{i+1} \cdots x_l.$$

und wir haben

$$\delta \Rightarrow^* x_{j+1} \cdots x_h \tag{12.1}$$

und

$$\mu \Rightarrow^* x_{h+1} \cdots x_l \tag{12.2}$$

für ein geeignetes $h \in \{j, \cdots, l+1\}$. Also können wir zunächst auf das Earley-Objekt

$$\langle A \to \alpha \bullet \beta\gamma, k\rangle \in Q_i$$

$(j-i)$ mal die Lese-Operation anwenden. Damit erhalten wir

$$\langle A \to \alpha x_{i+1} \cdots x_j \bullet D\mu\gamma, k\rangle \in Q_j. \tag{12.3}$$

Auf dieses Earley-Objekt wenden wir die Vorhersage-Operation an und erhalten

$$\langle D \to \bullet\, \delta, j\rangle \in Q_j. \tag{12.4}$$

Aus (12.4) und (12.1) folgt mit der Induktions-Voraussetzung

$$\langle D \to \delta \bullet, j\rangle \in Q_h. \tag{12.5}$$

Aus (12.3) und (12.5) folgt mit der Vervollständigungs-Operation

$$\langle A \to \alpha x_{i+1} \cdots x_j D \bullet \mu\gamma, k\rangle \in Q_h. \tag{12.6}$$

Aus (12.6) und (12.2) folgt mit der Induktions-Voraussetzung

$$\langle A \to \alpha x_{i+1} \cdots x_j D\mu \bullet \gamma, k\rangle \in Q_l. \tag{12.7}$$

Wegen $\beta = x_{i+1} \cdots x_j D\mu$ haben wir damit

$$\langle A \to \alpha\beta \bullet \gamma, k\rangle \in Q_l$$

gezeigt und das ist gerade die Behauptung. □

Korollar 39 (Vollständigkeit)
Falls $s = x_1 \cdots x_n \in L(G)$ ist, dann gilt $\langle \widehat{S} \to S\bullet, 0\rangle \in Q_n$.

Beweis: Der Algorithmus von Earley initialisiert die Menge Q_0 mit dem Earley-Objekt

$$\langle \widehat{S} \to \bullet S, 0\rangle.$$

Die Voraussetzung $x_1 \cdots x_n \in L(G)$ heißt gerade $S \Rightarrow^* x_1 \cdots x_n$. Also folgt die Behauptung aus dem eben bewiesenen Lemma, wenn wir dort $\alpha := \varepsilon$, $\beta := S$, $\gamma := \varepsilon$, $i := 0$, $k := 0$ und $l := n$ setzen. □

12.4 Analyse der Komplexität

Wir zeigen, dass sich die Anzahl der Schritte, die der Algorithmus von Earley bei einem String der Länge n durch $\mathcal{O}(n^3)$ nach oben abschätzen lässt. Falls die Grammatik eindeutig ist, wenn es also zu jedem String nur einen Parse-Baum gibt, kann dies zu $\mathcal{O}(n^2)$ verbessert werden. Zusätzlich hat Jay Earley in seiner Doktorarbeit [Ear68] gezeigt, dass der Algorithmus in vielen praktisch relevanten Fällen eine lineare Komplexität hat.

Der Schlüssel bei der Berechnung der Komplexität des Algorithmus von Earley ist die Betrachtung der Anzahl der Elemente der Menge Q_i. Es gilt offenbar:

$$\text{Wenn } \langle A \to \alpha \bullet \beta, k\rangle \in Q_i, \text{ dann } k \in \{0, \cdots, i\}.$$

Die Komponente $A \to \alpha \bullet \beta$ hängt nur von der Grammatik und nicht von dem zu parsenden String ab, während der Index i von der Länge des zu parsenden Strings abhängig ist, es gilt

$$i \in \{0, \cdots, n\}.$$

Interessiert nur das Wachstum der Mengen Q_i in Abhängigkeit von der Länge des zu parsenden Strings, so kann daher die Anzahl der Elemente der Menge Q_i durch $\mathcal{O}(n)$ abgeschätzt werden. Wir analysieren nun die Anzahl der Rechenschritte, die bei den einzelnen Operationen durchgeführt werden.

1. Bei der Implementierung der Vorhersage-Operation in der Methode *predict*() (Abbildung 12.7) haben wir drei Schleifen.

   1. Für die äußere `do-while`-Schleife finden wir, dass diese maximal so oft durchlaufen wird, wie neue Earley-Objekte in die Menge Q_i eingefügt werden. Alle mit der Vorhersage-Operation neu eingefügten Objekte haben aber die Form

      $$\langle A \to \bullet\gamma, i\rangle,$$

      der Index hat hier also immer den Wert i. Die Anzahl solcher Objekte ist nur von der Grammatik und nicht von dem Eingabe-String abhängig. Damit kann die Anzahl dieser Schleifen-Durchläufe durch $\mathcal{O}(1)$ abgeschätzt werden.
   2. Die äußere `for`-Schleife läuft über alle Earley-Objekte der Menge Q_i und wird daher maximal $\mathcal{O}(n)$-mal durchlaufen.
   3. Die innere `for`-Schleife läuft über alle Grammatik-Regeln und ist von der Länge des zu parsenden Strings unabhängig. Diese Schleife liefert also nur einen Beitrag, der durch $\mathcal{O}(1)$ abgeschätzt werden kann.

   Insgesamt hat ein Aufruf der Methode `predict`() daher die Komplexität $\mathcal{O}(1) \cdot O(n) \cdot \mathcal{O}(1) = \mathcal{O}(n)$.

2. Bei der Implementierung der Lese-Operation, die in Abbildung 12.8 gezeigt ist, haben wir eine `for`-Schleife, die über alle Elemente der Menge Q_i iteriert. Da diese Menge $\mathcal{O}(n)$ Elemente enthält, hat die Lese-Operation ebenfalls die Komplexität $\mathcal{O}(n)$.

3. Die von uns in Abbildung 12.6 auf Seite 158 gezeigte Implementierung der Vervollständigungs-Operation in der Methode `complete`() ist ineffizient, weil wir in der äußeren `for`-Schleife immer über alle Elemente der Menge Q_i iterieren. Effizienter wäre es, wenn wir nur über die beim letzten Schleifen-Durchlauf neu hinzu gekommenen Elemente iterieren würden. Dann würde im schlimmsten Falle für jedes Elemente der Menge Q_i einmal die innere `for`-Schleife, die über alle Elemente der Menge Q_j iteriert, ausgeführt. Da die Anzahl der Elemente von Q_i und Q_j jeweils durch $\mathcal{O}(n)$ abgeschätzt werden können, kann die Komplexität der Vervollständigungs-Operation insgesamt mit $\mathcal{O}(n^2)$ abgeschätzt werden.

Da die einzelnen Operationen für alle Mengen Q_i für $i = 0, \cdots, n$ durchgeführt werden müssen, hat der Algorithmus insgesamt die Komplexität $\mathcal{O}(n^3)$. Damit diese Komplexität auch tatsächlich erreicht wird, müssten wir die Implementierung der Methoden so umändern, dass kein Element der Menge Q_i mehrfach betrachtet wird. Da eine solche Implementierung schwerer zu verstehen wäre, haben wir uns aus didaktischen Gründen mit einer ineffizienteren Implementierung begnügt.

Der Nachweis, dass der Algorithmus bei einer eindeutigen Grammatik die Komplexität $\mathcal{O}(n^2)$ hat, geht über den Rahmen der Vorlesung hinaus und kann in dem Artikel von Earley [Ear70] nachgelesen werden. In seiner Doktorarbeit hat Jay Earley [Ear68] zusätzlich gezeigt, dass der Algorithmus in vielen praktisch relevanten Fällen nur eine lineare

Komplexität hat. Es gibt daher eine Reihe von Parser-Generatoren, die den Algorithmus von Earley umsetzen, z. B. das System *Accent*

http://accent.compilertools.net,

mit dessen Hilfe sich C-Parser für beliebige Grammatiken erzeugen lassen. Ein Problem bei der Verwendung solcher Systeme besteht in der Praxis darin, dass die Frage, ob eine Grammatik eindeutig ist, unentscheidbar ist. Bei der Definition einer neuen Grammatik kann es leicht passieren, dass die Grammatik aufgrund eines Design-Fehlers nicht eindeutig ist. Bei der Verwendung eines Earley-Parser-Generators können solche Fehler erst zur Laufzeit des erzeugten Parsers bemerkt werden. Bei Systemen wie *Antlr* oder *Bison*, die mit einer eingeschränkteren Klasse von Grammatiken arbeiten, tritt dieses Problem nicht auf, denn die Grammatiken, für die sich mit einem solchen System ein Parser erzeugen lässt, sind nach Konstruktion eindeutig. Ist also eine Grammatik aufgrund eines Design-Fehlers nicht eindeutig, so liegt Sie erst recht nicht in der eingeschränkten Klasse von LR(1)-Grammatiken, die sich beispielsweise mit *Bison* bearbeiten lassen und der Fehler wird bereits bei der Erstellung des Parsers bemerkt.

Chapter 13

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 ausführlich 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, vereinfacht man diese Tabellen etwas und erhält 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 *JavaCup* diskutieren, der ein LALR-Parser ist.

13.1 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 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 $F \rightarrow$ "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 $P \rightarrow F$ 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:

1. Welche Teilstrings ersetzen wir?

2. 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

$$E \rightarrow P$$

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.

13.2 Shift-Reduce-Parser

Wollen wir einen Bottom-Up-Parser implementieren, so müssen wir uns zunächst die Frage stellen, welche Datenstrukturen wir bei der Implementierung verwenden wollen. Wenn wir uns dabei für einen Stack entscheiden, dann sprechen wir von einem *Shift-Reduce-Parser*. Ist $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik, so wird ein Shift-Reduce-Parser P durch ein 4-Tupel

$$P = \langle Q, q_0, \mathit{action}, \mathit{goto} \rangle$$

beschrieben. Dabei gilt:

1. Q ist die Menge der *Zustände* des Shift-Reduce-Parsers.

   Zunächst werden wir die einzelnen Zustände rein abstrakt sehen und ihre Natur nicht näher analysieren. Später, wenn wir die Theorie der SLR-Parser diskutieren, werden wir erkennen, dass ein Zustand die Information speichert, mit welcher Grammatik-Regel der Parser gerade zu parsen versucht und welcher Teil der rechten Seite der Regel bereits erkannt worden ist.

2. $q_0 \in Q$ ist der Start-Zustand.

3. *action* ist eine Funktion, die als Argumente einen Zustand $q \in Q$ und ein Terminal $t \in T$ erhält. Das Ergebnis ist ein Element der Menge

   $$\mathit{Action} := \big\{\langle \texttt{shift}, q\rangle \mid q \in Q\big\} \cup \big\{\langle \texttt{reduce}, r\rangle \mid r \in R\big\} \cup \big\{\texttt{accept}\big\} \cup \big\{\texttt{error}\big\},$$

   wobei `shift`, `reduce`, `accept` und `error` hier einfach als Strings interpretiert werden, mit denen die verschiedenen Arten von Ergebnissen der Funktion *action*() unterschieden werden können. Zusammenfassend haben wir also:

   $$\mathit{action} : Q \times T \to \mathit{Action}.$$

4. *goto* ist eine Funktion, die jedem Zustand $q \in Q$ und jeder syntaktischen Variablen $v \in V$ einen neuen Zustand aus Q zuordnet:

   $$\mathit{goto} : Q \times V \to Q.$$

Ein Shift-Reduce-Parser arbeitet nun mit den folgenden Daten-Strukturen.

1. Einem Stack *states*, auf dem Zustände aus der Menge Q abgelegt werden:

   $$\mathit{states} \in \mathit{Stack}\langle Q \rangle.$$

2. Einem Stack *symbols*, auf dem Grammatik-Symbole, also Terminale und Variablen-Symbole abgelegt werden:

   $$\mathit{symbols} \in \mathit{Stack}\langle T \cup V \rangle.$$

Zur Vereinfachung der folgenden Überlegungen nehmen wir an, dass die Menge T der Terminale das spezielle Symbol "Eof" enthält und dass dieses Symbol das Ende des zu parsenden Strings spezifiziert aber sonst in dem String nicht auftritt.

Figure 13.1 on page 170 shows the implementation of the SetlX class `srParser` that implements shift-reduce-parsing via its method *parseSR*. This method assumes that the function *action* is coded as a binary relation that is stored in the member variable `mActionTable`. The function *goto* is also represented as a binary relation. It is stored in the member variable `mGotoTable`. The method *parseSR* is called with one argument *tl*. This is the list of tokens that have to be parsed. The last element of this list is the special token "Eof" denoting the end of file. The invocation *parseSR*(*tl*) returns `true` if the token list *tl* can be parsed successfully and `false` otherwise. The implementation of *parseSR* 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 1.

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.

```
class srParser(actionTable, gotoTable, stateTable) {
    mActionTable := actionTable;
    mGotoTable   := gotoTable;

    parseSR := procedure(tl) {
        index   := 1;      // point to next token
        symbols := [];     // stack of symbols
        states  := ["s0"]; // stack of states, "s0" is the start state
        while (true) {
            q := states[-1];
            t := tl[index];
            p := mActionTable[q,t];
            match (p) {
            case om:
                 return false;
            case Shift(s):
                 symbols := symbols + [t];
                 states  := states  + [s];
                 index   += 1;
            case Rule(head, body):
                 n       := #body;
                 symbols := symbols[.. -(n+1)];
                 states  := states[.. -(n+1)];
                 symbols := symbols + [head];
                 state   := states[-1];
                 states  := states + [ mGotoTable[state, head] ];
            case Accept():
                 return true;
            }
        }
    };
}
```

Figure 13.1: Implementation of a shift-reduce parser in SetlX

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 *action*(q, t).

   What happens next depends on this value of *action*(q, t).

   1. *action*(q, t) is undefined.
      If *action*(q, t) is undefined, then we really have *action*$(q, t) =$ `error`. However, for reasons of space efficiency, the action table does not store error entries. Therefore, in this case the parser has found a syntax error and returns `false`.
   2. *action*$(q, t) = \langle$`shift`$, s\rangle$.

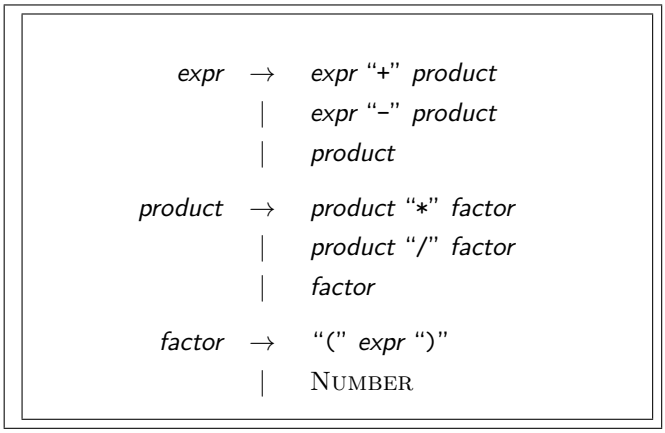
This action is represented in SetlX as the term `Shift(`s`)`. In this case, the token t is pushed onto the symbol stack in line 17, while the state s is pushed onto the stack of states. Furthermore, the variable *index* is incremented to point to the next unread token.

3. *action*$(q, t) = \langle$`reduce`$, A \rightarrow X_1 \cdots X_n \rangle$.
   This action is represented as the SetlX term `Rule(`$A, [X_1, \cdots, X_n]$`)`. In this case, we use the grammar rule
   $$r = (A \rightarrow X_1 \cdots X_n)$$
   to reduce the symbol stack. The SetlX variable *head* represents the left hand side A of this rule, while the list $[X_1, \cdots, X_n]$ is represented by the SetlX 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 *goto*(*state*, A) is put on top of the stack of states in line 26.
4. *action*$(q, t) =$ `accept`.
   In this case parsing is successful and therefore the function returns `true`.

In order to make the function *parseSR* work we have to provide an implementation of the functions *action* and *goto*. The tables 13.1 and 13.2 show these functions for the grammar given in Figure 13.2. 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. $\langle$*shft*$, s_i\rangle$ is short for $\langle$`shift`$, s_i\rangle$.
2. $\langle$*rdc*$, r_i\rangle$ is short for $\langle$`reduce`$, r_i\rangle$, where r_i denothes the grammar rule number i. Here, we have numbered the rules as follows:
   1. $r_1 = ($*expr* $\rightarrow$ *expr* "+" *product*$)$
   2. $r_2 = ($*expr* $\rightarrow$ *expr* "–" *product*$)$
   3. $r_3 = ($*expr* $\rightarrow$ *product*$)$
   4. $r_4 = ($*product* $\rightarrow$ *product* "∗" *factor*$)$
   5. $r_5 = ($*product* $\rightarrow$ *product* "/" *factor*$)$
   6. $r_6 = ($*product* $\rightarrow$ *factor*$)$
   7. $r_7 = ($*factor* $\rightarrow$ "(" *expr* ")" $)$
   8. $r_8 = ($*factor* $\rightarrow$ Number$)$

The corresponding grammar is shown in Figure 13.2. The coding of the functions *action* and *goto* is shown in the Figures 13.3, 13.4, and 13.5 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 subsection.



expr → *expr* "+" *product*
| *expr* "–" *product*
| *product*

product → *product* "*" *factor*
| *product* "/" *factor*
| *factor*

factor → "(" *expr* ")"
| NUMBER

Figure 13.2: 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 13.1: 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 13.2: The function *goto*().

```
actionTable := {};

r1 := Rule("E", ["E", "+", "P"]);      r4 := Rule("P", ["P", "*", "F"]);
r2 := Rule("E", ["E", "-", "P"]);      r5 := Rule("P", ["P", "/", "F"]);
r3 := Rule("E", ["P"]);                r6 := Rule("P", ["F"]);

r7 := Rule("F", ["(", "E", ")"]);
r8 := Rule("F", ["int"]);

actionTable["s0", "("  ] := Shift("s5");
actionTable["s0", "int"] := Shift("s2");

actionTable["s1", "EOF"] := r6;    actionTable["s2", "EOF"] := r8;
actionTable["s1", "+"  ] := r6;    actionTable["s2", "+"  ] := r8;
actionTable["s1", "-"  ] := r6;    actionTable["s2", "-"  ] := r8;
actionTable["s1", "*"  ] := r6;    actionTable["s2", "*"  ] := r8;
actionTable["s1", "/"  ] := r6;    actionTable["s2", "/"  ] := r8;
actionTable["s1", ")"  ] := r6;    actionTable["s2", "("  ] := r8;
                                   actionTable["s2", ")"  ] := r8;

actionTable["s3", "EOF"] := r3;
actionTable["s3", "+"  ] := r3;
actionTable["s3", "-"  ] := r3;
actionTable["s3", "*"  ] := Shift("s12");
actionTable["s3", "/"  ] := Shift("s11");
actionTable["s3", ")"  ] := r3;

actionTable["s4", "EOF"] := Accept();
actionTable["s4", "+"  ] := Shift("s8");
actionTable["s4", "-"  ] := Shift("s9");

actionTable["s5", "("  ] := Shift("s5");
actionTable["s5", "int"] := Shift("s2");

actionTable["s6", "+"  ] := Shift("s8");
actionTable["s6", "-"  ] := Shift("s9");
actionTable["s6", ")"  ] := Shift("s7");

actionTable["s7", "EOF"] := r7;
actionTable["s7", "+"  ] := r7;
actionTable["s7", "-"  ] := r7;
actionTable["s7", "*"  ] := r7;
actionTable["s7", "/"  ] := r7;
actionTable["s7", ")"  ] := r7;
```

Figure 13.3: Action table coded in SetlX, first part.

```
1  actionTable["s8", "("  ] := Shift("s5");
2  actionTable["s8", "int"] := Shift("s2");
3
4  actionTable["s9", "("  ] := Shift("s5");
5  actionTable["s9", "int"] := Shift("s2");
6
7  actionTable["s10", "EOF"] := r2;
8  actionTable["s10", "+" ] := r2;
9  actionTable["s10", "-" ] := r2;
10 actionTable["s10", "*" ] := Shift("s12");
11 actionTable["s10", "/" ] := Shift("s11");
12 actionTable["s10", ")" ] := r2;
13
14 actionTable["s11", "("  ] := Shift("s5");
15 actionTable["s11", "int"] := Shift("s2");
16
17 actionTable["s12", "("  ] := Shift("s5");
18 actionTable["s12", "int"] := Shift("s2");
19
20 actionTable["s13", "EOF"] := r4;
21 actionTable["s13", "+"  ] := r4;
22 actionTable["s13", "-"  ] := r4;
23 actionTable["s13", "*"  ] := r4;
24 actionTable["s13", "/"  ] := r4;
25 actionTable["s13", ")"  ] := r4;
26
27 actionTable["s14", "EOF"] := r5;
28 actionTable["s14", "+"  ] := r5;
29 actionTable["s14", "-"  ] := r5;
30 actionTable["s14", "*"  ] := r5;
31 actionTable["s14", "/"  ] := r5;
32 actionTable["s14", ")"  ] := r5;
33
34 actionTable["s15", "EOF"] := r1;
35 actionTable["s15", "+"  ] := r1;
36 actionTable["s15", "-"  ] := r1;
37 actionTable["s15", "*"  ] := Shift("s12");
38 actionTable["s15", "/"  ] := Shift("s11");
39 actionTable["s15", ")"  ] := r1;
```

Figure 13.4: Action table coded in SetlX, second part.

```
1  gotoTable   := {};
2
3  gotoTable["s0", "E"] := "s4";
4  gotoTable["s0", "P"] := "s3";
5  gotoTable["s0", "F"] := "s1";
6
7  gotoTable["s5", "E"] := "s6";
8  gotoTable["s5", "P"] := "s3";
9  gotoTable["s5", "F"] := "s1";
10
11 gotoTable["s8", "P"] := "s15";
12 gotoTable["s8", "F"] := "s1";
13
14 gotoTable["s9", "P"] := "s10";
15 gotoTable["s9", "F"] := "s1";
16
17 gotoTable["s11", "F"] := "s14";
18 gotoTable["s12", "F"] := "s13";
```

Figure 13.5: Goto table coded in SetlX.

13.3 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 40 (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 $\langle a, \beta, \gamma \rangle$ 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 können. Um diese Idee zu verschaulichen, betrachten wir ein konkretes Beispiel: Wir gehen von der in Abbildung 13.2 auf Seite 172 gezeigten Grammatik aus, wobei wir diese Grammatik noch um ein neues Start-Symbol $\widehat{s}$ und die Regel

$$\widehat{s} \to \mathit{expr}\ \textsc{Eof}$$

erweitern. Der Start-Zustand enthält offenbar die markierte Regel

$$\widehat{s} \to \varepsilon \bullet \mathit{expr}\,\textsc{Eof},$$

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 41 (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 13.2 auf Seite 172 gezeigten Grammatik 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 uns als erstes überlegen, wie wir die Menge Q der Zustände definieren wollen, denn dann funktioniert die Definition der restlichen Komponenten fast von alleine. Die Idee ist, dass wir die Zustände als Mengen von markierten Regeln definieren. Wir definieren zunächst

$$\Gamma := \{a \to \beta \bullet \gamma \mid (a \to \alpha\gamma) \in R\}$$

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 Grammatik-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 $\textsc{Eof}$ zu der Grammatik

$$\widehat{G} = \Big\langle V \cup \{\widehat{s}\}, T \cup \{\textsc{Eof}\}, R \cup \{\widehat{s} \to s\,\textsc{Eof}\}, \widehat{s}\Big\rangle.$$

Das Token $\textsc{Eof}$ steht dabei als Abkürzung für *end of file*. 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:

$$q_0 := \mathit{closure}\Big(\{\widehat{s} \to \bullet s\,\textsc{Eof}\}\Big).$$

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:

$$\mathit{goto}(\mathcal{M}, t) := \mathit{closure}\Big(\{a \to \beta\,t \bullet \delta \mid (a \to \beta \bullet t\,\delta) \in \mathcal{M}\}\Big).$$

Als Letztes spezifizieren wir, wie die Funktion $\mathit{action}(\mathcal{M}, t)$ für eine Menge von markierten Regeln $\mathcal{M}$ und ein Token t berechnet wird. Bei der Definition von $\mathit{action}(\mathcal{M}, t)$ unterscheiden wir vier Fälle.

1. Falls $\mathcal{M}$ eine markierte Regel der Form $a \to \beta \bullet t\,\delta$ enthält, 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 $\mathit{goto}(\mathcal{M}, t)$ berechnet wird. Insgesamt haben wir also

   $$\mathit{action}(\mathcal{M}, t) := \langle \texttt{shift}, \mathit{goto}(\mathcal{M}, t)\rangle \quad \text{falls} \quad (a \to \beta \bullet t\,\delta) \in \mathcal{M}.$$

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 $\widehat{s} \to s \bullet \textsc{Eof}$ enthält und wir den zu parsenden String vollständig gelesen haben, setzen wir

   $$\mathit{action}(\mathcal{M}, \textsc{Eof}) := \texttt{accept},$$

   denn in diesem Fall versucht der Parser, $\widehat{s}$ mit Hilfe der Regel $\widehat{s} \to s\,\textsc{Eof}$ zu erkennen und hat also bereits s erkannt. Ist nun das nächste Token im Eingabe-String das Datei-Ende-Zeichen $\textsc{Eof}$, so liegt der zu parsende String in der durch die Grammatik G spezifizierten Sprache $L(G)$. Wir haben daher

   $$\mathit{action}(\mathcal{M}, \textsc{Eof}) := \texttt{accept}, \quad \text{falls } (\widehat{s} \to s \bullet \textsc{Eof}) \in \mathcal{M}.$$

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. Eine *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 13.2 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

$$\mathit{start} \to \mathit{expr}\ \textsc{Eof}$$

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 wir führen wir die folgenden Abkürzungen ein:

$$s := \mathit{start},\ e := \mathit{expr},\ p := \mathit{product},\ f := \mathit{factor},\ N := \textsc{Number} \text{ und } \$:= \textsc{Eof}.$$

Wir beginnen mit dem Start-Zustand:

1. $s_0 := \mathit{closure}(\{\ s \to \bullet\, e\,\$\ \})$
$= \{\ s \to \bullet\, e\,\$,$
$e \to \bullet\, e\ \text{“+”}\ p,\ e \to \bullet\, e\ \text{“-”}\ p,\ e \to \bullet\, p,$
$p \to \bullet\, p\ \text{“*”}\ f,\ p \to \bullet\, p\ \text{“/”}\ f,\ p \to \bullet\, f,$
$f \to \bullet\ \text{“(”}\, e\ \text{“)”}\ ,\ f \to \bullet\, N\ \}.$

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. $s_3 := \mathit{goto}(s_0, p)$
$= \mathit{closure}(\{p \to p\bullet\ \text{“*”}\ f,\ p \to p\bullet\ \text{“/”}\ f,\ e \to p\bullet\})$
$= \{p \to p\bullet\ \text{“*”}\ f,\ p \to p\bullet\ \text{“/”}\ f, e \to p\bullet\}.$

5. $s_4 := \mathit{goto}(s_0, e)$
$= \mathit{closure}(\{s \to e\bullet\ \$,\ e \to e\bullet\ \text{“+”}\ p,\ e \to e\bullet\ \text{“-”}\ p\ \})$
$= \{\ s \to e\bullet\ \$,\ e \to e\bullet\ \text{“+”}\ p,\ e \to e\bullet\ \text{“-”}\ p\ \}.$

6. $s_5 := \mathit{goto}(s_0,\ \text{“(”}\ )$
$= \mathit{closure}(\{f \to\ \text{“(”}\ \bullet\, e\ \text{“)”}\ \})$
$= \{\ f \to\ \text{“(”}\ \bullet\, e\ \text{“)”}$
$e \to \bullet\, e\ \text{“+”}\ p,\ e \to \bullet\, e\ \text{“-”}\ p,\ e \to \bullet\, p,$
$p \to \bullet\, p\ \text{“*”}\ f,\ p \to \bullet\, p\ \text{“/”}\ f,\ p \to \bullet\, f,$
$f \to \bullet\ \text{“(”}\, e\ \text{“)”}\ ,\ f \to \bullet\, N\ \}.$

7. $s_6 := \mathit{goto}(s_5, e)$
$= \mathit{closure}(\{\ f \to\ \text{“(”}\ e\bullet\ \text{“)”}\ ,\ e \to e\bullet\ \text{“+”}\ p,\ e \to e\bullet\ \text{“-”}\ p\ \})$
$= \{\ f \to\ \text{“(”}\ e\bullet\ \text{“)”}\ ,\ e \to e\bullet\ \text{“+”}\ p,\ e \to e\bullet\ \text{“-”}\ p.\ \}$

8. $s_7 := \mathit{goto}(s_6,\ \text{“)”}\ )$
$= \mathit{closure}(\{\ f \to\ \text{“(”}\ e\ \text{“)”}\ \bullet\ \})$
$= \{\ f \to\ \text{“(”}\ e\ \text{“)”}\ \bullet\ \}.$

9. $s_8 := \mathit{goto}(s_4, \text{“+”}) = \mathit{closure}(\{e \to e \text{“+”} \bullet p\}) = \{ e \to e \text{“+”} \bullet p,\ p \to \bullet p \text{“*”} f,\ p \to \bullet p \text{“/”} f,\ p \to \bullet f,\ f \to \bullet \text{“(”} e \text{“)”},\ f \to \bullet N \}.$

10. $s_9 := \mathit{goto}(s_4, \text{“-”}) = \mathit{closure}(\{e \to e \text{“-”} \bullet p\}) = \{ e \to e \text{“-”} \bullet p,\ p \to \bullet p \text{“*”} f,\ p \to \bullet p \text{“/”} f,\ p \to \bullet f,\ f \to \bullet \text{“(”} e \text{“)”},\ f \to \bullet N \}.$

11. $s_{10} := \mathit{goto}(s_9, p) = \mathit{closure}(\{e \to e \text{“-”} p\bullet,\ p \to p \bullet \text{“*”} f,\ p \to p \bullet \text{“/”} f\}) = \{ e \to e \text{“-”} p\bullet,\ p \to p \bullet \text{“*”} f,\ p \to p \bullet \text{“/”} f \}.$

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

$$\begin{aligned} & \mathit{goto}(s_8, \text{“(”}) \\ = \ & \mathit{closure}(\{f \to \text{“(”} \bullet e \text{“)”}\}) \\ = \ & \{ f \to \text{“(”} \bullet e \text{“)”},\ e \to \bullet e \text{“+”} p,\ e \to \bullet e \text{“-”} p,\ e \to \bullet p, \\ & p \to \bullet p \text{“*”} f,\ p \to \bullet p \text{“/”} f,\ p \to \bullet f, \\ & f \to \bullet \text{“(”} e \text{“)”},\ f \to \bullet N \} \\ = \ & s_5. \end{aligned}$$

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 *Follow*(e) auftritt. Falls nun "*" $\in$ *Follow*(e) liegen würde, so hätten wir einen Shift-Reduce-Konflikt. Es gilt aber

$$\mathit{Follow}(e) = \{ \text{ "+" }, \text{ "–" }, \text{ ")" }, \text{ "\$" } \},$$

und daraus folgt "*" $\notin$ *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 *action*$(s_1,$ "+" $)$. Es gilt

$$\begin{array}{rcl} \mathit{action}(s_1, \text{ "+" }) & = & \mathit{action}(\{p \to f\bullet\}, \text{ "+" }) \\ & = & \langle \mathit{reduce}, p \to f \rangle, \end{array}$$

   denn wir haben "+" $\in$ *Follow*(P).

2. Als nächstes berechnen wir *action*$(s_4,$ "+" $)$. Es gilt

$$\begin{array}{rcl} \mathit{action}(s_4, \text{ "+" }) & = & \mathit{action}(\{s \to e \bullet \$,\ e \to e \bullet \text{ "+" } p,\ e \to e \bullet \text{ "–" } p \}, \text{ "+" }) \\ & = & \langle \mathit{shift}, \mathit{closure}(\{e \to e \text{ "+" } \bullet p\}) \rangle \\ & = & \langle \mathit{shift}, s_8 \rangle. \end{array}$$

Würden wir diese Rechnungen fortführen, so würden wir die Tabelle 13.1 erhalten, denn wir haben die Namen der Zustände so gewählt, dass diese mit den Namen der entsprechenden Zustände in den Tabellen 13.1 und 13.2 übereinstimmen.

Aufgabe 36: Berechnen Sie die Menge der SLR-Zustände für die in Abbildung 13.6 gezeigte Grammatik und geben Sie die Funktionen *action* und *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. ◇

conjunction	→	*conjunction* "&" *disjunction*
	|	*disjunction*
disjunction	→	*disjunction* "|" *literal*
	|	*literal*
literal	→	"!" IDENTIFIER
	|	IDENTIFIER

Figure 13.6: Eine Grammatik für Boole'sche Ausdrücke in konjunktiver Normalform.

13.3.1 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 13.7 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 NUMBER, so können wir mit dieser Grammatik den Ausdruck $1 + 2 * 3$ sowohl als

$(1 + 2) * 3$ als auch als $1 + (2 * 3)$

lesen.

$$\begin{array}{rcl} e & \to & e \text{ “+” } e \\ & | & e \text{ “*” } e \\ & | & N \end{array}$$

Figure 13.7: Eine Grammatik mit Shift-Reduce-Konflikten.

Wir berechnen zunächst den Start-Zustand s_0.

$$\begin{array}{rcl} s_0 & = & \mathit{closure}\big(\{s \to \bullet\, e\, \$\}\big) \\ & = & \big\{s \to \bullet\, e\, \$,\ e \to \bullet\, e \text{ “+” } e,\ e \to \bullet\, e \text{ “*” } e,\ e \to \bullet\, N\ \big\}. \end{array}$$

Als nächstes berechnen wir $s_1 := \mathit{goto}(s_0, e)$:

$$\begin{array}{rcl} s_1 & = & \mathit{goto}(s_0, e) \\ & = & \mathit{closure}\big(\{s \to e \bullet\ \$\ e \to e \bullet \text{ “+” } e,\ e \to e \bullet \text{ “*” } e\ \}\big) \\ & = & \big\{s \to e \bullet\ \$,\ e \to e \bullet \text{ “+” } e,\ e \to e \bullet \text{ “*” } e\ \}\big) \end{array}$$

Nun berechnen wir $s_2 := \mathit{goto}(s_1, \text{“+”})$:

$$\begin{array}{rcl} s_2 & = & \mathit{goto}(s_1, \text{“+”}) \\ & = & \mathit{closure}\big(\{e \to e \text{ “+” } \bullet\, e,\ \}\big) \\ & = & \big\{e \to e \text{ “+” } \bullet\, e,\ e \to \bullet\, e \text{ “+” } e,\ e \to \bullet\, e \text{ “*” } e,\ e \to \bullet\, N\ \}\big) \end{array}$$

Als nächstes berechnen wir $s_3 := \mathit{goto}(s_2, e)$:

$$\begin{array}{rcl} s_3 & = & \mathit{goto}(s_2, e) \\ & = & \mathit{closure}\big(\{e \to e \text{ “+” } e\bullet,\ e \to e \bullet \text{ “+” } e,\ e \to e \bullet \text{ “*” } e\}\big) \\ & = & \big\{e \to e \text{ “+” } e\bullet,\ e \to e \bullet \text{ “+” } e,\ e \to e \bullet \text{ “*” } e\ \}\big) \end{array}$$

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

$$\mathit{Follow}(e) = \{ \text{ “+”}, \text{ “*”}, \text{ “\$” } \},$$

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.

Aufgabe 37: Bei der in Abbildung 13.7 gezeigten Grammatik treten noch weitere Shift-Reduce-Konflikte auf. Berechnen Sie alle Zustände und geben Sie dann die restlichen Shift-Reduce-Konflikte an. ◇

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. ◇

$$
\begin{array}{lcl}
s & \to & a \text{ “x” } a \text{ “y”} \\
  & \mid & b \text{ “y” } b \text{ “x”} \\
a & \to & \varepsilon \\
b & \to & \varepsilon
\end{array}
$$

Figure 13.8: 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 13.8 gezeigte Grammatik. Diese Grammatik ist eindeutig, denn es gilt

$$L(s) = \{ \text{ “xy”}, \text{ “yx” } \}$$

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.

$$
\begin{array}{lcl}
s_0 & = & \mathit{closure}\big(\{\widehat{s} \to \bullet\, s\,\$\}\big) \\
    & = & \big\{\widehat{s} \to \bullet\, s\,\$,\ s \to \bullet\, a \text{ “x” } a \text{ “y” },\ s \to \bullet\, b \text{ “y” } b \text{ “x” },\ a \to \bullet\, \varepsilon,\ b \to \bullet\, \varepsilon\ \big\} \\
    & = & \big\{\widehat{s} \to \bullet\, s\,\$,\ s \to \bullet\, a \text{ “x” } a \text{ “y” },\ s \to \bullet\, b \text{ “y” } b \text{ “x” },\ a \to \varepsilon\bullet,\ b \to \varepsilon\bullet\ \big\},
\end{array}
$$

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

$$a \to \bullet\, \varepsilon \quad \text{und} \quad b \to \varepsilon\bullet.$$

Dieser Konflikt tritt bei der Berechnung von

$$\mathit{action}(s_0, \text{ “x” })$$

auf, denn wir haben

$$\mathit{Follow}(a) = \big\{\text{“x”}, \text{“y”}\big\} = \mathit{Follow}(b).$$

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 $\mathit{Follow}(b)$.

Es ist interessant zu bemerken, dass die obige Grammatik die $LL(1)$-Eigenschaft hat, denn es gilt

$$\mathit{First}(a \text{ “x” } a \text{ “y” }) = \{ \text{ “x” } \}, \quad \mathit{First}(b \text{ “y” } b \text{ “x” }) = \{ \text{ “y” } \}.$$

und daraus folgt sofort

$$\mathit{First}(a \text{ “x” } a \text{ “y” }) \cap \mathit{First}(b \text{ “y” } b \text{ “x” }) = \{\text{“x”}\} \cap \{\text{“y”}\} = \{\}.$$

Dieses Beispiel zeigt, dass SLR-Grammatiken im Allgemeinen nicht ausdruckstärker sind als LL(1)-Grammatiken. In der Praxis zeigt sich jedoch, dass viele Grammatiken, die nicht die LL(1)-Eigenschaft haben, SLR-Grammatiken sind.

Remark: As part of the resources provided with this lecture, the file slr-table-generator.stlx contains a SetlX-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. ◇

13.4 Kanonische LR-Parser

Der Reduce-Reduce-Konflikt, der in der in Abbildung 13.8 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 42 (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 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 43 (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 13.9 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  procedure 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 13.9: 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 $\textsc{Eof}$ hinzu. Dann hat der Start-Zustand die Form

$$q_0 := \mathit{closure}\big(\{\widehat{s} \to \bullet\, s : \textsc{Eof}\}\big),$$

denn auf das Start-Symbol muss das Datei-Ende "EOF" folgen. Als letztes zeigen wir, wie die Definition der Funktion *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. $(\widehat{s} \to s\bullet : \text{EOF}) \in \mathcal{M} \implies \mathit{action}(\mathcal{M}, \text{EOF}) := \texttt{accept}$.
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 13.8 gezeigten Grammatik wieder auf und berechnen zunächst die Menge aller Zustände. Um die Schreibweise zu vereinfachen, schreiben wir an Stelle von "EOF" kürzer "$".

1. $s_0 := \mathit{closure}\Big(\big\{\widehat{s} \to \bullet\, s : \$\big\}\Big)$
   $= \big\{\widehat{s} \to \bullet\, s : \$, s \to \bullet\, a\,\text{“x”}\,a\,\text{“y”} : \$, s \to \bullet\, b\,\text{“y”}\,b\,\text{“x”} : \$, a \to \bullet : \text{“x”}, b \to \bullet : \text{“y”}\big\}$.
2. $s_1 := \mathit{goto}(s_0, a)$
   $= \mathit{closure}\big(\big\{s \to a \bullet \text{“x”}\,a\,\text{“y”} : \$\big\}\big)$
   $= \big\{s \to a \bullet \text{“x”}\,a\,\text{“y”} : \$\big\}$.
3. $s_2 := \mathit{goto}(s_0, s)$
   $= \mathit{closure}\big(\big\{\widehat{s} \to s\bullet : \$\big\}\big)$
   $= \big\{\widehat{s} \to s\bullet : \$\big\}$.
4. $s_3 := \mathit{goto}(s_0, b)$
   $= \mathit{closure}\big(\big\{s \to b \bullet \text{“y”}\,b\,\text{“x”} : \$\big\}\big)$
   $= \big\{s \to b \bullet \text{“y”}\,b\,\text{“x”} : \$\big\}$.
5. $s_4 := \mathit{goto}(s_3, \text{“y”})$
   $= \mathit{closure}\big(\big\{s \to b\,\text{“y”} \bullet b\,\text{“x”} : \$\big\}\big)$
   $= \big\{s \to b\,\text{“y”} \bullet b\,\text{“x”} : \$, b \to \bullet : \text{“x”}\big\}$.
6. $s_5 := \mathit{goto}(s_4, b)$
   $= \mathit{closure}\big(\big\{s \to b\,\text{“y”}\,b \bullet \text{“x”} : \$\big\}\big)$
   $= \big\{s \to b\,\text{“y”}\,b \bullet \text{“x”} : \$\big\}$.

7. $s_6 := \mathit{goto}(s_5, \text{“x”}) = \mathit{closure}(\{s \to b\,\text{“y”}\,b\,\text{“x”}\,\bullet : \$\}) = \{s \to b\,\text{“y”}\,b\,\text{“x”}\,\bullet : \$\}.$

8. $s_7 := \mathit{goto}(s_1, \text{“x”}) = \mathit{closure}(\{s \to a\,\text{“x”}\,\bullet\,a\,\text{“y”} : \$\}) = \{s \to a\,\text{“x”}\,\bullet\,a\,\text{“y”} : \$, a \to \bullet : \text{“y”}\}.$

9. $s_8 := \mathit{goto}(s_7, a) = \mathit{closure}(\{s \to a\,\text{“x”}\,a\,\bullet\,\text{“y”} : \$\}) = \{s \to a\,\text{“x”}\,a\,\bullet\,\text{“y”} : \$\}.$

10. $s_9 := \mathit{goto}(s_8, \text{“y”}) = \mathit{closure}(\{s \to a\,\text{“x”}\,a\,\text{“y”}\,\bullet : \$\}) = \{s \to a\,\text{“x”}\,a\,\text{“y”}\,\bullet : \$\}.$

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

$$\mathit{Follow}(a) \cap \mathit{Follow}(b) = \{\text{“x”}, \text{“y”}\} \neq \{\}$$

gilt. Dieser Konflikt ist nun verschwunden, denn zwischen den e.m.R.s

$$a \to \bullet : \text{“x”} \quad \text{und} \quad b \to \bullet : \text{“y”}$$

gibt es wegen “x” $\neq$ “y” keinen Konflikt. Es ist leicht zu sehen, dass auch bei den anderen Zustände keine Konflikte auftreten.

Aufgabe 38: Berechnen Sie die Menge der Zustände eines LR-Parsers für die folgende Grammatik:

$$\begin{array}{rcl} e & \to & e\ \text{“+”}\ p \\ & | & p \\ p & \to & p\ \text{“*”}\ f \\ & | & f \\ f & \to & \text{“(”}\,e\,\text{“)”} \\ & | & \mathit{Number} \end{array}$$

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 lr-parser-generator.stlx contains a SetlX-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”. ◇

13.5 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. Anfangs, als der zur Verfügung stehenden Haupt-Speicher der meisten Rechner noch bescheidener dimensioniert waren, 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 13.10 gezeigt Grammatik, die ich dem *Drachenbuch* [ASUL06] entnommen habe. (Das "Drachenbuch" ist das Standardwerk im Bereich Compilerbau.)

$$
\begin{array}{lcl}
\widehat{s} & \to & s \\
s & \to & c\ c \\
c & \to & \text{“x”}\ c \\
 & | & \text{“y”}
\end{array}
$$

Figure 13.10: Eine Grammatik aus dem Drachenbuch.

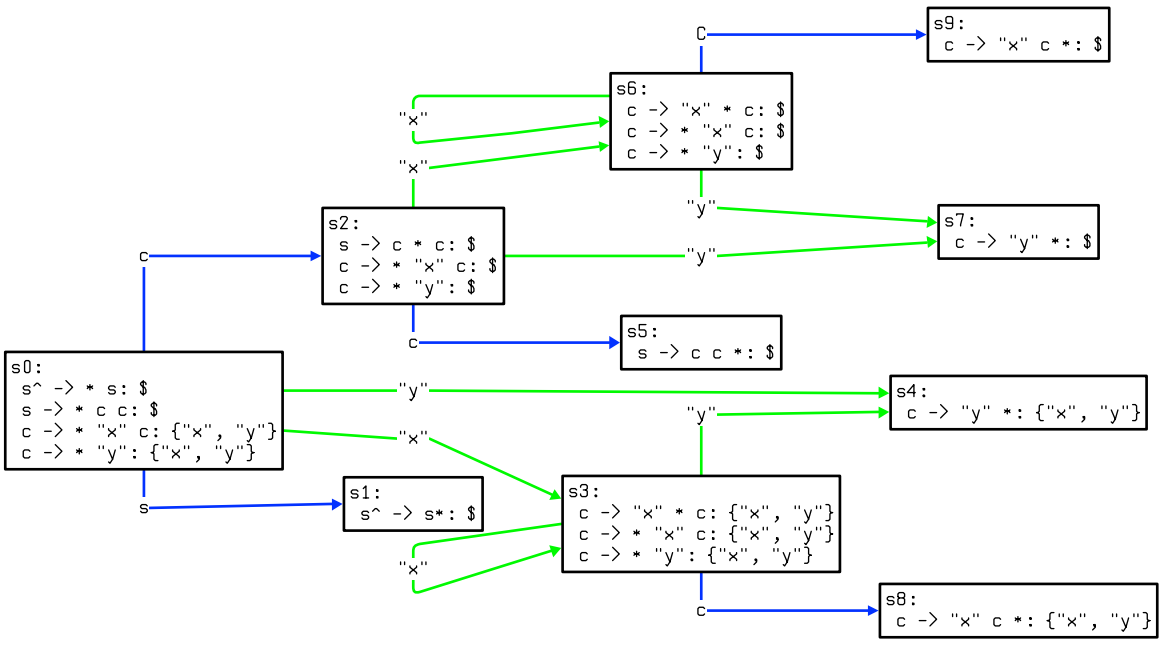


Figure 13.11: LR-Goto-Graph für die Grammatik aus Abbildung 13.10.

Abbildung 13.11 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

$$s_6 = \Big\{ s \to \text{“x”} \bullet c : \text{“\$”}, c \to \bullet\ \text{“x”}\ c : \text{“\$”}, c \to \bullet\ \text{“y”}\ : \text{“\$”} \Big\},$$

und andererseits haben wir

$$s_3 = \Big\{ s \to \text{“x”} \bullet c : \{\text{“x”}, \text{“y”}\}, c \to \bullet \ \text{“x”} \ c : \{\text{“x”}, \text{“y”}\}, c \to \bullet \ \text{“y”} \ : \{\text{“x”}, \text{“y”}\} \Big\}.$$

Offenbar entsteht die Menge s_3 aus der Menge s_6 indem überall “$” durch die Menge $\{\text{“x”}, \text{“y”}\}$ ersetzt wird. Genauso 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

$$\mathit{goto}(s_3, c) = s_8 \quad \text{und} \quad \text{und } \mathit{goto}(s_6, c) = s_9$$

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 $\{\text{“x”}, \text{“y”}\}$ 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 13.12 gezeigten Goto-Graphen.

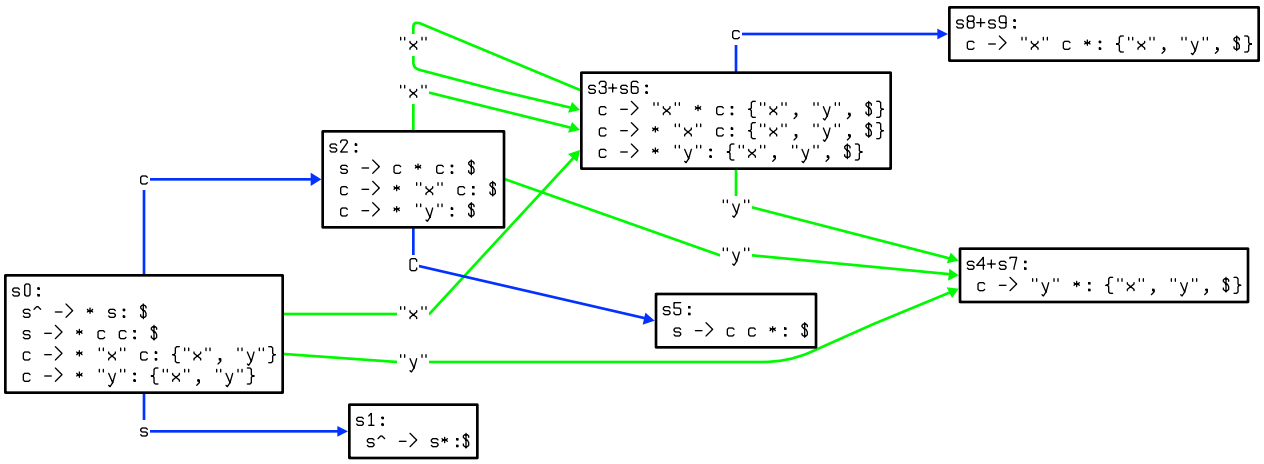


Figure 13.12: Der LALR-Goto-Graph für die Grammatik aus Abbildung 13.10.

Um die Beobachtungen, die wir bei der Betrachtung der in Abbildung 13.10 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 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 $\mathit{arb}()$ 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 $\mathit{goto}(\mathcal{M}, x)$ wohldefiniert. Die Berechnung des Ausdrucks $\mathit{action}(\mathcal{M}, t)$ ändert sich gegenüber der Berechnung für einen LR-Parser nicht.

13.6 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:

$$\textit{SLR-Sprache} \subsetneq \textit{LALR-Sprache} \subsetneq \textit{kanonische LR-Sprache} \qquad (\star)$$

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, für die die Menge der Folge-Token identisch war. 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.

13.6.1 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 185 gesehen, dass die Grammatik

$$s \;\to\; a \text{ “x” } a \text{ “y” } \mid b \text{ “y” } b \text{ “x” }, \quad a \;\to\; \varepsilon, \quad b \;\to\; \varepsilon$$

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. $s_0 = \{\widehat{s} \to \bullet\, s : \$, s \to \bullet\, a\,\text{“x”}\,a\,\text{“y”} : \$, s \to \bullet\, b\,\text{“y”}\,b\,\text{“x”} : \$, a \to \bullet : \text{“x”}, b \to \bullet : \text{“y”}\}$,
2. $s_1 = \{s \to a \bullet\, \text{“x”}\,a\,\text{“y”} : \$\}$,
3. $s_2 = \{\widehat{s} \to s\bullet : \$\}$,
4. $s_3 = \{s \to b \bullet\, \text{“y”}\,b\,\text{“x”} : \$\}$,
5. $s_4 = \{s \to b\,\text{“y”}\,\bullet\, b\,\text{“x”} : \$, b \to \bullet : \text{“x”}\}$,
6. $s_5 = \{s \to b\,\text{“y”}\,b \bullet\, \text{“x”} : \$\}$,
7. $s_6 = \{s \to b\,\text{“y”}\,b\,\text{“x”}\bullet : \$\}$,
8. $s_7 = \{s \to a\,\text{“x”}\,\bullet\, a\,\text{“y”} : \$, a \to \bullet : \text{“y”}\}$,
9. $s_8 = \{s \to a\,\text{“x”}\,a \bullet\, \text{“y”} : \$\}$,
10. $s_9 = \{s \to a\,\text{“x”}\,a\,\text{“y”}\bullet : \$\}$.

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.

13.6.2 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 13.13 zeigt eine solche Grammatik, die ich dem Drachenbuch entnommen habe.

$$
\begin{array}{rcl}
s & \to & \text{“v”}\ a\ \text{“y”} \\
 & | & \text{“w”}\ b\ \text{“y”} \\
 & | & \text{“v”}\ b\ \text{“z”} \\
 & | & \text{“w”}\ a\ \text{“z”} \\
a & \to & \text{“x”} \\
b & \to & \text{“x”}
\end{array}
$$

Figure 13.13: 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 folgende Mengen von erweiterten markierten Regeln:

1. $s_0 = \mathit{closure}(\widehat{s} \to \bullet\ s : \$) = \{\ \begin{array}{rcl} \widehat{s} & \to & \bullet\, s : \$, \\ s & \to & \bullet\,\text{“v”}\,a\,\text{“y”} : \$, \\ s & \to & \bullet\,\text{“v”}\,b\,\text{“z”} : \$, \\ s & \to & \bullet\,\text{“w”}\,a\,\text{“z”} : \$, \\ s & \to & \bullet\,\text{“w”}\,b\,\text{“y”} : \$\ \}, \end{array}$
2. $s_1 = \mathit{goto}(s_0, s) = \{\widehat{s} \to s\bullet : \$\}$

3. $s_2 = \mathit{goto}(s_0,\ \text{“v”}\ ) = \{\ s \ \to\ \text{“v”} \bullet b\,\text{“z”} : \$,$
$s \ \to\ \text{“v”} \bullet a\,\text{“y”} : \$,$
$a \ \to\ \bullet\,\text{“x”} : \text{“y”},$
$b \ \to\ \bullet\,\text{“x”} : \text{“z”}\ \},$
4. $s_3 = \mathit{goto}(s_0,\ \text{“w”}\ ) = \{\ s \ \to\ \text{“w”} \bullet a\,\text{“z”} : \$,$
$s \ \to\ \text{“w”} \bullet b\,\text{“y”} : \$,$
$a \ \to\ \bullet\,\text{“x”} : \text{“z”},$
$b \ \to\ \bullet\,\text{“x”} : \text{“y”}\ \},$
5. $s_4 = \mathit{goto}(s_2,\ \text{“x”}\ ) = \{a \to \text{“x”}\bullet : \text{“y”},\ b \to \text{“x”}\bullet : \text{“z”}\},$
6. $s_5 = \mathit{goto}(s_3,\ \text{“x”}\ ) = \{a \to \text{“x”}\bullet : \text{“z”},\ b \to \text{“x”}\bullet : \text{“y”}\},$
7. $s_6 = \mathit{goto}(s_2, a) = \{s \to \text{“v”}\,a \bullet \text{“y”} : \$\},$
8. $s_7 = \mathit{goto}(s_6,\ \text{“y”}\ ) = \{s \to \text{“v”}\,a\,\text{“y”}\bullet : \$\},$
9. $s_8 = \mathit{goto}(s_2, b) = \{s \to \text{“v”}\,b \bullet \text{“z”} : \$\},$
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 39: 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.

13.6.3 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 14

Der Parser-Generator Cup

LALR-Parser erlauben es, die Grammatiken vieler Programmiersprachen in natürlicher Weise zu parsen. Da die meisten von Ihnen in der Praxis vermutlich mit *Java* arbeiten, möchte ich Ihnen in diesem Kapitel einen LALR-Parser-Generator vorstellen, der einen Parser für die Programmiersprache *Java* erzeugt. Dieses Kapitel gibt daher eine Einführung in die Verwendung des Parser-Generators CUP [HFA+99], der auch unter dem Namen *JavaCup* bekannt ist. CUP selber ist nur ein Parser-Generator. Da wir zusätzlich einen Scanner benötigen, werden wir diesen in unseren Beispielen von *JFlex* erzeugen lassen. Wir werden die Version 0.11b des Parser-Generators CUP verwenden. Sie finden diese Version im Netz unter

`http://www2.cs.tum.edu/projects/cup/`.

Sie finden dort eine komprimierte "`.tar`"-Datei, in der sich zwei "`.jar`"-Dateien befinden.

1. Die Datei `java-cup-11b.jar` beinhaltet das ausführbare Programm.
2. Die Datei `java-cup-11b-runtime.jar` enthält die Klassen, die Sie beim Übersetzen des von CUP erzeugten Parsers benötigen.

14.1 Spezifikation einer Grammatik für Cup

Eine CUP-Spezifikation besteht aus vier Teilen.

1. Der erste Teil enthält eine (optionale) Paket-Deklaration sowie die benötigten Import-Deklarationen. Außerdem können wir hier den Namen der Klasse spezifizieren, die den von CUP erzeugten Parser enthält.
2. Der zweite Teil deklariert die verwendeten Symbole. Hier werden also die Terminale und die syntaktischen Variablen spezifiziert.
3. Der dritte Teil ist wieder optional und spezifiziert die Präzedenzen und Assoziativitäten der verwendeten Operator-Symbolen.
4. Der vierte Teil enthält die Grammatik-Regeln.

Abbildung 14.1 auf Seite 197 zeigt eine *Cup*-Spezifikation, mit deren Hilfe arithmetische Ausdrücke ausgewertet werden können. In dieser *Cup*-Spezifikation sind die Schlüsselwörter unterstrichen.

1. Die gezeigte CUP-Spezifikation enthält keine Paket-Deklarationen.
2. Die Spezifikation beginnt in Zeile 1 mit dem Import der Klassen von `java_cup.runtime`. Dieses Paket muss immer importiert werden, denn dort wird beispielsweise die Klasse `Symbol` definiert, die wir auch später noch in dem in Abbildung 14.4 gezeigten Scanner verwenden werden.

   Würden noch weitere Pakete benötigt, so könnten diese hier ebenfalls importiert werden.
3. In Zeile 2 spezifizieren wir, dass die erzeugte Parser-Klasse den Namen `ExprParser` haben soll.

```
1  import java_cup.runtime.*;
2  class ExprParser;
3
4  terminal            SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
5  terminal            LPAREN, RPAREN;
6  terminal Integer    NUMBER;
7
8  nonterminal         expr_list, expr_part;
9  nonterminal Integer expr, prod, fact;
10
11 start with expr_list;
12
13 expr_list ::= expr_list expr_part
14             |  expr_part
15             ;
16
17 expr_part ::= expr:e {: System.out.println("result = " + e); :} SEMI
18             ;
19
20 expr ::= expr:e PLUS  prod:p {: RESULT = e + p; :}
21       |  expr:e MINUS prod:p {: RESULT = e - p; :}
22       |  prod:p              {: RESULT = p;     :}
23       ;
24
25 prod ::= prod:p TIMES  fact:f {: RESULT = p * f; :}
26       |  prod:p DIVIDE fact:f {: RESULT = p / f; :}
27       |  prod:p MOD    fact:f {: RESULT = p % f; :}
28       |  fact:f               {: RESULT = f;     :}
29       ;
30
31 fact ::= LPAREN expr:e RPAREN {: RESULT = e;   :}
32       |  MINUS  fact:e        {: RESULT = - e; :}
33       |  NUMBER:n             {: RESULT = n;   :}
34       ;
```

Figure 14.1: CUP-Spezifikation eines Parsers für arithmetische Ausdrücke

4. In den Zeilen 4 bis 6 werden die Terminale deklariert. Es gibt zwei Arten von Terminalen:

   1. Terminale, die keinen zusätzlichen Wert haben. Hierbei handelt es sich um die Operator-Symbole, die beiden Klammer-Symbole und das Semikolon. Die Syntax zur Deklaration solcher Terminale ist

      `terminal` $t_1, \cdots, t_n$;

      Durch diese Deklaration werden die Symbole $t_1, \cdots, t_n$ als Terminale deklariert.

   2. Terminale, denen ein Wert zugeordnet ist. In diesem Fall muss der Typ dieses Wertes deklariert werden. Die Syntax dafür ist

      `terminal` *type* $t_1, \cdots, t_n$;

      Hierbei spezifiziert *type* den Typ, der den Terminalen $t_1, \cdots, t_n$ zugeordnet wird.

      Bei der Spezifikation eines Typs ist es wichtig zu beachten, dass zwischen dem Typ und dem ersten Terminal kein Komma steht, denn sonst würde der Typ ebenfalls als Terminal interpretiert.

      In Zeile 6 spezifizieren wir beispielsweise, dass dem Terminal `NUMBER` ein Wert vom Typ `Integer` zuge-

ordnet ist. Damit das funktioniert, muss der Scanner jedes Mal, wenn er ein Terminal `NUMBER` an den Parser zurück geben soll, ein Objekt der Klasse `Symbol` erzeugen, dass die entsprechende Zahl als Wert beinhaltet. In dem auf Seite 201 in Abbildung 14.4 gezeigten Scanner geschieht dies beispielsweise in dadurch, dass mit Hilfe der Methode `symbol()` der Konstruktor der Klasse `Symbol` aufgerufen wird, dem als zusätzliches Argument der Wert der Zahl übergeben wird.

5. In den Zeilen 8 und 9 werden die syntaktischen Variablen, die wir auch als Nicht-Terminale bezeichnen, deklariert. Die Syntax ist dieselbe wie bei der Deklaration der Terminale, nur dass wir jetzt das Schlüsselwort "`nonterminal`" an Stelle von "`terminal`" verwenden. Auch hier gibt es wieder zwei Fälle: Den in Zeile 8 deklarierten Nicht-Terminalen `expr_list` und `expr_part` wird kein Wert zugeordnet, während wir dem Nicht-Terminal *expr* einen Wert vom Typ `Integer` zuordnen, der sich aus der Auswertung des entsprechenden arithmetischen Ausdrucks ergibt.

6. Der letzte Teil einer CUP-Grammatik-Spezifikation enthält die Grammatik-Regeln. Die allgemeine Form einer CUP-Grammatik-Regel ist

   var "`::=`" $body_1$ "`{:`" $action_1$ "`:}`"
   "`|`" $body_2$ "`{:`" $action_2$ "`:}`"
   ⋮
   "`|`" $body_n$ "`{:`" $action_n$ "`:}`"
   "`;`"

   Dabei gilt

   1. *var* ist die syntaktische Variable, die von dieser Regel erzeugt wird.
   2. $body_i$ ist der Rumpf der i-ten Grammatik-Regel, der aus einer Liste von Terminalen und syntaktischen Variablen besteht.
   3. $action_i$ ist eine durch Semikolons getrennte Folge von *Java*-Anweisungen, die ausgeführt werden, falls der Stack des Shift/Reduce-Parsers, der die Symbole enthält, mit der zugehörigen Regel reduziert wird.

   Bei der Spezifikation einer Grammatik-Regel mit CUP weicht die Syntax von der Syntax, die in ANTLR verwendet wird, an mehreren Stellen ab:

   1. Statt eines einfachen Doppelpunkts "`:`" wird die Zeichenreihe "`::=`" verwendet, um die zu definierende Variable vom Rumpf der Grammatik-Regel zu trennen.
   2. Die Kommandos werden bei CUP in den Zeichenreihen "`{:`" und "`:}`" eingeschlossen, während bei ANTLR die geschweiften Klammern "{" und "}" ausgereicht haben.
   3. Um auf die Werte, die einem Terminal oder einer syntaktischen Variablen zugeordnet sind, zuzugreifen, hatten wir bei ANTLR Zuweisungen verwendet. Stattdessen müssen wir nun jedem Symbol, dessen Wert wir verwenden wollen, eine eigene Variable zuordnen, deren Namen wir getrennt von einem Doppelpunkt hinter das Symbol schreiben. Um der durch die Grammatik-Regel definierten syntaktischen Variablen einen Wert zuzuweisen, verwenden wir das Schlüsselwort "`RESULT`". Betrachten wir als Beispiel die folgende Regel:

      ```
      expr ::= expr:e PLUS prod:p {:  RESULT = e + p; :};
      ```

      Mit `expr:e` sucht der Parser nach einem arithmetischen Ausdruck, dessen Wert in der Variablen `e` gespeichert wird. Anschließend wird ein Plus-Zeichen gelesen und darauf folgt wieder ein Produkt, dessen Wert jetzt in `p` gespeichert wird. Der Wert des insgesamt gelesenen Ausdrucks wird dann durch die Zuweisung

      ```
      RESULT = e + p;
      ```

      berechnet und der linken Seite der Grammatik-Regel zugewiesen.
   4. Ein weiterer Unterschied zwischen CUP und ANTLR besteht darin, dass Nicht-Terminale nicht mehr durch den ihnen zugeordneten String repräsentiert werden können. Daher können wir den String "`PLUS`" nicht durch den String "`'+'`" ersetzen. An dieser Stelle sind CUP-Grammatiken leider nicht so gut lesbar wie

die entsprechenden ANTLR-Grammatiken. Der Grund dafür ist, dass bei CUP zum Scannen ein separates Werkzeug, nämlich *JFlex*, zum Scannen verwendet wird. Demgegenüber wird bei ANTLR der Scanner zusammen mit dem Parser in derselben Datei spezifiziert und das Werkzeug ANTLR erzeugt aus dieser Datei sowohl einen Parser als auch einen Scanner.

5. Ein weiterer wichtiger Unterschied zwischen CUP und ANTLR besteht darin, dass CUP im Gegensatz zu ANTLR keine EBNF-Grammatiken akzeptiert sondern nur gewöhnliche kontextfreie Grammatiken verarbeiten kann. Damit stehen uns die Operatoren "*", "+" und "?" bei der Spezifikation der Grammatik nicht zur Verfügung.

Um aus der in Abbildung 14.1 gezeigten CUP-Spezifikation einen Parser zu erzeugen, müssen wir diese zunächst mit dem Befehl

```
java -jar /usr/local/lib/java-cup-11b.jar calc.cup
```

übersetzen. Damit dies funktioniert müssen Sie die Datei

```
java-cup-11b.jar
```

in dem Verzeichnis

```
/usr/local/lib
```

ablegen. Sollten Sie unter *Windows* arbeiten, müssen Sie stattdessen ein geeignetes anderes Verzeichnis wählen. Eine Möglichkeit, den Aufruf zu vereinfachen, besteht unter Unix darin, dass Sie sich eine Datei cup mit folgendem Inhalt irgendwo in Ihrem Pfad ablegen:

```
#!/bin/bash
java -jar /usr/local/lib/java-cup-11b.jar  $@
```

Unter *Windows* können Sie sich stattdessen eine entsprechende `.bat`-Datei anlegen. Danach können Sie CUP einfacher mit dem Befehl

cup *calc.cup*

aufrufen. Dieser Befehl erzeugt verschiedene *Java*-Dateien.

1. Die Datei `ExprParser.java` enthält die Klasse `ExprParser`, die den eigentlichen Parser enthält. Wenn Sie diese Klassse später übersetzen wollen, müssen Sie dafür sorgen, dass die Datei `java-cup-11b-runtime.jar` im `CLASSPATH` liegt.

2. Die Datei `ExprParserSym.java` enthält die Klasse `ExprParserSym`, welche die verschiedenen Symbole als statische Konstanten der Klasse `ExprParserSym` definiert. Diese Konstanten werden später im von *JFlex* erzeugten Scanner verwendet. Abbildung 14.2 auf Seite 200 zeigt diese Klasse.

Neben dem Parser wird noch ein Scanner benötigt. Diesen werden wir im nächsten Abschnitt präsentieren. Um mit dem Parser arbeiten zu können, brauchen wir eine Klasse, welche die Methode `main()` enthält. Abbildung 14.3 auf Seite 200 zeigt eine solche Klasse. Wir erzeugen dort in Zeile 7 einen `InputStream`, der aus der Datei liest, deren Name wir als erstes Argument übergeben. In der darauf folgenden Zeile wird der `InputStream` in einen `Reader` verwandelt. Aus dem `Reader` erzeugen wir einen `Scanner`. Mit diesem `Scanner` können wir schließlich den Parser erzeugen, den wir dann in Zeile 11 mit Hilfe der Methode `parse()` starten, wobei eventuelle Ausnahmen noch abgefangen werden müssen. Falls mit dem Start-Symbol der Grammatik ein Wert assoziert ist, so wird dieser Wert von der Methode `parse()` als Ergebnis zurück gegeben, andernfalls gibt die Methode den Wert `null` zurück.

```
public class ExprParserSym {
  /* terminals */
  public static final int MINUS   = 4;
  public static final int DIVIDE  = 6;
  public static final int NUMBER  = 10;
  public static final int MOD     = 7;
  public static final int SEMI    = 2;
  public static final int EOF     = 0;
  public static final int PLUS    = 3;
  public static final int error   = 1;
  public static final int RPAREN  = 9;
  public static final int TIMES   = 5;
  public static final int LPAREN  = 8;
  public static final String[] terminalNames = new String[] {
      "EOF",
      "error",
      "SEMI",
      "PLUS",
      "MINUS",
      "TIMES",
      "DIVIDE",
      "MOD",
      "LPAREN",
      "RPAREN",
      "NUMBER"
  };
}
```

Figure 14.2: Die Klasse `ExprParserSym`.

```
import java_cup.runtime.*;
import java.io.*;

public class Calculator {
    public static void main(String[] args) {
        try {
            InputStream fileStream = new FileInputStream(args[0]);
            Reader      reader     = new InputStreamReader(fileStream);
            Scanner     scanner    = new Yylex(reader);
            ExprParser  parser     = new ExprParser(scanner);
            parser.parse();
        } catch (Exception e) {}
    }
}
```

Figure 14.3: Die Klasse `Calculator`.

14.2 Generierung eines Cup-Scanner mit Hilfe von Flex

Wir zeigen in diesem Abschnitt, wie wir mit Hilfe von *JFlex* einen Scanner für den im letzten Abschnitt erzeugten Parser erstellen können. Der Scanner, den wir benötigen, muss in der Lage sein, Zahlen und arithmetische Operatoren zu erkennen. Abbildung 14.4 auf Seite 201 zeigt einen solchen Scanner, den wir jetzt im Detail diskutieren.

```
1  import java_cup.runtime.*;
2
3  %%
4
5  %char
6  %line
7  %column
8  %cupsym ExprParserSym
9  %cup
10 %unicode
11
12 %{
13     private Symbol symbol(int type) {
14         return new Symbol(type, yychar, yychar + yylength());
15     }
16
17     private Symbol symbol(int type, Object value) {
18         return new Symbol(type, yychar, yychar + yylength(), value);
19     }
20 %}
21
22 %eofval{
23     return new Symbol(ExprParserSym.EOF);
24 %eofval}
25
26 %%
27
28 ";"           { return symbol( ExprParserSym.SEMI   ); }
29 "+"           { return symbol( ExprParserSym.PLUS   ); }
30 "-"           { return symbol( ExprParserSym.MINUS  ); }
31 "*"           { return symbol( ExprParserSym.TIMES  ); }
32 "/"           { return symbol( ExprParserSym.DIVIDE ); }
33 "%"           { return symbol( ExprParserSym.MOD    ); }
34 "("           { return symbol( ExprParserSym.LPAREN ); }
35 ")"           { return symbol( ExprParserSym.RPAREN ); }
36
37 0|[1-9][0-9]* { return symbol(ExprParserSym.NUMBER, new Integer(yytext())); }
38
39 [ \t\v\n\r]   { /* skip white space */ }
40
41 [^]           { throw new Error("Illegal character '" + yytext() +
42                                 "' at line " + yyline +
43                                 ", column "  + yycolumn); }
```

Figure 14.4: Ein Scanner für arithmetische Ausdrücke

1. In Zeile 1 importieren wir alle Klassen des Paketes `java_cup.runtime`. Dieses Paket enthält insbesondere die Definition der Klasse `Symbol`, mit der in einem CUP-Parser Terminale und Nicht-Terminale beschrieben werden. Daher muss dieses Paket bei jedem Scanner importiert werden, der an einen von CUP erzeugten Parser angeschlossen werden soll.

2. In den Zeilen 5 bis 7 spezifizieren wir, dass der Scanner die Anzahl der insgesamt gelesenen Zeichen, die Anzahl der gelesenen Zeilen und die Anzahl der in der aktuellen Zeile gelesenen Zeichen automatisch berechnen soll. Dadurch können wir später im Parser Syntax-Fehler präzise lokalisieren.

3. Zeile 8 spezifiziert mit dem Schlüsselwort "`%cupsym`", dass die vom Scanner zurückzugebenden Symbole in der Klasse `ExprParserSym` definiert werden. Diese Klasse wurde von CUP erzeugt. Der Name dieser Klasse ergibt sich aus dem Namen der Klasse des Parsers durch Anhängen von "`Sym`", das für *Symbol* steht. Die Klasse des Parsers hatten wir seinerzeit in der Datei "`calc.cup`" mit Hilfe des Schlüsselworts "`class`" spezifiziert.

4. Zeile 9 spezifiziert mit dem Schlüsselwort "`%cup`", dass der Scanner an einen CUP-Parser angeschlossen werden soll.

5. In den Zeilen 12 bis 20 definieren wir zwei Hilfs-Methoden, die Objekte vom Typ `Symbol` erzeugen. Der Scanner muss Objekte von diesem Typ an den Parser zurück liefern. Die in dem Paket `java_cup.runtime` definierte Klasse `Symbol` stellt verschiedene Konstruktoren für diese Klasse zur Verfügung. Wir stellen die wichtigsten Konstruktoren vor.

   1. `public Symbol(int symbolID);`
      Dieser Konstruktor bekommt als Argument eine natürliche Zahl, die festlegt, welche Art von Symbol definiert werden soll. Diese Zahl bezeichnen wir als *Symbol-Nummer*. Jedem Terminal und jeder syntaktische Variablen entspricht genau Symbol-Nummer. Die Kodierung der Symbol-Nummern wird von dem Parser-Generator CUP in der Klasse `ExprParserSym` festgelegt. Abbildung 14.2 auf Seite 200 zeigt diese von CUP erzeugte Klasse.
   2. `public Symbol(int symbolID, Object value);`
      Dieser Konstruktor bekommt zusätzlich zur Symbol-Nummer einen *Wert*, der im Symbol abgespeichert wird. Dieser Wert hat den Typ `Object`, wodurch der Typ so allgemein wie möglich ist. Dieser Konstruktor wird benutzt, wenn Terminale, die einen Wert haben, wie beispielsweise Zahlen, vom Scanner an den Parser zurück gegeben werden sollen.
   3. `public Symbol(int symbolID, int start, int end)`
      Dieser Konstruktor hat zusätzlich zur Symbol-Nummer die Argumente `start` und `stop`, die den Anfang und das Ende des erkannten Terminals festlegen. Die Variable `start` gibt die Position des ersten Zeichens im Text an, während `end` die Position des letzten Zeichens des Tokens angibt. Diese Information ist nützlich, um später im Parser Syntax-Fehler besser lokalisieren zu können.
   4. `public Symbol(int symbolID, int start, int end, Objekt value)`
      Dieser Konstruktor erhält zusätzlich zur Symbol-Nummer und Position des gelesenen Tokens noch den Wert, der diesem Token zugeordnet ist.

   Bei der Implementierung der beiden Hilfs-Methoden mit dem Namen `symbol()` verwenden wir die Funktion `yylength()`. Diese Funktion wird von *jflex* zur Verfügung gestellt und gibt die Länge des Strings zurück, der dem zuletzt erkannten Tokens entspricht.

6. In den Zeilen 28 bis 35 erkennen wir die arithmetischen Operatoren und die Klammer-Symbole. Zur Spezifikation des zurückgegebenen Token verwenden wir dabei die in der Klasse `ExprParserSym` definierten Konstanten `SEMI`, `PLUS`, etc.

7. In Zeile 37 erkennen wir mit dem regulären Ausdruck "`0|[1-9][0-9]*`" eine natürliche Zahl. Diese wandeln wir durch den Konstruktor-Aufruf

   ```
   new Integer(yytext())
   ```

   in ein Objekt vom Typ `Integer` um, wobei der String, der in eine Zahl umgewandelt wird, von der Funktion

`yytext()` geliefert wird. Anschließend geben wir ein Objekt der Klasse `Symbol` zurück, in dem diese Zahl als mit dem Symbol assoziierter Wert abgespeichert wird.

8. In Zeile 39 überlesen wir Leerzeichen, Tabulatoren und Zeilen-Umbrüche. Das Überlesen geschieht dadurch, dass wir in diesem Fall kein Symbol an den Parser zurück geben, denn die semantische Aktion enthält keinen `return`-Befehl.

9. Falls ein beliebiges anderes Zeichen gelesen wird, geben wir mit der Regel, die in Zeile 41 beginnt, eine Fehlermeldung aus. Dabei greifen wir auf die Variablen `yyline` und `yycolumn` zurück, damit der Fehler lokalisiert werden kann.

Durch den Aufruf

```
jflex calc.jflex
```

erzeugt *JFlex* aus der Datei `clac.jflex` den Scanner in der Klasse `Yylex`. Wir hatten in Abbildung 14.3 gesehen, wie diese Klasse an den Parser angebunden wird.

14.3 Shift/Reduce und Reduce/Reduce-Konflikte

```
1  import java_cup.runtime.*;
2
3  terminal           PLUS, MINUS, TIMES, DIVIDE, MOD;
4  terminal           UMINUS, LPAREN, RPAREN;
5  terminal Integer   NUMBER;
6
7  nonterminal Integer expr;
8
9  expr ::= expr PLUS   expr
10       |  expr MINUS  expr
11       |  expr TIMES  expr
12       |  expr DIVIDE expr
13       |  expr MOD    expr
14       |  NUMBER
15       |  MINUS expr
16       |  LPAREN expr RPAREN
17       ;
```

Figure 14.5: CUP-Spezifikation eines Parsers für arithmetische Ausdrücke

Wir betrachten nun ein weiteres Beispiel. Abbildung 14.5 zeigt eine CUP-Spezifikation einer Grammatik, die offenbar mehrdeutig ist, da die Präzedenzen der arithmetischen Operatoren durch diese Grammatik nicht festgelegt werden. Mit dieser Grammatik ist beispielsweise nicht klar, ob der String

"`1 + 2 * 3`" als "`(1 + 2) * 3`" oder als "`1 + (2 * 3)`"

interpretiert werden soll. Wir hatten im letzten Kapitel schon gesehen, dass es in einer mehrdeutigen Grammatik immer Shift/Reduce- oder Reduce/Reduce-Konflikte geben muss, denn jede LALR-Grammatik ist eindeutig. Wenn wir versuchen, die Grammatik aus Abbildung 14.5 mit CUP zu übersetzen und wenn wir dabei zusätzlich die Option "`-dump`" angeben, der Aufruf von CUP hat dann die Form

```
cup -dump calc.cup
```

so erhalten wir eine große Zahl von Shift/Reduce-Konflikten angezeigt. Beispielsweise erhalten wir die folgende Fehlermeldung:

```
Warning : *** Shift/Reduce conflict found in state #12
  between expr ::= expr MINUS expr (*)
  and     expr ::= expr (*) PLUS expr
  under symbol PLUS
  Resolved in favor of shifting.
```

Statt des Zeichens "•" benutzt CUP den String "(*)" zur Darstellung der Position in einer markierten Regel. Die obige Fehlermeldung zeigt uns an, dass es zwischen der markierten Regel

$$R_1 \;:=\; \Big(\mathit{expr} \to \mathit{expr}\ \text{“–”}\ \mathit{expr}\ \bullet\Big)$$

und der markierten Regel

$$R_2 \;:=\; \Big(\mathit{expr} \to \mathit{expr}\ \bullet\ \text{“+”}\ \mathit{expr}\Big)$$

einen Shift/Reduce-Konflikt gibt: Die beiden markierten Regeln R_1 und R_2 sind Elemente eines Zustands, der von CUP intern mit der Nummer 12 versehen worden ist. Der Zustand mit der Nummer 12 hat folgende Form:

```
lalr_state [12]: {
  [expr ::= expr (*) MOD expr ,    {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN }]
  [expr ::= expr (*) MINUS expr ,  {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN }]
  [expr ::= expr (*) DIVIDE expr , {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN }]
  [expr ::= expr (*) TIMES expr ,  {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN }]
  [expr ::= expr (*) PLUS expr ,   {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN }]
  [expr ::= expr MINUS expr (*) ,  {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN }]
}
```

Damit können wir jetzt den Shift/Reduce-Konflikt interpretieren: Im Zustand 12 ist der Parser entweder dabei, die Eingabe mit der Regel

$$\mathit{expr} \to \mathit{expr}\ \text{“–”}\ \mathit{expr}$$

zu reduzieren, oder der Parser ist gerade dabei, die rechte Seite der Regel

$$\mathit{expr} \to \mathit{expr}\ \text{“+”}\ \mathit{expr}$$

zu erkennen, wobei er bereits eine *expr* erkannt hat und nun als nächstes das Token "+" erwartet wird. Da das Token "+" auch in der Follow-Menge der erweiterten markierten Regel R_1 liegen kann, ist an dieser Stelle unklar, ob das Token "+" auf den Stack geschoben werden soll, oder ob stattdessen mit der Regel R_1 reduziert werden muss. Bei einem Shift/Reduce-Konflikt entscheidet sich der von CUP erzeugte Parser immer dafür, das Token auf den Stack zu schieben.

14.4 Operator-Präzendenzen

Es ist mit CUP möglich, Shift/Reduce-Konflikte durch die Angabe von *Operator-Präzedenzen* aufzulösen. Abbildung 14.6 zeigt die Spezifikation einer Grammatik zur Erkennung arithmetischer Ausdrücke, die aus Zahlen und den binären Operatoren "+", "–", "*", "/" und "^" aufgebaut sind. Mit Hilfe der Schlüsselwörter

"`precedence left`" und "`precedence right`"

haben wir festgelegt, dass die Operatoren "+", "–", "*" und "/" *links-assoziativ* sind, ein Ausdruck der Form

$3-2-1$ wird also als $(3-2)-1$ und nicht als $3-(2-1)$

gelesen. Demgegenüber ist der Operator "^", der in der CUP-Grammatik mit "`POW`" bezeichnet wird und die Potenzbildung bezeichnet, *rechts-assoziativ*, der Ausdruck

`4^3^2` wird daher als $4^{(3^2)}$ und nicht als $\left(4^3\right)^2$

interpretiert. Die Reihenfolge, in der die Assoziativität der Operatoren spezifiziert werden, legt die *Präzedenzen*, die auch als *Bindungsstärken* bezeichnet werden, fest. Dabei ist die Bindungsstärke umso größer, je später der Operator

spezifiziert wird. In unserem konkreten Beispiel bindet der Exponentiations-Operator "^" also am stärksten, während die Operatoren "+" und "–" am schwächsten binden. Bei der in Abbildung 14.6 gezeigten Grammatik ordnet CUP den Operatoren die Bindungsstärke nach der folgenden Tabelle zu:

Operator	Bindungsstärke	Assoziativität
"+"	1	links
"–"	1	links
"*"	2	links
"/"	2	links
"^"	3	rechts

```
1  import java_cup.runtime.*;
2  class  ExprParser;
3
4  terminal         SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD, POW;
5  terminal         UMINUS, LPAREN, RPAREN;
6
7  terminal Double  NUMBER;
8
9  nonterminal        expr_list, expr_part;
10 nonterminal Double expr;
11
12 precedence left   PLUS, MINUS;
13 precedence left   TIMES, DIVIDE, MOD;
14 precedence right  UMINUS, POW;
15
16 start with expr_list;
17
18 expr_list ::= expr_list expr_part
19            |  expr_part
20            ;
21
22 expr_part ::= expr:e {: System.out.println("result = " + e); :} SEMI
23            ;
24
25 expr ::= expr:e1 PLUS   expr:e2 {: RESULT = e1 + e2; :}
26       |  expr:e1 MINUS  expr:e2 {: RESULT = e1 - e2; :}
27       |  expr:e1 TIMES  expr:e2 {: RESULT = e1 * e2; :}
28       |  expr:e1 DIVIDE expr:e2 {: RESULT = e1 / e2; :}
29       |  expr:e1 MOD    expr:e2 {: RESULT = e1 % e2; :}
30       |  expr:e1 POW    expr:e2 {: RESULT = Math.pow(e1, e2); :}
31       |  NUMBER:n               {: RESULT = n;       :}
32       |  MINUS expr:e           {: RESULT = - e;     :} %prec UMINUS
33       |  LPAREN expr:e RPAREN   {: RESULT = e;       :}
34       ;
```

Figure 14.6: Auflösung der Shift/Reduce-Konflikte durch Operator-Präzedenzen.

Wie erläutern nun, wie diese Bindungsstärken benutzt werden, um Shift/Reduce-Konflikte aufzulösen. CUP geht folgendermaßen vor:

1. Zunächst wird jeder Grammatik-Regel eine *Präzedenz* zugeordnet. Die Präzedenz ist dabei die Bindungsstärke des letzten in der Regel auftretenden Operators. Für den Fall, dass eine Regel mehrere Operatoren enthält, für die eine Bindungsstärke spezifiziert wurde, wird zur Festlegung der Bindungsstärke also der Operator herangezogen, der in der Regel am weitesten rechts steht. In unserem Beispiel haben die einzelnen Regeln damit die folgenden Präzedenzen:

Regel	Präzedenz
$E \to E$ "+" E	1
$E \to E$ "-" E	1
$E \to E$ "*" E	2
$E \to E$ "/" E	2
$E \to E$ "^" E	3
$E \to$ "(" E ")"	—
$E \to N$	—

   Für die Regeln, die keinen Operator enthalten, für den eine Bindungsstärke spezifiziert ist, bleibt die Präzedenz unspezifiziert.

2. Ist s ein Zustand, in dem zwei Regeln r_1 und r_2 der Form

   $$r_1 = (a \to \beta \bullet o\ \delta : L_1) \quad \text{und} \quad r_2 = (c \to \gamma\bullet : L_2) \quad \text{mit} \quad o \in L_2$$

   vorkommen, so gibt es bei der Berechnung von

   $$\mathit{action}(s, o)$$

   zunächst einen Shift/Reduce-Konflikt. Falls dem Operator o die Präzedenz $p(o)$ zugeordnet worden ist und wenn außerdem die Regel r_2, mit der reduziert werden würde, die Präzedenz $p(r_2)$ hat, so wird der Shift/Reduce-Konflikt in Abhängigkeit von der relativen Größe der beiden Zahlen $p(o)$ und $p(r_2)$ aufgelöst. Hier werden fünf Fälle unterschieden:

   1. $p(o) > p(c \to \gamma)$: In diesem Fall bindet der Operator o stärker. Daher wird das Token o in diesem Fall auf den Stack geschoben:

      $\mathit{action}(s, o) = \langle \texttt{shift}, \mathit{goto}(s, o)\rangle$.

      Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

      ```
      1+2*3
      ```

      mit den Grammatik-Regeln

      $$E \to E \text{ "+" } E \mid E \text{ "*" } E \mid \textsc{Number}$$

      parsen. Betrachten wir die Situation, bei der der Teilstring "1+2" bereits gelesen wurde und nun als nächstes das Token "*" verarbeitet werden soll. Der LALR-Parser ist dann in dem folgenden Zustand:

      $$\left\{ \begin{array}{lcl} E & \to & E \bullet \text{"*"} E : \{\$, \text{"*"}, \text{"+"}\}, \\ E & \to & E \bullet \text{"+"} E : \{\$, \text{"*"}, \text{"+"}\}, \\ E & \to & E \text{"+"} E \bullet : \{\$, \text{"*"}, \text{"+"}\} \end{array} \right\}.$$

      Wenn in diesem Zustand als nächstes Zeichen ein "*" gelesen wird, so darf der bisher gelesene String "1+2" nicht mit der Regel $E \to E$"+"E reduziert werden, denn wir wollen die 2 ja zunächst mit 3 multiplizieren. Stattdessen muss also das Zeichen "*" auf den Stack geschoben werden.

   2. $p(o) < p(c \to \gamma)$: Jetzt bindet der Operator, der in der Regel r_2 auftritt, stärker als der Operator o. Daher wird in diesem Fall zunächst mit der Regel r_2 reduziert, wir haben also

      $\mathit{action}(s, o) = \langle \texttt{reduce}, r_2\rangle$.

      Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

```
1*2+3
```

mit den Grammatik-Regeln

$$E \to E \text{ “+” } E \mid E \text{ “*” } E \mid \textsc{Number}$$

parsen. Betrachten wir die Situation, bei der der Teilstring “1*2” bereits gelesen wurde und nun als nächstes das Token “+” verarbeitet werden soll. Der LALR-Parser ist dann in dem folgenden Zustand:

$$\left\{ \begin{array}{lcl} E & \to & E \bullet \text{“*”} E : \{\$, \text{“*”}, \text{“+”}\}, \\ E & \to & E \bullet \text{“+”} E : \{\$, \text{“*”}, \text{“+”}\}, \\ E & \to & E \text{“*”} E \bullet : \{\$, \text{“*”}, \text{“+”}\} \end{array} \right\}.$$

Wenn in diesem Zustand als nächstes Zeichen ein “+” gelesen wird, so soll der bisher gelesene String “1*2” mit der Regel $E \to E \text{“*”} E$ reduziert werden, denn wir wollen die 1 ja zunächst mit 2 multiplizieren.

3. $p(o) = p(c \to \gamma)$ und der Operator o ist links-assoziativ: Dann wird zunächst mit der Regel r_2 reduziert, wir haben also

   $$\mathit{action}(s, o) = \langle \texttt{reduce}, r_2 \rangle.$$

   Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

   ```
   1-2-3
   ```

   mit den Grammatik-Regeln

   $$E \to E \text{ “+” } E \mid E \text{ “-” } E \mid \textsc{Number}$$

   parsen. Betrachten wir die Situation, bei der der Teilstring “1-2” bereits gelesen wurde und nun als nächstes das Token “-” verarbeitet werden soll. Der LALR-Parser ist dann in dem folgenden Zustand:

   $$\left\{ \begin{array}{lcl} E & \to & E \bullet \text{“-”} E : \{\$, \text{“-”}, \text{“+”}\}, \\ E & \to & E \bullet \text{“+”} E : \{\$, \text{“-”}, \text{“+”}\}, \\ E & \to & E \text{“-”} E \bullet : \{\$, \text{“-”}, \text{“+”}\} \end{array} \right\}.$$

   Wenn in diesem Zustand als nächstes Zeichen ein “-” gelesen wird, so soll der bisher gelesene String “1-2” mit der Regel $E \to E \text{“-”} E$ reduziert werden, denn wir wollen von der Zahl 1 ja zunächst die Zahl 2 subtrahieren.

4. $p(o) = p(c \to \gamma)$ und der Operator o ist rechts–assoziativ: In diesem Fall wird o auf den Stack geschoben:

   $$\mathit{action}(s, o) = \langle \texttt{shift}, \mathit{goto}(s, o) \rangle.$$

   Wenn wir diesen Fall verstehen wollen, reicht es aus, den String

   ```
   2^3^4
   ```

   mit den Grammatik-Regeln

   $$E \to E \text{ “\^{}” } E \mid \textsc{Number}$$

   zu parsen und die Situation zu betrachten, bei der der Teilstring “1^2” bereits verarbeitet wurde und als nächstes Zeichen nun der Operator “^” gelesen wird. Der LALR-Parser ist dann in dem folgenden Zustand:

   $$\big\{E \to E \bullet \text{ “\^{}” } E : \{\$, \text{ “\^{}” }\}, E \to E \text{“\^{}” } E \bullet : \{\$, \text{ “\^{}” }\}\big\}.$$

   Hier muss als nächstes das Token “^” auf den Stack geschoben werden, denn wir wollen ja zunächst den Ausdruck “3^4” berechnen.

5. $p(o) = p(c \to \gamma)$ und der Operator o hat keine Assoziativität: In diesem Fall liegt ein Syntax-Fehler vor:

   $$\mathit{action}(s, o) = \mathit{error}.$$

   Diesen Fall verstehen Sie, wenn Sie versuchen, einen String der Form

   ```
   1 < 1 < 1
   ```

mit den Grammatik-Regeln

$$E \to E \text{ “<” } E \mid E \text{ “+” } E \mid \text{NUMBER}$$

zu parsen. In dem Moment, in dem Sie den Teilstring “`1 < 1`” gelesen haben und nun das nächste Token das Zeichen “<” ist, erkennen Sie, dass es ein Problem gibt.

Bemerkung: Beachten Sie, dass auch in diesem Fall der Shift/Reduce-Konflikt aufgelöst wird, denn den Syntax-Fehler erhalten Sie erst beim Parsen, während die Erstellung des Parsers selber fehlerfrei (sprich: ohne verbleibende Konflikte) verläuft.

Um in CUP einen Operator o als nicht-assoziativ zu deklarieren, schreiben Sie:

`precedence nonassoc` o

6. $p(o)$ ist undefiniert oder $p(c \to \gamma)$ ist undefiniert.
   In diesem Fall erzeugt CUP einen Shift/Reduce-Konflikt und gibt bei der Erzeugung des Parsers eine entsprechende Warnung aus. Diese Warnung wird als Fehler gewertet, wenn sie nicht durch Angabe der Option

   `-expect` n

   beim Aufruf von CUP unterdrückt wird. Bei dieser Option ist n die Anzahl der vom Benutzer erwarteten Konflikte. In diesem Fall wird der Konflikt dann per Default dadurch aufgelöst, dass das betreffende Token auf den Stack geschoben wird. Wir werden dieses Verhalten im nächsten Abschnitt anhand eines Beispiels im Detail diskutieren.

Die von CUP mit der Option “`-dump`” erzeugte Ausgabe zeigt im Detail, wie die Shift/Reduce-Konflikt Konflikte aufgelöst worden sind. Wir betrachten exemplarisch zwei Zustände in der Datei, die für die in Abbildung 14.6 gezeigte Grammatik erzeugt wird.

1. Der Zustand Nummer 14 hat die in Abbildung 14.7 gezeigte Form. Hier gibt es unter anderem einen Shift/Reduce-Konflikt zwischen den beiden markierten Regeln

   $$E \to E \bullet \text{ “+” } E \quad \text{und} \quad E \to E \text{ “*” } E \bullet,$$

   denn die erste Regel verlangt nach einem Shift, während die zweite Regel eine Reduktion fordert. Da die Regel $E \to E$ “*” E dieselbe Präzedenz wie der Operator “*” und dieser eine höhere Präzedenz als “+” hat, wird beispielsweise beim Lesen des Zeichens “+” mit der Regel $E \to E$ “*” E reduziert. Wird hingegen das Zeichen “^” gelesen, so wird dieses geshiftet, denn dieses Zeichen hat eine höhere Priorität als die Regel $E \to E$ “*” E. Weiterhin gibt es einen Shift/Reduce-Konflikt zwischen den beiden markierten Regeln

   $$E \to E \bullet \text{ “*” } E \quad \text{und} \quad E \to E \text{ “*” } E \bullet.$$

   Hier haben beide Regeln die gleiche Präzedenz. Daher entscheidet die Assoziativität. Da der Operator “*” links-assoziativ ist, wird mit der Regel $E \to E$ “*” E reduziert, falls das nächste Zeichen ein Multiplikations-Operator “*” ist.

2. Der Zustand Nummer 18 hat die in Abbildung 14.8 gezeigte Form. Zunächst gibt es hier einen Shift/Reduce-Konflikt zwischen den Regeln

   $$E \to E \bullet \text{ “^” } E \quad \text{und} \quad E \to E \text{ “^” } E \bullet,$$

   wenn das nächste Token der Operator “^” ist. Da der Operator dieselbe Präzedenz hat wie die Regel, entscheidet die Assoziativität. Nun ist der Operator “^” rechts-assoziativ, daher wird in diesem Fall geshiftet.

   Hier gibt es noch viele andere Shift/Reduce-Konflikte, die aber alle dieselbe Struktur haben. Exemplarisch betrachten wir den Shift/Reduce-Konflikt zwischen den Regeln

   $$E \to E \bullet \text{ “+” } E \quad \text{und} \quad E \to E \text{ “^” } E \bullet,$$

   der auftritt, wenn das nächste Token ein “+” ist. Da die Regel $E \to E$ “^” E die Präzedenz 3 hat, die größer ist als die Präzedenz 1 des Operators “+” wird dieser Konflikt dadurch aufgelöst, dass mit der Regel $E \to E$ “^” E reduziert wird.

```
lalr_state [14]: {
  [expr ::= expr (*) PLUS expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
  [expr ::= expr TIMES expr (*) , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
  [expr ::= expr (*) POW expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
  [expr ::= expr (*) TIMES expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
  [expr ::= expr (*) MOD expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
  [expr ::= expr (*) MINUS expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
  [expr ::= expr (*) DIVIDE expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
}

From state #14
 [term 2:REDUCE(with prod 7)] [term 3:REDUCE(with prod 7)]
 [term 4:REDUCE(with prod 7)] [term 5:REDUCE(with prod 7)]
 [term 6:REDUCE(with prod 7)] [term 7:REDUCE(with prod 7)]
 [term 8:SHIFT(to state 9)] [term 11:REDUCE(with prod 7)]
```

Figure 14.7: Der Zustand Nummer 14.

```
lalr_state [18]: {
  [expr ::= expr POW expr (*) , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
  [expr ::= expr (*) PLUS expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
  [expr ::= expr (*) POW expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
  [expr ::= expr (*) TIMES expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
  [expr ::= expr (*) MOD expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
  [expr ::= expr (*) MINUS expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
  [expr ::= expr (*) DIVIDE expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
}

From state #18
 [term 2:REDUCE(with prod 10)] [term 3:REDUCE(with prod 10)]
 [term 4:REDUCE(with prod 10)] [term 5:REDUCE(with prod 10)]
 [term 6:REDUCE(with prod 10)] [term 7:REDUCE(with prod 10)]
 [term 8:SHIFT(to state 9)] [term 11:REDUCE(with prod 10)]
```

Figure 14.8: Der Zustand Nummer 18.

Aufgabe 40: Implementieren Sie einen CUP-Parser, der in der Lage ist, eine CUP-Grammatik zu lesen und zusätzlich die folgenden Anforderungen erfüllt:

1. Die Grammatik soll intern in Form eines abstrakten Syntax-Baums abgespeichert werden. In dem Verzeichnis

   Exercises/Grammar2HTML-Cup/

   finden Sie verschiedene *Java*-Klassen, mit denen Sie einen solchen Syntax-Baum darstellen können. Diese Klassen implementieren eine Methode `toString()`, mit deren Hilfe sich der Syntax-Baum bequem im HTML-Format ausgeben lässt. Außerdem enthält das Verzeichnis auch die Datei `Grammatik.java`, welche die Klasse `Grammatik` implementiert. Diese Klasse enthält eine Methode `main`, mit der Sie Ihren Parser testen können.

2. Die semantischen Aktionen der gelesenen Grammatik sollen unterdrückt werden.

3. Testen Sie Ihr Programm, indem Sie es sowohl auf sich selbst als auch auf die im Unterricht vorgestellte Grammatik für arithmetische Ausdrücke anwenden. Zusätzlich können Sie es auch auf die C-Grammatik, die Sie unter

https://github.com/karlstroetmann/Formal-Languages/blob/master/Cup/Grammars/c-grammar.cup

im Netz finden, anwenden.

14.5 Das Dangling-Else-Problem

```
1  class Dangling;
2
3  terminal LPAREN, RPAREN, IF, ELSE, WHILE, LBRACE, RBRACE, ASSIGN, SEMI;
4  terminal EQUAL, ID;
5
6  nonterminal stmnt, stmntList, expr;
7
8  start with stmnt;
9
10 stmnt ::= IF LPAREN expr RPAREN stmnt
11        |  IF LPAREN expr RPAREN stmnt ELSE stmnt
12        |  WHILE LPAREN expr RPAREN stmnt
13        |  LBRACE stmntList RBRACE
14        |  ID ASSIGN expr SEMI
15        ;
16
17 stmntList ::= stmntList stmnt
18             |  /* epsilon */
19             ;
20
21 expr ::= ID EQUAL ID
22        | ID
23        ;
```

Figure 14.9: Fragment einer Grammatik für die Sprache `C`

Bei der syntaktischen Beschreibung von Befehlen der Sprache `C` tritt bei der Behandlung von *if-then-else* Konstrukten ein Shift/Reduce-Konflikt auf, den wir jetzt analysieren wollen. Abbildung 14.9 zeigt die Grammatik `dangling.cup`, die einen Teil der Syntax von Befehlen der Sprache `C` beschreibt. Um uns auf das Wesentliche konzentrieren zu können, sind dort die Ausdrücke nur Gleichungen und Variablen. Das Token "`ID`" (Abkürzung für *Identifier*) steht für eine Variable, die Grammatik beschreibt also Befehle, die aus Zuweisungen, *If-Abfragen*, *If-Else-Abfragen* und *While-Schleifen* aufgebaut sind. Übersetzen wir diese Grammatik mit CUP, so erhalten wir den in Abbildung 14.10 ausschnittsweise gezeigten Shift/Reduce-Konflikt.

Der Konflikt entsteht bei der Berechnung von `action(`"`state #10`", "`else`"`)` zwischen den beiden markierten Regeln

stmnt → "`if`" "`(`" *expr* "`)`" *stmnt* • und

stmnt → "`if`" "`(`" *expr* "`)`" *stmnt* • "`else`" *stmnt*.

Die erste Regel verlangt nach einer Reduktion, die zweite Regel sagt, dass das Token `else` geshiftet werden soll. Das dem Konflikt zu Grunde liegende Problem ist, dass die in Abbildung 14.9 gezeigte Grammatik mehrdeutig ist, denn ein *stmnt* der Form

```
if (a == b) if (c == d) s = t; else u = v;
```

kann auf die folgenden beiden Arten gelesen werden:

```
1  Warning : *** Shift/Reduce conflict found in state #10
2    between stmnt ::= IF LPAREN expr RPAREN stmnt (*)
3    and     stmnt ::= IF LPAREN expr RPAREN stmnt (*) ELSE stmnt
4    under symbol ELSE
5    Resolved in favor of shifting.
6
7  lalr_state [10]: {
8    [stmnt ::= IF LPAREN expr RPAREN stmnt (*) , {EOF IF ELSE WHILE LBRACE RBRACE ID }]
9    [stmnt ::= IF LPAREN expr RPAREN stmnt (*) ELSE stmnt , {EOF IF ELSE WHILE LBRACE RBRACE ID }]
10 }
```

Figure 14.10: Ein Shift/Reduce-Konflikt.

1. Die erste (und nach der Spezifikation der Sprache `C` auch korrekte) Interpretation besteht darin, dass wir den Befehl wie folgt klammern:

```
if (a == b) {
    if (c == d) {
        s = t;
    } else {
        u = v;
    }
}
```

2. Die zweite Interpretation, die nach der in Abbildung 14.9 gezeigten Grammatik ebenfalls zulässig wäre, würde den Befehl in der folgenden Form interpretieren:

```
if (a == b) {
    if (c == d) {
        s = t;
    }
} else {
    u = v;
}
```

   Hier wird das "`else`" dem äußeren "`if`" zugeordnet, was nicht der Spezifikation der Sprache `C` entspricht.

Es gibt vier Möglichkeiten, das Problem zu lösen.

1. Tritt ein Shift/Reduce-Konflikt auf, der nicht durch Operator-Präzedenzen gelöst wird, so ist der Default, dass das nächste Token auf den Stack geschoben wird. In dem konkreten Fall ist dies genau das, was wir wollen, weil dadurch das "`else`" immer mit dem letzten "`if`" assoziert wird. Um die normalerweise bei Konfliken von CUP ausgelöste Fehlermeldung zu unterdrücken, müssen wir CUP mit der Option "`-expect`" wie folgt aufrufen:

```
cup -expect 1 -dump dangling.cup
```

   Die Zahl `1` gibt hier die Anzahl der Konflikte an, die wir erwarten. So lange die spezifizierte Zahl der Konflikte mit der tatsächlich gefundenen Zahl übereinstimmt, wird von CUP nur eine Warnung und keine Fehlermeldung ausgegeben. Insbesondere wird in diesem Fall ein Parser erzeugt.

2. Die zweite Möglichkeit besteht darin, die Grammatik so umzuschreiben, dass die Mehrdeutigkeit verschwindet. Die grundsätzliche Idee ist hier, zwischen zwei Arten von Befehlen zu unterscheiden.

   (a) Einerseits gibt es Befehle, bei denen jedem "`if`" auch ein "`else`" zugeordnet ist. Zwischen einem "`if`" und einem "`else`" dürfen nur solche Befehle auftreten.

Wir bezeichnen Befehle dieser Form als *geschlossene Befehle*. Die Idee bei dieser Sprechweise besteht darin, dass "`if`" als öffnende Klammer zu interpretieren, während das "`else`" einer schließenden Klammer entspricht. Bei einem geschlossenen Befehl entspricht jeder öffnenden Klammer eine schließende Klammer.

(b) Andererseits gibt es Befehle, bei denen dem letzten "`if`" kein "`else`" zugeordnet ist. Solche Befehle bezeichnen wir als *offene Befehle*. Offene Befehle dürfen nicht zwischen einem "`if`" und einem "`else`" auftreten, denn dann müsste das "`else`" dem "`if`" des offenen Befehls zugeordnet werden und der offene Befehl wäre in Wahrheit geschlossen.

Abbildung 14.11 zeigt die Grammatik `dangling.cup`, die diese Idee umsetzt. In der Abbildung sind nur noch die Grammatik-Regeln gezeigt, denn die Deklaration der Terminale und syntaktischen Variablen hat sich gegenüber der ursprünglichen Grammatik nicht verändert. Die syntaktische Kategorie *matchedStmnt* beschreibt dabei die Befehle, bei denen jedem "`if`" ein "`else`" zugeordnet ist, während die Kategorie *unMatchedStmnt* die restlichen Befehle erfasst.

```
1   stmnt ::= matchedStmnt
2           |  unmatchedStmnt
3           ;
4
5   matchedStmnt ::= IF LPAREN expr RPAREN matchedStmnt ELSE matchedStmnt
6                  |  WHILE LPAREN expr RPAREN matchedStmnt
7                  |  LBRACE stmntList RBRACE
8                  |  ID ASSIGN expr SEMI
9                  ;
10
11  unmatchedStmnt ::= IF LPAREN expr RPAREN stmnt
12                   |  IF LPAREN expr RPAREN matchedStmnt ELSE unmatchedStmnt
13                   |  WHILE LPAREN expr RPAREN unmatchedStmnt
14                   ;
15
16  stmntList ::= stmntList stmnt
17              |  /* epsilon */
18              ;
19
20  expr ::= ID EQUAL ID
21         | ID
22         ;
```

Figure 14.11: Eine eindeutige Grammatik für C-Befehle.

Aus theoretischer Sicht ist das Umschreiben der Grammatik der sauberste Weg. Aus diesem Grund haben die Entwickler der Sprache *Java* [GJS96] bei der Spezifikation der Syntax den oben skizzierten Weg beschritten. Der Nachteil ist allerdings, dass bei diesem Vorgehen die Grammatik aufgebläht wird.

3. Die nächste Möglichkeit um das *Dangling-Else*-Problem zu lösen, besteht darin, dass wir "`if`" und "`else`" als Operatoren auffassen, denen wir eine Präzedenz zuordenen. Abbildung 14.12 zeigt die Grammatik `dangling.cup`, bei der dieser Weg beschritten wurde.

   (a) Zunächst haben wir in den Zeilen 1 und 2 die Terminale "`IF`" und "`ELSE`" als nicht-assoziative Operatoren deklariert, wobei "`ELSE`" die höhere Präzedenz hat. Dadurch erreichen wir, dass ein "`ELSE`" auf den Stack geschoben wird, wenn der Parser in dem in Abbildung 14.10 gezeigten Zustand ist.

   (b) In Zeile 6 haben wir der Regel

   stmnt `::= IF LPAREN` *expr* `RPAREN` *stmnt*

   explizit mit Hilfe der nachgestellten Option

```
%prec IF
```

die Präzedenz des Operators "IF" zugewiesen. Dies ist notwendig, weil der letzte Operator, der in dieser Regel auftritt, die schließende runde Klammer "RPAREN" ist, der wir keine Priorität zugewiesen haben. Der Klammer eine Priorität zuzuweisen wäre einerseits kontraintuitiv, andererseits problematisch, da die Klammer ja auch noch an anderen Stellen verwendet werden könnte. Mit Hilfe der "%prec"-Deklaration können wir einer Regel unmittelbar die Präzedenz eines Operators zuweisen und so das Problem umgehen. In dem vorliegenden Fall ist die Präzedenz des Operators "ELSE" höher als die Präzedenz von "IF", so dass der Shift/Reduce-Konflikt dadurch aufgelöst wird, dass das Token "ELSE" auf den Stack geschoben wird, wodurch eine "else"-Klausel tatsächlich mit der unmittelbar davor stehenden "if"-Klausel verbunden wird, wie es die Definition der Sprache C fordert.

```
1  precedence nonassoc IF;
2  precedence nonassoc ELSE;
3
4  start with stmnt;
5
6  stmnt ::= IF LPAREN expr RPAREN stmnt            %prec IF
7          |  IF LPAREN expr RPAREN stmnt ELSE stmnt
8          |  WHILE LPAREN expr RPAREN stmnt
9          |  LBRACE stmntList RBRACE
10         |  ID ASSIGN expr SEMI
11         ;
12
13 stmntList ::= stmntList stmnt
14             |  /* epsilon */
15             ;
16
17 expr ::= ID EQUAL ID
18         | ID
19         ;
```

Figure 14.12: Auflösung des Shift/Reduce-Konflikts mit Hilfe von Operator-Präzedenzen.

Operator-Präzedenzen sind ein mächtiges Mittel um eine Grammatik zu strukturieren. Sie sollten allerdings mit Vorsicht eingesetzt werden, denn Sprachen wie die Programmier-Sprache C, bei der es 15 verschiedene Operator-Präzendenzen gibt, überfordern die meisten Benutzer.

4. Die letzte Möglichkeit, das *Dangling-Else*-Problem zu lösen, besteht darin, dass wir fordern, dass die Befehle, die in einem "if"-Befehl verwendet werden, immer in den geschweiften Klammern "{" und "}" eingeschlossen werden. Dies ist allerdings nur dann eine Option, wenn wir die Syntax der Sprache selber definieren können. Bei der Sprache SETLX wurde dieser Weg beschritten. Abbildung 14.13 auf Seite 214 zeigt die Grammatik `no-dangling.cup`, bei der durch die Verwendung von geschweiften Klammern erreicht wurde, dass keine Mehrdeutigkeit mehr auftritt.

14.6 Auflösung von Reduce/Reduce-Konflikten

Im Gegensatz zu Shift/Reduce-Konflikten können Reduce/Reduce-Konflikte nicht durch Operator-Präzedenzen aufgelöst werden. Wir diskutieren in diesem Abschnitt die Möglichkeiten, die wir haben um Reduce/Reduce-Konflikte aufzulösen. Wir beginnen unsere Diskussion damit, dass wir die Reduce/Reduce-Konflikte in verschiedene Kategorien einteilen.

1. *Mehrdeutigkeits-Konflikte* sind Reduce/Reduce-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 14.5 in der Grammatik

```
1  stmnt ::= IF LPAREN expr RPAREN block
2          |  IF LPAREN expr RPAREN block ELSE block
3          |  WHILE LPAREN expr RPAREN stmnt
4          |  block
5          |  ID ASSIGN expr SEMI
6          ;
7
8  block ::= LBRACE stmntList RBRACE
9          ;
10
11 stmntList ::= stmntList stmnt
12             |  /* epsilon */
13             ;
14
15 expr ::= ID EQUAL ID
16         | ID
17         ;
```

Figure 14.13: Eine Grammatik ohne das *Dangling-Else*-Problem.

`calc.cup` 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 aber 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.

14.6.1 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 14.14 zeigt die Grammatik `lookAhead.cup`[1], die zwar eindeutig ist, aber nicht die LR(1)-Eigenschaft hat.

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

$$a \to b \text{ “U” “V”}$$

verwendet werden und folglich ist das “X” als ein b zu interpretieren. Ist das zweite Token hinter dem “X” hingegen

[1] 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.

```
1  terminal    U, V, W, X;
2  nonterminal a, b, c;
3
4  a ::= b U V
5     |  c U W
6     ;
7  b ::= X
8     ;
9  c ::= X
10    ;
```

Figure 14.14: Eine eindeutige Grammatik ohne die LR(1)-Eigenschaft.

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     ;
4  b ::= X U
5     ;
6  c ::= X U
7     ;
```

Figure 14.15: Eine zu 14.14 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 14.15 zeigt die entsprechend modifizierte Grammatik `solved.cup`,

14.6.2 Mysteriöse Reduce/Reduce-Konflikte

Wir sprechen dann von einem *mysteriösen Reduce/Reduce-Konflikt*, wenn die gegebene Grammatik eine LR(1)-Grammatik ist, sich aber beim Übergang von den LR-Zuständen zu den LALR-Zuständen Reduce/Reduce-Konflikte ergeben. Die in Abbildung 14.16 gezeigte Grammatik `mysterious.cup` habe ich dem *Bison*-Handbuch entnommen.

Übersetzen wir diese Grammatik mit CUP, so erhalten wir unter anderem den folgenden Zustand:

```
lalr_state [1]: {
```

```
1  terminal    ID, COMMA, COLON;
2  nonterminal def, param_spec, return_spec, type, name_list, name;
3
4  def
5      ::= param_spec return_spec COMMA
6       ;
7  param_spec
8      ::= type
9       |  name_list COLON type
10      ;
11 return_spec
12     ::= type
13      |  name COLON type
14      ;
15 type
16     ::= ID
17      ;
18 name
19     ::= ID
20      ;
21 name_list
22     ::= name
23      |  name COMMA name_list
24      ;
```

Figure 14.16: Eine CUP-Grammatik mit einem mysteriösen Reduce/Reduce-Konflikt.

```
  [name ::= ID (*) , {COMMA COLON }]
  [type ::= ID (*) , {ID COMMA }]
}
```

Da in beiden Mengen von Folgetoken das Token `COMMA` auftritt, gibt es hier offensichtlich einen Reduce/Reduce-Konflikt. Um diesen Konflikt besser zu verstehen, berechnen wir zunächst die Zustände eines kanonischen LR-Parsers für diese Grammatik. Wir erhalten dann eine Menge von Zuständen, von denen die für den späteren Konflikt ursächlichen Zuständen in Abbildung 14.17 gezeigt sind.

Analysieren wir die Zustände, so stellen wir fest, dass beim Übergang von LR-Zuständen zu den LALR-Zuständen die beiden Zustände s_7 und s_8 zu einem Zustand zusammengefasst werden, denn diese beiden Zustände haben den selben Kern. Bei der Zusammenfassung entsteht der Zustand, der von CUP als "`lalr_state [1]`" bezeichnet hat. Die Zustände s_7 und s_8 selber haben noch keinen Konflikt, weil dort die Mengen der Folgetoken disjunkt sind. Der Konflikt tritt erst durch die Vereinigung dieser beiden Mengen auf, denn dadurch ist das Token "`,`" als Folgetoken für beide in dem Zustand enthaltenen Regeln zulässig. Um den Konflikt aufzulösen müssen wir verhindern, dass die beiden Zustände s_7 und s_8 zusammengefasst werden. Dazu analysieren wir zunächst, wo diese Zustände herkommen.

1. Den Zustand s_7 erhalten wir, wenn wir im Zustand s_0 das Token `ID` lesen, denn es gilt

   $$s_7 = \mathit{goto}(s_0, \texttt{ID}).$$

2. Der Zustand s_8 entsteht, wenn das Token `ID` im Zustand s_2 gelesen wird, wir haben

   $$s_8 = \mathit{goto}(s_2, \texttt{ID}).$$

Die Idee zur Auflösung des Konflikts ist, dass wir den Zustand s_2 so ändern, dass die Kerne von $s_7 = \mathit{goto}(s_0, \texttt{ID})$ und $s_8 = \mathit{goto}(s_0, \texttt{ID})$ unterschiedlich werden. Die erweiterten markierten Regeln in den Zuständen s_0 und s_2, die letztlich für den Konflikt verantwortlich sind, sind die Grammatik-Regeln für die syntaktische Variablen *param_spec*

```
s0 = { S -> <*> def: [$],
       def -> <*> param_spec return_spec ',': [$],
       param_spec -> <*> name_list ':' type: [ID],
       param_spec -> <*> type: [ID],
       name_list -> <*> name: [':'],
       name_list -> <*> name ',' name_list: [':'],
       name -> <*> ID: [',', ':'],
       type -> <*> ID: [ID]
     }
s2 = { def -> param_spec <*> return_spec ',': [$],
       return_spec -> <*> name ':' type: [','],
       return_spec -> <*> type: [','],
       name -> <*> ID: [':'],
       type -> <*> ID: [',']
     }
s7 = { name -> ID <*>: [',', ':'],
       type -> ID <*>: [ID]
     }
s8 = { name -> ID <*>: [':'],
       type -> ID <*>: [',']
     }
```

Figure 14.17: LR-Zustände der in Abbildung 14.16 gezeigten Grammatik.

und *return_spec*, denn von diesen Regeln werden die Zustände s_7 und s_8 abgeleitet. Um zu verhindern, dass diese Zustände zusammengefasst werde, ändern wir die Regeln für die syntaktische Variable *return_spec* wie in der in Abbildung 14.18 gezeigten Grammatik `mysterious-solved.cup` ab, indem wir eine zusätzliche Grammatik-Regel

return_spec → `ID BOGUS`

einführen. Wenn das Terminal `BOGUS` nie vom Scanner erzeugt werden kann, dann ändert sich durch die Hinzunahme dieser Regel die von der Grammatik erzeugte Sprache nicht. Allerdings ändern sich nun die LR-Zustände. Abbildung 14.19 zeigt, wie sich die entsprechenden Zustände ändern. Insbesondere sehen wir, dass der Zustand s_8 nun eine weitere markierte Regel enthält, zu der es in dem Zustand s_7 kein äquivalent gibt. Die Konsequenz ist, dass diese Zustände beim Übergang von den LR-Zuständen zu den LALR-Zuständen nicht mehr zusammengefasst werden können, da sie unterschiedliche Kerne haben. Daher gibt es dann auch keinen Konflikt mehr.

14.7 Auflösung von Shift/Reduce-Konflikten

Es gibt im Wesentlichen zwei Arten von Shift/Reduce-Konflikten:

1. Konflikte, die auf eine Mehrdeutigkeit der Grammatik zurückzuführen sind.

   Solche Mehrdeutigkeits-Konflikte hatten wir beispielsweise in der Grammatik für arithmetische Ausdrücke, die in Abbildung 13.7 auf Seite 184 gezeigt ist, diskutiert. In dem Beispiel waren diese Konflikte darauf zurückzuführen, dass durch die Grammatik keine Präzedenzen für die arithmetischen Operatoren festgelegt wurde, so dass am Ende nicht klar war, ob ein Ausdruck der Form "`1+2*3`" als "`1+(2*3)`" oder als "`(1+2)*3`" zu interpretieren ist.

   Wir haben bereits früher in Abschnitt 14.4 besprochen, wie solche Konflikte durch die Spezifikation von Operator-Präzedenzen aufgelöst werden können.

2. Konflikte, die entstehen, weil ein Look-Ahead von einem Token nicht ausreichend ist um zu entscheiden, ob das nächste Token der Eingabe auf den Stack geschoben werden soll, oder ob stattdessen der Stack durch

```
def : param_spec return_spec ','
    ;
param_spec
    : type
    |  name_list ':' type
    ;
return_spec
    : type
    | name ':' type
    | ID BOGUS      // this never happens
    ;
type: ID
    ;
name: ID
    ;
name_list
    : name
    | name ',' name_list
    ;
```

Figure 14.18: Auflösung des mysteriösen Reduce/Reduce-Konflikts.

```
s0 = { S -> <*> def: [$],
       def -> <*> param_spec return_spec ',': [$],
       name -> <*> ID: [',', ':'],
       name_list -> <*> name: [':'],
       name_list -> <*> name ',' name_list: [':'],
       param_spec -> <*> name_list ':' type: [ID],
       param_spec -> <*> type: [ID],
       type -> <*> ID: [ID]
     }
s2 = { def -> param_spec <*> return_spec ',': [$],
       name -> <*> ID: [':'],
       return_spec -> <*> ID BOGUS: [','],
       return_spec -> <*> name ':' type: [','],
       return_spec -> <*> type: [','],
       type -> <*> ID: [',']
     }
s7 = { name -> ID <*>: [',', ':'],
       type -> ID <*>: [ID]
     }
s8 = { name -> ID <*>: [':'],
       return_spec -> ID <*> BOGUS: [','],
       type -> ID <*>: [',']
     }
```

Figure 14.19: Einige Zustände der in Abbildung 14.18 gezeigten Grammatik.

Anwendung einer Grammatik-Regel reduziert werden muss. Solche *Look-Ahead*-Konflikte werden wir in diesem Abschnitt diskutieren.

Als Beispiel für einen Look-Ahead-Konflikt betrachten wir die in Abbildung 14.20 gezeigte Grammatik `lambda.cup`. Diese Grammatik beschreibt drei Arten von Ausdrücken:

1. λ-Ausdrücke haben syntaktisch die Form

   `[`x_1`, ···, `x_n`] |->` e.

   Ein solcher Ausdruck steht für eine Funktion, die n Argumente x_1, $\cdots$, x_n verarbeitet und als Ergebnis den Ausdruck e zurück liefert, wobei der Ausdruck e im Allgemeinen von den Parametern x_1, $\cdots$, x_n abhängen wird. Ein konkretes Beispiel wäre etwa der λ-Ausdruck

   `[x, y] |-> [x, y, x]`.

2. Zusätzlich sind als Ausdrücke Variablen-Namen zugelassen.

3. Außerdem sind auch Listen von Ausdrücken möglich, wobei diese Listen in eckigen Klammern eingefasst werden. Diese Listen können dabei beliebig geschachtelt sein.

```
1   terminal MAPSTO, LBRACKET, RBRACKET, COMMA, IDENTIFIER;
2
3   nonterminal expr, exprList, lambdaDefinition, identifierList;
4
5   expr ::= lambdaDefinition
6         |  IDENTIFIER
7         |  LBRACKET exprList RBRACKET
8         ;
9   lambdaDefinition
10       ::= LBRACKET identifierList RBRACKET MAPSTO expr
11        ;
12  identifierList
13       ::= IDENTIFIER COMMA identifierList
14        |  IDENTIFIER
15        ;
16  exprList
17       ::= expr COMMA exprList
18        |  expr
19        ;
```

Figure 14.20: Eine CUP-Grammatik für λ-Ausdrücke.

Versuchen wir mit CUP einen Parser für diese Grammatik zu erzeugen, so erhalten wir verschiedene Shift/Reduce-Konflikte. Diese Konflikte entstehen in dem Zustand Nummer 6, der in Abbildung 14.21 gezeigt ist. Da auf eine *expr* ein "`,`" folgen kann, dieses aber in dem Zustand gleichzeitig auch auf den Stack geschoben werden kann, ist nicht klar, ob mit der Regel

expr $\rightarrow$ `IDENTIFIER`

reduziert werden darf, wenn das Look-Ahead-Token ein Komma ist. Um zu entscheiden, ob der Parser versucht eine Liste von `IDENTIFIER`n zu parsen, müsste der Parser bis zu dem Token `MAPSTO` schauen können. Wenn später ein solches Token noch kommt, dann würde der Parser im Zustand 6 versuchen, eine `identifierList` zu parsen und das Komma sollte auf den Stack geschoben werden. Andernfalls könnte mit der Regel

expr $\rightarrow$ `IDENTIFIER`

reduziert werden. Leider lässt ein LR-Parser-Generator nicht zu, dass wir den noch ungelesenen Teil der Eingabe inspizieren. Wir müssen daher eine andere Lösung suchen.

```
1  lalr_state [6]: {
2    [identifierList ::= IDENTIFIER (*) COMMA identifierList , {RBRACKET }]
3    [expr ::= IDENTIFIER (*) , {RBRACKET COMMA }]
4    [identifierList ::= IDENTIFIER (*) , {RBRACKET }]
5  }
```

Figure 14.21: Der Zustand mit der Nummer 6 zu der Grammatik aus Abbildung 14.20

Wir können den Shift/Reduce-Konflikt lösen, indem wir die Grammatik wie in der in Abbildung 14.22 gezeigten Grammatik `lambda-generalized.cup` **verallgemeinern**. Diese Grammatik lässt auch Ausdrücke zu, bei denen in der Argument-Liste nicht nur Variablen-Namen enthält, sondern Ausdrücke beliebiger Komplexität. Damit beschreibt diese Grammatik eine Sprache, die eigentlich zu allgemein ist. Es ist aber ein leichtes, später den resultierenden Syntax-Baum drauf hin zu untersuchen, ob in der Parameter-Liste tatsächlich nur Variablen stehen oder nicht. Daher bietet eine solche Verallgemeinerung der Grammatik eine praktische Möglichkeit um Shift/Reduce-Konflikte zu lösen.

```
1  terminal    MAPSTO, LBRACKET, RBRACKET, COMMA, IDENTIFIER;
2
3  nonterminal expr, exprList, lambdaDefinition;
4
5  expr ::= lambdaDefinition
6         |  IDENTIFIER
7         |  LBRACKET exprList RBRACKET
8         ;
9
10 lambdaDefinition
11       ::= LBRACKET exprList RBRACKET MAPSTO expr
12        ;
13
14 exprList
15       ::= expr COMMA exprList
16        |  expr
17        ;
```

Figure 14.22: Die verallgemeinerte Grammatik für λ-Ausdrücke.

Chapter 15

Types and Type Checking*

There are two fundamentally different approaches to typing. Either the user is required to declare the types of variables or instead the program has to check the types of objects at runtime. In the first case, the programming language is called *statically typed*, in the second case it is called *dynamically typed*.

1. Statically typed languages like *Java* or `C` require that the user declares the types of functions and variables. The compiler is then able to check that these variables will indeed have the declared type at runtime. This approach has the following advantages:

   1. A number of runtime errors can be excluded. For example, if a variable is declared as a `float` in `C`, we are guaranteed that the program will not try to store a string in this variable.
   2. The program does not need to check the type of variables at runtime and can this be more efficient.
   3. Adding type information serves as a form of documentation that can be checked automatically. This enhances the readability of typed programs.
   4. Typed programs are easier to maintain as many violations of interfaces between different parts of a typed program will actually manifest itself as type errors. Therefore, the compiler is able to detect these violations.

2. Dynamically typed languages like *Perl*, *Python*, or *JavaScript* do not require the programmer to declare any types. Rather, the types are checked at runtime. For example, if a program in a dynamically typed language contains an expression of the form

   $$x + y,$$

   the compiler generates code that checks the type of x and y at runtime. Then, if x and y are discovered to be integers, the program performs an integer addition. However, if x and y happen to be strings, these strings are concatenated. Dynamic typing has the following advantages:

   1. Programs written in dynamically typed languages are typically shorter than the corresponding programs in statically typed languages.
   2. If the types used in an algorithm are very complex, then using an untyped language is sometimes the only way to code an algorithm in the intended way. The reason is that in some cases the expressiveness of current type system is not sufficient to be able to code certain algorithms conveniently.

The disadvantages of statically typed languages are the advantages of dynamically typed languages and vice versa. Therefore, dynamically typed languages are often used for prototyping, while statically typed languages are used for production systems.

In practice, most statically typed languages offer some escape mechanism to cover the cases where the type system gets to unwieldy. For example, in `C` the programmer can declare a variable x to have type `void*` and then cast this pointer to any other pointer type. In *Java*, programmer can declare a variable x as having type `Object`. This variable can then hold any object and when this variable is used the user needs to cast it to the appropriate type.

In this chapter we are going to show how the compiler is able to type check a statically typed program. To this end, we first introduce a very simple statically typed programming language and then we develop a type checker for this language.

15.1 Eine Beispielsprache

Wir stellen jetzt die Sprache TTL (*typed term language* vor. Dabei handelt es sich um eine sehr einfache Beispielsprache, die es dem Benutzer ermöglicht

- Typen zu definieren,
- Funktionen zu deklarieren und
- Terme anzugeben,

für die dann die Typ-Korrektheit nachgewiesen wird. Die Sprache TTL ist sehr einfach gehalten, damit wir uns auf die wesentlichen Ideen der Typ-Überprüfung konzentrieren können. Daher können wir in TTL auch keine wirklichen Programme schreiben, sondern einzig und allein überprüfen, ob Ausdrücke wohlgetypt sind.

```
1  type list(X) := nil + cons(X, list(X));
2
3  signature concat: list(T) * list(T) -> list(T);
4  signature x: int;
5  signature y: int;
6  signature z: int;
7
8  concat(nil, nil): list(int);
9  concat(cons(x, nil), cons(y, cons(z, nil))): list(int);
```

Figure 15.1: Ein TTL-Beispiel-Programm

Abbildung 15.1 zeigt ein einfaches Beispiel-Programm. Die Schlüsselwörter habe ich unterstrichen. Wir diskutieren dieses Programm jetzt im Detail.

1. In Zeile 1 definieren wir den *generischen* Typ $\texttt{list}(X)$. Das X ist hier der Typ-Parameter, der später durch einen konkreten Typ wie z.B. `int`, `string` oder $\texttt{list}(\texttt{string})$ ersetzt werden kann.

   Semantisch ist die Zeile als induktive Definition zu lesen, durch die eine Menge von Termen definiert wird, wobei X eine gegebene Menge bezeichnet. *nil* und *cons* werden in diesem Zusammenhang als Funktions-Zeichen verwendet. Formal hat die induktive Definition die folgende Gestalt:

   1. Induktions-Anfang: Der Term *nil* ist ein Element der Menge $\mathit{List}(X)$:

      $$\mathit{nil} \in \mathit{list}(X)$$

   2. Induktions-Schritt: Falls a ein Element der Menge X und l ein Element der Menge $\mathit{list}(X)$ ist, dann ist der Term $\mathit{cons}(a, l)$ ebenfalls ein Element der Menge $\mathit{list}(X)$:

      $$a \in X \wedge l \in \mathit{list}(X) \rightarrow \mathit{cons}(a, l) \in \mathit{list}(X).$$

2. In Zeile 3 deklarieren wir die Funktion *concat* als zweistellige Funktion. Die beiden Argumente haben jeweils den Typ $\texttt{list}(T)$ und das Ergebnis hat ebenfalls diesen Typ.
3. In den Zeilen 4 – 6 legen wir fest, dass die Variablen x, y und z jeweils den Typ `int` haben.

4. In Zeile 8 und 9 werden schließlich die beiden Terme

   $$\texttt{concat}(\texttt{nil}, \texttt{nil}) \quad \text{und} \quad \texttt{concat}(\texttt{cons}(\texttt{x}, \texttt{nil}), \texttt{cons}(\texttt{y}, \texttt{cons}(\texttt{z}, \texttt{nil})))$$

   angegeben und es wird behauptet, dass diese den Typ $\texttt{list}(\texttt{int})$ haben. Die Aufgabe der Typ-Überprüfung besteht darin, diese Aussage zu verifizieren.

```
1  grammar ttl;
2
3  program   : typeDef* signature* typedTerm*
4            ;
5
6  typeDef   : 'type' FCT ':=' type ('+' type)* ';'
7            | 'type' FCT '(' PARAM (',' PARAM)* ')' ':=' type ('+' type)* ';'
8            ;
9
10 type      : FCT '(' type (',' type)* ')'
11           | FCT
12           | PARAM
13           ;
14
15 signature : 'signature' FCT ':' type ('*' type)* '->' type ';'
16           | 'signature' FCT ':' type ';'
17           ;
18
19 term      : FCT '(' term (',' term)* ')'
20           | FCT
21           ;
22
23 typedTerm : term ':' type ';'
24           ;
25
26 PARAM   : ('A'..'Z')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
27 FCT     : ('a'..'z')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

Figure 15.2: Eine EBNF-Grammatik für die getypte Beispielsprache

Die genaue Syntax der Sprache TTL wird durch die in Abbildung 15.2 gezeigte Grammatik definiert. Diese Grammatik verwendet neben den Zeichen-Reihen, die in doppelten Anführungs-Zeichen gesetzt sind, die folgenden Terminale:

1. FUNCTION bezeichnet entweder einen Typ-Konstruktor wie `list`, eine Variable wie `x` oder `y` oder einen Funktionsnamen wie `concat`. Syntaktisch werden Typ-Konstruktoren, Variablen und Funktionsnamen dadurch erkannt, dass sie mit einem Kleinbuchstaben beginnen.
2. PARAMETER bezeichnet einen Typ-Parameter wie z.B. das X in $\texttt{list}(X)$. Diese beginnen immer mit einem Großbuchstaben.

Bevor wir einen Algorithmus zur Typ-Überprüfung vorstellen können ist es erforderlich, einige grundlegende Begriffe wie den Begriff der Substitution und die Anwendung von Substitutionen auf Typen zu diskutieren.

15.2 Grundlegende Begriffe

Als erstes definieren wir den Begriff der Signatur eines Funktions-Zeichens. Die *Signatur* eines Funktionszeichens legt fest,

- welchen Typ die Argumente der Funktion haben und
- von welchem Typ das Ergebnis der Funktion ist.

Wir geben die Signatur einer Funktion f in der Form

$$f : \sigma_1 \times \cdots \times \sigma_n \to \varrho$$

an. Damit spezifizieren wir:

1. Die Funktion f erwartet n Argumente.
2. Das i-te Argument hat den Typ σ_i.
3. Das von der Funktion berechnete Ergebnis hat den Typ ϱ.

Als nächstes definieren wir, was wir unter einem Typ verstehen wollen. Anschaulich sind das Ausdrücke wie

$$\mathit{map}(K, V), \quad \mathit{double}, \quad \text{oder} \quad \mathit{list}(\mathit{int}).$$

Formal werden Typen aus Typ-Parametern und Typ-Konstruktoren aufgebaut. In dem obigen Beispiel sind *map*, *double*, *list* und *int* Typ-Konstruktoren, während K und V Typ-Parameter sind. Wir nehmen an, dass einerseits eine Menge $\mathbb{P}$ von Typ-Parametern und andererseits eine Menge von Typ-Konstruktoren $\mathbb{K}$ gegeben sind. In dem letzten Beispiel könnten wir

$$\mathbb{K} = \{\mathit{map}, \mathit{list}, \mathit{int}, \mathit{double}\} \quad \text{und} \quad \mathbb{P} = \{K, V\}$$

setzen. Zusätzlich muss noch eine Funktion

$$\mathit{arity} : \mathbb{K} \to \mathbb{N}$$

gegeben sein, die für jeden Typ-Konstruktor festlegt, wieviel Argumente er erwartet. In dem obigen Beispiel hätten wir

$$\mathit{arity}(\mathit{map}) = 2, \quad \mathit{arity}(\mathit{list}) = 1, \quad \mathit{arity}(\mathit{int}) = 0 \quad \text{und} \quad \mathit{arity}(\mathit{double}) = 0.$$

Dann wird die Menge $\mathcal{T}$ der Typen induktiv definiert:

1. Jeder Typ-Parameter X ist ein Typ:
$$X \in \mathbb{P} \Rightarrow X \in \mathcal{T}.$$
2. Ist c ein Typ-Konstruktor mit $\mathit{arity}(c) = 0$, so ist auch c ein Typ:
$$c \in \mathbb{K} \wedge \mathit{arity}(c) = 0 \Rightarrow c \in \mathcal{T}.$$
3. Ist f ein n-stelliger Typ-Konstruktor und sind τ_1, $\cdots$, τ_n Typen, so ist auch $f(\tau_1, \cdots, \tau_n)$ ein Typ:
$$f \in \mathbb{K} \wedge \mathit{arity}(f) = n \wedge n > 0 \wedge \tau_1 \in \mathcal{T} \wedge \cdots \wedge \tau_n \in \mathcal{T} \Rightarrow f(\tau_1, \cdots, \tau_n) \in \mathcal{T}.$$

Examples: If the type constructors *map*, *int*, *list*, and *double* have the arities given above and if, furthermore, K and V are type parameters, then the following are types:

1. *int*,
2. *double*,
3. *list*(*double*),
4. *list*(V),
5. *map*(K, *list*(*double*)). ◇

Definition 44 (Parameter Substitution) *A* parameter substitution *is a finite set of pairs of the form*

$$\sigma = \{\langle X_1, \tau_1\rangle, \cdots, \langle X_n, \tau_n\rangle\}$$

such that

1. X_i *is a type parameter for all* $i \in \{1, \cdots, n\}$,
2. τ_i *is a type for all* $i \in \{1, \cdots, n\}$,
3. *the type parameters occurring in* σ *are pairwise distinct, that is we have*

$$i \neq j \rightarrow X_i \neq X_j \quad \text{for all } i, j \in \{1, \cdots, n\}.$$

If $\sigma = \big\{\langle X_1, \tau_1\rangle, \cdots, \langle X_n, \tau_n\rangle\big\}$ *is a parameter substitution, then* σ *is written as*

$$\sigma = \big[X_1 \mapsto \tau_1, \cdots, X_n \mapsto \tau_n\big].$$

In this case, the domain *of* σ *is defined as*

$$\mathit{dom}(\sigma) = \{X_1, \cdots, X_n\}.$$

The set of all parameter substitutions is denoted as SUBST. *In the following,* parameter substitutions *are just called* substitutions. □

Substitutions can be *applied* to types. If τ is a type and $\vartheta = \big[X_1 \mapsto \tau_1, \cdots, X_n \mapsto \tau_n\big]$ is a substitution, then $\tau\vartheta$ is the type that we get if we replace all occurrences of X_i in τ by τ_i, The formal definition follows.

Definition 45 (Application of a Substitution)
If τ *is a type and* $\vartheta = \big[X_1 \mapsto \tau_1, \cdots, X_n \mapsto \tau_n\big]$ *is a substitution, then the* application *of* ϑ *to* τ *(written* $\tau\vartheta$*) is defined by induction on* τ*:*

1. *If* τ *is a type parameter, there are two cases:*
   1. $\tau = X_i$ *for some* $i \in \{1, \cdots, n\}$. *Then*
      $$X_i\vartheta := \tau_i.$$
   2. $\tau = Y$ *where* Y *is a type parameter such that* $Y \notin \{X_1, \cdots, X_n\}$. *Then*
      $$Y\vartheta := Y.$$
2. *Otherwise* τ *must have the form* $\tau = f(\sigma_1, \cdots, \sigma_m)$. *Then* $\tau\vartheta$ *is defined as*
   $$f(\sigma_1, \cdots, \sigma_m)\vartheta := f(\sigma_1\vartheta, \cdots, \sigma_m\vartheta).$$

□

Examples: Define the substitution ϑ as

$$\vartheta := \big[X_1 \mapsto \texttt{double},\ X_2 \mapsto \texttt{list(int)}\big].$$

Then we have the following:

1. $X_3\vartheta = X_3$,
2. $\texttt{list}(X_2)\vartheta = \texttt{list}\big(\texttt{list(int)}\big)$,
3. $\texttt{map}(X_1, \texttt{set}(X_2))\vartheta = \texttt{map}\big(\texttt{double}, \texttt{set}(\texttt{list(int)})\big)$.

Next, we show how substitutions can be composed.

Definition 46 (Composition of Substitutions) *If*

$$\vartheta = \big[X_1 \mapsto \sigma_1, \cdots, X_m \mapsto \sigma_m\big] \quad \text{and} \quad \eta = \big[Y_1 \mapsto \tau_1, \cdots, Y_n \mapsto \tau_n\big]$$

are substitutions such that $\mathit{dom}(\vartheta) \cap \mathit{dom}(\eta) = \{\}$, *then we define the* composition $\vartheta\eta$ *of* ϑ *and* η *as*

$$\vartheta\eta := \big[X_1 \mapsto \sigma_1\eta, \cdots, X_m \mapsto \sigma_m\eta,\ Y_1 \mapsto \tau_1, \cdots, Y_n \mapsto \tau_n\big].$$

□

Example: Define

$$\vartheta := \big[X_1 \mapsto \texttt{double},\ X_2 \mapsto \texttt{list}(\texttt{X}_3)\big] \quad \text{and}$$

$$\eta := \big[X_3 \mapsto \texttt{map}(\texttt{int}, \texttt{double}),\ X_4 \mapsto \texttt{char}\big].$$

Then we have

$$\vartheta\eta = \big[X_1 \mapsto \texttt{double},\ X_2 \mapsto \texttt{list}(\texttt{map}(\texttt{int}, \texttt{double})),\ X_3 \mapsto \texttt{map}(\texttt{int}, \texttt{double}),\ X_4 \mapsto \texttt{char}\big]. \quad \diamond$$

Exercise 41: How would we have to change the definition of the composition of ϑ and η if we drop the condition $dom(\vartheta) \cap dom(\eta) = \{\}$?

The following theorem is a consequence of the previous definition.

Theorem 47 (Associativity of Composition)
If τ is a type and ϑ and η are substitutions such that $dom(\vartheta) \cap dom(\eta) = \{\}$ holds, then we have

$$(\tau\vartheta)\eta = \tau(\vartheta\eta). \quad \square$$

This theorem is proved by an easy induction on τ.

Next, we formalize the notion of a term t being of type τ.

Definition 48 ($t : \tau$)
For a term t and a type τ the relation $t : \tau$ (read: t has type τ) is defined by induction on t.

1. *If c is a nullary function symbol with signature $c : \tau$ and if ϑ is any parameter substitution, then*

   $$c : \tau\vartheta.$$

2. *Assume that*

   1. *f is an n-ary function symbol such that $n > 0$.*
   2. *f has the signature*

      $$f : \sigma_1 \times \cdots \times \sigma_n \to \varrho.$$

   3. *ϑ is a substitution such that we have $t_i : \sigma_i\vartheta$.*

   Then $f(t_1, \cdots, t_n) : \varrho\vartheta$.

Examples: The following examples assume that the types and signatures are given as in Figure 15.1 on page 222.

1. We have *nil* : $\texttt{list}(\texttt{int})$, because *nil* has the signature

   nil : $\texttt{list}(\texttt{X})$.

   Therefore, defining

   $$\vartheta = [X \mapsto \texttt{int}]$$

   gives $\texttt{list}(X)\vartheta = \texttt{list}(\texttt{int})$ showing the claim.

2. Next, we show

   concat(*nil*, *nil*) : $\texttt{list}(\texttt{int})$.

   The signature of *concat* is

   concat : $list(T) \times list(T) \to list(T)$.

   Define

   $$\vartheta = [T \mapsto \texttt{int}].$$

   We have already seen that

   nil : $\texttt{list}(\texttt{int})$

   holds. Because of $\texttt{list}(\texttt{T})\vartheta = \texttt{list}(\texttt{int})$ this shows the claim. $\diamond$

15.3 A Type Checking Algorithm

Assume a term $t = f(t_1, \cdots, t_n)$ and a type τ is given and we want to check whether $t : \tau$ holds. Furthermore, assume that the signature of f is given as

$$f : \sigma_1 \times \cdots \times \sigma_n \to \varrho.$$

According to the definition of $t : \tau$ the term t has type τ iff there is a parameter substitution ϑ such that $\varrho\vartheta = \tau$ and $t_i : \sigma_i\vartheta$. Therefore

$$f(t_1, \cdots, t_n) : \tau \;\Leftrightarrow\; \exists \vartheta \in \textsc{Subst} : \big(\varrho\vartheta = \tau \wedge \forall i \in \{1, \cdots, n\} : t_i : \sigma_i\vartheta\big)$$

The problem of type checking is to compute the substitution ϑ or to show that ϑ can not exist.

Examples: We discuss the previous examples (that is *nil* : `list(int)` and *concat*(*nil*, *nil*) : `list(int)`) again using the formula

$$f(t_1, \cdots, t_n) : \tau \;\Leftrightarrow\; \exists \vartheta \in \textsc{Subst} : \big(\varrho\vartheta = \tau \wedge \forall i \in \{1, \cdots, n\} : t_i : \sigma_i\vartheta\big).$$

1. Lets prove *nil* : `list(int)` again. We have the signature

   nil : `list(X)`.

   Therefore, we have

   nil : `list(int)` $\;\Leftrightarrow\; \exists \vartheta \in \textsc{Subst} : \big(\texttt{list}(X)\vartheta = \texttt{list(int)}\big)$.

   Obviously, we can define

   $$\vartheta = [X \mapsto \texttt{int}].$$

   Then, the equation $\texttt{list}(X)\vartheta = \texttt{list(int)}$ is true and we conclude that *nil* : `list(int)` holds.

2. Next, we prove *concat*(*nil*, *nil*) : `list(int)`. The signature of *concat* is given as

   concat : *list*$(T) \times$ *list*$(T) \to$ *list*(T).

   Therefore, our claim is equivalent to

   $$\begin{aligned} & \textit{concat}(\textit{nil}, \textit{nil}) : \texttt{list(int)} \\ \Leftrightarrow\quad & \exists \vartheta \in \textsc{Subst} : \big(\texttt{list}(T)\vartheta = \texttt{list(int)} \wedge \textit{nil} : \textit{list}(T)\vartheta \wedge \textit{nil} : \textit{list}(T)\vartheta\big). \end{aligned}$$

   The equation $\texttt{list}(T)\vartheta = \texttt{list(int)}$ is solved by the substitution

   $$\vartheta = [T \mapsto \texttt{int}]$$

   and therefore the correctness of

   concat(*nil*, *nil*) : `list(int)`

   is reduced to the correctness of

   nil : *list*$(T)\vartheta$,

   As *list*$(T)\vartheta$ = *list*(*int*) and *nil* : *list*(*int*) holds, the proof is complete. □

The previous examples show that the correctness of an expression $t : \tau$ can be reduced to the solution of a set of *type equations* of the form $\varrho\vartheta = \tau$. We formalize this observation by defining a function

$$\mathit{typeEqs} : \texttt{Term} \times \texttt{Type} \to \texttt{set}(\texttt{Equation}).$$

For a term t and a type τ

$$\mathit{typeEqs}(t, \tau)$$

yields a set of type equations. These type equations will be solvable if and only if $t : \tau$ is correct. The function *typeEqs* is defined as follows:

$$\big(f : \sigma_1 \times \cdots \times \sigma_n \to \varrho\big) \implies \mathit{typeEqs}\big(f(t_1, \cdots, t_n), \tau\big) := \{\varrho = \tau\} \cup \bigcup_{i=1}^{n} \mathit{typeEqs}(t_i, \sigma_i).$$

This definition has to be read as follows: If the function f has the signature $f : \sigma_1 \times \cdots \times \sigma_n \to \varrho$, then $f(t_1, \cdots, t_n) : \tau$ is true if and only if the set of *type equations*

$$\{\varrho = \tau\} \cup \bigcup_{i=1}^{n} \mathit{typeEqs}(t_i, \sigma_i)$$

has a solution ϑ. We formalize the notion of a *type equation* next.

Definition 49 (Type Equation) *If σ and τ are types, then the expression*

$$\sigma \doteq \tau$$

is called a type equation. *A* system of type equations *is a set of type equations. A substitution ϑ* solves *a type equation $\sigma \doteq \tau$ if and only if $\sigma\vartheta = \tau\vartheta$. A substitution ϑ solves a system of type equations E iff it solves every type equation in E.* □

In order to implement type checking we still need an algorithm to solve systems of type equations. The crucial observation is that type equations can be solved by unification. We do not need the general form of unification because if we have a type equation of the form $\varrho = \tau$ then only the left hand side ϱ will contain type parameters. We use the algorithm given by Martelli and Montanari [MM82]. This algorithm works with pairs of the form

$$\langle E, \vartheta \rangle$$

where E is a set of type equations and ϑ is a substitution. We start with the pair

$$\langle E, [] \rangle.$$

Here, the substitution is empty, while E is the set of type equations we want to solve. These pairs will be gradually transformed until we arrive a a pair of the form

$$\langle \{\}, \vartheta \rangle$$

such that ϑ is the solution of the system of type equations E. We use the following rules to rewrite the pairs.

1. If X is a type parameter, then

   $$\Big\langle E \cup \big\{X \doteq \tau\big\}, \vartheta \Big\rangle \quad \rightsquigarrow \quad \Big\langle E[X \mapsto \tau], \vartheta\big[X \mapsto \tau\big] \Big\rangle.$$

   This works as follows: If E contains a type equation of the form $X \doteq \tau$, then we can remove this type equation from E if we incorporate this equation in the substitution ϑ by transforming ϑ into the new substitution $\vartheta\big[X \mapsto \tau\big]$. Of course, we also have to apply the substitution $[X \mapsto \tau]$ to the remaining type equations in E.

2. If f is an n-ary type constructor we have the rule

   $$\Big\langle E \cup \big\{f(\sigma_1, \cdots, \sigma_n) \doteq f(\tau_1, \cdots, \tau_n)\big\}, \vartheta \Big\rangle \rightsquigarrow \Big\langle E \cup \big\{\sigma_1 \doteq \tau_1, \cdots, \sigma_n \doteq \tau_n\big\}, \vartheta \Big\rangle.$$

   Therefore, a type equation of the form $f(\sigma_1, \cdots, \sigma_n) \doteq f(\tau_1, \cdots, \tau_n)$ is replaced by the n type equations $\sigma_1 \doteq \tau_1$, $\cdots$, $\sigma_n \doteq \tau_n$.

A special case of this occurs if $n = 0$. It reads

$$\Big\langle E \cup \big\{c \doteq c\big\}, \vartheta \Big\rangle \;\rightsquigarrow\; \Big\langle E, \vartheta \Big\rangle.$$

Here c is a nullary type constructor. This rule just says that trivial type equations can be dropped.

3. A system of type equations of the form $E \cup \big\{f(\sigma_1, \cdots, \sigma_m) \doteq g(\tau_1, \cdots, \tau_n)\big\}$ has no solution if f and g are different. Therefore, we have

$$\Big\langle E \cup \big\{f(\sigma_1, \cdots, \sigma_m) \doteq g(\tau_1, \cdots, \tau_n)\big\}, \vartheta \Big\rangle \;\rightsquigarrow\; \Omega \quad \text{if } f \neq g.$$

Here Ω denotes unsolvability. If the solution of *typeEqs*(t, τ) yields Ω, then the set of type equations E can not be solved and $t : \tau$ is wrong. ◇

Example: We demonstrate the algorithm by solving the type equation

$$\texttt{map}(X_2, X_3) \doteq \texttt{map}(\texttt{char}, \texttt{list}(\texttt{int})).$$

We proceed as follows:

$$\begin{aligned}
& \big\langle \big\{\texttt{map}(X_2, X_3) \doteq \texttt{map}(\texttt{char}, \texttt{list}(\texttt{int}))\big\}, [] \big\rangle \\
\rightsquigarrow\; & \big\langle \big\{X_2 \doteq \texttt{char},\; X_3 \doteq \texttt{list}(\texttt{int})\big\}, [] \big\rangle \\
\rightsquigarrow\; & \big\langle \big\{X_3 \doteq \texttt{list}(\texttt{int})\big\}, \big[X_2 \mapsto \texttt{char}\big] \big\rangle \\
\rightsquigarrow\; & \big\langle \{\}, \big[X_2 \mapsto \texttt{char},\; X_3 \mapsto \texttt{list}(\texttt{int})\big] \big\rangle
\end{aligned}$$

In this case, the algorithm is successful and the resulting substitution

$$\big[X_2 \mapsto \texttt{char},\; X_3 \mapsto \texttt{list}(\texttt{int})\big]$$

is a solution of the type equation given above.

15.4 Implementierung eines Typ-Checkers für TTL

Zur Illustration der dargestellten Theorie implementieren wir einen Typ-Checker für TTL. Die Grammatik hatten wir ja bereits in Abbildung 15.2 auf Seite 223 präsentiert. Die Abbildungen 15.3, 15.4 und 15.5 auf den folgenden Seiten zeigen die *JavaCup*-Spezifikation eines Parsers für diese Grammatik. Der zugehörige Scanner ist in Abbildungen 15.7 auf Seite 234 gezeigt. Der Scanner unterscheidet in den Zeilen 34 und 35 zwischen Namen, die mit einem großen Buchstaben beginnen und solchen Namen, die mit einem kleinen Buchstaben beginnen. Erstere bezeichnen Typ-Parameter, letztere bezeichnen sowohl Funktionen als auch Typ-Konstruktoren. Der von *JavaCup* generierte Parser baut einen abstrakten Syntax-Baum auf. Die zugehörigen Klassen wurden mit Hilfe des im vorhergehenden Abschnitts diskutierten Klassen-Generators EP aus der in Abbildung 15.6 gezeigten Spezifikation erzeugt.

Die Klasse `MartelliMontanari` enthält die in Abbildung 15.8 gezeigte Methode *solve*(), mit der sich ein syntaktisches Gleichungs-System lösen lässt. Wir diskutieren diese Methode jetzt im Detail.

1. Am Anfang wählen wir in Zeilen 3 willkürlich die erste syntaktische Gleichung aus der Menge `mEquations` der zu lösenden syntaktischen Gleichungen aus.

   Das weitere Vorgehen richtet sich dann nach der Art der Gleichung.

2. In den Zeilen 6 – 14 behandeln wir den Fall, dass auf der linken Seite der syntaktischen Gleichung ein Typ-Parameter steht. In diesem Fall formen wir das syntaktische Gleichungs-System nach der folgenden Regel um:

   $$\Big\langle E \cup \big\{X \doteq \tau\big\}, \vartheta \Big\rangle \quad \rightsquigarrow \quad \Big\langle E[X \mapsto \tau], \vartheta\big[X \mapsto \tau\big] \Big\rangle.$$

   Der Typ-Parameter, der in der obigen Formel mit `X` bezeichnet wird, trägt im Programm den Namen `var` und der Typ τ wird im Programm mit `rhs` bezeichnet.

3. In Zeile 15 betrachten wir den Fall, dass auf der linken Seite der syntaktischen Gleichung ein zusammengesetzter Typ steht. Wenn die Typ-Gleichung lösbar sein soll, muss dann auch auf der rechten Seite ein zusammengesetzter Typ stehen. Dies wird in Zeile 16 überprüft.

   Der restliche Code beschäftigt sich nun mit dem Fall, dass auf beiden Seiten der syntaktischen Gleichung ein zusammengesetzter Typ steht.

4. Die Zeilen 22 – 25 behandeln den Fall, dass die Typ-Konstruktoren auf beiden Seiten verschieden sind, es wird also der Fall

   $$\Big\langle E \cup \big\{f(\sigma_1, \cdots, \sigma_m) \doteq g(\tau_1, \cdots, \tau_n)\big\}, \vartheta \Big\rangle \;\rightsquigarrow\; \Omega.$$

   behandelt. Die Methode liefert in diesem Fall statt einer Substitution den Wert `null` zurück.

5. Die Zeilen 27 – 32 behandeln schließlich den Fall, in dem wir das Gleichungs-System nach der Regel

   $$\Big\langle E \cup \big\{f(\sigma_1, \cdots, \sigma_n) \doteq f(\tau_1, \cdots, \tau_n)\big\}, \vartheta \Big\rangle \;\rightsquigarrow\; \Big\langle E \cup \big\{\sigma_1 \doteq \tau_1, \cdots, \sigma_n \doteq \tau_n\big\}, \vartheta \Big\rangle.$$

   umformen.

Als letztes diskutieren wir die Berechnung der Typ-Gleichungen, die in der Methode *typeEqs*() der Klasse `Term` implementiert ist. Ein Term $f(s_1, \cdots, s_n)$ hat genau dann den Typ τ, falls die Funktion f eine Signatur

$$f : \sigma_1 \times \cdots \times \sigma_n \to \varrho$$

hat und es darüber hinaus eine Parameter-Substitution ϑ gibt, so dass $\tau = \varrho\vartheta$ gilt und weiterhin die Terme s_i vom Typ $\sigma_i\vartheta$ sind:

$$f(s_1, \cdots, s_n) : \tau \;\Leftrightarrow\; \exists \vartheta \in \textsc{Subst} : \varrho\vartheta = \tau \wedge s_1 : \sigma_1\vartheta \wedge \cdots \wedge s_n : \sigma_n\vartheta.$$

1. Um die obige Definition von $t : \tau$ umzusetzen, suchen wir für einen Term der Form $f(s_1, \cdots, s_n)$ zunächst in Zeile 2 nach der Signatur des Funktions-Zeichens f. Diese Signaturen sind in der Symboltabelle `map` hinterlegt. Findet sich für das Funktions-Zeichen keine Signatur, so liegt offenbar ein Fehler vor, der in Zeile 4 ausgegeben wird.

```
1  import java_cup.runtime.*;
2  import java.util.*;
3
4  terminal           TYPE, SIGNATURE, LEFT_PAR, RIGHT_PAR;
5  terminal           COMMA, COLON, SEMICOLON, ASSIGN, ARROW, PLUS, TIMES;
6  terminal String    FUNCTION, PARAMETER;
7
8  nonterminal Program         program;
9  nonterminal Term            term;
10 nonterminal List<Term>      termList;
11 nonterminal List<Parameter>  varList;
12 nonterminal Type            type;
13 nonterminal List<Type>      typeList;
14 nonterminal List<Type>      typeSum;
15 nonterminal TypeDef         typeDef;
16 nonterminal List<TypeDef>   typeDefList;
17 nonterminal Signature       signature;
18 nonterminal List<Signature> signatures;
19 nonterminal List<Type>      argTypes;
20 nonterminal TypedTerm       typedTerm;
21 nonterminal List<TypedTerm> typedTerms;
22
23 program     ::= typeDefList:typDefs signatures:signList typedTerms:termList
24                 {: RESULT = new Program(typDefs, signList, termList); :}
25                ;
26 typeDefList ::= typeDefList:l typeDef:t {: l.add(t); RESULT = l; :}
27                |  typeDef:t
28                   {: List<TypeDef> l = new LinkedList<TypeDef>();
29                      l.add(t); RESULT = l;
30                   :}
31                ;
32 typeDef     ::= TYPE FUNCTION:f ASSIGN typeSum:s SEMICOLON
33                 {: RESULT = new SimpleTypeDef(f, s); :}
34                |  TYPE FUNCTION:f LEFT_PAR varList:a RIGHT_PAR ASSIGN
35                       typeSum:s SEMICOLON
36                 {: RESULT = new ParamTypeDef(f, a, s); :}
37                ;
38 type        ::= FUNCTION:f LEFT_PAR typeList:t RIGHT_PAR
39                             {: RESULT = new CompositeType(f, t); :}
40                |  FUNCTION:f  {: RESULT = new CompositeType(f);    :}
41                |  PARAMETER:v {: RESULT = new Parameter(v);        :}
42                ;
43 typeList    ::= typeList:l COMMA type:t {: l.add(t); RESULT = l; :}
44                |  type:t {: List<Type> l = new LinkedList<Type>();
45                            l.add(t);
46                            RESULT = l;
47                         :}
48                ;
```

Figure 15.3: *JavaCup*-Spezifikation der TTL-Grammatik, 1. Teil

```
typeSum     ::= typeSum:l PLUS type:t {: l.add(t); RESULT = l; :}
             |  type:t
                {:
                   List<Type> l = new LinkedList<Type>();
                   l.add(t);
                   RESULT = l;
                :}
             ;
signature   ::= SIGNATURE FUNCTION:f COLON argTypes:a
                           ARROW type:t SEMICOLON
                {: RESULT = new Signature(f, a, t); :}
             |  SIGNATURE FUNCTION:f COLON type:t SEMICOLON
                {: List a = new LinkedList<Type>();
                   RESULT = new Signature(f, a, t);
                :}
             ;
signatures  ::= signatures:l signature:s {: l.add(s); RESULT = l; :}
             |  signature:s
                {: List<Signature> l = new LinkedList();
                   l.add(s);
                   RESULT = l;
                :}
             ;
argTypes    ::= argTypes:l TIMES type:t {: l.add(t); RESULT = l; :}
             |  type:t
                {: List<Type> l = new LinkedList();
                   l.add(t); RESULT = l;
                :}
             ;
varList     ::= varList:l COMMA PARAMETER:v
                            {: l.add(new Parameter(v)); RESULT = l; :}
             |  PARAMETER:v {: List<Parameter> l = new LinkedList();
                               l.add(new Parameter(v));
                               RESULT = l;
                            :}
             ;
term        ::= FUNCTION:f LEFT_PAR termList:l RIGHT_PAR
                           {: RESULT = new Term(f, l); :}
             |  FUNCTION:f {: List<Term> l = new LinkedList<Term>();
                              RESULT = new Term(f, l);
                           :}
             ;
termList    ::= termList:l COMMA term:t
                      {: l.add(t); RESULT = l;            :}
             |  term:t {: List<Term> l = new LinkedList();
                         l.add(t); RESULT = l;
                      :}
             ;
```

Figure 15.4: *JavaCup*-Spezifikation der TTL-Grammatik, 2. Teil

```
typedTerm   ::= term:t COLON type:s SEMICOLON
                {: RESULT = new TypedTerm(t, s); :}
             ;

typedTerms  ::= typedTerms:l typedTerm:t
                {: l.add(t); RESULT = l; :}
             |  typedTerm:t
                {: List<TypedTerm> l = new LinkedList<TypedTerm>();
                   l.add(t);
                   RESULT = l;
                :}
             ;
```

Figure 15.5: *JavaCup*-Spezifikation der TTL-Grammatik, 3. Teil

```
Program = Program(List<TypeDef>   typeDefs,
                  List<Signature> signatures,
                  List<TypedTerm> typedTerms);

TypeDef = SimpleTypeDef(String name, List<Type> typeSum)
        + ParamTypeDef(String       name,
                       List<String> parameters,
                       List<Type>   typeSum);

Type = Parameter(String name)
     + CompositeType(String name, List<Type> argTypes);

substitute: Type * Parameter * Type -> Type;

Signature = Signature(String name, List<Type> argList, Type result);

Term = Term(String function, List<Term> termList);

typeEqs: Term * Type * Map<String, Signature> -> List<Equation>;

TypedTerm = TypedTerm(Term term, Type type);

Substitution = Substitution(List<Parameter> variables, List<Type> types);

Equation = Equation(Type lhs, Type rhs);

substitute: Equation * Parameter * Term -> Equation;

MartelliMontanari =
    MartelliMontanari(List<Equation> equations, Substitution theta);
```

Figure 15.6: Definition der benötigten *Java*-Klassen mit Hilfe von EP.

2. Ansonsten speichern wir in Zeile 8 die Argument-Typen $\sigma_1, \cdots, \sigma_n$ in der Variablen `argTypes`. Die Signatur von f muss natürlich genau so viele Argument-Typen haben, wie das Funktions-Zeichen f Argumente hat.

```
1  import java_cup.runtime.*;
2
3  %%
4
5  %char
6  %line
7  %column
8  %cup
9
10 %{
11     private Symbol symbol(int type) {
12         return new Symbol(type, yychar, yychar + yylength());
13     }
14
15     private Symbol symbol(int type, Object value) {
16         return new Symbol(type, yychar, yychar + yylength(), value);
17     }
18 %}
19
20 %%
21
22 "+"               { return symbol( sym.PLUS      ); }
23 "*"               { return symbol( sym.TIMES     ); }
24 "("               { return symbol( sym.LEFT_PAR  ); }
25 ")"               { return symbol( sym.RIGHT_PAR ); }
26 ","               { return symbol( sym.COMMA     ); }
27 ":"               { return symbol( sym.COLON     ); }
28 ";"               { return symbol( sym.SEMICOLON ); }
29 ":="              { return symbol( sym.ASSIGN    ); }
30 "->"              { return symbol( sym.ARROW     ); }
31 "type"            { return symbol( sym.TYPE      ); }
32 "signature"       { return symbol( sym.SIGNATURE ); }
33
34 [a-z][a-zA-Z_0-9]* { return symbol(sym.FUNCTION,  yytext()); }
35 [A-Z][a-zA-Z_0-0]* { return symbol(sym.PARAMETER, yytext()); }
36
37 [ \t\n]           { /* skip white space */ }
38 "//" [^\n]*       { /* skip comments    */ }
39
40 [^] { throw new Error("Illegal character '" + yytext() +
41                       "' at line " + yyline + ", column " + yycolumn); }
```

Figure 15.7: *JFlex*-Spezifikation des Scanners

Dies wird in Zeile 9 überprüft.

3. Falls bis hierhin keine Probleme aufgetreten sind, erzeugen wir nun die Typ-Gleichungen nach der Formel

$$\big(f : \sigma_1 \times \cdots \times \sigma_n \to \varrho\big) \implies \mathit{typeEqs}\big(f(t_1, \cdots, t_n), \tau\big) := \{\varrho = \tau\} \cup \bigcup_{i=1}^{n} \mathit{typeEqs}(t_i, \sigma_i).$$

Dazu erstellen wir in Zeile 17 zuächst die Typ-Gleichungen

$$\varrho \doteq \tau$$

```
1  public Substitution solve() {
2      while (mEquations.size() != 0) {
3          Equation eq = mEquations.remove(0);
4          Type lhs = eq.getLhs();
5          Type rhs = eq.getRhs();
6          if (lhs instanceof Parameter) {
7              Parameter var = (Parameter) lhs;
8              List<Equation> newEquations = new LinkedList<Equation>();
9              for (Equation equation: mEquations) {
10                 Equation neq = equation.substitute(var, rhs);
11                 newEquations.add(neq);
12             }
13             mEquations = newEquations;
14             mTheta     = mTheta.substitute(var, rhs);
15         } else if (lhs instanceof CompositeType) {
16             CompositeType compLhs = (CompositeType) lhs;
17             CompositeType compRhs = (CompositeType) rhs;
18             if (!compLhs.getName().equals(compRhs.getName())) {
19                 // different type constructors, no solution
20                 System.err.println("Error: different type constructors\n");
21                 return null;
22             }
23             List<Type> lhsArgs = compLhs.getArgTypes();
24             List<Type> rhsArgs = compRhs.getArgTypes();
25             for (int i = 0; i < lhsArgs.size(); ++i) {
26                 Type sigmaLhs = lhsArgs.get(i);
27                 Type sigmaRhs = rhsArgs.get(i);
28                 mEquations.add(0, new Equation(sigmaLhs, sigmaRhs));
29             }
30         }
31     }
32     return mTheta;
33 }
```

Figure 15.8: Die Methode *solve*() aus der Klasse `MartelliMontanari`.

und berechnen dann in der Schleife in den Zeilen 19 – 22 rekursiv die Typ-Gleichungen, die sich aus den Forderungen

$$s_i : \sigma_i \quad \text{für } i = 1, \cdots, n$$

ergeben. Als Ergebnis erhalten wir eine Menge von Typ-Gleichungen, die genau dann lösbar sind, wenn der Term $f(s_1, \cdots, s_n)$ den Typ τ hat.

Als letztes diskutieren wir die Klasse `Program`. Abbildung 15.10 zeigt den Konstruktor dieser Klasse. Dieser Konstruktor bekommt als Argumente

- eine Liste von Typ-Definitionen `typeDefs`,
- eine Liste von Signaturen `signatures` und
- eine Liste von getypten Termen `typedTerms`, deren Typ-Korrektheit überprüft werden soll.

Die wesentliche Aufgabe des Konstruktors besteht darin, die Member-Variable `mSignatureMap` zu initialisieren. Hierbei handelt es sich um eine Symboltabelle, in der zu jedem Funktions-Namen die zugehörige Signatur abgelegt

```
1  public List<Equation> typeEqs(Type tau, Map<String, Signature> map) {
2      Signature sign = map.get(mFunction);
3      if (sign == null) {
4          System.err.println("The function " + mFunction +
5                             " has not been declared!");
6          throw new Error("Undeclared function in " + myString());
7      }
8      List<Type> argTypes = sign.getArgList();
9      if (argTypes.size() != mTermList.size()) {
10         System.err.println("Wrong number of parameters for function " +
11                            mFunction);
12         System.err.println("expected: " + argTypes.size());
13         System.err.println("found:    " + mTermList.size());
14         throw new Error("Wrong number of parameters in " + myString());
15     }
16     List<Equation> result = new LinkedList<Equation>();
17     Equation eq = new Equation(sign.getResult(), tau);
18     result.add(eq);
19     for (int i = 0; i < mTermList.size(); ++i) {
20         Term argI   = mTermList.get(i);
21         Type sigmaI = argTypes.get(i);
22         result.addAll( argI.typeEqs(sigmaI, map) );
23     }
24     return result;
25 }
```

Figure 15.9: Berechnung der Typ-Gleichungen

ist. Dazu werden zunächst in der Schleife in Zeile 9 – 11 alle Signaturen aus der als Argument übergebenen Liste `signatures` in dieser Liste eingetragen. Dies alleine ist allerdings nicht ausreichend, denn durch die Typ-Definitionen werden implizit noch weitere Funktions-Zeichen deklariert. Beispielsweise werden durch die Typ-Definition

```
type  list(X) := nil + cons(X, list(X));
```

implizit die Funktionszeichen *nil* und *cons* definiert. Diese Funktionszeichen haben die folgenden Signaturen:

1. `nil:  List(T)`,
2. `cons:  T * List(T) -> List(T)`.

Die Aufgabe des Konstruktors der Klasse `Program` besteht also darin, aus den in dem Argument `typeDefs` enthaltenen Typ-Definitionen die Signaturen der implizit deklarierten Funktionszeichen zu generieren. Dazu iteriert die Schleife in Zeile 12 zunächst über alle Typ-Definitionen. Es gibt zwei Arten von Typ-Definitionen: Typ-Definitionen, bei denen der erzeugte Typ noch von einem oder mehreren Parametern abhängt, wie dies z.B. bei der Typ-Definition von `List(T)` der Fall ist, oder solche Typ-Definitionen, bei denen der erzeugte Typ keine Parameter hat. Letztere Typ-Definitionen werden durch die Klasse `SimpleTypeDef` dargestellt, erstere durch die Klasse `CompositeType`. Diese beiden Fälle werden durch die `if`-Abfrage in Zeile 13 unterschieden.

1. Falls die untersuchte Typ-Definition `td` eine einfache Typ-Definition der Form

   $$\tau := f_1(\sigma_1^{(1)}, \cdots, \sigma_{n_1}^{(1)}) + \cdots f_k(\sigma_1^{(k)}, \cdots, \sigma_{n_k}^{(k)})$$

   ist, so wird der Typ τ von den Funktionen f_1, $\cdots$, f_k erzeugt. Die i-te Funktion f_i hat die Signatur

   $$f_i : \sigma_1^{(i)} \times \cdots \times \sigma_{n_i}^{(i)} \to \tau.$$

```
1  public Program(List<TypeDef>   typeDefs,
2                 List<Signature> signatures,
3                 List<TypedTerm> typedTerms)
4  {
5      mTypeDefs   = typeDefs;
6      mSignatures = signatures;
7      mTypedTerms = typedTerms;
8      mSignatureMap = new TreeMap<String, Signature>();
9      for (Signature s: mSignatures) {
10         mSignatureMap.put(s.getName(), s);
11     }
12     for (TypeDef td: mTypeDefs) {
13         if (td instanceof SimpleTypeDef) {
14             SimpleTypeDef std = (SimpleTypeDef) td;
15             Type rho = new CompositeType(std.getName(), new LinkedList<Type>());
16             for (Type tau: std.getTypeSum()) {
17                 CompositeType c = (CompositeType) tau;
18                 String name = c.getName();
19                 Signature s = new Signature(name, c.getArgTypes(), rho);
20                 mSignatureMap.put(name, s);
21             }
22         } else {
23             ParamTypeDef ctd = (ParamTypeDef) td;
24             List<Type> paramList = new LinkedList<Type>();
25             for (Parameter v : ctd.getParameters()) {
26                 paramList.add(v);
27             }
28             Type rho = new CompositeType(ctd.getName(), paramList);
29             for (Type tau: ctd.getTypeSum()) {
30                 CompositeType c = (CompositeType) tau;
31                 String name = c.getName();
32                 Signature s = new Signature(name, c.getArgTypes(), rho);
33                 mSignatureMap.put(name, s);
34             }
35         }
36     }
37 }
```

Figure 15.10: Der Konstruktor der Klasse `Program`.

Diese Signatur wird in Zeile 19 aus dem Ergebnis-Typ τ und den Argument-Typen `c.getArgTypes()` der Funktion f_i zusammengebaut. Die so konstruierte Signatur wird dann in der Symboltabelle `mSignatureMap` abgelegt.

2. Falls es sich bei der untersuchten Typ-Definition um die Definition eines parametrisierten Typs handelt, muss darauf geachtet werden, dass der Ergebnis-Typ der Signatur der Funktionen f_i auch von diesen Parametern abhängt. Zu diesem Zweck wird in Zeile 24 zunächst eine neue Parameter-Liste `paramList` angelegt, in die dann in der Schleife in Zeile 25 — 27 die Parameter des Typs hineinkopiert werden. Der Rest dieses Falls ist analog zum ersten Fall.

Neben dem Konstruktor enthält die Klasse `Program` noch die Methode `typeCheck()`, welche die Typüberprüfung ausführt. Die Implementierung dieser Methode ist in der Abbildung 15.11 gezeigt. Diese Methode iteriert in der Schleife, die in Zeile 2 beginnt, über alle getypten Terme $t : \tau$, die dem Konstruktor der Klasse `Program` übergeben

wurden. Dazu wird in Zeile 6 zu jedem Term t und Typ τ die Menge der Typ-Gleichungen *typeEqs*(t, τ) berechnet. Dazu wird der Methode `typeEqs()`, die die Funktion *typeEqs*() implementiert, noch die vom Konstruktor berechnete Symboltabelle übergeben. In dieser Symboltabelle sind die Signaturen der verschiedenen Funktions-Zeichen abgespeichert. Die berechneten Typ-Gleichungen werden anschließend mit Hilfe der Methode `solve()` der Klasse `MartelliMontanari` gelöst. Falls die Typ-Gleichungen gelöst werden konnten, hat der Term t tatsächlich den Typ τ. Andernfalls wird eine Fehlermeldung ausgegeben.

```
1  public void typeCheck() {
2      for (TypedTerm tt: mTypedTerms) {
3          System.out.println("\nChecking " + tt.myString());
4          Term t   = tt.getTerm();
5          Type tau = tt.getType();
6          List<Equation> typeEquations = t.typeEqs(tau, mSignatureMap);
7          MartelliMontanari mm = new MartelliMontanari(typeEquations);
8          Substitution theta = mm.solve();
9          if (theta != null) {
10             System.out.println(tt.myString() + " has been verified!");
11             System.out.println(theta);
12         } else {
13             System.out.println(tt.myString() + " type ERROR!!!");
14         }
15     }
16 }
```

Figure 15.11: Die Methode *typeCheck()* in der Klasse `Program`.

15.5 Inklusions-Polymorphismus

Interpretieren wir Typen als Mengen von Werten, so stellen wir fest, dass es bei Typen eine Inklusions-Hierarchy gibt. In der Sprache *Java* hat jeder Wert vom Typ `String` auch gleichzeitig den Typ `Object`, es gilt also

$$\texttt{String} \subseteq \texttt{Object}.$$

Allgemein gilt: Ist e ein Ausdruck, der den Typ A hat und ist A weiter eine Klasse, die von der Klasse B abgeleitet ist, so kann der Ausdruck e überall dort verwendet werden, wo ein Ausdruck vom Typ B benötigt wird. Damit kann der Ausdruck e sowohl als ein A, als auch als ein B verwendet werden: e kann also *polymorph* verwendet werden. Im Gegensatz zu dem *parametrischen Polymorphismus*, den wir bisher betrachtet haben, sprechen wir hier von *Inklusions-Polymorphismus*. Die Behandlung von Inklusions-Polymorphismus ist besonders dann interessant, wenn dieser zusammen mit parametrischen Polymorphismus auftritt. Wir zeigen exemplarisch ein Beispiel in Abbildung 15.12.

1. In Zeile 4 wird hier zunächst ein Feld `x` von Strings angelegt.
2. In Zeile 5 definieren wir ein Feld `y` von Objekten und initialisieren dieses Feld mit dem bereits erzeugten Feld `x`. Da jeder String zugleich auch ein Objekt ist, ist dies möglich.
3. In Zeile 6 weisen wir dem zweiten Element des Feldes `y` die Zahl 2 zu. Da `y` ein Feld von Objekten ist und eine Zahl ebenfalls als Objekt angesehen werden kann, sollte auch dies kein Problem sein.

In der Tat lässt sich das Programm fehlerfrei übersetzen. Bei der Ausführung wird aber eine Ausnahme ausgelöst. Der Grund ist, dass mit der Zuweisung in Zeile 5 die Variable `y` eine Referenz auf das Feld von Strings ist, das in Zeile 4 angelegt worden ist. Wenn wir nun versuchen, in dieses Feld eine Zahl einzufügen, dann gibt es eine `ArrayStoreException`. Dieses Beispiel zeigt, dass die Sprache *Java* nicht statisch auf Typ-Sicherheit geprüft

werden kann. Es zeigt auch, dass die Behandlung der Kombiation von Inklusions-Polymorphismus und parametrischen Polymorphismus deutlich komplexer ist, als die Behandlung von parametrischem Polymorphismus alleine.

```
1  public class TypeSurprise {
2
3      public static void main(String[] args) {
4          String[] x = { "a", "b", "c" };
5          Object[] y = x;
6          y[1] = new Integer(2);
7          System.out.println(x[1]);
8      }
9  }
```

Figure 15.12: Ein Typ-Problem in *Java*.

In dem Papier "*Type Inference for Java 5*" von Mazurak und Zdancewic wird das Typ-System der Sprache *Java* näher untersucht. Die Autoren kommen zu dem Schluss, dass die Frage, ob ein gegebenes *Java*-Programm korrekt getypt ist, unentscheidbar ist. Es ist daher auch nicht verwunderlich, dass sich der *Java*-Compiler bei manchen Programmen mit einem Stack-Overflow verabschiedet, der seine Ursache im Typ-Checker des Compilers hat. Des Weiteren ist der folgende Satz aus der "*Java Language Specifiaction*", also der verbindlichen Definition der Sprache *Java*, aufschlussreich:

> *The type inference algorithm should be viewed as a heuristic, designed to perform well in practice. If it fails to infer the desired result, explicit type parameters may be used instead.*

Dies zeigt, dass die Kombination von parametrischem Polymorphismus und Inklusions-Polymorphismus zu sehr komplexen Problemen führen kann, die wir aber aus Zeitgründen nicht tiefer betrachten können.

Chapter 16

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.

16.1 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 16.1 on page 240 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 16.1: 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.

   1. 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`".

   2. Line 11 uses the directive ".limit" 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 16.1 on page 240 corresponds to the *Java* code shown in Figure 16.2 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 16.2: Printing `Hello world` in *Java*.

   3. 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 ".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
      ```

      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.

[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.

4. Line 15 calls the method `println`, which is a method of the class "`java.io.PrintStream`". It also specifies that `println` is used to print a string.
5. Line 16 returns from the method `main`.
6. 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 with their sum onto the stack.

16.2 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 16.3 on page 243 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.

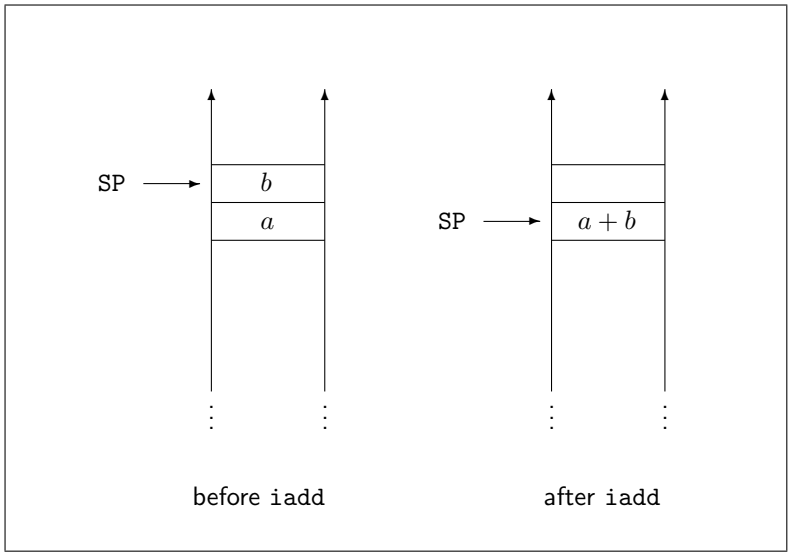


Figure 16.3: The effect of iadd.

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 16.4 on page 244 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.

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.

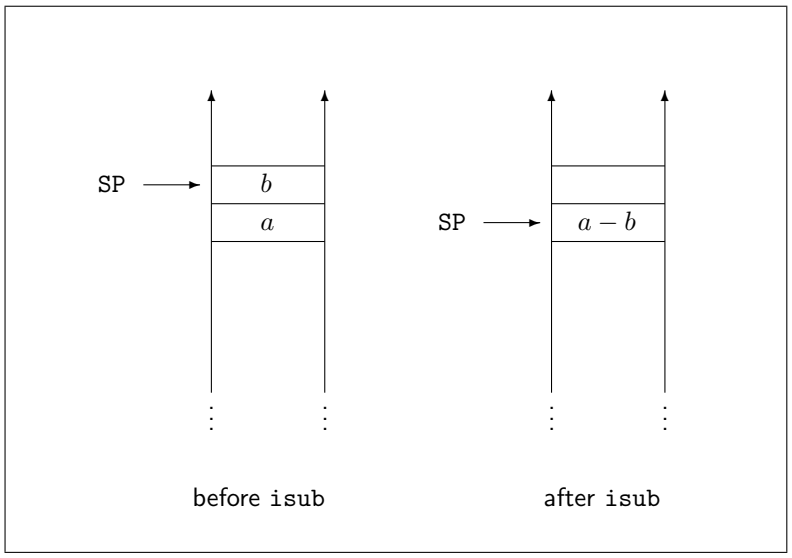


Figure 16.4: The effect of `isub`.

6. The instruction "`iand`" computes the bitwise conjunction of the values on top of the stack.
7. The instruction "`ior`" computes the bitwise disjunction 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 16.5 on page 244 pictures this command.

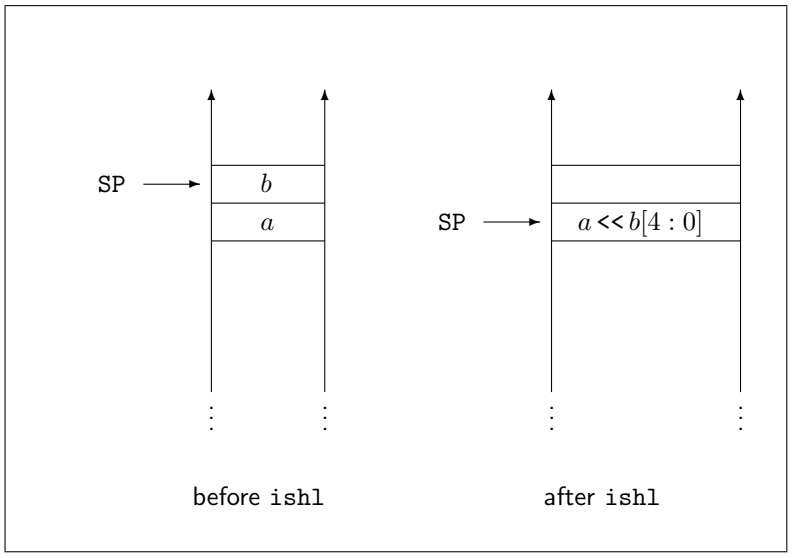


Figure 16.5: The effect of `ishl`.

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.

16.2.1 Instructions to Manipulate the Stack

1. The instruction "dup" duplicates the value that is on top of the stack. Figure 16.6 on page 245 pictures this command.

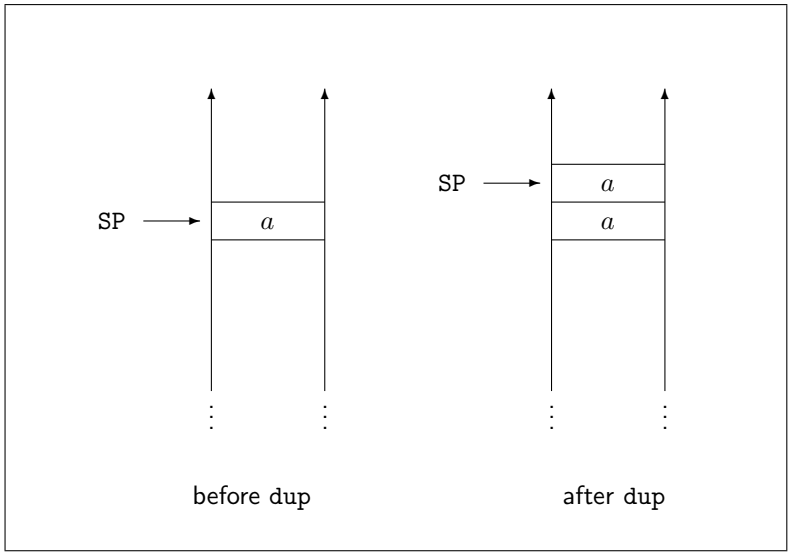


Figure 16.6: The effect of dup.

2. The instruction "pop" removes the value that is on top of the stack. Figure 16.7 on page 245 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.

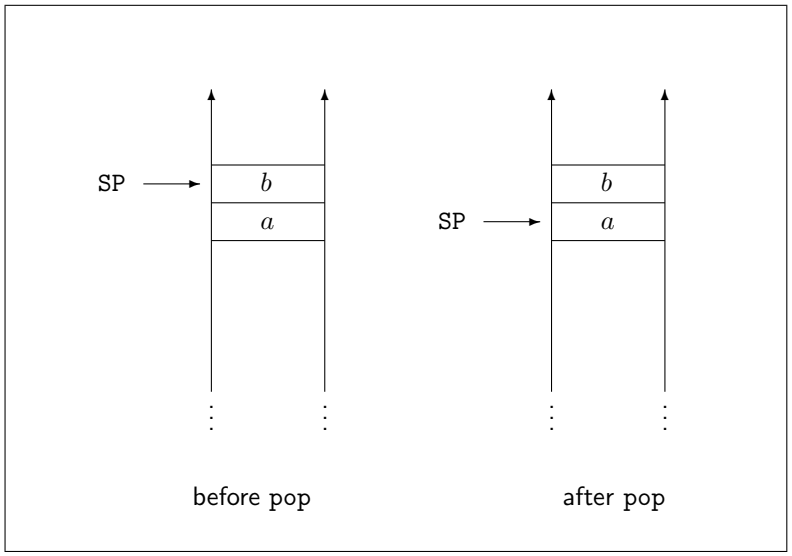


Figure 16.7: The effect of pop.

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 16.8 on page 246 pictures this command.

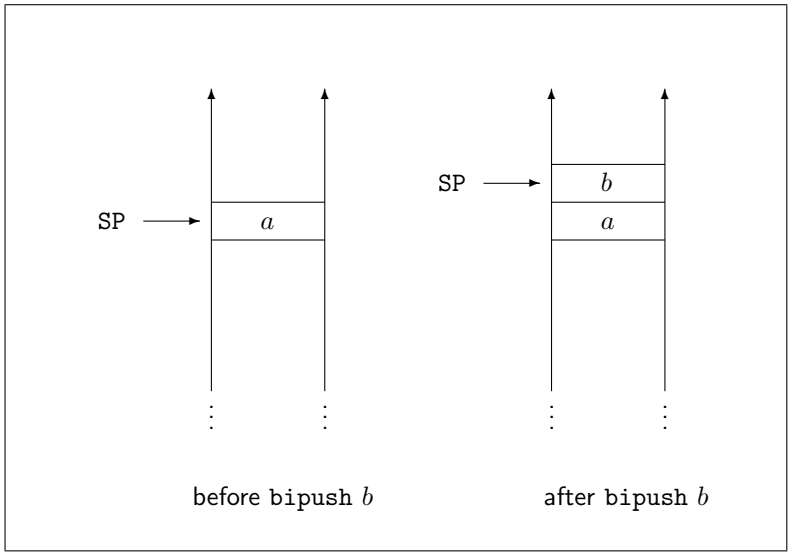


Figure 16.8: The effect of `bipush` b.

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 16.9 on page 247 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 16.10 on page 247 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.

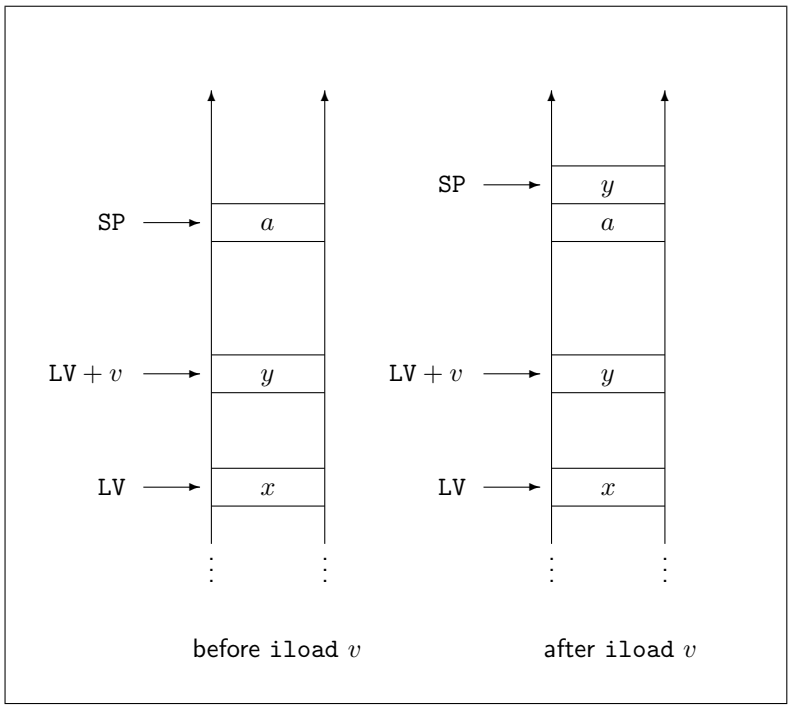


Figure 16.9: The effect of `iload` v.

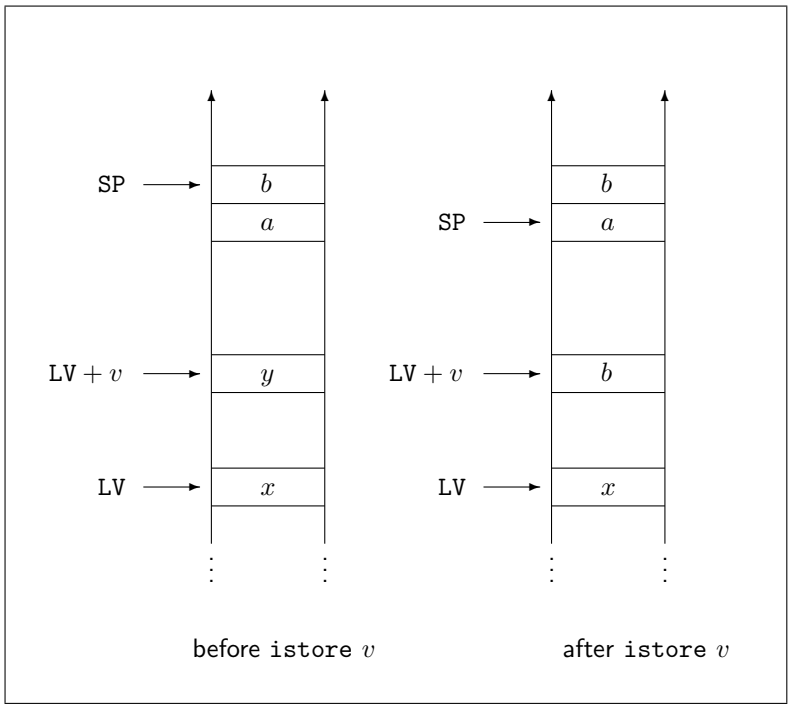


Figure 16.10: The effect of `istore` v.

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.

16.3 An Example Program

Figure 16.11 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 16.12 on page 250 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 16.11: 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 16.12: 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 42: Implement an assembler program that computes the factorial function. Test your program by printing $n!$ for $n = 1, \cdots, 10$.

16.4 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 16.13 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 16.13: 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 16.14. 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 16.14: The output of "`javap -c Sum.class`".

Chapter 17

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. 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 *JFlex* entwickeln.

2. Der Parser liest die Folge von Token und produziert als Ergebnis einen abstrakten Syntax-Baum.

   Wir werden den Parser mit Hilfe von CUP generieren.

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 betrachten.

5. Der Code-Generator übersetzt schließlich den Parse-Baum in eine Folge von JAVA-Assembler-Befehlen.

Bei unseren Compiler sind wir an dieser Stelle schon fertig. 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 einen gegebene Prozessor-Architektur zur erzeugen. Die schwierigste Aufgabe besteht hier darin, für die verwendeten Variablen eine Register-Zuordnung zu finden, bei der möglichst alle Variablen in Registern vorgehalten werden können. Aus Zeitgründen können wir das Thema der Register-Zuordnung in dieser Vorlesung nicht behandeln.

17.1 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 17.1 zeigt die Grammatik der Quellsprache in erweiterter Backus-Naur-Form (EBNF). Die Grammatik für *Integer*-`C` verwendet die folgenden beiden 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.

2. NUMBER steht für eine Folge von Ziffern, die als Dezimalzahl interpretiert wird.

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. Jeder Befehl wird durch ein Semikolon abgeschlossen. Der Aufbau der einzelnen Befehle ist dann ähnlich wie bei der Sprache SL, für die wir im Kapitel 10 einen Interpreter entwickelt haben. Die in Abbildung 17.1 gezeigte Grammatik ist mehrdeutig:

$$
\begin{array}{rcl}
\textit{program} & \rightarrow & \textit{function}^* \\
\textit{function} & \rightarrow & \text{“}\texttt{int}\text{”}\ \textsc{Id}\ \text{“(”}\ \textit{paramList}\ \text{“)”}\ \ \text{“\{”}\ \textit{decl}^*\ \textit{stmnt}^*\ \text{“\}”} \\
\textit{paramList} & \rightarrow & (\text{“}\texttt{int}\text{”}\ \textsc{Id}\ (\text{“,”}\ \ \text{“}\texttt{int}\text{”}\ \textsc{Id})^*)? \\
\textit{decl} & \rightarrow & \text{“}\texttt{int}\text{”}\ \textsc{Id}\ \text{“;”} \\
\textit{stmnt} & \rightarrow & \text{“\{”}\ \textit{stmnt}^*\ \text{“\}”} \\
 & | & \textsc{Id}\ \text{“=”}\ \textit{expr}\ \text{“;”} \\
 & | & \text{“}\texttt{if}\text{”}\ \ \text{“(”}\ \textit{boolExpr}\ \text{“)”}\ \textit{stmnt} \\
 & | & \text{“}\texttt{if}\text{”}\ \ \text{“(”}\ \textit{boolExpr}\ \text{“)”}\ \textit{stmnt}\ \text{“}\texttt{else}\text{”}\ \textit{stmnt} \\
 & | & \text{“}\texttt{while}\text{”}\ \ \text{“(”}\ \textit{boolExpr}\ \text{“)”}\ \textit{stmnt} \\
 & | & \text{“}\texttt{return}\text{”}\ \textit{expr}\ \text{“;”} \\
 & | & \textit{expr}\ \text{“;”} \\
\textit{boolExpr} & \rightarrow & \textit{expr}\ (\text{“==”} \mid \text{“!=”} \mid \text{“<=”} \mid \text{“>=”} \mid \text{“<”} \mid \text{“>”})\ \textit{expr} \\
 & | & \text{“!”}\ \textit{boolExpr} \\
 & | & \textit{boolExpr}\ (\text{“\&\&”} \mid \text{“||”})\ \textit{boolExpr} \\
\textit{expr} & \rightarrow & \textit{expr}\ (\text{“+”} \mid \text{“-”} \mid \text{“*”} \mid \text{“/”})\ \textit{expr} \\
 & | & \text{“(”}\ \textit{expr}\ \text{“)”} \\
 & | & \textsc{Number} \\
 & | & \textsc{Id}\ (\text{“(”}(\textit{expr}\ (\text{“,”}\ \textit{expr})^*)?\ \text{“)”})?
\end{array}
$$

Figure 17.1: Eine EBNF-Grammatik für *Integer*-C

1. Die Grammatik hat das *Dangling-Else-Problem*.

   Da wir im Kapitel 14 bereits gesehen haben, wie dieses Problem professionell gelöst werden kann, nehme ich mir hier die Freiheit, CUP mit der Option

   ```
   "-expect 1
   ```

   aufzurufen und dadurch die Fehlermeldung zu unterdrücken, denn wir hatten ja bereits gesehen, dass CUP per default den durch die Mehrdeutigkeit entstehenden Shift-Reduce-Konflikt in unserem Sinne auflöst.

2. Für die bei arithmetischen und Boole'schen Ausdrücken verwendeten Operatoren müssen Präzedenzen festgelegt werden.

Abbildung 17.2 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.

```
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 17.2: Ein einfaches INTEGER-C-Programm.

17.2 Developing the Scanner and the Parser

We construct the scanner using *JFlex* while we use CUP to implement the parser. Figure 17.3 shows the scanner. The scanner recognizes the operators, keywords, identifiers, and numbers. When we compare this scanner to the scanners we have discussed previously, we realize that there is nothing that we haven't seen before. Therefore, we do not discuss this scanner in more detail.

Before we can discuss the parser, we have to specify the classes that are used to represent the abstract syntax tree that is generated by the parser. I have chosen to describe these classes using a formal notation. The specification is shown in Figure 17.4 on page 257. We discuss this specification now line by line.

1. In line 1, the equation

   ```
   Program = Program(List<Function> functionList)
   ```

   specifies that the class `Program` has one member variable called `mFunctionList`. (Note that it is called "`mFunctionList`" rather than "`functionList`". The reason is that in order to distinguish the member variables of a class from the local variables of a method I have decided to start all member variables with the letter "`m`".) The variable `mFunctionList` has the type `List<Function>`. Therefore, an object of class `Program` is essentially a list of objects of class `Function`.

2. Similarly, in line 3 to 6 the equation

   ```
   Function = Function(String            name,
                       List<String>      parameterList,
                       List<Declaration> mDeclarations,
                       List<Statement>   body);
   ```

   specifies that the class `Function` has four attributes:

   1. `mName` is a `String` storing the name of the function,
   2. `mParameterList` is the list of formal parameters of the function,
   3. `mDeclarations` is the list of local variable declarations occurring in the body of the function, while
   4. `mBody` is the list of statements that make up the definition of the function.

```
1  import java_cup.runtime.*;
2  %%
3  %char
4  %line
5  %column
6  %cupsym IntegerCParserSym
7  %cup
8  %unicode
9  %{  private Symbol symbol(int type) {
10         return new Symbol(type, yychar, yychar + yylength());
11     }
12     private Symbol symbol(int type, Object value) {
13         return new Symbol(type, yychar, yychar + yylength(), value);
14     }
15 %}
16 %eofval{ return new Symbol(IntegerCParserSym.EOF);
17 %eofval}
18 %%
19 "+"                   { return symbol( IntegerCParserSym.PLUS      ); }
20 "-"                   { return symbol( IntegerCParserSym.MINUS     ); }
21 "*"                   { return symbol( IntegerCParserSym.TIMES     ); }
22 "/"                   { return symbol( IntegerCParserSym.SLASH     ); }
23 "("                   { return symbol( IntegerCParserSym.LPAREN    ); }
24 ")"                   { return symbol( IntegerCParserSym.RPAREN    ); }
25 "{"                   { return symbol( IntegerCParserSym.LBRACE    ); }
26 "}"                   { return symbol( IntegerCParserSym.RBRACE    ); }
27 ","                   { return symbol( IntegerCParserSym.COMMA     ); }
28 ";"                   { return symbol( IntegerCParserSym.SEMICOLON ); }
29 "="                   { return symbol( IntegerCParserSym.ASSIGN    ); }
30 "=="                  { return symbol( IntegerCParserSym.EQ        ); }
31 "!="                  { return symbol( IntegerCParserSym.NE        ); }
32 "<"                   { return symbol( IntegerCParserSym.LT        ); }
33 ">"                   { return symbol( IntegerCParserSym.GT        ); }
34 "<="                  { return symbol( IntegerCParserSym.LE        ); }
35 ">="                  { return symbol( IntegerCParserSym.GE        ); }
36 "&&"                  { return symbol( IntegerCParserSym.AND       ); }
37 "||"                  { return symbol( IntegerCParserSym.OR        ); }
38 "!"                   { return symbol( IntegerCParserSym.NOT       ); }
39 "int"                 { return symbol( IntegerCParserSym.INT       ); }
40 "return"              { return symbol( IntegerCParserSym.RETURN    ); }
41 "if"                  { return symbol( IntegerCParserSym.IF        ); }
42 "else"                { return symbol( IntegerCParserSym.ELSE      ); }
43 "while"               { return symbol( IntegerCParserSym.WHILE     ); }
44 [a-zA-Z][a-zA-Z_0-9]* { return symbol(IntegerCParserSym.ID,     yytext());              }
45 0|[1-9][0-9]*         { return symbol(IntegerCParserSym.NUMBER, new Integer(yytext())); }
46 [ \t\v\n\r]           { /* skip white space          */ }
47 "//" [^\n]*           { /* skip single line comments */ }
48 "/*" ~"*/"            { /* skip multi  line comments */ }
```

Figure 17.3: Der Scanner für *Integer-C*

```
1  Program = Program(List<Function> functionList);
2
3  Function = Function(String            name,
4                      List<String>      parameterList,
5                      List<Declaration> mDeclarations,
6                      List<Statement>   body);
7
8  Statement = Block(List<Statement> statementList)
9            + Assign(String var, Expr expr)
10           + IfThen(BoolExpr boolExpr, Statement statement)
11           + IfThenElse(BoolExpr condition, Statement then, Statement else)
12           + While(BoolExpr condition, Statement statement)
13           + Return(Expr expr)
14           + ExprStatement(Expr expr);
15
16 Declaration = Declaration(String var);
17
18 Expr = Sum(Expr lhs, Expr rhs)                        // +
19      + Difference(Expr lhs, Expr rhs)                // -
20      + Product(Expr lhs, Expr rhs)                   // *
21      + Quotient(Expr lhs, Expr rhs)                  // /
22      + MyNumber(Integer number)
23      + Variable(String name)
24      + FunctionCall(String name, List<Expr> args);
25
26 BoolExpr = Equation(Expr lhs, Expr rhs)               // ==
27          + Inequation(Expr lhs, Expr rhs)            // !=
28          + LessOrEqual(Expr lhs, Expr rhs)           // <=
29          + GreaterOrEqual(Expr lhs, Expr rhs)        // >=
30          + LessThan(Expr lhs, Expr rhs)              // <
31          + GreaterThan(Expr lhs, Expr rhs)           // >
32          + Negation(BoolExpr expr)                   // !
33          + Conjunction(BoolExpr lhs, BoolExpr rhs)   // &&
34          + Disjunction(BoolExpr lhs, BoolExpr rhs);  // ||
```

Figure 17.4: Spezifikation der Klassen zur Darstellung des Syntax-Baums.

3. Next, the equation

```
Statement = Block(List<Statement> statementList)
          + Assign(String var, Expr expr)
          + IfThen(BoolExpr boolExpr, Statement statement)
          + IfThenElse(BoolExpr condition, Statement then, Statement else)
          + While(BoolExpr condition, Statement statement)
          + Return(Expr expr)
          + ExprStatement(Expr expr);
```

tells us that there is an abstract class `Statement` and that the classes `Block`, `Assign`, $\cdots$, `ExprStatement` are derived from this class.

1. `Block` is a class representing a list of statements enclosed in the curly braces "{" and "}". This list of statemments is stored in the member variable `mStatementList`.

2. `Assign` is a class representing an assignment statement. Therefore, this class has two attributes:
   - `mVar` is the name of the variable on the left hand side of the assignment and
   - `mExpr` is the expression on the right hand side of the assignment.
3. `IfThen` is a class representing an `if-then` statement without a trailing `else` clause. This class has two member variables:
   - `mBoolExpr` is the Boolean expression controlling whether the
   - `mStatement` is to be executed.
4. `IfThenElse` is a class representing an `if-then-else` statement. This class has three member variables:
   - `mBoolExpr` is the Boolean expression controlling the execution.
   - `mThen` is the statement that is executed if `mBoolExpr` evaluates as `true`.
   - `mElse` is the statement that is executed if `mBoolExpr` evaluates as `false`.
5. `While` is a class representing a `while` loop. This class has two member variables:
   - `mBoolExpr` is the Boolean expression controlling the loop.
   - `mStatement` is a statement that is executed as long as `mBoolExpr` evaluates to `true`.
6. `Return` is a class representing a `return` statement. This class has the member variable `mExpr`. Evaluation of this expression yields the value to be returned. Note that in the grammar given in Figure 17.1 on page 254 the expression following the return statement is not optional.
7. `ExprStatement` is a class representing an expression that is to be evaluated as a statement.

4. The equation in line 16 specifies that the class `Declaration` has one member variable with name `mVar`. This variable stores the name of the variable that is declared in the variable declaration associated with the corresponding object of class `Declaration`.

5. Similarly, the equations defining `Expr` and `BoolExpr` specify the representation of arithmetical and Boolean expressions. We refrain from a detailed discussion here, since by now you should have gotten the idea.

Now we are ready to present the CUP specification of our parser. For reasons of space, we have split the CUP grammar into three parts.

1. Figure 17.5 shows the specification of the syntactical variables, terminals, and the specification of the precedences of the terminals. Note that the arithmetical operators have a higher precedence than the logical operators.

2. Figure 17.6 and 17.7 show the grammar including the actions needed to construct the syntax tree. Note that the grammar allows the list of functions to be empty. Of course, an empty program wouldn't be of any use, but this approach simplifies the construction of the associated lists.

   Observe that the actions given in the grammar only construct the abstract syntax tree. Everything else, in particular the generation of code, is then delegated to appropriate methods in the classes representing the syntax tree. We will discuss the code generation later in a separate section.

```
1  import java_cup.runtime.*;
2  import java.util.*;
3
4  class  IntegerCParser;
5
6  terminal           COMMA, PLUS, MINUS, TIMES, SLASH, LPAREN, RPAREN, LBRACE, RBRACE;
7  terminal           ASSIGN, EQ, LT, GT, LE, GE, NE, AND, OR, NOT;
8  terminal           IF, ELSE, WHILE, RETURN, SEMICOLON;
9  terminal           INT;
10 terminal String    ID;
11 terminal Integer   NUMBER;
12
13 nonterminal Program           program;
14 nonterminal List<Function>    functionList;
15 nonterminal Function          function;
16 nonterminal List<String>      paramList, neParamList;
17 nonterminal Declaration       declaration;
18 nonterminal List<Declaration> declarations;
19 nonterminal Statement         stmnt;
20 nonterminal List<Statement>   stmntList;
21 nonterminal Expr              expr;
22 nonterminal List<Expr>        exprList, neExprList;
23 nonterminal BoolExpr          boolExpr;
24
25 precedence left     OR;
26 precedence left     AND;
27 precedence right    NOT;
28 precedence left     PLUS, MINUS;
29 precedence left     TIMES, SLASH;
```

Figure 17.5: Declaration of terminals, syntactical variables, and operator precedences.

```
30  start with program;
31
32  program ::= functionList:l {: RESULT = new Program(l); :} ;
33
34  functionList ::= /* epsilon */ {: RESULT = new LinkedList<Function>(); :}
35                 |  functionList:l function:f {: l.add(f); RESULT = l; :}
36                 ;
37
38  function ::= INT ID:f LPAREN paramList:p RPAREN
39               LBRACE declarations:d stmntList:l RBRACE
40               {: RESULT = new Function(f, p, d, l); :}
41             ;
42
43  paramList ::= /* epsilon */ {: RESULT = new LinkedList<String>(); :}
44              |  neParamList:l {: RESULT = l;                        :}
45              ;
46
47  neParamList ::= INT ID:v {: RESULT = new LinkedList<String>(); RESULT.add(v); :}
48                |  neParamList:l COMMA INT ID:v {: RESULT = l; RESULT.add(v);    :}
49                ;
50
51  declaration ::= INT ID:v SEMICOLON {: RESULT = new Declaration(v); :} ;
52
53  declarations ::= /* epsilon */ {: RESULT = new LinkedList<Declaration>(); :}
54                 |  declarations:l declaration:d {: RESULT = l; RESULT.add(d); :}
55                 ;
56
57  stmnt ::= LBRACE stmntList:l RBRACE           {: RESULT = new Block(l);     :}
58          |  ID:v ASSIGN expr:e SEMICOLON       {: RESULT = new Assign(v, e); :}
59          |  IF LPAREN boolExpr:b RPAREN stmnt:s {: RESULT = new IfThen(b, s); :}
60          |  IF LPAREN boolExpr:b RPAREN stmnt:t ELSE stmnt:e
61                {: RESULT = new IfThenElse(b, t, e); :}
62          |  WHILE LPAREN boolExpr:b RPAREN stmnt:s
63                {: RESULT = new While(b, s); :}
64          |  RETURN expr:e SEMICOLON   {: RESULT = new Return(e);        :}
65          |  expr:e SEMICOLON          {: RESULT = new ExprStatement(e); :}
66          ;
67
68  stmntList ::= /* epsilon */       {: RESULT = new LinkedList<Statement>(); :}
69              |  stmntList:l stmnt:s {: l.add(s); RESULT = l;                 :}
70              ;
```

Figure 17.6: The first part of the CUP grammar.

```
71  boolExpr ::= expr:l EQ expr:r          {: RESULT = new Equation(      l, r); :}
72             |  expr:l NE expr:r          {: RESULT = new Inequation(    l, r); :}
73             |  expr:l LE expr:r          {: RESULT = new LessOrEqual(   l, r); :}
74             |  expr:l GE expr:r          {: RESULT = new GreaterOrEqual(l, r); :}
75             |  expr:l LT expr:r          {: RESULT = new LessThan(      l, r); :}
76             |  expr:l GT expr:r          {: RESULT = new GreaterThan(   l, r); :}
77             |  NOT boolExpr:e            {: RESULT = new Negation(      e   ); :}
78             |  boolExpr:l AND boolExpr:r {: RESULT = new Conjunction(   l, r); :}
79             |  boolExpr:l OR  boolExpr:r {: RESULT = new Disjunction(   l, r); :}
80             ;
81
82  expr ::= expr:l PLUS  expr:r            {: RESULT = new Sum(       l, r);   :}
83         |  expr:l MINUS expr:r            {: RESULT = new Difference(l, r);   :}
84         |  expr:l TIMES expr:r            {: RESULT = new Product(   l, r);   :}
85         |  expr:l SLASH expr:r            {: RESULT = new Quotient(  l, r);   :}
86         |  LPAREN expr:e RPAREN           {: RESULT = e;                      :}
87         |  NUMBER:n                       {: RESULT = new MyNumber(n);        :}
88         |  ID:v                           {: RESULT = new Variable(v);        :}
89         |  ID:n LPAREN exprList:l RPAREN  {: RESULT = new FunctionCall(n, l); :}
90         ;
91
92  exprList ::= /* epsilon */ {: RESULT = new LinkedList<Expr>(); :}
93             |  neExprList:l  {: RESULT = l;                     :}
94             ;
95
96  neExprList ::= expr:e {: RESULT = new LinkedList<Expr>(); RESULT.add(e); :}
97               |  neExprList:l COMMA expr:e {: RESULT = l; RESULT.add(e);   :}
98               ;
```

Figure 17.7: The last part of the CUP grammar.

17.3 Darstellung der Assembler-Befehle

Die Abbildungen 17.8 und 17.9 zeigen eine mögliche Übersetzung des Programms zur Berechnung der Summe $\sum_{i=1}^{n} i$ aus Abbildungen 17.2 in Java-Assembler. Um solche Assembler-Programme innerhalb des Programms darstellen zu können, implementieren wir für jeden Java-Assembler-Befehl eine eigene Klasse, die diesen Befehl darstellen kann. Auch die Deklarationen, wie beispielsweise "`.limit`" oder "`.end method`" werden jeweils durch eine Klasse dargestellt. Abbildung 17.10 zeigt die Spezifikation dieser Klassen.

```
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 main([Ljava/lang/String;)V
11     .limit locals 1
12     .limit stack 2
13     ldc 6
14     ldc 6
15     imul
16     istore 0
17     getstatic java/lang/System/out Ljava/io/PrintStream;
18     iload 0
19     invokestatic MySum/sum(I)I
20     invokevirtual java/io/PrintStream/println(I)V
21     bipush 42
22     pop
23     return
24 .end method
```

Figure 17.8: Eine Übersetzung des Programms aus Abbildung 17.2 in Java-Assembler, 1. Teil.

1. Wir haben in den Klassen `GOTO`, `IFEQ` und `IFLT` das Sprungziel als Zahl dargestellt. Später wird bei der Ausgabe eines Sprungziels diesen Zahlen noch der Buchstabe "`l`" vorangestellt, so dass das Sprungziel als String interpretiert werden kann. Um die Generierung von Sprungzielen zu verstehen, betrachten wir die Klasse `LABEL`, die in Abbildung 17.11 gezeigt wird. Diese Klasse verfügt über die statische Variable `sLabelCount`, die in Zeile 2 mit 0 initialisiert wird. Der Konstruktor der Klasse `LABEL` erzeugt bei jedem Aufruf ein neues Objekt, dessen Member-Variable `mLabel` einen nur einmal vergebenen Wert hat. Dies wird dadurch erreicht, dass die statische Variable `sLabelCount` nach jedem Anlegen eines neuen Objektes vom Typ `LABEL` inkrementiert wird. Dadurch wird sichergestellt, dass zwei verschiedene Objekte der Klasse `LABEL` später tatsächlich verschiedene Sprungziele bezeichnen.

   Zeile 9 zeigt die Umwandlung eines `LABEL`-Objektes in einen String, die zur Ausgabe der Assembler-Kommandos benutzt wird. Der eindeutigen Zahl wird der Buchstabe "`l`" vor- und der Doppelpunkt "`:`" nachgestellt, damit die Ausgabe der Syntax des *Jasmin*-Assemblers entspricht.

2. Neben den eigentlichen Assembler-Kommandos zeigt Abbildung 17.10 noch die Definition von Klassen wie beispielsweise der Klasse `MAIN`, deren Implementierung in Abbildung 17.12 gezeigt wird. Diese Klasse dient nur dazu, die Direktive

   "`.method public static main([Ljava/lang/String;)V`"

```
1  .method public static sum(I)I
2      .limit locals 2
3      .limit stack 2
4      ldc 0
5      istore 1
6    l1:
7      iload 0
8      ldc 0
9      if_icmpne l3
10     bipush 0
11     goto l4
12   l3:
13     bipush 1
14   l4:
15     ifeq l2
16     iload 1
17     iload 0
18     iadd
19     istore 1
20     iload 0
21     ldc 1
22     isub
23     istore 0
24     goto l1
25   l2:
26     iload 1
27     ireturn
28 .end method
```

Figure 17.9: Übersetzung des Programms aus Abbildung 17.2 in Java-Assembler, 2. Teil.

auszugeben. Die Klassen `METHOD`, `END_METHOD` und `LIMIT` dienen ebenfalls dazu, Direktiven auszugeben.

```
AssemblerCmd = METHOD(String name, Integer numberArgs)
             + END_METHOD()
             + GETSTATIC(String all)
             + IADD()
             + ISUB()
             + IMUL()
             + IDIV()
             + IAND()
             + IOR()
             + POP()
             + BIPUSH(Integer number)
             + ILOAD(String var)
             + ISTORE(String var)
             + INVOKE(String name)
             + IRETURN()
             + LABEL(Integer label)
             + GOTO(Integer label)
             + IFEQ(Integer label)
             + IFLT(Integer label)
             + LDC(Integer number)
             + LIMIT(String what, Integer bound)
             + MAIN()
             + NEWLINE()
             + POP()
             + PRINTLN()
             + RETURN();
```

Figure 17.10: Darstellung der Assembler-Befehle als Klassen.

```
public class LABEL extends AssemblerCmd {
    private static  Integer sLabelCount = 0;
    private         Integer mLabel;

    public LABEL() {
        mLabel = ++sLabelCount;
    }
    public Integer getLabel() { return mLabel;             }
    public String  toString() { return "l" + mLabel + ":"; }
}
```

Figure 17.11: Die Klasse `LABEL`.

17.4 Die Code-Erzeugung

Nun haben wir alles Material zusammen, um die eigentliche Code-Erzeugung diskutieren zu können. Wir gliedern unsere Darstellung, indem wir die Übersetzung arithmetischer Ausdrücke, Boole'schen Ausdrücke, Befehle und Funktionen getrennt behandeln. Wir beginnen damit, dass wir zeigen, wie arithmetische Ausdrücke übersetzt werden.

```
1  package Assembler;
2
3  public class MAIN extends AssemblerCmd {
4
5      public MAIN() {}
6      public String toString() {
7          return ".method public static main([Ljava/lang/String;)V";
8      }
9  }
10
```

Figure 17.12: Die Klasse `MAIN`.

17.4.1 Übersetzung arithmetischer Ausdrücke

Die Übersetzung eines arithmetischen Ausdrucks *expr* soll Code erezeugen, durch dessen Ausführung das Ergebnis der Auswertung auf den Stack gelegt wird. Zu diesem Zweck deklariert die abstrakte Klasse `Expr`, die in Abbildung 17.13 gezeigt ist, die Methode

```
public abstract List<AssemblerCmd> compile(Map<String, Integer> symbolTable);
```

die als Ergebnis eine Liste von Assembler-Kommandos erzeugt. Werden diese Kommandos ausgeführt, so liegt anschließend der Wert des Ausdrucks auf dem Stack. Bei dem Parameter `symbolTable`, welcher der Methode `compile` als Argument übergeben wird, handelt es sich um eine Funktion, die jeder Variablen eine eindeutige Position im Stack zuordnet, an der diese Variable im Stack gespeichert wird. Diese Positionen werden bei der Deklaration der einzelnen Variablen berechnet. Die Details dieser Berechnung diskutieren wir, wenn wie die Implementierung der Klasse `Function` diskutieren.

Außerdem deklariert die Klasse `Expr` noch die Methode `stackSize`. Aufgabe dieser Methode ist es zu berechnen, wie groß der Stack bei der Auswertung des Ausdrucks maximal werden kann. Diese Information benötigen wir um später mit Hilfe der Direktive "`.limit stack`" die maximale Größe des Stacks spezifizieren zu können. Dies ist erforderlich, da in *Java* jede Methode angeben muss, wie groß der Stack maximal werden kann.

Im Folgenden betrachten wir die Übersetzung der verschiedenen arithmetischen Ausdrücke der Reihe nach.

```
1  public abstract class Expr {
2      public abstract List<AssemblerCmd> compile(Map<String, Integer> symbolTable);
3      public abstract Integer stackSize();  // maximum size of stack needed
4  }
```

Figure 17.13: Die Klasse `Expr`.

Übersetzung einer Variablen

Um eine Variable v auszuwerten, laden wir diese Variable mit dem Kommando

```
iload v
```

auf den Stack. Daher hat die Klasse `Variable` die in Abbildung 17.14 gezeigte Form. Die Klasse Variable verwaltet eine Member-Variable mit dem Namen `mName`, die den Namen der Variablen angibt. Die Methode *compile*() legt zunächst in Zeile 8 eine neue Liste von Assembler-Kommandos an und erzeugt dann in Zeile 9 das Assembler-Kommando

```
iload v,
```

das als einziges Kommando in diese Liste eingefügt wird. Hierbei ist v die Nummer der Variablen, die in der Symbol-Tabelle, die dem Konstruktor als Argument übergeben wird, gespeichert ist. Anschließend kann die Liste als Ergebnis zurück gegeben werden.

Die Methode `stackSize` gibt beim Laden einer Variablen den Wert 1 zurück, denn es wird ja nur ein Objekt auf dem Stack abgelegt.

```
1  public class Variable extends Expr {
2      private String mName;
3
4      public Variable(String name) {
5          mName = name;
6      }
7      public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
8          List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
9          AssemblerCmd iload = new ILOAD(symbolTable.get(mName));
10         result.add(iload);
11         return result;
12     }
13     public Integer stackSize() {
14         return 1;
15     }
16 }
```

Figure 17.14: Die Klasse `Variable`.

Übersetzung einer Konstanten

Eine Konstante c kann mit Hilfe des Befehls

`ldc` c

auf den Stack geladen werden.

```
1  public class MyNumber extends Expr {
2      private Integer mNumber;
3
4      public MyNumber(Integer number) {
5          mNumber = number;
6      }
7      public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
8          List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
9          AssemblerCmd ldc = new LDC(mNumber);
10         result.add(ldc);
11         return result;
12     }
13     public Integer stackSize() {
14         return 1;
15     }
16 }
```

Figure 17.15: Die Klasse `MyNumber`.

Abbildung 17.15 zeigt die Implementierung der Klasse `MyNumber`, die eine Konstante darstellt. Die Konstante selbst wird in der Member-Variablen `mNumber` gespeichert. Die Methode *compile*() gibt eine Liste zurück, die den Befehl `ldc` enthält.

Die Methode `stackSize` gibt den Wert 1 zurück, den es wird ja nur eine Zahl auf den Stack gelegt.

Übersetzung zusammengesetzter Ausdrücke

Um einen Ausdruck der Form

lhs "+" *rhs*

zu übersetzen, muss zunächst Code erzeugt werden, der die Ausdrücke *lhs* und *rhs* rekursiv übersetzt. Wird dieser Code ausgeführt, so liegen auf dem Stack anschließend die Werte von *lhs* und *rhs*. Durch den Befehl `iadd` werden diese nun addiert. Die Übersetzung kann also wie folgt spezifiziert werden:

$$\textit{compile}(\textit{lhs}\ \text{"+"}\ \textit{rhs}) = \textit{lhs}.\textit{compile}() + \textit{rhs}.\textit{compile}() + [\texttt{iadd}],$$

wobei der Operator "+" auf der rechten Seite dieser Gleichung der Verkettung von Listen dient. Abbildung 17.16 zeigt die Umsetzung dieser Überlegung.

Bei der Berechnung der Größe des benötigten Stacks gehen wir von folgenden Überlegungen aus:

1. Zunächst benötigen wir für die Auswertung von *lhs* einen Stack der Größe *lhs*.`stackSize`(). Nachdem *lhs* *ausgewertet* worden ist, verbleibt aber nur der Wert von *lhs* auf dem Stack.

2. Falls wir für die Auswertung von *rhs* weniger Platz auf dem Stack benötigen als für die Auswertung von *lhs*, dann reicht insgesamt der Stack aus, der für die Auswertung von *lhs* allokiert worden ist, denn das Ergebnis der Auswertung von *lhs* benötigt nur eine Speicherstelle und da die Auswertung von *rhs* nach Voraussetzung weniger Platz auf dem Stack benötigt als die Auswertung von *lhs* benötigt hat, reicht der verbleibende Platz auf dem Stack aus um *rhs* auszuwerten.

3. Falls die Auswertung von *rhs* genauso viel oder mehr Platz braucht als die Auswertung von *lhs*, dann muss der Stack insgesamt die Höhe

   $$\textit{rhs}.\texttt{stackSize}() + 1$$

   haben, denn wir müssen zusätzlich ja noch das Ergebnis der Auswertung von *lhs* speichern.

Insgesamt sehen wir, dass die Höhe des Stacks durch die Formel

$$\texttt{max}(\textit{lhs}.\texttt{stackSize}(), \textit{rhs}.\texttt{stackSize}() + 1)$$

gegeben ist. Die Übersetzung von Ausdrücken der Form

$$\textit{lhs} - \textit{rhs}, \quad \textit{lhs} * \textit{rhs} \quad \text{und} \quad \textit{lhs}/\textit{rhs}$$

verläuft nach demselben Schema. Statt des Befehls `iadd` verwenden wir hier die entsprechenden Befehle `isub`, `imul` und `idiv`.

Übersetzung von Funktions-Aufrufen

Ein Funktions-Aufruf der Form $f(e_1, \cdots, e_n)$ kann übersetzt werden, indem zunächst die Ausdrücke e_1, $\cdots$, e_n übersetzt werden. Anschließend wird dann die Funktion f mit Hilfe des Kommandos `invokevirtual` aufgerufen. Damit hat die Übersetzung im Allgemeinen die folgende Form:

$$\textit{compile}\big(f(e_1, \cdots, e_n)\big) = \textit{compile}(e_1) + \cdots + \textit{compile}(e_n) + [\texttt{invokevirtual}\ f]$$

Allerdings müssen wir noch einen Sonderfall berücksichtigen. Falls es sich bei der Funktion f um die Funktion `println`() handelt, so müssen wir vor der eigentlichen Ausgabe noch den `PrintStream` auf den Stack legen, anschließend ist das Argument auszuwerten und zum Schluss können wir dann die Methode `println` aufrufen. Da die Funktion `println` selber kein Ergebnis berechnet, ist es in diesem Fall erforderlich, einen Dummy-Wert auf dem Stack

```
1  public class Sum extends Expr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Sum(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
10         List<AssemblerCmd> result = mLhs.compile(symbolTable);
11         result.addAll(mRhs.compile(symbolTable));
12         result.add(new IADD());
13         return result;
14     }
15     public Integer stackSize() {
16         return Math.max(mLhs.stackSize(), mRhs.stackSize() + 1);
17     }
18 }
```

Figure 17.16: Die Klasse `Sum`.

abzulegen. Philosophische Betrachtungen, die über den Rahmen der Vorlesung hinausgehen, legen nahe, hierfür den Wert 42 zu wählen.

Abbildung 17.17 zeigt die Implementierung der Klasse `FunctionCall`, die einen Funktions-Aufruf repräsentiert. Die Klasse hat zwei Member-Variablen.

1. `mName` ist der Name der aufgerufenen Funktion.

2. `mArgs` ist die Liste der Argumente, mit der die Funktion aufgerufen wird.

Falls es sich bei der Funktion nicht um die Methode `println` handelt, werden zunächst alle Argumente der Funktion ausgewertet und die Ergebnisse dieser Argumente auf dem Stack abgelegt. Schließlich wird die Funktion über den Befehl `invokevirtual` aufgerufen, der durch die Klasse `INVOKE` dargestellt wird.

Bei der Berechnung der Größe des benötigten Stacks ist zu berücksichtigen, das bei der Auswertung des Arguments mit dem Index i bereits i Werte auf dem Stack liegen.

```
1  public class FunctionCall extends Expr {
2      private String     mName;
3      private List<Expr> mArgs;
4
5      public FunctionCall(String name, List<Expr> args) {
6          mName = name;
7          mArgs = args;
8      }
9      public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
10         List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
11         if (mName.equals("println")) {
12             AssemblerCmd getStatic =
13                 new GETSTATIC("java/lang/System/out Ljava/io/PrintStream;");
14             result.add(getStatic);
15             for (Expr arg: mArgs) {
16                 result.addAll(arg.compile(symbolTable));
17             }
18             AssemblerCmd println = new PRINTLN();
19             AssemblerCmd bipush  = new BIPUSH(42);
20             result.add(println);
21             result.add(bipush);
22             return result;
23         }
24         for (Expr arg: mArgs) {
25             result.addAll(arg.compile(symbolTable));
26         }
27         String descr = Compiler.sClassName + "/" + mName + "(";
28         for (int i = 0; i < mArgs.size(); ++i) {
29             descr += "I";
30         }
31         descr += ")I";
32         AssemblerCmd invoke = new INVOKE(descr);
33         result.add(invoke);
34         return result;
35     }
36     public Integer stackSize() {
37         Integer biggest = 0;
38         for (int i = 0; i < mArgs.size(); ++i) {
39             biggest = Math.max(biggest, i + mArgs.get(i).stackSize());
40         }
41         if (mName.equals("println")) {
42             ++biggest;
43         }
44         return Math.max(biggest, 1);
45     }
46 }
```

Figure 17.17: Die Klasse `FunctionCall`.

17.4.2 Übersetzung von Boole'schen Ausdrücken

Boole'sche Ausdrücke werden aus Gleichungen und Ungleichungen mit Hilfe der logischen Operatoren "!" (Negation), "&&" (Konjunktion) und "||" (Disjunktion) aufgebaut. Wir beginnen mit der Übersetzung von Gleichungen.

Übersetzung von Gleichungen

Bevor wir eine Gleichung der Form

lhs == *rhs*

übersetzen können, müssen wir uns überlegen, was der erzeugte Code überhaupt erreichen soll. Eine naheliegende Forderung ist, dass am Ende auf dem Stack eine 1 abgelegt wird, wenn die Werte der beiden Ausdrücke *lhs* und *rhs* übereinstimmen. Andernfalls soll auf dem Stack eine 0 abgelegt werden. Die Übersetzung kann unter diesen Annahmen wie folgt ablaufen:

```
1  public class Equation extends BoolExpr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Equation(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
10         List<AssemblerCmd> result = mLhs.compile(symbolTable);
11         result.addAll(mRhs.compile(symbolTable));
12         LABEL        trueLabel = new LABEL();
13         LABEL        nextLabel = new LABEL();
14         AssemblerCmd if_icmpeq = new IF_ICMPEQ(trueLabel.getLabel());
15         AssemblerCmd bipush0   = new BIPUSH(0);
16         AssemblerCmd gotoNext  = new GOTO(nextLabel.getLabel());
17         AssemblerCmd bipush1   = new BIPUSH(1);
18         result.add(if_icmpeq);
19         result.add(bipush0);
20         result.add(gotoNext);
21         result.add(trueLabel);
22         result.add(bipush1);
23         result.add(nextLabel);
24         return result;
25     }
26     public Integer stackSize() {
27         return Math.max(mLhs.stackSize(), mRhs.stackSize() + 1);
28     }
29 }
```

Figure 17.18: Die Klasse `Equation`.

1. Zunächst erzeugen wir in den Zeilen 10 und 11 den Code zur Auswertung von *lhs* und *rhs*. Wenn dieser Code abgearbeitet worden ist, liegen die Werte von *lhs* und *rhs* auf dem Stack.
2. Anschließend überprüfen wir mit Hilfe des Befehls `if_icmpeq`, ob die beiden Werte gleich sind. Falls dies so ist, legen wir eine 1 auf den Stack, sonst eine 0.

Damit hat der erzeugte Code insgesamt die folgende Form

$$\begin{array}{lcl} \mathit{compile}(\mathit{lhs} \texttt{ == } \mathit{rhs}) & = & \mathit{lhs}.\mathit{compile}() \\ & + & \mathit{rhs}.\mathit{compile}() \\ & + & [\,\texttt{if\_icmpeq}\ \mathit{true}\,] \\ & + & [\,\texttt{bipush}\ 0\,] \\ & + & [\,\texttt{goto}\ \mathit{next}\,] \\ & + & [\,\mathit{true}\texttt{:}\,] \\ & + & [\,\texttt{bipush}\ 1\,] \\ & + & [\,\mathit{next}\texttt{:}\,] \end{array}$$

Diese Gleichung ist in der Methode *compile*() der Klasse `Equation` eins zu eins umgesetzt worden.

Übersetzung von negierten Gleichungen

Die Übersetzung einer negierten Gleichung der Form

lhs `!=` *rhs*

verläuft analog zu der Übersetzung einer Gleichung, denn wir müssen hier nur den Befehl `if_icmpeq` durch den Befehl `if_icmpne` ersetzen. Daher lautet die Spezifikation

$$\begin{array}{lcl} \mathit{compile}(\mathit{lhs} \texttt{ != } \mathit{rhs}) & = & \mathit{lhs}.\mathit{compile}() \\ & + & \mathit{rhs}.\mathit{compile}() \\ & + & [\,\texttt{if\_icmpne}\ \mathit{true}\,] \\ & + & [\,\texttt{bipush}\ 0\,] \\ & + & [\,\texttt{goto}\ \mathit{next}\,] \\ & + & [\,\mathit{true}\texttt{:}\,] \\ & + & [\,\texttt{bipush}\ 1\,] \\ & + & [\,\mathit{next}\texttt{:}\,] \end{array}$$

Dies kann wieder eins zu eins umgesetzt werden. Aus Platzgründen verzichten wir darauf, die Klasse `Inequation` zu präsentieren.

Übersetzung von Ungleichungen

The compilation of inequations of the form

lhs `<=` *rhs*, *lhs* `<` *rhs*, *lhs* `>=` *rhs*, and *lhs* `<` *rhs*,

is essentially the same as the compilation of equations. We only have to replace the assembler command `if_icmpeq` with either `if_icmple`, `if_icmplt`, `if_icmpge`, or `if_icmpgt`.

Übersetzung von Konjunktionen

Die Übersetzung einer Konjunktion der Form

lhs `&&` *rhs*

kann wie folgt spezifiziert werden:

$$\begin{array}{lcl} \mathit{compile}(\mathit{lhs} \texttt{ \&\& } \mathit{rhs}) & = & \mathit{lhs}.\mathit{compile}() \\ & + & \mathit{rhs}.\mathit{compile}() \\ & + & [\,\texttt{iand}\,] \end{array}$$

Abbildung 17.19 zeigt die Umsetzung dieser Gleichung.

Die obige Umsetzung entspricht allerdings nicht dem, was in der Sprache `C` tatsächlich passiert. Dort wird die Auswertung eines Ausdrucks der Form

lhs `&&` *rhs*

```
1  public class Conjunction extends BoolExpr {
2      private BoolExpr mLhs;
3      private BoolExpr mRhs;
4
5      public Conjunction(BoolExpr lhs, BoolExpr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
10         List<AssemblerCmd> result = mLhs.compile(symbolTable);
11         result.addAll(mRhs.compile(symbolTable));
12         AssemblerCmd iand = new IAND();
13         result.add(iand);
14         return result;
15     }
16     public Integer stackSize() {
17         return Math.max(mLhs.stackSize(), mRhs.stackSize() + 1);
18     }
19 }
```

Figure 17.19: Die Klasse `Conjunction`.

abgebrochen, sobald das Ergebnis der Auswertung feststeht. Liefert die Auswertung von *lhs* als Ergebnis eine 0, so wird der Ausdruck *rhs* nicht mehr ausgewertet. Falls dieser Ausdruck Seiteneffekte hat, ist das Ergebnis der Auswertung dann also verschieden von unserer Auswertung.

Eine Disjunktion wird in analoger Weise auf den Assembler-Befehl `ior` zurück geführt.

Übersetzung von Negationen

Die Übersetzung einer Negation der Form !*expr* kann nicht so geradlinig behandelt werden wie die Übersetzung von Konjunktionen und Disjunktionen. Das liegt daran, dass es einen Assembler-Befehl `inot`, der den oben auf dem Stack liegenden Wert negiert, nicht gibt. Aber es geht auch anders, denn weil wir die Wahrheitswerte durch 1 und 0 darstellen, können wir die Negation arithmetisch wie folgt spezifizieren:

$$!x = 1 - x.$$

Damit verläuft die Übersetzung einer Negation nach dem folgenden Schema:

$$\begin{array}{lcl} \mathit{compile}(!\,\mathit{expr}) & = & [\,\texttt{bipush}\ 1\,] \\ & + & \mathit{expr}.\mathit{compile}() \\ & + & [\,\texttt{isub}\,] \end{array}$$

Abbildung 17.20 zeigt die Umsetzung dieser Idee.

17.4.3 How to Compile a Statement

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. 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.

```
1  public class Negation extends BoolExpr {
2      private BoolExpr mExpr;
3
4      public Negation(BoolExpr expr) {
5          mExpr = expr;
6      }
7      public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
8          List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
9          AssemblerCmd bipush1 = new BIPUSH(1);
10         AssemblerCmd isub    = new ISUB();
11         result.add(bipush1);
12         result.addAll(mExpr.compile(symbolTable));
13         result.add(isub);
14         return result;
15     }
16     public Integer stackSize() {
17         return mExpr.stackSize() + 1;
18     }
19 }
```

Figure 17.20: Die Klasse `Negation`.

Übersetzung von Zuweisungen

Wir untersuchen als erstes, wie eine Zuweisung der Form

x = *expr*

übersetzt werden kann. Die Grundidee besteht darin, zunächst den Ausdruck *expr* auszuwerten. Als Folge dieser Auswertung wird dann ein Wert auf dem Stack zurück bleiben, der das Ergebnis dieser Auswertung ist. Diesen Wert können wir mit dem Befehl `istore` unter der Variable x abspeichern. Folglich kann die Übersetzung einer Zuweisung wie folgt spezifiziert werden:

$$\begin{array}{lcl} \mathit{compile}(x\text{=}\mathit{expr}) & = & \mathit{expr}.\mathit{compile}() \\ & + & [\,\texttt{istore}\ x\,] \end{array}$$

Die Idee wird in der in Abbildung 17.21 gezeigten Klasse `Assign` umgesetzt.

Übersetzung von Ausdrücken als Befehlen

Die Übersetzung eines Ausdrucks, der als Befehl verwendet wird, birgt eine Tücke: Die Übersetzung des Ausdrucks selber hinterlässt auf dem Stack einen Wert. Dieser muss aber bei Beendigung des Befehls vom Stack entfernt werden! Daher müssen wir den Befehl `pop` an das Ende der Liste der Assembler-Befehle anfügen, die bei der Übersetzung des Ausdrucks erzeugt werden. Die Übersetzung eines Befehls vom Typ `ExprStatement` wird also wie folgt spezifiziert:

$$\begin{array}{lcl} \mathit{compile}(\mathit{expr};) & = & \mathit{expr}.\mathit{compile}() \\ & + & [\,\texttt{pop}\,] \end{array}$$

Abbildung 17.22 zeigt die Klasse `ExprStatement`, in der diese Überlegung umgesetzt wird.

Die Übersetzung von Verzweigungs-Befehlen

Als nächstes überlegen wir, wie ein Verzweigungs-Befehl der Form

`if` (*expr*) *statement*

```
1  public class Assign extends Statement {
2      private String mVar;
3      private Expr   mExpr;
4
5      public Assign(String var, Expr expr) {
6          mVar  = var;
7          mExpr = expr;
8      }
9      public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
10         List<AssemblerCmd> result   = mExpr.compile(symbolTable);
11         AssemblerCmd       storeCmd = new ISTORE(symbolTable.get(mVar));
12         result.add(storeCmd);
13         return result;
14     }
15     public Integer stackSize() {
16         return mExpr.stackSize();
17     }
18 }
```

Figure 17.21: Die Klasse `Assign`.

```
1  public class ExprStatement extends Statement {
2      private Expr mExpr;
3
4      public ExprStatement(Expr expr) {
5          mExpr = expr;
6      }
7      public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
8          List<AssemblerCmd> result = mExpr.compile(symbolTable);
9          AssemblerCmd       popCmd = new POP();
10         result.add(popCmd);
11         return result;
12     }
13     public Integer stackSize() {
14         return mExpr.stackSize();
15     }
16 }
```

Figure 17.22: Die Klasse `ExprStatement`.

übersetzt werden kann. Offenbar muss zunächst der Boole'sche Ausdruck *expr* übersetzt werden. Die Auswertung dieses Ausdrucks wird auf dem Stack entweder eine 1 oder eine 0 hinterlassen, je nachdem, ob die Bedingung des Tests wahr oder falsch wahr. Mit dem Befehl `ifeq` können wir überprüfen, welcher dieser beiden Fälle vorliegt. Das führt zu der folgenden Spezifikation:

$$\begin{aligned} \mathit{compile}(\texttt{if}\ (\mathit{expr})\ \mathit{statement}) &= \mathit{expr}.\mathit{compile}() \\ &+ [\,\texttt{ifeq}\ \mathit{else}\,] \\ &+ \mathit{statement}.\mathit{compile}() \\ &+ [\,\mathit{else}\texttt{:}\,] \end{aligned}$$

Diese Spezifikation ist in der Abbildung 17.23 umgesetzt worden.

```
1  public class IfThen extends Statement {
2      private BoolExpr  mBoolExpr;
3      private Statement mStatement;
4
5      public IfThen(BoolExpr boolExpr, Statement statement) {
6          mBoolExpr = boolExpr;
7          mStatement = statement;
8      }
9      public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
10         List<AssemblerCmd> result = mBoolExpr.compile(symbolTable);
11         LABEL        elseLabel = new LABEL();
12         AssemblerCmd ifeq      = new IFEQ(elseLabel.getLabel());
13         result.add(ifeq);
14         result.addAll(mStatement.compile(symbolTable));
15         result.add(elseLabel);
16         return result;
17     }
18     public Integer stackSize() {
19         return Math.max(mBoolExpr.stackSize(), mStatement.stackSize());
20     }
21 }
```

Figure 17.23: Die Klasse `IfThen.java`

Die Übersetzung eines Verzweigungs-Befehls der Form

`if` (*expr*) *thenStmnt* `else` *elseStmnt*

erfolgt in analoger Art und Weise. Diesmal lautet die Spezifikation:

$$\begin{aligned} \mathit{compile}(\texttt{if}\ (\mathit{expr})\ \mathit{thenStmnt}\ \texttt{else}\ \mathit{elseStmnt}) \;=\; & \mathit{expr.compile}() \\ + \;& [\ \texttt{ifeq}\ \mathit{else}\ ] \\ + \;& \mathit{thenStmnt.compile}() \\ + \;& [\ \texttt{goto}\ \mathit{next}\ ] \\ + \;& [\ \mathit{else}\texttt{:}\ ] \\ + \;& \mathit{elseStmnt.compile}()\ ] \\ + \;& [\ \mathit{next}\texttt{:}\ ] \end{aligned}$$

Diese Spezifikation ist in der Abbildung 17.24 umgesetzt worden.

Die Übersetzung einer Schleife

Die Übersetzung einer `while`-Schleife der Form

`while` (*cond*) *statement*

orientiert sich an der folgenden Spezifikation:

$$\begin{aligned} \mathit{compile}(\texttt{while}\ (\mathit{cond})\ \mathit{stmnt}) \;=\; & [\ \mathit{loop}\texttt{:}\ ] \\ + \;& \mathit{cond.compile}() \\ + \;& [\ \texttt{ifeq}\ \mathit{next}\ ] \\ + \;& \mathit{stmnt.compile}() \\ + \;& [\ \texttt{goto}\ \mathit{loop}\ ] \\ + \;& [\ \mathit{next}\texttt{:}\ ] \end{aligned}$$

Die Umsetzung dieser Spezifikation sehen Sie in Abbildung 17.25.

```
1  public class IfThenElse extends Statement {
2      private BoolExpr  mExpr;
3      private Statement mThen;
4      private Statement mElse;
5
6      public IfThenElse(BoolExpr expr, Statement thenStmnt, Statement elseStmnt) {
7          mExpr = expr;
8          mThen = thenStmnt;
9          mElse = elseStmnt;
10     }
11     public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
12         List<AssemblerCmd> result = mExpr.compile(symbolTable);
13         LABEL        elseLabel = new LABEL();
14         LABEL        nextLabel = new LABEL();
15         AssemblerCmd ifeq      = new IFEQ(elseLabel.getLabel());
16         AssemblerCmd gotoNext  = new GOTO(nextLabel.getLabel());
17         result.add(ifeq);
18         result.addAll(mThen.compile(symbolTable));
19         result.add(gotoNext);
20         result.add(elseLabel);
21         result.addAll(mElse.compile(symbolTable));
22         result.add(nextLabel);
23         return result;
24     }
25     public Integer stackSize() {
26         return Math.max(mExpr.stackSize(), Math.max(mThen.stackSize(), mElse.stackSize()));
27     }
28 }
```

Figure 17.24: Die Klasse `IfThenElse`.

Übersetzen einer Liste von Befehlen

Eine in geschweiften Klammern eingeschlossene Liste von Befehlen der Form

$$\{stmnt_1\,;\ \cdots\ stmnt_n\,;\}$$

wird dadurch übersetzt, dass die Listen, die bei der Übersetzung der einzelnen Befehle $stmnt_i$ entstehen, aneinander gehängt werden:

$$compile\big(\{stmnt_1\,;\ \cdots\ stmnt_n\,;\}\big) \quad = \quad compile(stmnt_1) + \cdots + compile(stmnt_n).$$

Diese Idee ist in der Klasse `Block` realisiert worden. Abbildung 17.26 zeigt diese Klasse.

17.4.4 Zusammenspiel der Komponenten

Nachdem wir jetzt gesehen haben, wie die einzelnen Teile eines Programms in Listen von Assembler-Befehlen übersetzt werden können, müssen wir noch zeigen, wie die einzelnen Komponenten unseres Programms zusammen spielen. Dazu sind noch zwei Klassen zu diskutieren:

1. Die Klasse `Function` repräsentiert die Definition einer Funktion.
2. Die Klasse `Program` repräsentiert das vollständige Programm.

```
1  public class While extends Statement {
2      private BoolExpr  mCondition;
3      private Statement mStatement;
4
5      public While(BoolExpr condition, Statement statement) {
6          mCondition = condition;
7          mStatement = statement;
8      }
9      public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
10         List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
11         LABEL        loopLabel = new LABEL();
12         LABEL        nextLabel = new LABEL();
13         AssemblerCmd ifeq      = new IFEQ(nextLabel.getLabel());
14         AssemblerCmd gotoLoop  = new GOTO(loopLabel.getLabel());
15         result.add(loopLabel);
16         result.addAll(mCondition.compile(symbolTable));
17         result.add(ifeq);
18         result.addAll(mStatement.compile(symbolTable));
19         result.add(gotoLoop);
20         result.add(nextLabel);
21         return result;
22     }
23     public Integer stackSize() {
24         return Math.max(mCondition.stackSize(), mStatement.stackSize());
25     }
26 }
```

Figure 17.25: Die Klasse `While`.

Wir beginnen mit der Diskussion der Klasse `Function`. Abbildung 17.27 zeigt die Klasse `Function`, allerdings ohne die Implementierung der Methode *compile*(), die wir aus Platzgründen in die Abbildung 17.28 ausgelagert haben.

Die Klasse `Function` enthält vier Member-Variablen:

1. `mName` gibt den Namen der Funktion an.
2. `mParameterList` ist die Liste der Parameter, mit der die Funktion aufgerufen wird.
3. `mDeclarations` ist die Liste der Variablen-Deklarationen.
4. `mBody` ist die Liste von Befehlen, die im Rumpf der Funktion ausgeführt werden.

```
1  public class Block extends Statement {
2      private List<Statement> mStatementList;
3
4      public Block(List<Statement> statementList) {
5          mStatementList = statementList;
6      }
7      public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
8          List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
9          for (Statement stmnt: mStatementList) {
10             result.addAll(stmnt.compile(symbolTable));
11         }
12         return result;
13     }
14     public Integer stackSize() {
15         Integer biggest = 0;
16         for (Statement stmnt: mStatementList) {
17             biggest = Math.max(biggest, stmnt.stackSize());
18         }
19         return biggest;
20     }
21 }
```

Figure 17.26: Die Klasse `Block`.

```
1  public class Function {
2      private String            mName;
3      private List<String>      mParameterList;
4      private List<Declaration> mDeclarations;
5      private List<Statement>   mBody;
6
7      private Integer           mLocals; // number of local variables
8
9      public Function(String            name,
10                     List<String>      parameterList,
11                     List<Declaration> declarations,
12                     List<Statement>   body)
13     {
14         mName          = name;
15         mParameterList = parameterList;
16         mDeclarations  = declarations;
17         mBody          = body;
18         mLocals        = mParameterList.size() + mDeclarations.size();
19     }
20     public List<AssemblerCmd> compile() { ... }
21     public Integer stackSize() { ... }
22 }
```

Figure 17.27: Die Klasse `Function`.

```
1  public List<AssemblerCmd> compile() {
2      Map<String, Integer> symbolTable = new TreeMap();
3      Integer count = 0;
4      for (String var: mParameterList) {
5          symbolTable.put(var, count);
6          ++count;
7      }
8      for (Declaration decl: mDeclarations) {
9          symbolTable.put(decl.getVar(), count);
10         ++count;
11     }
12     Integer stackSize = 0;
13     for (Statement stmnt: mBody) {
14         stackSize = Math.max(stackSize, stmnt.stackSize());
15     }
16     List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
17     AssemblerCmd nl = new NEWLINE();
18     result.add(nl);
19     if (mName.equals("main")) {
20         AssemblerCmd main        = new MAIN();
21         AssemblerCmd limitLocals = new LIMIT("locals", mLocals);
22         AssemblerCmd limitStack  = new LIMIT("stack", stackSize);
23         result.add(main);
24         result.add(limitLocals);
25         result.add(limitStack);
26         for (Statement stmnt: mBody) {
27             result.addAll(stmnt.compile(symbolTable));
28         }
29         AssemblerCmd myReturn = new RETURN();
30         AssemblerCmd endMain  = new END_METHOD();
31         result.add(myReturn);
32         result.add(endMain);
33     } else {
34         AssemblerCmd method      = new METHOD(mName, mParameterList.size());
35         AssemblerCmd limitLocals = new LIMIT("locals", mLocals);
36         AssemblerCmd limitStack  = new LIMIT("stack", stackSize);
37         result.add(method);
38         result.add(limitLocals);
39         result.add(limitStack);
40         for (Statement stmnt: mBody) {
41             result.addAll(stmnt.compile(symbolTable));
42         }
43         AssemblerCmd endMethod = new END_METHOD();
44         result.add(endMethod);
45     }
46     return result;
47 }
```

Figure 17.28: Die Methode *compile*().

Die eigentliche Arbeit der Klasse `Funktion` wird in der Methode *compile*(), die in Abbildung 17.28 gezeigt ist, geleistet. Es sind zwei Fälle zu unterscheiden:

1. Falls die zu übersetzende Funktion den Namen "`main`" hat, so hat der erzeugte Code die folgende 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-main
```

   Hier bezeichnet l die Anzahl der in der Funktion `main` verwendeten lokalen Variablen, s ist die maximale Höhe des Stacks und s_1, $\cdots$, s_n bezeichnen die einzelnen Assemblerbefehle, die bei der Übersetzung des Rumpfes der Funktion erzeugt werden.

2. Andernfalls hat der erzeugte Code die folgende Form:

```
1           .method public static f(I···I)I
2           .limit locals l
3           .limit stack s
4              s_1
5              ⋮
6              s_n
7           .end-method
```

   Hier bezeichnet f den Namen der Funktion, l ist die Anzahl der in der Funktion verwendeten lokalen Variablen und s ist die maximale Höhe des Stacks. Weiter sind s_1, $\cdots$, s_n die Assemblerbefehle des Rumpfes der Funktion.

Abbildung 17.29 zeigt die Implementierung der Funktion `stackSize`. Da die einzelnen Befehle nichts auf dem Stack zurück lassen dürfen, ergibt sich die Höhe des Stacks, der zur Ausführung aller Befehle benötigt wird, als das Maximum der Höhen der einzelnen Befehle.

```
1  public Integer stackSize() {
2      Integer biggest = 0;
3      for (Statement stmnt: mBody) {
4          biggest = Math.max(biggest, stmnt.stackSize());
5      }
6      return biggest;
7  }
```

Figure 17.29: Computing the size of the stack

Zum Abschluss diskutieren wir die Klasse `Program`, die in Abbildung 17.30 gezeigt wird. Diese Klasse verwaltet in der Member-Variablen `mFunctionList` die Liste aller zu übersetzenden Funktionen.

Bei der Übersetzung der Funktionen ist darauf zu achten, dass zuerst die Funktion *main*() übersetzt wird, denn diese muss am Anfang der erzeugten Assembler-Datei stehen. In der `C`-Datei ist die Funktion *main*() aber die letzte Funktion, denn in der Sprache `C` müssen alle Funktionen vor ihrer Verwendung deklariert worden sein.

```
1  public class Program {
2      private List<Function> mFunctionList;
3
4      public Program(List<Function> functionList) {
5          mFunctionList = functionList;
6      }
7      public List<AssemblerCmd> compile() {
8          List<AssemblerCmd> fctList = new LinkedList<AssemblerCmd>();
9          int indexMain = mFunctionList.size() - 1;
10         Function main = mFunctionList.get(indexMain);
11         fctList.addAll(main.compile());
12         for (int i = 0; i < indexMain; ++i) {
13             Function f = mFunctionList.get(i);
14             fctList.addAll(f.compile());
15         }
16         return fctList;
17     }
18 }
```

Figure 17.30: Die Klasse `Program`.

Wie übersetzen in Zeile 11 als erstes die Funktion *main*(). Anschließend werden in der Schleife, die sich von Zeile 12 bis 15 erstreckt, die restlichen Funktionen übersetzt. Der erzeugte Code befindet sich dann in der Liste `fctList`, die als Ergebnis zurück gegeben wird.

Übersetzen wir die in Abbildung 17.2 gezeigte Funktion zur Berechnung der Summe $\sum_{i=1}^{n} i$ mit dem Compiler, so erhalten wir die in den Abbildungen 17.8 und 17.9 gezeigten Assembler-Datei. Vergleichen wir dieses Programm mit dem in Abbildung 17.8 gezeigten Assembler-Programm, das wir von Hand geschrieben haben, so fällt auf, dass das vom Compiler erzeugte Programm deutlich länger ist. Es wäre nun Aufgabe eines Code-Optimierers, den erzeugten Code zu verkürzen und dadurch zu optimieren. Eine Diskussion von Techniken zur Code-Optimierung geht allerdings über den Rahmen der Vorlesung heraus.

Exercise 43: The goal of this exercise is to extend the language supported by the compiler presented in this chapter.

(a) Extend the compiler so that `for`-loops are supported. The syntax of these `for`-loops should be similar to the syntax of `for`-loops in the programming language `C`.

(b) Extend the compiler so that increment and decrement statements of the form

++var; and *--var*;

are supported. As we only intend to use these statements in the update statement of a `for`-loop, there is no need to support these operators inside expressions.

In order to test your version of the compiler, rewrite the example integer-`C` program presented in this chapter to use both a `for`-loop and the increment operator. ◇

Bibliography

[ASUL06] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling: Volume I: Parsing*. Prentice-Hall, 1972.

[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.

[Ear68] Jay C. Earley. *An efficient context-free parsing algorithm*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1968.

[Ear70] Jay C. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java^{TM} Language Specification*. Sun Microsystems, 1.0 edition, August 1996. appeared also as book with same title in Addison-Wesleys 'The Java Series'.

[HFA+99] Scott E. Hudson, Frank Flannery, C. Scott Ananian, Dan Wang, and Andrew W. Appel. CUP – LALR parser generator for *Java*, 1999. Available at `http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html`.

[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.

[Kle09] Gerwin Klein. JFlex User's Manual: Version 1.4.3. Technical report, Technische Universität München, 2009. Available at: `http://jflex.de/jflex.pdf`.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, December 1965.

[Les75] Michael E. Lesk. Lex – A lexical analyzer generator. Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, 1975.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[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.

[Ner58] Anil Nerode. Linear automaton transformations. *Proceedings of the AMS*, 9:541–544, 1958.

[Nic93] G. T. Nicol. *Flex: The Lexical Scanner Generator, for Flex Version 2.3.7*. FSF, 1993.

[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.

[Sal66] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM*, 13(1):158–169, 1966.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, second edition, 2006.

[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