

## Formal Languages and Their Applications

— Autumn 2020 —

Prof. Dr. Karl Stroetmann

November 1, 2020

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

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

The previous time I gave this lecture was in 2015. Then, I had used *Java* as the main programming language. Currently, I am rewriting all programs using *Python*. Until this task is finished, these lecture notes are going to change frequently. 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**

1	Introduction and Motivation				4
	1.1 Basic Definitions				 4
	1.2 Overview				
	1.3 Literature				
	1.4 Check your Understanding				 7
2	Regular Expressions				8
	2.1 Preliminary Definitions				 8
	2.2 The Formal Definition of Regular Expressions				 9
	2.3 Algebraic Simplification of Regular Expressions				 12
	2.4 Check your Understanding				 13
3	Building Scanners with Ply				15
	3.1 The Structure of a PLY Scanner Specification				 16
	3.2 The Syntax of Regular Expressions in <i>Python</i>				
	3.3 A Complex Example: Evaluating an Exam				
	3.4 Scanner States				
4	Finite State Machines				24
Ċ	4.1 Deterministic Finite State Machines				
	4.2 Non-Deterministic Finite State Machines				
	4.3 Equivalence of Deterministic and Non-Deterministic FSMs				
	4.3.1 Implementation				
	4.4 From Regular Expressions to Deterministic Finite State Machines				
	4.4.1 Implementation				 39
	4.5 Translating a Deterministic FSM into a Regular Expression				 39
	4.5.1 Implementation				 43
	4.6 Minimierung endlicher Automaten				 43
	4.6.1 Implementation				 46
	4.7 Conclusion				 46
	4.8 Check your Understanding				 46
5	The Theory of Regular Languages				47
	5.1 Closure Properties of Regular Languages				 47
	5.2 Recognizing Empty Languages				 50
	5.3 Equivalence of Regular Expressions				
	5.4 Limits of Regular Languages				 51
6	Context-Free Languages				56
	6.1 Kontextfreie Grammatiken				
	6.1.1 Ableitungen				
	6.12 Absorbing Processing Control of the Control of	•	•	•	6.4

CONTENTS

		6.1.3 Mehrdeutige Grammatiken	
	6.2	Top-Down-Parser	
		6.2.1 Umschreiben der Grammatik	
		6.2.2 Implementing a Top Down Parser in <i>Python</i>	
		6.2.3 Implementing a Recursive Descent Parser that Uses an EBNF Grammar	76
7	Intro		78
	7.1	A Parser for Arithmetic Expressions	78
	7.2	Evaluation of Arithmetical Expressions	31
	7.3	Generating Abstract Syntax Trees with Antle 8	35
		7.3.1 Implementing the Parser	35
8	Inte	preter 8	39
9	Grer	zen kontextfreier Sprachen	95
	9.1	Beseitigung nutzloser Symbole*	
	9.2	Parse-Bäume als Listen	
	9.3	Das Pumping-Lemma für kontextfreie Sprachen	
	9.4	Anwendungen des Pumping-Lemmas	
		9.4.1 Die Sprache $L=\{\mathbf{a}^k\mathbf{b}^k\mathbf{c}^k\mid k\in\mathbb{N}\}$ ist nicht kontextfrei	
10			
ΙU		ey-Parser*	
		Der Algorithmus von Earley	
	10.2	Implementing Earley's Algorithm in SetlX	
		10.2.1 The class earleyItem	
		10.2.2 The class grammar	
	10.0	10.2.3 The class earleyParser	
		Korrektheit und Vollständigkeit	
	10.4	Analyse der Komplexität	24
11		om-Up-Parser 12	_
		Bottom-Up-Parser	
		Shift-Reduce-Parser	
	11.3	SLR-Parser         13	
		11.3.1 Shift-Reduce- und Reduce-Reduce-Konflikte	
		Kanonische LR-Parser	
		LALR-Parser	
	11.6	Vergleich von SLR-, LR- und LALR-Parsern	
		11.6.1 $SLR$ -Sprache $\subsetneq LALR$ -Sprache	
		11.6.2 LALR-Sprache $\subsetneq$ kanonische LR-Sprache	
		11.6.3 Bewertung der verschiedenen Methoden	54
12		Parser-Generator Cup 15	
	12.1	Spezifikation einer Grammatik für CUP	55
	12.2	Generierung eines Cup-Scanner mit Hilfe von Flex	50
	12.3	Shift/Reduce und Reduce/Reduce-Konflikte	52
		Operator-Präzendenzen	
	12.5	Das Dangling-Else-Problem	59
		Auflösung von Reduce/Reduce-Konflikten	
		12.6.1 Look-Ahead-Konflikte	
		12.6.2 Mysteriöse Reduce/Reduce-Konflikte	74
	12.7	Auflösung von Shift/Reduce-Konflikten	76

CONTENTS

<b>13</b>	Types and Type Checking*	180
	13.1 Eine Beispielsprache	181
	13.2 Grundlegende Begriffe	182
	13.3 A Type Checking Algorithm	
	13.4 Implementierung eines Typ-Checkers für TTL	
	13.5 Inklusions-Polymorphismus	
14	Assembler	199
	14.1 Introduction into JASMIN Assembler	199
	14.2 Assembler Instructions	201
	14.2.1 Instructions to Manipulate the Stack	
	14.3 An Example Program	
	14.4 Disassembler*	
15	Entwicklung eines einfachen Compilers	212
	15.1 Die Programmiersprache <i>Integer</i> -C	212
	15.2 Developing the Scanner and the Parser	
	15.3 Darstellung der Assembler-Befehle	
	15.4 Die Code-Erzeugung	
	15.4.1 Übersetzung arithmetischer Ausdrücke	
	15.4.2 Übersetzung von Boole'schen Ausdrücken	
	15.4.3 How to Compile a Statement	
	15.4.4 Zusammenspiel der Komponenten	
	13.4.4 Zusammenspiel der Nombonenlen	∠აე

# 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  $\Sigma$  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.  $\Sigma = \{a, \dots, z, A, \dots, Z\}$  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  $\{A, \cdots, Z\}$ .

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

$$w = c_1 \cdots c_n$$
 instead of  $w = [c_1, \cdots, c_n]$ .

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

#### Examples:

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

$$w_1 := "01110"$$
 and  $w_2 := "11001"$ ,

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

$$w_1 \in \Sigma^*$$
 and  $w_2 \in \Sigma^*$ .

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

$$w := "example",$$

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

$$w_1 \cdot w_2 = "0110"$$
 and  $w_2 \cdot w_1 = "1001"$ .

#### **Definition 3 (Formal Language)**

If  $\Sigma$  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

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

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  $\Sigma^*$  that start with the character 1 as well as the string 0, which only contains the character 0. For example, we have

"100" 
$$\in L_{\mathbb{N}}$$
, but 010  $ot\in L_{\mathbb{N}}$ .

Let us define a function

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

0

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

- (a) 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:

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

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

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

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

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

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  $\dagger$  is some new symbol that is different from all symbols in  $\Sigma_{\text{ASCII}}$ . The universal language  $L_u$  is the set of all strings of the form

$$p \dagger x \dagger y$$

such that

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

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

### 1.2 Overview

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

- 1. In the first part we discuss the theory of regular languages.
  - (a) We will start with *regular expressions*. After a formal definition of this notion, we discuss how regular expressions are used in *Python*.
  - (b) Next, we show how the tool PLY can be used to generate scanners.
  - (c) Then, we turn to the implementation of regular expressions via finite state machines.
  - (d) We show how the equivalence of regular expressions can be checked.
  - (e) We finish our discussion of regular languages by discussing their limits. In particular, we discuss the Pumping Lemma.

- 2. In the second part of this lecture we discuss context free languages. Context free languages are a formalism to describe the syntax of programming languages. In particular, the theory of regular languages can be used to implement parsers.
  - (a) We discuss the definition of context free grammars. These are used to specify context free languages.
  - (b) We discuss the parser generators ANTLR and PLY.
  - (c) We present the theory that is necessary to understand the parser generator PLY.
  - (d) We proceed to discuss the limits of context free languages.
- 3. In the last part of this lecture we discuss interpreters and compilers.

### 1.3 Literature

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

- (a) Introduction to Automata Theory, Languages, and Computation [HMU06]
  - This book is the bible with respect to the theory of formal languages and it contains all the theoretical results discussed in this lecture. Obviously, we will only be able to cover a small part of the results discussed in this book.
- (b) Introduction to the Theory of Computation [Sip12]
  - This is another readable introduction to the theory of formal languages. It also discusses the theory of computability, which is not covered in this lecture.
- (c) Compilers Principles, Techniques and Tools [ASUL06]
  - This book is one of the standard references with respect to the theory of compilers. It also covers a fair amount of the theory of formal languages.

## 1.4 Check your Understanding

- (a) Define the notion of an alphabet.
- (b) Given an alphabet, define the notion of a string.
- (c) Define the notion of a formal language.

# Chapter 2

# Regular Expressions

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

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

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

## 2.1 Preliminary Definitions

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

**Definition 4 (Product of Languages)** If  $\Sigma$  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

$$L_1 \cdot L_2 := \{ w_1 \cdot w_2 \mid w_1 \in L_1 \land w_2 \in L_2 \}.$$

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

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

then the product of  $L_1$  and  $L_2$  is given as

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

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

0

B.C.: n = 0:

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

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

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

$$L^{n+1} = L^n \cdot L$$

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

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

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

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

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

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

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

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

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

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

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  $\Sigma$ . This set is denoted as RegExp $_{\Sigma}$  and is defined by induction. Simultaneously, we define the function

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

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

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

<sup>&</sup>lt;sup>1</sup> Given a set M the power set of M, i.e. the set of all subsets of M, is denoted as  $2^M$ .

#### 1. $\emptyset \in \mathtt{RegExp}_{\Sigma}$

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

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

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

#### 2. $\varepsilon \in \text{RegExp}_{\Sigma}$

The regular expression  $\varepsilon$  denotes the language that only contains the empty string  $\varepsilon$ :

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

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

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

#### 3. $c \in \Sigma \to c \in \text{RegExp}_{\Sigma}$ .

Every character from the alphabet  $\Sigma$  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. $r_1 \in \text{RegExp}_{\Sigma} \land r_2 \in \text{RegExp}_{\Sigma} \rightarrow r_1 + r_2 \in \text{RegExp}_{\Sigma}$

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  $\Sigma$ , i.e. we have "+"  $\notin \Sigma$ .

### 5. $r_1 \in \mathtt{RegExp}_\Sigma \land r_2 \in \mathtt{RegExp}_\Sigma \to r_1 \cdot r_2 \in \mathtt{RegExp}_\Sigma$

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

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

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

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

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:

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

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

#### 7. $r \in \text{RegExp}_{\Sigma} \to (r) \in \text{RegExp}_{\Sigma}$

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

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

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

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

$$a + b \cdot c$$

is to be interpreted as

$$(a+b)\cdot c$$
 or  $a+(b\cdot c)$ .

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

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

Using these conventions, the regular expression

$$a + b \cdot c^*$$
 is interpreted as  $a + (b \cdot (c^*))$ .

**Examples**: In the following examples, the alphabet  $\Sigma$  is defined as

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

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

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

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

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

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

$$L(r_2) = \{ w \in \Sigma^* \mid |w| > 1 \}.$$

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

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  $(b+c)^*$  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. 
$$r_4 := (b+c)^* \cdot a \cdot (b+c)^* + (a+c)^* \cdot b \cdot (a+c)^*$$

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".

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

**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  $\Sigma = \{a, b, c\}$ . 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  $(0+1\cdot0)^*$ . The remaining part of w is shorter and, by induction, can be shown to be generated by  $(0+1\cdot0)^*\cdot1^*$ .
- (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

$$(1+\varepsilon)\cdot(0\cdot0^*\cdot1)^*\cdot0^*?$$

**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

$$r_1 \doteq r_2$$
 iff  $L(r_1) = L(r_2)$ ,

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 operator is commutative for sets:

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

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

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

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

$$L((r_1 \cdot r_2) \cdot r_3) = \{xw \mid x \in L(r_1 \cdot r_2) \land w \in L(r_3)\}$$

$$= \{(uv)w \mid u \in L(r_1) \land v \in L(r_2) \land w \in L(r_3)\}$$

$$= \{u(vw) \mid u \in L(r_1) \land v \in L(r_2) \land w \in L(r_3)\}$$

$$= \{uy \mid u \in L(r_1) \land y \in L(r_2 \cdot r_3)\}$$

$$= L(r_1 \cdot (r_2 \cdot r_3)).$$

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

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

holds. In order to prove the inclusion

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

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

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

we have  $w \in L((r^*)^*)$  if and only if there is an  $n \in \mathbb{N}$  such that there are strings  $u_1, \dots, u_n \in L(r^*)$  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  $i \in \{1, \dots, n\}$  such that for  $j = 1, \dots, m(i)$  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

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

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

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 RegExp<sub>\Sigma</sub> defined?

- (c) How is the function  $L:\mathtt{RegExp}_\Sigma\to 2^{\Sigma^*}$  defined?
- (d) Given  $r_1, r_2 \in \mathtt{RegExp}_{\Sigma}$ , 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", "for", or "switch".
- 2. Operator symbols like "+", "+=", "<", or "<=".
- 3. Parentheses like "(", "[", and "{" and the corresponding closing symbols.
- 4. Constants. The language C distinguishes between three different kinds of constants:
  - (a) Numbers, for example the integer "123" or the floating point number "1.23e2".
  - (b) Strings, which are enclosed in double. For example, ""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 character ',', the tabulator '\t', the line break '\n', and the carriage return '\r' are white space symbols.

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

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

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

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

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

```
/* Hello World program */
#include<stdio.h>

int main() {
    printf("Hello World!\n");
    return 1;
}
```

Figure 3.1: A simple C program.

## 3.1 The Structure of a Ply Scanner Specification

In this section we introduce Python Lex-Yacc which is also known as PLY. As the home page of PLY states, "PLY is an implementation of the lex and yacc parsing tools for Python". The tool has been developed by David Beazley. In this section, we only discuss PLY as a scanner generator. In Chapter 12 we will discuss how PLY can be used to generate a parser.

We begin with a simple example. In general, a  $\operatorname{PLY}$  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  $\operatorname{PLY}$  documentation.

- 1. The module ply.lex contains the definition of the function ply.lex.lex() that is able to generate a scanner. Therefore, this module is imported in line 1.
- 2. The first part of a scanner specification is the token declaration section. Syntactically, this is just a list containing the names of all tokens. Note that all token names have to start with a capital letter.
  - In Figure 3.2 the token declaration section extends from line 3 to line 11.
- 3. The second part contains the token definitions. There are two kinds of token definitions:
  - (a) Immediate token definitions have the following form:

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

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

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

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

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

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

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

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

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

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

Note that the function t\_newline does not return a token.

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

```
"' \t.'"
```

is <u>not</u> 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 " $\mathbf{r}$ ", for otherwise the character sequence " $\mathsf{t}$ " 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 = [
3
        'NUMBER',
        'PLUS',
5
        'MINUS',
        'TIMES',
        'DIVIDE',
        'LPAREN',
        'RPAREN'
10
    ]
11
12
    t_PLUS
                = r' + '
13
                = r' - '
    t_MINUS
14
                = r'\*'
    t_TIMES
    t_DIVIDE
              = r'/'
16
               = r'\('
    t_LPAREN
    t_RPAREN
               = r'\)'
18
    def t_NUMBER(t):
20
         r'0|[1-9][0-9]*'
         t.value = int(t.value)
22
         return t
23
24
    def t_newline(t):
25
         r' n+'
26
         t.lexer.lineno += len(t.value)
27
    t_ignore = ' \t'
29
30
    def t_error(t):
31
         print("Illegal character '%s'" % t.value[0])
         t.lexer.skip(1)
33
34
    lexer = lex.lex()
35
    data = '3 + 4 * 10 + 007 + (-20) * 2'
37
    lexer.input(data)
39
    for tok in lexer:
40
         print(tok)
41
```

Figure 3.2: A simple scanner Specification for PLY.

## 3.2 The Syntax of Regular Expressions in Python

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

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

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Regexp-Tutorial.ipynb that points to this tutorial.

## 3.3 A Complex Example: Evaluating an Exam

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

```
Class: Algorithms and Complexity
Group: TIT09AID
MaxPoints = 60

Exercise: 1. 2. 3. 4. 5. 6.
Jim Smith: 9 12 10 6 6 0
John Slow<sup>1</sup>: 4 4 2 0 - -
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.ipynbulk

### 3.4 Scanner States

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

<sup>&</sup>lt;sup>1</sup>You know nothing, John Slow.

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

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

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

#### Exercises/LaTeX2HTML.

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

Figure 3.4: An example LATEX input file.

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

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

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

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

The area A of a circle is given as

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

while its circumference satisfies

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

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

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

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

- 1. A LATEX input file has the following structure:
  - (a) The first line list the type of the document. In our example, it reads

```
\documentclass{article}.
```

This line will be transformed into the following HTML:

```
<html>
<head>
<script type="text/javascript"
src="http://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS-MML_HTMLorMML">
</script>
```

Here, the <script> tag is necessary in order for the MATHML to be displayed correctly.

(b) The next line has the form:

```
\begin{document}
```

This line precedes the content and should be translated into the tag

<body>.

- (c) After that, the LATEX file consists of text that contains mathematical formula.
- (d) The LATEX input file finishes with a line of the form

```
\end{document}.
```

This line should be translated into the tags

```
</body></html>.
```

2. In LATEX, an inline formula is started and ended with a single dollar symbol "\$". In MATHML, an inline formula is written as

```
<math xmlns="http://www.w3.org/1998/Math/MathML" display='inline'>···</math>.
```

Here, I have used " $\cdots$ " 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  $\[Mathematical\]$  a mathematical variable does not need any special markup, in  $\[Mathematical\]$  a mathematical variable is written using the tags  $\[mathematical\]$  and  $\[mathematical\]$  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  $\pi$  is written using the command "\pi". In MATHML, we have to make use of the HTML entity "π" and hence we would write  $\pi$  as

```
<mn>&pi;</mn>.
```

7. In LATEX the multiplication operator "·" is written using the command "\cdot". In MATHML, we have to make use of the HTML entity "⋅" and hence we would write "·" as

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

9. In  $\triangle T_E X$ , raising an expression e to the nth power is done using the operator " $^{\circ}$ ". 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

$$\mbox{msup}<\mbox{mi}>\mbox{mi}>\mbox{mn}>2<\mbox{msup}>.$$

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

$$\label{local_model} $$ \mbox{mi>a++b2.$$

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

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

In  ${\rm MATHML}$ , a sum with lower and upper limits is created via the tags <munderover> and </munderover> and the  ${\rm HTML}$  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:

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

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

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

# Chapter 4

# **Finite State Machines**

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

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

### 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 me explain this idea more precisely:

- 1. Initially, the  $F_{\rm SM}$  is in a state that is known as the start state.
- 2. In every step of its computation, the  $F_{SM}$  reads one character of the input string s. Every time a character is processed, the state of the  $F_{SM}$  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 tsmall returns t

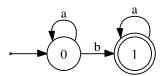


Figure 4.1: An FSM to recognize the language  $L(a^* \cdot b \cdot a^*)$ .

Finite state machines are best presented graphically. Figure 4.1 depicts a simple  $\mathrm{Fsm}$  that recognizes those strings that are specified by the regular expression

$$a^* \cdot b \cdot a^*$$
.

This  $F_{SM}$  has but two states. These states are called 0 and 1.

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

If the  $F_{SM}$  is in the state 0 and reads the character "a", then the  $F_{SM}$  stays in the state 0. This is specified in the figure by an arrow labeled with the character "a" that both starts and ends in the state 0. On the other hand, if the character "b" is read while the  $F_{SM}$  is in state 0, then the  $F_{SM}$  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  $F_{SM}$  is in state 1, then the  $F_{SM}$  does not change its state. On the other hand, if the  $F_{SM}$  reads the character "b" while in state 1, then the next state is undefined since there is no arrow originating from state 1 that is labeled with the character "b".

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

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

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

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

where the components Q,  $\Sigma$ ,  $\delta$ ,  $q_0$ , and A have the following properties:

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

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

In the figures depicting FSMs transitions of the form  $\delta(q,c)=\Omega$  are not shown.

- 4.  $q_0 \in Q$  is the start state.
- 5.  $A \subseteq Q$  is the set of accepting states.

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

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

where we have:

- 1.  $Q = \{0, 1\},\$
- 2.  $\Sigma = \{a, b\},\$
- 3.  $\delta = \{\langle 0, a \rangle \mapsto 0, \langle 0, b \rangle \mapsto 1, \langle 1, a \rangle \mapsto 1, \langle 1, b \rangle \mapsto \Omega \}$ ,
- 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  $\delta$  to a new function

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

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

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

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

because if a deterministic  $F_{\rm SM}$  does not read any character, it cannot change its state.

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

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

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

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

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

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

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

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

is complete if the transition function  $\delta$  never returns the undefined value  $\Omega$ , i.e. we have

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

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

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

**Proof**: Assume F is given as

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

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

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

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

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

1. 
$$\hat{Q} := Q \cup \{ 2 \},$$

the dead state is added to the set Q.

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

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

- (a)  $\delta(q,c) \neq \Omega \rightarrow \widehat{\delta}(q,c) = \delta(q,c)$  for all  $q \in Q$  and  $c \in \Sigma$ . If the state transition function is defined for the state q and the character c, then  $\widehat{\delta}(q,c)$  is the same as  $\delta(q,c)$ .
- (b)  $\delta(q,c)=\Omega \to \widehat{\delta}(q,c)=$   $\mbox{\cite{a}}$  for all  $q\in Q$  and  $c\in \Sigma.$  If the state transition function  $\delta$  is undefined for the state q and the character c, then  $\widehat{\delta}(q,c)$  returns the dead state  $\mbox{\cite{a}}$ .
- (c)  $\hat{\delta}(\mathbf{Z},c) = \mathbf{Z}$  for all  $c \in \Sigma$ , because there is no escape from death<sup>1</sup>.

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  $\mathfrak Z$  and remains in this state regardless of the rest of the input string. As the dead state  $\mathfrak Z$  is not an accepting state, the languages accepted by F and  $\widehat{F}$  are identical.  $\square$ 

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

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

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

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

- (a) At least three characters have been read. In this case, the last three characters are "aaa".
- (b) Two characters have been read. In this case, the string that has been read so far is the string "aa".
- (c) Only one character has been read so far. In this case, the string that has been is "a".
- (d) Nothing has yet been read and therefore the string that has been read is  $\varepsilon$ .

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

State 1: "aab".

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

State 2: "aba".

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

State 3: "abb".

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

State 4: "bab".

State 5: "bba".

State 6: "bbb".

<sup>&</sup>lt;sup>1</sup>Or, as the disciples of the Drowned Good say: "What is dead may never die".

#### State 7: "baa".

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

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

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

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

$$\delta(0, b) = 1.$$

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

$$\delta(1, a) = 2.$$

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

$$\delta(1, b) = 3.$$

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

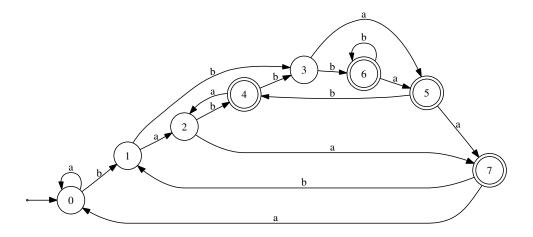


Figure 4.2: An FSM accepting  $L(a + b)^* \cdot b \cdot (a + b) \cdot (a + b)$ .

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

### 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

$$L((a+b)^* \cdot b \cdot (a+b) \cdot (a+b))$$

needs 8 different states since the  $F_{SM}$  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  $F_{SM}$  if the  $F_{SM}$  would be permitted to *choose* its next state from a given set of states.



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

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

$$(a+b)^* \cdot b \cdot (a+b) \cdot (a+b)$$
.

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  $F_{SM}$  reads the letter a while it is in this state, the  $F_{SM}$  will stay in state 0. However, if the  $F_{SM}$  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  $F_{SM}$  switches to state 3 if it reads either the character a or the character b
- 4. State 3 is the accepting state. There is no transition from this state. Hence, if the  $F_{SM}$  is in state 3 and there are still characters to read, then the  $F_{SM}$  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":

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

In this computation, the  $F_{\rm SM}$  has chosen the correct transition when reading the first occurrence of the character "b". If the  $F_{\rm SM}$  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:

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

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

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

Here, the  $F_{\rm SM}$  has switched to early into the state 1. In this case, the  $F_{\rm SM}$  dies when reading the last character "b". If the  $F_{\rm SM}$  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:

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

The previous examples show that in order to avoid premature death, the given non-deterministic  $F_{SM}$  has to choose its successor state wisely. If F is a non-deterministic  $F_{SM}$  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  $F_{SM}$  is far more powerful than the concept of a deterministic  $F_{SM}$ . After all, a non-deterministic  $F_{SM}$  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  $F_{SMS}$  have the same power to recognize languages: Every language recognized by a non-deterministic  $F_{SM}$  is also recognized by a deterministic  $F_{SM}$ . In order to prove this claim, we have to formalize the notion of a non-deterministic  $F_{SM}$ . The definition that follows is more general than the informal description of non-deterministic  $F_{SMS}$  given so far, as we will allow the  $F_{SM}$  to also have  $\varepsilon$  transitions. An  $\varepsilon$  transition allows the  $F_{SM}$  to switch its state without reading any character. For example, if there is an  $\varepsilon$  transition from the state 1 into the state 2, we write

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

**Definition 11 (NFA)** A non-deterministic FSM (abbreviated as NFA for non-deterministic automaton) is a 5-tupel  $\langle Q, \Sigma, \delta, q_0, A \rangle$ ,

such that the following holds:

- 1. Q is the finite set of states.
- 2.  $\Sigma$  is the input alphabet.
- 3.  $\delta$  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,  $\delta$  is called the transition function.

- 4.  $q_0 \in Q$  is the start state.
- 5.  $A \subseteq Q$  is the set of accepting states.

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

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

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

$$q_1 \stackrel{c}{\mapsto} q_2$$
.

In order to distinguish a deterministic  $F_{SM}$  from a non-deterministic  $F_{SM}$ , deterministic  $F_{SM}$ s are also called  $D_{FA}$  which is short for deterministic finite automaton.

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

$$F = \langle Q, \Sigma, \delta, 0, A \rangle$$
 where

- 1.  $Q = \{0, 1, 2, 3\}.$
- 2.  $\Sigma = \{a, b\}.$

3. 
$$\delta = \{ \langle 0, \mathbf{a} \rangle \mapsto \{0\}, \langle 0, \mathbf{b} \rangle \mapsto \{0, 1\}, \langle 0, \varepsilon \rangle \mapsto \{\}, \langle 1, \mathbf{a} \rangle \mapsto \{2\}, \langle 1, \mathbf{b} \rangle \mapsto \{2\}, \langle 1, \varepsilon \rangle \mapsto \{\}, \langle 2, \mathbf{a} \rangle \mapsto \{3\}, \langle 2, \mathbf{b} \rangle \mapsto \{3\}, \langle 2, \varepsilon \rangle \mapsto \{\}\}.$$

It is more convenient to specify the transition function  $\delta$  as follows:

$$0 \stackrel{\text{a}}{\mapsto} 0$$
,  $0 \stackrel{\text{b}}{\mapsto} 0$ ,  $0 \stackrel{\text{b}}{\mapsto} 1$ ,  $1 \stackrel{\text{a}}{\mapsto} 2$ ,  $1 \stackrel{\text{b}}{\mapsto} 2$ .  $2 \stackrel{\text{a}}{\mapsto} 3$  and  $2 \stackrel{\text{b}}{\mapsto} 3$ .

- 4. The start state is 0.
- 5.  $A = \{3\}$ , hence the only accepting state is 3.

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

$$\langle q, s \rangle$$

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

$$\langle q_1, cs \rangle \leadsto \langle q_2, s \rangle$$
 iff  $q_1 \stackrel{c}{\mapsto} q_2$ , i.e. if  $q_2 \in \delta(q_1, c)$ .

Therefore, we have  $\langle q_1,cs\rangle \leadsto \langle q_2,s\rangle$  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

$$\langle q_1, s \rangle \leadsto \langle q_2, s \rangle$$
 iff  $q_1 \stackrel{\varepsilon}{\mapsto} q_2$ , i.e. if  $q_2 \in \delta(q_1, \varepsilon)$ .

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

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

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

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

$$L(F) = \{ w \in \{ a, b \} \mid |w| \ge 3 \land w[-3] = b \}$$

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

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

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

## 4.3 Equivalence of Deterministic and Non-Deterministic FSMs

In this section we show how a non-deterministic  $\ensuremath{\mathrm{Fsm}}$ 

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

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

$$L(F) = L(det(F))$$

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

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

$$ec: Q \rightarrow 2^Q$$

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

B.C.: 
$$q \in ec(q)$$
.

I.S.: 
$$p \in ec(q) \land r \in \delta(p, \varepsilon) \rightarrow r \in ec(q)$$
.

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

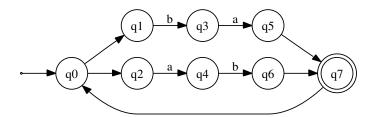


Figure 4.4: A non-deterministic FSM with  $\varepsilon$ -transitions.

**Example**: Figure 4.4 shows a non-deterministic  $F_{SM}$  with  $\varepsilon$ -transitions. In the figure, the  $\varepsilon$ -transitions are shown as unlabelled arrows. We compute the  $\varepsilon$ -closure for all states:

- 1.  $ec(q_0) = \{q_0, q_1, q_2\},\$
- 2.  $ec(q_1) = \{q_1\},\$
- 3.  $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(F) 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 F could reach when it reads the character c in state q and then performs an arbitrary number of  $\varepsilon$ -transitions. Formally, the definition of  $\delta^*$  is as follows:

$$\delta^*(q_1,c) := \bigcup \{ \operatorname{ec}(q_2) \mid q_2 \in \delta(q_1,c) \}.$$

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  $\varepsilon$ -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. 
$$\delta^*(q_0, \mathbf{a}) = \{\},$$

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

2.  $\delta^*(q_1, 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  $\varepsilon$ -transitions.

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

because when the letter 'a' is read in the state  $q_3$  the  $F_{SM}$  switches into the state  $q_5$ . From  $q_5$  the states  $q_7$ ,  $q_0$ ,  $q_1$  and  $q_2$  are reachable by  $\varepsilon$ -transitions.

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

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

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

$$\Delta(M,c) := \left\{ \int \left\{ \delta^*(q,c) \mid q \in M \right\}. \right.$$

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  $\varepsilon$ -transitions. Then we take the union of these sets.

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

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

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

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

where the components of this tuple are defined as follows:

- 1. The set of states of det(F) is the set of all subsets of Q and therefore it is equal to the power set  $2^Q$ . Later we will see that we do not need all of these subsets. The reason is that the states are those subsets that could be reached from the start state  $q_0$  when some string has been read. In most cases there are some combinations of states that can not be reached and the corresponding sets are not really needed as states.
- 2. The input alphabet  $\Sigma$  does not change when going from F to det(F). After all, the deterministic  $Fsm\ det(F)$  has to recognize the same language as the non-deterministic  $Fsm\ F$ .
- 3. The previously defined function  $\Delta$  specifies how the set of states changes when a character is read.
- 4. The start state  $ec(q_0)$  of the non-deterministic Fsm det(F) is the set of all states that can be reached from the start state  $q_0$  of the non-deterministic Fsm F via  $\varepsilon$ -transitions.
- 5. The set of accepting states  $\widehat{A}$  is the set of those subsets of Q that contain an accepting state of the FSM Q:

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

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

Solution: We start by computing the set of states.

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

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

2. As we have  $\delta(0, \mathbf{a}) = \{0\}$  and there are no  $\varepsilon$ -transitions we have

$$\Delta(S_0, \mathbf{a}) = \Delta(\{0\}, \mathbf{a}) = \{0\} = S_0.$$

3. As we have  $\delta(0, b) = \{0, 1\}$  we conclude

$$S_1 := \Delta(S_0, \mathbf{b}) = \Delta(\{0\}, \mathbf{b}) = \{0, 1\}.$$

4. We have that  $\delta(0, \mathbf{a}) = \{0\}$  and  $\delta(1, \mathbf{a}) = \{2\}$ . Hence

$$S_2 := \Delta(S_1, \mathbf{a}) = \Delta(\{0, 1\}, \mathbf{a}) = \{0, 2\}.$$

5. We have  $\delta(0, b) \in \{0, 1\}$  and  $\delta(1, b) = \{2\}$ . Therefore

$$S_4 := \Delta(S_1, b) = \Delta(\{0, 1\}, b) = \{0, 1, 2\}$$

Similarly we derive the following:

6. 
$$S_3 := \Delta(S_2, \mathbf{a}) = \Delta(\{0, 2\}, \mathbf{a}) = \{0, 3\}.$$

7. 
$$S_5 := \Delta(S_2, b) = \Delta(\{0, 2\}, b) = \{0, 1, 3\}.$$

8. 
$$S_6 := \Delta(S_4, \mathbf{a}) = \Delta(\{0, 1, 2\}, \mathbf{a}) = \{0, 2, 3\}.$$

9. 
$$S_7 := \Delta(S_4, b) = \Delta(\{0, 1, 2\}, b) = \{0, 1, 2, 3\}.$$

10. 
$$\Delta(S_3, \mathbf{a}) = \Delta(\{0, 3\}, \mathbf{a}) = \{0\} = S_0.$$

11. 
$$\Delta(S_3, b) = \Delta(\{0, 3\}, b) = \{0, 1\} = S_1.$$

12. 
$$\Delta(S_5, \mathbf{a}) = \Delta(\{0, 1, 3\}, \mathbf{a}) = \{0, 2\} = S_2.$$

13. 
$$\Delta(S_5, b) = \Delta(\{0, 1, 3\}, b) = \{0, 1, 2\} = S_4.$$

14. 
$$\Delta(S_6, \mathbf{a}) = \Delta(\{0, 2, 3\}, \mathbf{a}) = \{0, 3\} = S_3.$$

15. 
$$\Delta(S_6, b) = \Delta(\{0, 2, 3\}, b) = \{0, 1, 3\} = S_5.$$

16. 
$$\Delta(S_7, \mathbf{a}) = \Delta(\{0, 1, 2, 3\}, \mathbf{a}) = \{0, 2, 3\} = S_6$$

17. 
$$\Delta(S_7, b) = \Delta(\{0, 1, 2, 3\}, b) = \{0, 1, 2, 3\} = S_7.$$

These are all possible sets of states that the deterministic Fsm det(F) can reach. For a better overview let us summarize the definitions of the individual states of the deterministic 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 det(F) is given as follows:

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

The transition function  $\Delta$  is shown as a table:

$\Delta$	$S_0$	$S_1$	$S_2$	$S_3$	$S_4$	$S_5$	$S_6$	$S_7$
a	$S_0$	$S_2$	$S_3$	$S_0$	$S_6$	$S_2$	$S_3$	$S_6$
b	$S_1$	$S_4$	$S_5$	$S_1$	$S_7$	$S_4$	$S_5$	$S_7$

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

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

Therefore we have now found the deterministic FSM det(F). We have

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

This FSM is shown in Figure 4.5 on page 36.

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

**Exercise 7**: Transform the non-deterministic FSM F that is shown in Figure 4.4 on page 32 into an equivalent deterministic FSM det(F).

### 4.3.1 Implementation

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

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

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

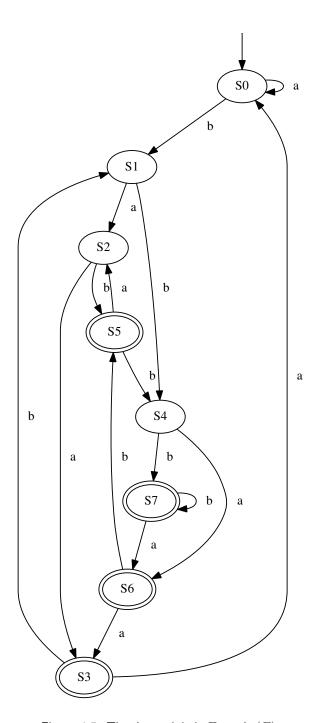


Figure 4.5: The deterministic FSM det(F).

### 4.4 From Regular Expressions to Deterministic Finite State Machines

In this section we construct a non-deterministic  $\operatorname{Fsm} A(r)$  that accepts the same language as a given regular expression r, i.e. we will have

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

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

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

In the following we assume that  $\Sigma$  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  $F_{SM}$  has no transitions at all.



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

Figure 4.6 shows the FSM  $A(\emptyset)$ . It is obvious that we have  $L(A(\emptyset)) = \{\}$ .

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

Figure 4.7 shows the  $Fsm\ A(\varepsilon)$ . We have that  $L(A(\varepsilon))=\{\text{''}\}$ , i.e. the automaton only accepts the empty string.

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

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

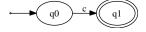


Figure 4.8: The FSM A(c).

Figure 4.8 shows A(c). We have that  $L(A(c)) = \{c\}$ , 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  $\varepsilon$ -transition from  $q_2$  to  $q_3$ . Formally, this transition function  $\delta$  can be specified as follows:

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

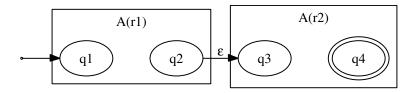


Figure 4.9: The FSM  $A(r_1 \cdot r_2)$ .

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

Instead of having an  $\varepsilon$ -transition from  $q_2$  to  $q_3$  we can identify the states  $q_2$  and  $q_3$ . The advantage is that the resulting  $F_{SM}$  is smaller. We will do this when creating  $F_{SM}$  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:

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



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

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

- (a)  $q_0$  is the start state of the FSM  $A(r_1 + r_2)$ ,
- (b)  $q_5$  is the only accepting state of the FSM  $A(r_1 + r_2)$ .

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

- (a) The new start state  $q_0$  has two  $\varepsilon$ -transitions leading to the start states  $q_1$  and  $q_2$  of the FSMs  $A(r_1)$  and  $A(r_2)$ .
- (b) Each of the accepting states  $q_3$  and  $q_4$  of the FSMs  $A(r_1)$  and  $A(r_2)$  has an  $\varepsilon$ -transition to the new accepting state  $q_5$ .

In order to simplify this  $F_{SM}$  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  $F_{SM}$  would be more difficult to understand and hence we are <u>not</u> doing this when creating  $F_{SM}$ s by hand.

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

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

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

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



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

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

- (a)  $q_0$  is the start state of  $A(r^*)$ ,
- (b)  $q_3$  is the only accepting state of  $A(r^*)$ .

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

- (a) The new start state  $q_0$  has  $\varepsilon$ -transitions to the states  $q_1$  and  $q_3$ .
- (b)  $q_2$  has an  $\varepsilon$ -transition back to the state  $q_1$ .
- (c)  $q_2$  also has an  $\varepsilon$ -transition to the state  $q_3$ .

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

**Exercise 8**: Construct a non-deterministic  $F_{SM}$  that accepts the language specified by the regular expression  $a^* \cdot b^*$ .

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

**Exercise 9**: Construct a non-deterministic  $F_{\rm SM}$  for the regular expression

$$(a+b) \cdot a^* \cdot b$$
.

#### 4.4.1 Implementation

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

### 4.5 Translating a Deterministic Fsm into a Regular Expression

In this last section we start with a deterministic  $F_{SM}$  F and construct a regular expression r such that we have

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

We assume that the  $Fsm\ F$  is given as follows:

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

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

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

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

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

To this end we define the ternary relation

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

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

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

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

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

(a)  $p \stackrel{\varepsilon}{\mapsto}_k p$ ,

because when the empty string is read we can only reach the state p if we start in the state p.

(b)  $\delta(p,c) = q \Rightarrow p \stackrel{c}{\mapsto}_k q$ ,

because when the  $\mathrm{Fsm}$  reads the character c and switches from state p directly to the state q, there are no states "inbetween".

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

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

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

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

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

B.C.: k = 0.

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

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

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

as the set of all letters that take the  $F_{SM}$  from the state  $p_1$  to the state  $p_2$ . If this set is not empty we define

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

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

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

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

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

If this set is not empty we have

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

Otherwise we have

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

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

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

- (a) We already have  $p_1 \stackrel{w}{\mapsto}_k p_2$ .
- (b) The string w can be written as  $w = w_1 s_1 \cdots s_l w_2$  where we have:
  - $p_1 \stackrel{w_1}{\mapsto}_k q_k$ ,
  - $q_k \stackrel{s_i}{\mapsto}_k q_k$  for all  $i = \{1, \dots, l\}$ ,
  - $\bullet q_k \stackrel{w_2}{\mapsto}_k p_2.$

Therefore we define

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

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

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

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

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

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

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

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

Then the regular expression r(A) is defined as

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

This regular expression specifies those strings that take the  $Fsm\ F$  from its start state  $q_0$  into any of its accepting states

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

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

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

$$r^{(1)}(0,0) = r^{(0)}(0,0) + r^{(0)}(0,0) \cdot \left(r^{(0)}(0,0)\right)^* \cdot r^{(0)}(0,0)$$
  
$$\stackrel{\cdot}{=} r^{(0)}(0,0) \cdot \left(r^{(0)}(0,0)\right)^*$$

In the last step we used the fact that

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Exercise 11**: Take the  $F_{\rm SM}$  shown in Figure 4.12 on page 42 and construct a regular expression specifying the same language as this  $F_{\rm SM}$ .

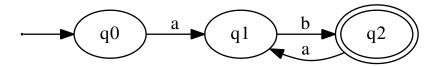


Figure 4.12: A deterministic finite state machine.

#### 4.5.1 Implementation

The Jupyter notebook

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

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

### 4.6 Minimierung endlicher Automaten

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

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

minimiert werden kann. Ohne Beschränkung der Allgemeinheit wollen wir dabei voraussetzen, dass der Automat F 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

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

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

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

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

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

die als zweites Argument nicht nur einen Buchstaben sondern auch einen String akzeptiert. Der Funktions-Aufruf  $\delta^*(q,s)$  berechnet den Zustand p, in den der Automat F gelangt, wenn der Automat im Zustand q den String s liest. Da die Funktion  $\delta^*$  eine Verallgemeinerung der Funktion  $\delta$  ist, werden wir in der Notation nicht zwischen  $\delta$  und  $\delta^*$  unterscheiden und einfach nur  $\delta$  schreiben.

Offensichtlich können wir in einem endlichen Automaten  $F=\langle Q,\Sigma,\delta,q_0,A\rangle$  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.

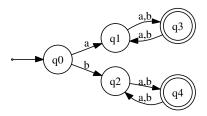


Figure 4.13: Ein endlicher Automat mit äquivalenten Zuständen.

Im Allgemeinen können wir einen Automaten dadurch minimieren, dass wir bestimmte Zustände identifizieren. Betrachten wir beispielsweise den in Abbildung 4.13 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.

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

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

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

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

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

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  $\delta$  in a way that the new version of  $\delta$  will return  $p_1$  in all those cases where the old version of  $\delta$  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

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

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

1. Zunächst initialisieren wir V mit allen 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  $\varepsilon$  von einem nicht-akzeptierenden Zustand unterschieden werden:

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

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 Zustände  $\delta(p,c)$  und  $\delta(q,c)$  bereits unterscheidbar sind, fügen wir dieses Paar zur Menge V hinzu:

```
while (\exists \langle p,q \rangle \in Q \times Q : \exists c \in \Sigma : \langle \delta(p,c), \delta(q,c) \rangle \in V \land \langle p,q \rangle \notin V) { choose \langle p,q \rangle \in Q \times Q such that \langle \delta(p,c), \delta(q,c) \rangle \in V \land \langle p,q \rangle \notin V { V := V \cup \{ \langle p,q \rangle, \langle q,p \rangle \}; }
```

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  $\langle p,q \rangle \not\in V$  gilt. Es lässt sich zeigen, dass der so konstruierte Automat tatsächlich minimal ist.

**Beispiel**: Wir betrachten den in Abbildung 4.13 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 i 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

$$\delta(q_0, b) = q_2$$
,  $\delta(q_2, b) = q_4$  und  $q_2 \not\sim q_4$ .

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

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

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

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

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

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

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 4.14 zeigt den entsprechenden reduzierten endlichen Automaten.



Figure 4.14: Der reduzierte endliche Automat.

**Aufgabe 12**: 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.

### 4.6.1 Implementation

The Jupyter notebook

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

contains a program that takes a deterministic  $\mathrm{FSM}\ F$  and returns an equivalent  $\mathrm{FSM}\ that$  has the minimum number of states.

### 4.7 Conclusion

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

- (a) Every deterministic finite state machine can be translated into an equivalent regular expression.
- (b) Every regular expression can be translated into an equivalent non-deterministic FSM.
- (c) Every non-deterministic  $F_{\rm SM}$  can be transformed into an equivalent deterministic  $F_{\rm SM}$ .

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

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

### 4.8 Check your Understanding

- (a) How are deterministic and non-deterministic finite state machines defined?
- (b) Given a non-deterministic  $FSM\ F$ , can you convert it into a deterministic  $FSM\ det(F)$  that accepts the same language as the  $FSM\ F$ ?
- (c) Given a regular expression r, can you describe the steps necessary to construct a non-deterministic FSM F that accepts the language specified by r?
- (d) Given a deterministic  $F_{SM}$  F can you describe how to construct a regular expression r that specifies the language accepted by F?
- (e) How do we mimimize a finite state machine?

## Chapter 5

# The Theory of Regular Languages

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

$$L = L(r)$$

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

- (a) The union  $L_1 \cup L_2$  of two regular languages  $L_1$  and  $L_2$  is a regular language.
- (b) The intersection  $L_1 \cap L_2$  of two regular languages  $L_1$  und  $L_2$  is a regular language.
- (c) The complement  $\Sigma^* \setminus L$  of a regular language L is a regular language.

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

$$r_1 \doteq r_2$$

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

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

is not regular. To summarize, this chapter

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

### 5.1 Closure Properties of Regular Languages

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

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

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

$$L_1 = L(r_1)$$
 and  $L_2 = L(r_2)$ 

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

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

Therefore,  $L_1 \cup L_2$  is a regular language.

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

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

$$L_1 = L(r_1)$$
 und  $L_2 = L(r_2)$ .

First, we construct two complete deterministic  $F_{\rm SMS}$   $F_1$  and  $F_2$  that accept these languages, i.e. we have

$$L(F_1) = L_1$$
 and  $L(F_2) = L_2$ .

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

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

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

$$F := \langle Q_1 \times Q_2, \Sigma, \delta, \langle q_1, q_2 \rangle, A_1 \times A_2 \rangle$$

where the state transition function

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

is defined as

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

Effectively, the FSM F runs the FSM  $F_1$  and  $F_2$  in parallel. In order to do so, the states of F are pairs of the form  $\langle p_1,p_2\rangle$  where  $p_1$  is a state from  $F_1$  and  $p_2$  is a state from  $F_2$  and the transition function  $\delta$  computes the state that follows on  $\langle p_1,p_2\rangle$  when a character c is read, by simultaneously computing the next states of  $p_1$  and  $p_2$  in the FSMs  $F_1$  and  $F_2$  when these FSMs read the character c. A string s is accepted by F if and only if both  $F_1$  and  $F_2$  accept s. Therefore, the set  $F_1$ 0 of accepting states of the FSM  $F_2$ 1 is defined as:

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

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

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

Therefore we have shown that

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

and this completes the proof.

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

$$\land : \textit{RegExp} \times \textit{RegExp} \rightarrow \textit{RegExp}$$

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

0

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

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

$$L(F_1) = L(r_1)$$
 and  $L(F_2) = L(r_2)$ 

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

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

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

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

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

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

Then we have

$$\begin{aligned} &w \in L\Big(\overline{\det(F)}\Big) \\ \Leftrightarrow &\delta(q_0,w) \in Q \backslash A \\ \Leftrightarrow &\delta(q_0,w) \not \in A \\ \Leftrightarrow &w \not \in L\Big(\det(F)\Big) \\ \Leftrightarrow &w \not \in L(F) \\ \Leftrightarrow &w \not \in L(r) \\ \Leftrightarrow &w \not \in L \\ \Leftrightarrow &w \in \Sigma^* \backslash L \end{aligned}$$

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

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

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

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

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

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

**Exercise 13**: Assume  $\Sigma$  to be some alphabet. For a string  $s=c_1c_2\cdots c_{n-1}c_n\in\Sigma^*$  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=abc, then  $s^R=cba$ . 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.

**\rightarrow** 

### 5.2 Recognizing Empty Languages

In this section we develop an algorithm that take a deterministic  $\operatorname{Fsm}$ 

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

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

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

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

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

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

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

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

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

- 1.  $L(\emptyset) = \{\}.$
- 2.  $L(\varepsilon) \neq \{\}$ .
- 3.  $L(c) \neq \{\}$  for all  $c \in \Sigma$ .
- 4.  $L(r_1 \cdot r_2) = \{\} \Leftrightarrow L(r_1) = \{\} \lor L(r_2) = \{\}.$
- 5.  $L(r_1 + r_2) = \{\} \iff L(r_1) = \{\} \land L(r_2) = \{\}.$
- 6.  $L(r^*) \neq \{\}.$

### 5.3 Equivalence of Regular Expressions

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

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

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

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

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

$$L(r_1) = L(r_2) \Leftrightarrow L(r_1) \subseteq L(r_2) \wedge L(r_2) \subseteq L(r_1)$$
  
$$\Leftrightarrow L(r_1) \backslash L(r_2) = \{\} \wedge L(r_2) \backslash L(r_1) = \{\}$$

Next, assume that  $F_1$  and  $F_2$  are deterministic  $F_{\rm SMS}$  such that

$$L(F_1) = L(r_1)$$
 and  $L(F_2) = L(r_2)$ 

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

$$L(r_1) \setminus L(r_2) = L(F_{1,2})$$
 and  $L(r_2) \setminus L(r_1) = L(F_{2,1})$ 

holds. Hence we have

$$r_1 \doteq r_2 \iff L(F_{1,2}) = \{\} \land L(F_{2,1}) = \{\}$$

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

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

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Equivalence.ipynb implements the theory discussed in this section.

### 5.4 Limits of Regular Languages

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

#### Theorem 18 (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.  $\forall h \in \mathbb{N} : uv^h w \in L$ .

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

$$\exists n \in \mathbb{N} : \Big( n > 0 \land \forall s \in L : \big( |s| \ge n \to \exists u, v, w \in \Sigma^* : s = uvw \land v \ne \varepsilon \land |uv| \le n \land \forall h \in \mathbb{N} : uv^h w \in L \big) \Big).$$

**Proof**: As L is a regular language, there exists a deterministic  $F_{SM}$ 

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

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:

$$n := card(Q)$$
.

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

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

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

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

and since we have  $s \in L$  we conclude that  $q_m$  must be an accepting state, i.e.  $q_m \in A$ . As  $m \ge n$  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

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

we know, that even in the list

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

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

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

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

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

As k < l we have that  $v \neq \varepsilon$  and l < n implies |uv| < 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

$$q_0 \stackrel{u}{\longmapsto} q_k.$$
 (5.1)

2. Reading the string v changes the state of the  $Fsm\ F$  from the state  $q_k$  to the state  $q_l$ . As we have  $q_l=q_k$ , this implies

$$q_k \stackrel{v}{\longmapsto} q_k.$$
 (5.2)

3. Reading the string w changes the state of the  $F_{SM}$  F from the state  $q_l = q_k$  to the accepting state  $q_m$ :

$$q_k \xrightarrow{w} q_m$$
. (5.3)

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

$$q_k \stackrel{v}{\longmapsto} q_k \stackrel{v}{\longmapsto} q_k$$
, hence  $q_k \stackrel{v^2}{\longmapsto} q_k$ .

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

$$q_k \xrightarrow{\nu^h} q_k$$
 for all  $h \in \mathbb{N}$ . (5.4)

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

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

This can be condensed to

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

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

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

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

Then the language L is not regular.

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

$$s = uvw$$

such that

$$|uv| \le n$$
,  $v \ne \varepsilon$ , and  $\forall h \in \mathbb{N} : uv^h w \in L$ 

holds. Let us define the string s as

$$s := a^n b^n$$
.

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

$$\mathbf{a}^n \mathbf{b}^n = uvw$$
,  $|uv| \le n$ ,  $v \ne \varepsilon$ , and  $\forall h \in \mathbb{N} : uv^h w \in L$ .

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

$$v = \mathbf{a}^k$$
 for some  $k \in \mathbb{N}$ .

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

$$uw \in L. (5.5)$$

In order to facilitate our argument, we define the function

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

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

$$t \in L \Rightarrow count(t, "a") = count(t, "b").$$

On one hand we have:

$$\begin{array}{lll} \textit{count}(uw, \text{``a''}) & = & \textit{count}(uvw, \text{``a''}) - \textit{count}(v, \text{``a''}) \\ & = & \textit{count}(s, \text{``a''}) - \textit{count}(v, \text{``a''}) \\ & = & \textit{count}(\mathbf{a}^n \mathbf{b}^n, \text{``a''}) - \textit{count}(\mathbf{a}^k, \text{``a''}) \\ & = & n - k \\ & < & n \end{array}$$

But on the other hand we have

$$\begin{array}{lll} \textit{count}(uw, \text{``b''}) & = & \textit{count}(uvw, \text{``b''}) - \textit{count}(v, \text{``b''}) \\ & = & \textit{count}(s, \text{``b''}) - \textit{count}(v, \text{``b''}) \\ & = & \textit{count}(\mathbf{a}^n \mathbf{b}^n, \text{``b''}) - \textit{count}(\mathbf{a}^k, \text{``b''}) \\ & = & n - 0 \\ & = & n \end{array}$$

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 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.

**Remark**: 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

$$\{\binom{n}{n} \mid n \in \mathbb{N}\}$$

is not regular. Hence, regular expressions are unable to check even such simple questions as to whether the parentheses in an expression are balanced. Therefore, the concept of regular expressions is not strong enough to describe the syntax of a programming language. The next chapter introduces the notion of context-free languages. These languages are powerful enough to describe modern programming languages.

**Exercise 14**: The language  $L_{\text{square}}$  is the set of all strings of the form  $a^n$  where n is a square, we have

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

Prove that the language  $L_{\mathrm{square}}$  is not a regular language.

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

**Solution**: The proof is a proof by contradiction. We assume that  $L_{\text{square}}$  is a regular language. According to the

Pumping Lemma there exists an natural number n > 0 such that every string  $s \in L_{\text{square}}$  such that  $|s| \ge n$  can be split into three parts u, v, and w such that we have:

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

Let us define  $s := a^{n^2}$ . We have  $s \in L_{\text{square}}$  and we see that

$$|s| = |a^{n^2}| = n^2 \ge n.$$

Hence there have to be strings u, v and w such that s=uvw and u, v, and w have the properties specified above. As 'a' is the only character that occurs in s, the strings u, v, and w also contain only this character. Hence there must be natural numbers x, y, and z such that

$$u = a^x$$
,  $v = a^y$  und  $w = a^z$ 

holds. Then we have the following.

1. The partition s=uvw has the form  $a^{n^2}=a^xa^ya^z$  and hence we have

$$n^2 = x + y + z. ag{5.6}$$

2. The inequality  $|uv| \le n$  implies  $x + y \le n$ , which implies

$$y \le n. (5.7)$$

3. From the condition  $v \neq \varepsilon$  we get

$$y > 0. (5.8)$$

4. The formula  $\forall h \in \mathbb{N} : uv^h w \in L_{\text{square}}$  implies

$$\forall h \in \mathbb{N} : a^x a^{y \cdot h} a^z \in L_{\text{square}}. \tag{5.9}$$

In particular, this must hold for h=2:

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

According to the definition of  $L_{
m square}$  there is a natural number k such that

$$x + 2 \cdot y + z = k^2. ag{5.10}$$

If we add y on both sides of equation (5.6) we get

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

Because of y > 0 this implies

$$n < k. (5.11)$$

On the other hand we have

$$k^2 = x + 2 \cdot y + z$$
 according to (5.10)  
 $= x + y + z + y$   
 $\leq x + y + z + n$  according to (5.7)  
 $= n^2 + n$  according to (5.6)  
 $< n^2 + 2 \cdot n + 1$  since  $n + 1 > 0$   
 $= (n + 1)^2$ 

0

This shows that  $k^2 < (n+1)^2$  holds and therefore we have

$$k < n+1. (5.12)$$

The inequalities (5.11) and (5.12) imply

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

Since k is a natural number and n is also a natural number this is a contradiction because there is no natural number between n and n+1.

**Exercise 15**: Define  $\Sigma := \{a\}$ . Prove that the language

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

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| < n und
- 3.  $\forall h \in \mathbb{N} : uv^h w \in L$ .

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

$$a^p = uvw$$

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:

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

Für x, y und z gilt dann Folgendes:

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

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 16**: Die Sprache  $L_{\text{power}}$  beinhaltet alle Wörter der Form a $^n$  für die n eine Zweier-Potenz ist, es gilt also

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

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

## Chapter 6

# **Context-Free Languages**

In this chapter we present the notion of a *context-free language*. This concept is much more powerful than the notion of a regular language. The syntax of most modern programming languages can be described by a context-free language. Furthermore, checking whether a string is a member of a context-free language, structures the string into a recursive structure known as a parse tree. These parse trees are the basis for *understanding* the meaning of a string that is to be interpreted as a program fragment. A program that checks whether a given string is an element of a context-free language is called a parser. Usually, a parser builds a blueparse tree from a given string. Parsing is therefore the first step in an interpreter or a compiler. In this chapter, we first define the notion of context-free languages. Next, we discuss parse trees. We conclude this chapter by introducing some of the less complex algorithms that are available for parsing a string into a parse tree.

### 6.1 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 Number \Rightarrow C \in Factor.
```

2. Jede Variable ist ein Faktor:

$$V \in Variable \Rightarrow V \in Factor.$$

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 ArithExpr \Rightarrow "(" A ")" \in 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 \textit{Factor} \Rightarrow F \in \textit{Product}.
```

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

```
P \in Product \land F \in Factor \Rightarrow P"*"F \in Product \land P"/"F \in Product.
```

6. Jedes Produkt ist gleichzeitig auch ein arithmetischer Ausdruck

```
P \in Product \Rightarrow P \in 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 ArithExpr \land P \in Product \Rightarrow A"+"P \in ArithExpr \land A"-"P \in 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:

```
arithExpr 
ightharpoonup arithExpr 
ightharpoonup arithExpr 
ightharpoonup arithExpr 
ightharpoonup arithExpr 
ightharpoonup arithExpr 
ightharpoonup product 
ightharpoonup product 
ightharpoonup product 
ightharpoonup product 
ightharpoonup factor 
ightharpoonup factor 
ightharpoonup "(" arithExpr ")" factor 
ightharpoonup Variable factor 
ightharpoonup Number
```

Die Ausdrücke auf der linken Seite einer Grammatik-Regel bezeichnen wir als syntaktische Variablen oder auch als Nicht-Terminale. Alle anderen Ausdrücke werden als Terminale bezeichnet. Wir werden syntaktische Variablen üblicherweise klein schreiben, denn das ist die Konvention bei den Parser-Generatoren Anter und Ply, 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 ")" sind die 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:

- 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.

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:

```
arithExpr 
ightharpoonup arithExpr "+" product \mid arithExpr "-" product \mid product \mid product \mid product \mid product \mid product | factor \mid factor \mid mumber \mid Variable
```

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 20 (Kontextfreie Grammatik) Eine kontextfreie Grammatik G ist ein 4-Tupel

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

für welches das Folgende gilt:

1. V ist eine Menge von Namen, die wir als syntaktische Variablen oder auch Nicht-Terminale bezeichnen. In dem obigen Beispiel gilt

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

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

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

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

geschrieben. Formal steht diese Regel für das Paar

$$\langle arithExpr, [arithExpr, "+", product] \rangle$$
.

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

#### 6.1.1 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  $a \to \gamma$ . 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 G aus G Ableitungs-Schritten besteht, so schreiben wir

$$a \Rightarrow^n w$$
.

Wir geben ein Beispiel:

arithExpr ⇒ arithExpr "+" product ⇒ product "+" product

⇒ product "\*" factor "+" product

⇒ factor "\*" factor "+" product

⇒ Number "\*" factor "+" product

⇒ Number "\*" Number "+" product

⇒ Number "\*" Number "+" factor

⇒ Number "\*" Number "+" Number

Damit haben wir also gezeigt, dass

oder genauer

$$arithExpr \Rightarrow ^{8} Number "*" Number "+" Number$$

gilt. Ersetzen wir hier das Terminal 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

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

wird von der Grammatik

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

erzeugt, wobei die Regeln R wie folgt gegeben sind:

$$s \rightarrow$$
 "("  $s$  ")" |  $\varepsilon$ .

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

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

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

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

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:

$$s \Rightarrow$$
 "("  $s$  ")"  $\Rightarrow^n w$ 

Daraus folgt

$$w = "("v")" \wedge s \Rightarrow^n v.$$

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

$$w = \text{``('' } v \text{ '')''} = ((^k)^k) = (^{k+1})^{k+1} \in L.$$

**Aufgabe 17**: Wir definieren für  $w \in \Sigma^*$  und  $c \in \Sigma$  die Funktion

0

 $\Diamond$ 

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

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

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

Dann wird count(dv, c) durch eine Fall-Unterscheidung definiert:

$$\mathit{count}(dv,c) := \left\{ \begin{array}{ll} \mathit{count}(v,c) + 1 & \mathsf{falls} \ c = d; \\ \mathit{count}(v,c) & \mathsf{falls} \ c \neq d. \end{array} \right. \diamond$$

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:

$$L := \left\{ w \in \Sigma^* \mid \operatorname{count}(w, \text{``A''}) = \operatorname{count}(w, \text{``B''}) \right\}$$

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

**Exercise 18**: 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

$$w_1 = \mathtt{ABABA}$$
 and  $w_2 = \mathtt{ABBA}$ 

are both palindromes, while the string ABB is not a palindrome. The language of palindromes  $L_{\rm palindrome}$  is the set of all strings in  $\Sigma^*$  that are palindromes, i.e. we have

$$L_{\text{palindrome}} := \{ w \in \Sigma^* \mid w = w^R \}.$$

- (a) Prove that the language  $L_{
  m palindrome}$  is a context-free language.
- (b) Prove that the language  $L_{\rm palindrome}$  is not regular.

**Aufgabe 19**\*: Es sei  $\Sigma := \{$  "A", "B" $\}$ . Wir definieren die Menge L als die Menge der Strings s, die sich <u>nicht</u> 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

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

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

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

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:

$$\hat{s} := s[n+1]$$
 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:

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

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

$$x := s[1:2 \cdot k - 1]$$
 und  $y := s[2 \cdot k:2 \cdot n]$ .

Für die Längen von x und y folgt daraus

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

Dann gilt einerseits

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

und andererseits haben wir

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

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:

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

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:

$$x, y \in \Sigma^* \land |x| = 2 \cdot m + 1 \land |y| = 2 \cdot n + 1 \land \hat{x} \neq \hat{y} \to xy \in L. \tag{*}$$

**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

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

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:

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

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:

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

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:

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. 
$$L(a) = \{ w \in \Sigma^* \mid \exists k \in \mathbb{N} : |w| = 2 \cdot k - 1 \land w[k] = \text{``A''} \},$$

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

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. 
$$L(s) = \{ w \in \Sigma^* | |w| \% 2 = 1 \}$$
  
 $\cup \{ w \in \Sigma^* | \exists x, y \in \Sigma^* : w = xy \land |x| \% 2 = 1 \land |y| \% 2 = 1 \land \hat{x} \neq \hat{y} \}$   
 $= \{ w \in \Sigma^* | \neg (\exists v \in \Sigma^* : w = vv) \}$ 

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

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

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

#### 6.1.2 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 6.1 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 6.1 gezeigten Art sehr schnell zu groß werden, vereinfachen wir diese Bäume mit Hilfe der folgenden Regeln:

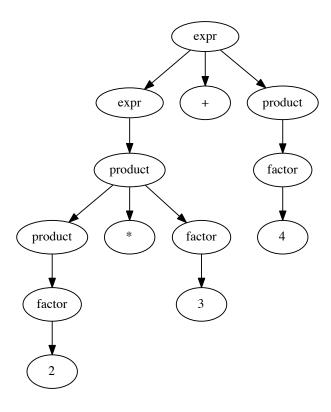


Figure 6.1: Ein Parse-Baum für den String "2\*3+4".

- 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 6.2 zeigt den abstrakten Syntax-Baum den wir erhalten, wenn wir den in Abbildung 6.1 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.

### 6.1.3 Mehrdeutige Grammatiken

Die zu Anfang des Abschnitts 6.1 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:

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

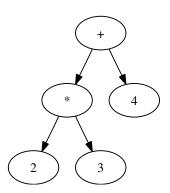


Figure 6.2: Ein abstrakter Syntax-Baum für den String "2\*3+4".

wobei die Regeln R wie folgt gegeben sind:

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

```
expr ⇒ expr "+" expr

⇒ expr "*" expr "+" expr

⇒ 2 "*" expr "+" expr

⇒ 2 "*" 3 "+" expr

⇒ 2 "*" 3 "+" 4
```

Diese Ableitung entspricht dem abstrakten Syntax-Baum, der in Abbildung 6.2 gezeigt ist. Es gibt aber noch eine andere Ableitung des Strings "2\*3+4" mit dieser Grammatik:

```
expr ⇒ expr "*" expr

⇒ expr "*" expr "+" expr

⇒ 2 "*" expr "+" expr

⇒ 2 "*" 3 "+" expr

⇒ 2 "*" 3 "+" 4
```

Diese Ableitung entspricht dem abstrakte Syntax-Baum, der in Abbildung 6.3 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  $\Sigma^*$ , bei denen die Buchstaben "A" and "B" mit der gleichen Häufigkeit auftreten, es gilt also

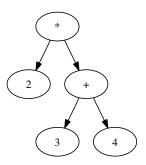


Figure 6.3: Ein anderer abstrakter Syntax-Baum für den String "2\*3+4".

$$L = \{ w \in \Sigma^* \mid count(w, \text{``A''}) = count(w, \text{``B''}) \}.$$

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:

$$s \rightarrow$$
 "A"  $s$  "B"  $s$  | "B"  $s$  "A"  $s$  |  $\varepsilon$ 

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

$$s \rightarrow$$
 "A"  $s$  "B"  $s$ 

Die Regel

$$s \rightarrow$$
 "B"  $s$  "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:

$$s \Rightarrow$$
 "A"  $s$  "B"  $s$ 

$$\Rightarrow$$
 "A" "B"  $s$ 

$$\Rightarrow$$
 "A" "B" "A"  $s$  "B"  $s$ 

$$\Rightarrow$$
 "A" "B" "A" "B"  $s$ 

$$\Rightarrow$$
 "A" "B" "A" "B"

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

Abbildung 6.4 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".

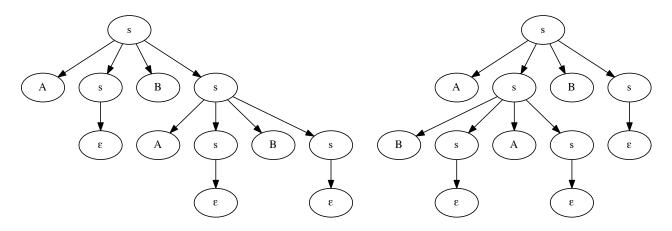


Figure 6.4: Zwei strukturell verschiedene Parse-Bäume für den String "ABAB".

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:

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

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

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

$$\begin{split} L(x) &= \left\{ \omega \in L \mid \forall \sigma \preceq \omega : \mathit{count}(\sigma, \text{``B''}) \leq \mathit{count}(\sigma, \text{``A''}) \right\} \quad \text{unc} \\ L(y) &= \left\{ \omega \in L \mid \forall \sigma \preceq \omega : \mathit{count}(\sigma, \text{``A''}) \leq \mathit{count}(\sigma, \text{``B''}) \right\}. \end{split}$$

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

"A" 
$$w$$
 "B"  $\min w \in L(x)$ 

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

"B" 
$$w$$
 "A"  $\min w \in L(y)$ 

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

$$s \rightarrow u s$$

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  $\Sigma = \{$  "A", "B", "C", "D" $\}$  die Sprache

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

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].

### 6.2 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

$$a \rightarrow X_1 X_2 \cdots X_n$$

zu parsen, versuchen wir, zunächst ein  $X_1$  zu parsen. Dabei zerlegen wir den String w in die Form  $w=w_1r_1$  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  $r_1=w_2r_2$  mit  $w_2\in L(X_2)$  gilt. Setzen wir diesen Prozess fort, so haben wir zum Schluss den String w in

$$w = w_1 w_2 \cdots w_n$$
 mit  $w_i \in L(X_i)$  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

$$a \rightarrow a \beta$$

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 zwei Möglichkeiten, um mit diesem Problem umzugehen:

- 1. Wir können die Grammatik so umschreiben, dass sie danach nicht mehr links-rekursiv ist.
- 2. Eine einfachere Methode besteht darin, denn Begriff der kontextfreien Grammatik zu erweitern. Wir werden 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 die beiden oben genannten Verfahren anhand der Grammatik für arithmetische Ausdrücke ausführlich diskutieren.

#### 6.2.1 Umschreiben der Grammatik

In der folgenden Grammatik ist a eine syntaktische Variable und die griechischen Buchstaben  $\beta$  und  $\gamma$  stehen für irgendwelche Strings, die aus syntaktischen Variablen und Tokens bestehen. Wird die syntaktische Variable a durch die beiden Regeln

$$a \rightarrow a\beta$$

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:

$$L(a) = \{ w \in \Sigma^* \mid \exists n \in \mathbb{N} : \gamma \beta^n \Rightarrow^* w \}.$$

Diese Sprache kann offenbar auch durch die folgenden Regeln für a beschrieben werden:

$$\begin{array}{ccc} a & \to & \gamma b \\ b & \to & \beta b \\ & \mid & \varepsilon \end{array}$$

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

$$L(b) = \{ w \in \Sigma \mid \exists n \in \mathbb{N} : \beta^n \Rightarrow w \}.$$

Damit ist klar, dass auch mit der oben angegeben Grammatik

$$L(a) = \left\{ w \in \Sigma^* \mid \exists n \in \mathbb{N} : \gamma \beta^n \Rightarrow^* w \right\}$$

gilt. Um die Links-Rekursion aus der in Abbildung 6.5 auf Seite 71 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

$$\begin{array}{cccc}
a & \rightarrow & a\beta_1 \\
& | & a\beta_2 \\
& \vdots & \vdots \\
& | & a\beta_k \\
& | & \gamma_1 \\
& \vdots & \vdots \\
& | & \gamma_l
\end{array}$$

beschrieben wird. Wir können diesen Fall durch Einführung zweier Hilfs-Variablen b und c auf den ersten Fall zurückführen:

$$a \rightarrow ab \mid c$$

$$b \rightarrow \beta_1 \mid \dots \mid \beta_k$$

$$c \rightarrow \gamma_1 \mid \dots \mid \gamma_l$$

Dann können wir die Grammatik umschreiben, indem wir eine neue Hilfs-Variable, nennen wir sie l für Liste, einführen und erhalten

$$\begin{array}{ccc} a & \rightarrow & c \ l \\ \\ l & \rightarrow & b \ l \mid \varepsilon. \end{array}$$

Die Hilfs-Variablen b und c können nun wieder eliminiert werden und dann bekommen wir die folgende Grammatik:

$$a \rightarrow \gamma_1 l \mid \gamma_2 l \mid \cdots \mid \gamma_l l$$
  
$$l \rightarrow \beta_1 l \mid \beta_2 l \mid \cdots \mid \beta_k l \mid \varepsilon$$

Wenden wir dieses Verfahren auf die in Abbildung 6.5 gezeigte Grammatik für arithmetische Ausdrücke an, so erhalten wir die in Abbildung 6.6 gezeigte Grammatik.

```
expr 
ightarrow expr "+" product \ | expr "-" product \ | product \ | product \ | product "*" factor \ | product "/" factor \ | factor \ | factor \ | Number
```

Figure 6.5: Links-rekursive Grammatik für arithmetische Ausdrücke.

Figure 6.6: Grammatik für arithmetische Ausdrücke ohne Links-Rekursion.

 $\Diamond$ 

Die Variablen exprRest und productRest können wie folgt interpretiert werden:

1. exprRest beschreibt eine Liste der Form

```
\textit{op product} \; \cdots \; \textit{op product,} wobei \textit{op} \in \{ \text{ "+" }, \text{ "-" } \} \; \text{gilt.}
```

2. productRest beschreibt eine Liste der Form

```
\label{eq:opfactor} \textit{op factor} \; \cdots \; \textit{op factor}, wobei \textit{op} \in \{ \text{ "*" }, \text{ "/" } \} \; \text{gilt}.
```

#### Aufgabe 20:

(a) Die folgende Grammatik beschreibt reguläre Ausdrücke:

Diese Grammatik verwendet nur die syntaktische Variable {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.

#### 6.2.2 Implementing a Top Down Parser in Python

Now we are ready to implement a parser for recognizing arithmetic expressions. We will use the grammar that is shown in Figure 6.6 on page 71. Before we can implement the parser, we need a scanner. We will use a hand-coded scanner that is shown in Figure 6.7 on page 73. The function tokenize implemented in this scanner receives a string s as argument and returns a list of tokens. The string s is supposed to represent an arithmetical expression. In order to understand the implementation, you need to know the following:

- (a) We need to set the flag re.VERBOSE in our call of the function findall below because otherwise we are not able to format the regular expression lexSpec the way we have done it.
- (b) The regular expression lexSpec contains 5 parenthesized groups. Therefore, findall returns a list of 5-tuples where the 5 components correspond to the 5 groups of the regular expression. As the 5 groups are non-overlapping, exactly one of the 5 components will be a non-empty string.

Figure 6.8 on page 74 shows an implementation of a recursive descent parser in PYTHON.

(a) The main function is the function parse. This function takes a string s representing an arithmetic expression. This string is tokenized using the function tokenize. The function tokenize turns a string into a list of tokens. For example, the expression

```
import re
2
    def tokenize(s):
3
         lexSpec = r'''([ \t]+) | # blanks and tabs
4
                       ([1-9][0-9]*|0) | # number
                                      | # parentheses
6
                       ([-+*/])
                                       | # arithmetical operators
                       (.)
                                           # unrecognized character
         tokenList = re.findall(lexSpec, s, re.VERBOSE)
10
11
         for ws, number, parenthesis, operator, error in tokenList:
12
                           # skip blanks and tabs
13
14
                 pass
             if number:
15
                result += [ number ]
             if parenthesis:
17
                 result += [ parenthesis ]
18
             if operator:
19
                 result += [ operator ]
             if error:
21
                 result += [ f'ERROR({error})']
22
        return result
23
```

Figure 6.7: A scanner for arithmetic expressions.

```
tokenize('(1 + 2) * 3')

returns the result

['(', 1, '+', 2, ')', '*', 3].
```

This list of tokens is then parsed by the function parseExpr. That function returns a pair:

- (a) The first component is the value of the arithmetic expression.
- (b) The second component is the list of those tokens that have not been consumed when parsing the expression. Of course, on a successful parse this list should be empty.
- (b) The function parseExpr implements the grammar rule

```
expr \rightarrow product \ exprRest.
```

It takes a token list TL as input. It will return a pair of the form

```
(v, Rest),
```

where v is the value of the arithmetic expression that has been parsed, while Rest is the list of the remaining tokens. For example, the expression

```
parseExpr(['(', 1, '+', 2, ')', '*', 3, ')', '*', 2])
```

returns the result

```
[9, [')', '*', 2]].
```

Here, the part ['(', 1, '+', 2, ')', '\*', 3] has been parsed and evaluated as the number 9 and [')', '\*', 2] is the list of tokens that have not yet been processed.

```
def parse(s):
2
          TL
                        = tokenize(s)
          result, Rest = parseExpr(TL)
          assert Rest == [], f'Parse Error: could not parse {TL}'
          return result
     def parseExpr(TL):
         product, Rest = parseProduct(TL)
         return parseExprRest(product, Rest)
10
    def parseExprRest(sum, TL):
11
         if TL == []:
12
             return sum, []
13
         elif TL[0] == '+':
14
             product, Rest = parseProduct(TL[1:])
15
             return parseExprRest(sum + product, Rest)
         elif TL[0] == '-':
17
             product, Rest = parseProduct(TL[1:])
             return parseExprRest(sum - product, Rest)
19
         else:
             return sum, TL
21
22
     def parseProduct(TL):
23
24
         factor, Rest = parseFactor(TL)
25
         return parseProductRest(factor, Rest)
26
     def parseProductRest(product, TL):
27
         if TL == []:
28
             return product, []
29
30
         elif TL[0] == '*':
             factor, Rest = parseFactor(TL[1:])
31
             return parseProductRest(product * factor, Rest)
32
         elif TL[0] == '/':
33
             factor, Rest = parseFactor(TL[1:])
34
             return parseProductRest(product / factor, Rest)
35
36
             return product, TL
37
38
     def parseFactor(TL):
         if TL[0] == '(':
40
             expr, Rest = parseExpr(TL[1:])
             assert Rest[0] == ')', 'Parse Error: expected ")"'
42
             return expr, Rest[1:]
         else:
44
             return int(TL[0]), TL[1:]
45
```

Figure 6.8: A top down parser for arithmetic expressions.

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

Here, the function parseProduct will return the result

$$(2, ['+', 3]),$$

where 2 is the result of parsing and evaluating the token list [1, '\*', 2], while ['+', 3] is the part of the input token list that is not used by parseProduct. Next, the list ['+', 3] needs to be parsed as the rest of an expression and then 3 needs to be added to 2.

(c) The function parseExprRest takes a number and a list of tokens. It implements the following grammar rules:

exprRest 
$$\rightarrow$$
 "+" product exprRest  $\mid$  "-" product exprRest  $\mid$   $\varepsilon$ 

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

is used. Therefore, in that case we have not consumed any tokens and hence the input argument is already the result.

(d) The function parseProduct implements the rule

$$product \rightarrow factor \ exprRest.$$

The implementation is similar to the implementation of parseExpr.

(e) The function parseProductRest implements the rules

The implementation is similar to the implementation of *parseExprRest*.

(f) The function parseFactor implements the rules

Therefore, we first check whether the next token is "(" because in that case, we have to use the first grammar rule, otherwise we use the second.

The parser shown in Figure 6.8 does not contain any error handling. Appropriate error handling will be discussed once we have covered the theory of top-down parsing.

### 6.2.3 Implementing a Recursive Descent Parser that Uses an EBNF Grammar

The previous solution to parse an arithmetical expression was not completely satisfying: The reason is that we did not really fix the problem of left recursion but rather cured the symptoms. The underlying reason for left recursion is that context free grammars are not that convenient to describe the structure of programming languages since a description of this structure needs both recursion and iteration, but context-free grammars provide no direct way to describe iteration. Rather, they simulate iteration via recursion. Let us therefore extend the power of context-free languages slightly by admitting regular expression on the right hand side of grammar rules. These new type of grammars are known as *extended Backus Naur form* grammars, which is abbreviated as EBNF grammars. An EBNF grammar admits the operators "\*", "?", and "+" on the right hand side of a grammar rule. The meaning of these operators is the same as when these operators are used in the regular expressions of the programming language *Python*. Furthermore, the right hand side of a grammar rule can be structured using parentheses.

Figure 6.9: EBNF grammar for arithmetical expressions.

It can be shown that the languages described by  $\rm EBNF$  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 <u>convenient</u> to describe a language using an  $\rm EBNF$  grammar rather than using a context-free grammar. Figure 6.9 displays an  $\rm EBNF$  grammar for arithmetical expressions.

Obviously, the grammar in Figure 6.9 is more concise than the context-free grammar shown in Figure 6.6 on page 71. For example, the first rule clearly expresses that an arithmetical expression is a list of products that are separated by the operators "+" and "-".

Figure 6.10 shows a recursive descent parser that implements this grammar.

- 1. The function parseExpr recognizes a product in line 2. The value of this product is stored in the variable result together with the list Rest of those tokens that have not been consumed yet. If the list Rest is not empty and the first token in this list is either the operator "+" or the operator "-", then the function parseExpr tries to recognize more products. These are added to or subtracted from the result computed so far in line 7 or 9. If there are no more products to be parsed, the while loop terminates and the function returns the result together with the list of the remaining tokens Rest.
- 2. The function parseProduct recognizes a factor in line 13. The value of this factor is stored in the variable result together with the list Rest of those tokens that have not been consumed yet. If the list Rest is not empty and the first token in this list is either the operator "\*" or the operator "/", then the function parseProduct tries to recognize more factors. The result computed so far is multiplied with or divided by these factors in line 18 or 20. If there are no more products to be parsed, the while loop terminates and the function returns the result together with the list Rest of tokens that have not been consumed.
- 3. The function parseFactor recognizes a factor. This is either an expression in parentheses or a number.
  - If the first token is a an opening parenthesis, the function tries to parse an expression next. This expression has to be followed by a closing parenthesis. The tokens following this closing parenthesis are not consumed but rather are returned together with the result of evaluating the expression.
  - If the first token is a number, this number is returned together with the list of all those tokens that have not been consumed.

```
def parseExpr(TL):
         result, Rest = parseProduct(TL)
2
         while len(Rest) > 1 and Rest[0] in {'+', '-'}:
             operator = Rest[0]
             arg, Rest = parseProduct(Rest[1:])
             if operator == '+':
                 result += arg
                                # operator == '-':
             else:
                 result -= arg
         return result, Rest
10
11
    def parseProduct(TL):
12
         result, Rest = parseFactor(TL)
13
         while len(Rest) > 1 and Rest[0] in {'*', '/'}:
             operator = Rest[0]
15
             arg, Rest = parseFactor(Rest[1:])
             if operator == '*':
17
                 result *= arg
                                # operator == '/':
             else:
19
                 result /= arg
         return result, Rest
21
    def parseFactor(TL):
23
         if TL[0] == '(':
             expr, Rest = parseExpr(TL[1:])
25
             assert Rest[0] == ')', "ERROR: ')' expected, got {Rest[0]}"
26
             return expr, Rest[1:]
27
         else:
28
             assert isinstance(TL[0], int), "ERROR: Number expected, got {TL[0]}"
29
             return TL[0], TL[1:]
30
```

Figure 6.10: A recursive descent parser for the grammar in Figure 6.9.

**Historical Notes** The language ALGOL [Bac59, NBB+60] was the first programming language with a syntax that was based on an EBNF grammar.

## Chapter 7

# Introducing ANTLR

If your task is to implement a parser, it is best to use one of the tools that is available for this purpose. The wikipedia page Comparison of parser generators shows the large number of parser generators that are available. A parser generator is a program that takes a grammar as its input and generates a parser that can parse according to this grammar. In my opinion, the parser generator that is both most mature<sup>1</sup> and most powerful is Antle [Par12, PHF14]. The name is short for <u>another tool for language recognition</u><sup>2</sup>. Antle can be downloaded at

In this lecture we will use Antle version 4.8, which was released in January 2020. The tool Antle is written in Java and therefore its main target language is Java. However, Antle can also be used to generate parsers for the programming languages C++, C#, Python, JavaScript, Go, and Swift. We will introduce Antle via some examples that demonstrate the most important features of this tool. For a discussion of all the features offered by Antle I recommend the book "The Definitive Antle Reference" by Terrence Parr [Parl2]. However, this book only covers the generation of Java parsers.

## 7.1 A Parser for Arithmetic Expressions

We start with a parser for arithmetic expressions. This parser will do nothing fancy, it will just check whether a given string has the format of an arithmetic expression and adheres to the grammar that is shown in Figure 6.5 on page 71. If we use Antermode we can implement this grammar as shown in Figure 7.1. We discuss the implementation line by line.

- 1. In line 1 the keyword grammar specifies the name of the grammar. In this case the grammar is called Expr. This grammar has to be stored in a file with the name "Expr.g4". The rule is that the file name has to be the same name as the name of the grammar and the file extension has to be ".g4".
- 2. Line 3 gives the grammar rule for the non-terminal start. In the last chapter we would have used the notation

$$start \rightarrow expr$$

to specify this rule. With Anter R the left and right part of a grammar rule are separated by a colon. Anter R terminates every grammar rule with the character ";".

- 3. Line 6–9 give the grammar rules for the non-terminal *expr*. Note that we have to enclose the terminals "+" and "-" in single quotes. The different alternatives for the non-terminal expr are separated by the character "|".
- 4. Similarly, the lines 11–14 and 16–18 show the grammar rules for the non-terminals *product* and *factor*.

 $<sup>^{1}</sup>$  The first version of  $\rm Antlr$  became available in 1992 and  $\rm Antlr$  4.0 was released in 2012.

<sup>&</sup>lt;sup>2</sup> The name ANTLR can also be interpreted as an acronym for  $\frac{1}{LR}$ , where "LR" is short for  $\frac{1}{LR}$  parser. LR parsers are a kind of bottom-up parsers. We will discuss these parsers later in chapter  $\frac{1}{1}$ .

```
grammar Expr;
2
3
     start
              : expr
              : expr '+' product
     expr
              | expr '-' product
              product
10
     product : product '*' factor
11
              | product '/' factor
12
              | factor
13
14
15
     factor : '(' expr ')'
16
              NUMBER
17
18
19
     NUMBER : '0' | [1-9] [0-9] *;
20
              : [ \t\n\r] -> skip;
21
```

Figure 7.1: ANTLR grammar for arithmetical expressions.

- 5. With ANTLR the grammar and the specification of the tokens can be given in the same file. In order to be able to distinguish terminals and non-terminals, terminals have to begin with an upper case letter, while non-terminals start with a lower case letter. Therefore, "NUMBER" is the name of a terminal.
- 6. In line 20 the lexical specification of the non-terminal NUMBER is given by a regular expression. The regular expression

```
'0'|[1-9][0-9]*;
```

describes a sequence of digits. This sequence can only start with the digit "0" when "0" is not followed by any other digit.

Notice that we have to enclose the first occurrence of "0" in single quotes. On the other hand, we must not put the digits occurring in the square brackets "[" and "]" in quotes, since these occur inside a range and characters inside a range must never be quoted.

7. Line 21 defines the terminal WS, where the name WS is short for *white space*. This terminal specifies a single character that is either a blank, a tabulator, a line break, or a carriage return. The lexical specification of the terminal WS is followed by the operator "->" which in turn is followed by a semantic action. The semantic action "skip", which is executed once a white space character has been recognized, simply discards the white space character. Therefore, the net effect of line 21 is to discard all white space characters.

In most programming languages<sup>4</sup>, white space has no purpose other than that of separating tokens. Therefore, most  $A_{NTLR}$  parsers will have a scanner rule that is similar to the rule shown in line 21.

To conclude, the lines 3–18 specify the grammar, while the lines 20–21 specify the lexical structure. If the grammar that is shown in Figure 7.1 is stored in the file Expr.g4 we can generate a parser by using the following command:

<sup>&</sup>lt;sup>3</sup>It is a convention that the names of non-terminals consist of only upper case letters, but this is not required.

<sup>&</sup>lt;sup>4</sup> Unfortunately, *Python* is an exception to this rule. Therefore, parsing *Python* is considerably harder than parsing languages like C or *Java*.

```
java -jar /usr/local/lib/antlr-4.8-complete.jar -Dlanguage=Python3 Expr.g4
```

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:

ExprParser.py
 This file contains the parser.

ExprLexer.pyThis is the scanner.

3. ANTLR generates some more files. However, these are not relevant for us.

In order to run the parser we need a driver program. Figure 7.2 shows such a program.

```
from ExprLexer import ExprLexer
from ExprParser import ExprParser

import antlr4

def parse_string(string):
    inputStream = antlr4.InputStream(string)
    lexer = ExprLexer(inputStream)
    tokenStream = antlr4.CommonTokenStream(lexer)
    parser = ExprParser(tokenStream)
    parser.start()
```

Figure 7.2: Driver program for the parser generated by ANTLR.

- 1. In Line 1 and 2 we import the scanner and the parser that has been generated by  ${
  m ANTLR}.$
- 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 shown in Figure 7.3.

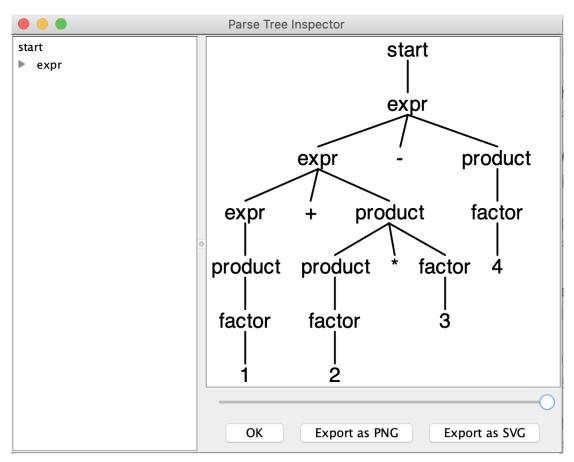


Figure 7.3: Parse tree for the string "1 + 2 \* 3 - 4".

## 7.2 Evaluation of Arithmetical Expressions

The last example isn't very exciting as the arithmetical expressions that the parser reads are not evaluated. In this section we show how arithmetical expressions can be evaluated using Antering Anterior Ante

- Firstly, we present a grammar for a small symbolic calculator.
   Here, the attribute symbolic means that the calculator supports the use of variables.
- 2. Secondly, we show how this grammar can be extended with actions so that the expressions can be evaluated.

Figure 7.4 presents the grammar. In comparison with grammar for arithmetical expressions that was shown in Figure 7.1 there are the following changes:

- 1. The start-symbol start now recognizes a list of statements. Note that ANTLR provides the postfix operator "+" to specify non-empty sequences. The postfix operators "\*" and "?" are also supported. They have the same semantics as in regular expressions.
- 2. A statement is either an assignment or an expression.

```
grammar Program;
     start: statement+ ;
     statement
         : IDENTIFIER ':=' expr ';'
          | expr ';'
     expr: expr '+' product
10
          | expr '-' product
11
          | product
12
13
14
     product
15
          : product '*' factor
          | product '/' factor
17
          | factor
19
20
21
          : 'sqrt' '(' expr ')'
          | '(' expr ')'
23
          | FLOAT
          | IDENTIFIER
25
26
27
28
     IDENTIFIER: [a-zA-Z][a-zA-Z0-9_]*;
29
                : '0'([.][0-9]+)?
30
                | [1-9][0-9]*([.][0-9]+)?
31
32
     WS
                : [ \t\n\r] -> skip;
33
```

Figure 7.4: A grammar for a symbolic calculator.

- 3. The rules for expressions and products are the same as previously.
- 4. The rule for the non-terminal factor has changed.
  - (a) Expressions are now allowed to contain calls of the function sqrt, which is supposed to compute the square root of a given number.
  - (b) We can still put expressions in parenthesis.
  - (c) Instead of integers the grammar now supports floating point numbers. These are recognized by the terminal FLOAT.
  - (d) Expressions can also contain variables. These are recognized by the terminal IDENTIFIER.
- 5. The terminal IDENTIFIER recognizes variable names. A variable name starts with a letter from the Latin alphabet, i.e. a character from the range [a-z]. This letter can be either upper or lower case. The remaining characters of a variable name can be either letters from the Latin alphabet, digits, or the underscore.

6. The terminal FLOAT recognizes floating point numbers, i.e. numbers that have an optional fractional part like 1.23. Note that the dot "." has to be enclosed in square brackets because otherwise it would match any character that is different from a newline.

A parser for this grammar is able to parse strings like the following:

```
x := 2.3 * 3.4; y := sqrt(2 * x); z := x * x + y * y; sqrt(z);
```

Our next task is to develop an interpreter for the language specified by the grammar shown in Figure 7.4. Figure 7.5 shows how an interpreter can be implemented.

```
grammar Calculator;
     @header {
3
     import math
4
     }
6
     start: statement+ ;
     statement
         : IDENTIFIER ':=' expr ';' {self.Values[$IDENTIFIER.text] = $expr.result}
10
         expr';'
                                     {print($expr.result)
11
12
     expr returns[result]
14
         : e=expr '+' p=product {$result = $e.result + $p.result}
         | e=expr '-' p=product {$result = $e.result - $p.result}
16
                                 {$result = $p.result
         | p=product
17
18
19
     product returns[result]
20
         : p=product '*' f=factor {$result = $p.result * $f.result}
21
         | p=product '/' f=factor {$result = $p.result / $f.result}
22
                                   {$result = $f.result
23
24
25
     factor returns[result]
         : '(' expr ')'
                                {\$result = \$expr.result
                                                                           }
27
         | FLOAT
                                {\$result = float(\$FLOAT.text)
                                                                           }
         IDENTIFIER
                                {\$result = self.Values[\$IDENTIFIER.text]}
29
         | 'sqrt' '(' expr ')' {$result = math.sqrt($expr.result)
31
32
     IDENTIFIER: [a-zA-Z][a-zA-Z0-9_]*;
33
     FLOAT : '0'([.][0-9]+)?
34
               | [1-9][0-9]*([.][0-9]+)?;
35
     WS
               : [ \t\n\r] -> skip;
36
```

Figure 7.5: An interpreter for evaluating arithmetical expressions.

1. As we have to use the function sqrt that is located in the module math we need to import this module into our parser. This is achieved by the header declaration shown in line 3-5. The header declaration is started

with the keyword header that is followed by an opening brace. It ends at the matching closing brace. ANTLR puts all the code that is between the braces at the beginning of the generated parser.

Note that we have to be careful to **not** indent the code. The reason is that *Python* is very picky about indentation.

2. If the parser recognizes an assignment of the form

IDENTIFIER := 
$$expr$$
;

it evaluates the expression *expr* and then stores the result of this evaluation in the dictionary Values. In order to do this, the grammar rule is followed by some action code that is enclosed in curly braces. This code will be executed once the parser has recognized the assignment.

The dictionary Values that is used to store the values assigned to variable names is a member of the parser object that is generated by ANTLR. We can refer to this object via the variable self. The value that is computed for the expression *expr* is available as the member \$expr.result. The fact that this member has the name result is due to the returns-specification in line 14.

- 3. Line 11 deals with the case where the parser has seen an expression followed by a semicolon. In this case, the result of the evaluation of the expression is printed.
- 4. The returns-declaration in line 14 specifies that when the parser reads an expression, i.e. a string that has the syntactical form specified by the grammar rule for expr it will return the result of this evaluation in the member variable result.
- 5. Line 15 deals with the case that the parser has found a sum of the form

In order to evaluate this expression, the parser has to compute the sum of the *expr* and the *product*. The notation "e=expr" assigns the result of evaluating *expr* to the variable e and "p=product" assigns the result of evaluating *product* to the variable p. The action code adds these variables and assigns the sum to the variable result. Note that all these variable names have to prefixed by a dollar symbol.

- 6. Line 29 shows how a string representing a floating point number is converted into a floating point number. The expression \$FLOAT.text references the string that is matched by the regular expression FLOAT defined in line 34–35.
- 7. Similarly, in line 29 the expression \$IDENTIFIER.text references the string that is matched by the regular expression IDENTIFIER defined in line 33. This string is then used as a key in the dictionary Values that stores the values of the variables.

Next, we need a driver program to call the parser that ANTLR generates from the grammar shown in Figure 7.5. Figure 7.6 shows how we can utilize the parser and scanner that is generated by ANTLR.

- (a) Line 6 creates a parser. Since the constructor CalculatorParser is called with the argument None, this parser is not yet connected to a scanner.
- (b) Line 7 creates and sets the member variable Values for this parser. This member variable is a dictionary. This dictionary associates variables with their values. Initially, this dictionary is empty.
- (c) Line 8 reads a line of input. As long as there still is input, the while loop in line 9 will process this input.
- (d) Line 10 transforms the string that has been read into a stream that is suitable for ANTLR.
- (e) Line 11 creates a scanner for this input stream.
- (f) Line 12 transforms this input stream into a token stream via the previously generated scanner.
- (g) Line 13 connects the token stream to the parser.
- (h) Line 14 starts the parser. The parser will now consume and process the given line of input.

```
from CalculatorLexer import CalculatorLexer
    from CalculatorParser import CalculatorParser
    import antlr4
    def main():
                       = CalculatorParser(None)
        parser
        parser.Values = {}
                       = input('> ')
        line
        while line != '':
             input_stream = antlr4.InputStream(line)
10
                           = CalculatorLexer(input_stream)
11
             token_stream = antlr4.CommonTokenStream(lexer)
12
             parser.setInputStream(token_stream)
13
             parser.start()
             line = input('> ')
15
        return parser. Values
```

Figure 7.6: A program to utilize the generated parser.

- (i) Line 15 reads the next line of input. If a non-empty line is read, the while loop proceeds.
- (j) When there is no more input, line 16 returns the dictionary associating variables with values.

## 7.3 Generating Abstract Syntax Trees with Antlr

The evaluation of arithmetical expressions was relatively easy as it is possible to evaluate an arithmetical expression directly via semantic actions that are embedded in the grammar. If the problem is more complex than the evaluation of an expression it is usually easier to first generate an abstract syntax tree and then use the syntax tree to solve the problem at hand. We demonstrate this approach using the problem of symbolic differentiation. For example, if the task is to find the derivative of the expression  $x \cdot \ln(x)$  with respect to x, then the product rule tells us that

$$\frac{\mathrm{d}}{\mathrm{d}x}(x \cdot \ln(x)) = 1 \cdot \ln(x) + x \cdot \frac{1}{x}.$$

As the arithmetical expressions that we want to differentiate contain the function symbols for the natural logarithm and for exponentiation in addition to the four arithmetical operators we have to modify the grammar given in the last section. Figure 7.7 shows the grammar that we are going to use for symbolic differentiation.

### 7.3.1 Implementing the Parser

Figure 7.8 shows how the grammar from Figure 7.7 is implemented with ANTLR.

- 1. The return specification "returns [result]" in line 3 specifies that the expression object that is generated when the parser parses the non-terminal expr has a member variable with the name result. When referring to this variable inside a semantic action we have to prefix the variable name with the dollar symbol as shown below.
- 2. Line 4 recognizes a sum of the form

$$e + p$$

where e is an expression and p is a product. Hence the parser has to recognize an expression e, followed by the symbol "+" followed by a product p. The abstract syntax tree when reading e is stored in the variable

```
expr "+"
                          product
               expr "-"
                          product
              product
product \rightarrow
              product "*" factor
              product "/" factor
product
              factor
               "(" expr ")"
  factor \rightarrow
               "exp" "(" expr ")"
               "log" "(" expr ")"
               Variable
               Number
```

Figure 7.7: A grammar for arithmetical expressions with exponential function and logarithm.

```
grammar Differentiator;
2
    expr returns [result]
         : e=expr '+' p=product {$result = ('+', $e.result, $p.result)}
         | e=expr '-' p=product {$result = ('-', $e.result, $p.result)}
         | p=product
                                {$result = $p.result
    product returns [result]
         : p=product '*' f=factor {$result = ('*', $p.result, $f.result)}
         | p=product '/' f=factor {$result = ('/', $p.result, $f.result)}
11
                                  {$result = $f.result
         | f=factor
12
13
    factor returns [result]
15
         : '(' e=expr ')'
                                {$result = $e.result;
         | 'exp' '(' e=expr ')' {$result = ('exp', $e.result)}
17
         | 'ln' '(' e=expr ')' {$result = ('ln' , $e.result)}
         | v=VAR
                                {$result = $v.text
19
         n=NUMBER
                                {\$result = int(\$n.text)
                                                              }
21
22
           : [a-zA-Z][a-zA-Z0-9]*;
23
    NUMBER : '0' | [1-9] [0-9]*;
24
    WS
            : [ \t\n\r] -> skip;
```

Figure 7.8: ANTLR implementation of the grammar form Figure 7.7.

\$e.result, while the syntax tree for the product is stored in the variable \$p.result. To build a syntax tree

for the sum e+p we create the triple

```
('+', $e.result, $p.result)
```

and assign this triple to the variable \$result.

- 3. The remaining grammar rules work in a similar way.
- 4. In line 23 we can get the actual text that is matched by the terminal VAR by writing \$v.text.
- 5. In line 24 we have to convert the string recognized by the terminal NUMBER into an integer by calling the function int.

Our second task is to implement symbolic differentiation. As we have discussed this topic already in our lecture on logic, we confine ourselves with presenting the function diff that is shown in Figure 7.9. This function takes a nested tuple representing the abstract syntax tree of an arithmetic expression and computes the derivative with respect to variable x.

```
def diff(e):
         if isinstance(e, int):
             return '0'
3
         if e[0] == '+':
             f , g = e[1:]
             fs, gs = diff(f), diff(g)
             return ('+', fs, gs)
         if e[0] == '-':
             f , g = e[1:]
             fs, gs = diff(f), diff(g)
10
             return ('-', fs, gs)
11
         if e[0] == '*':
12
             f, g = e[1:]
13
             fs, gs = diff(f), diff(g)
14
             return ('+', ('*', fs, g), ('*', f, gs))
15
         if e[0] == '/':
             f, g = e[1:]
17
             fs, gs = diff(f), diff(g)
18
             return ('/', ('-', ('*', fs, g), ('*', f, gs)), ('*', g, g))
19
         if e[0] == 'ln':
             f = e[1]
21
             fs = diff(f)
22
             return ('/', fs, f)
23
         if e[0] == 'exp':
             f = e[1]
25
             fs = diff(f)
26
             return ('*', fs, e)
         if e == 'x':
             return '1'
29
         return '0'
```

Figure 7.9: A function to compute the derivative of a given expression.

Finally, Figure 7.10 shows how the parser can be invoked. Invoking the parser in line 10 creates an abstract syntax tree that is stored in the variable context.result in line 10. This abstract syntax tree is then used as input to the function diff.

```
def main():
    parser = DifferentiatorParser(None)
    line = input('> ')
    while line != '':
        input_stream = antlr4.InputStream(line)
        lexer = DifferentiatorLexer(input_stream)
        token_stream = antlr4.CommonTokenStream(lexer)
    parser.setInputStream(token_stream)
    term = parser.expr()
    d = diff(term.result)
    print(d)
    line = input('> ')
```

Figure 7.10: A driver program for the grammar shown in Figure 7.8

#### Exercise 21: 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 H<sub>TML</sub>.

#### Remark:

(a) The directory

#### Exercises/Grammar2HTML

contains the notebook Grammar-2-HTML.ipynb that contains a number of function that can be used to transform an abstract syntax tree of a grammar into HTML.

(b) ANTLR provides a negation operator that is written as "~". This operator is handy when matching quoted strings. For example, the token definition

```
STRING : '"' ~ ('"') + '"';
```

can be used to match strings that are enclosed in double quotes.

(c) For historical reasons, ANTLR treats the string "rule" as a keyword. Therefore, it is not possible to have a syntactical variable that is called "rule".

# Chapter 8

# Interpreter

```
grammar Pure;
    program : statement+
    statement: VAR ':=' expr ';'
              | VAR ':=' 'read' '(' ')' ';'
               'print' '(' expr ')' ';'
                       '(' boolExpr ')' '{' statement* '}'
              | 'while' '(' boolExpr ')' '{' statement* '}'
10
    boolExpr : expr '==' expr
11
              | expr '<' expr
12
13
    expr
             : expr '+' product
14
              | expr '-' product
              | product
16
             : product '*' factor
    product
18
              | product '/' factor
19
              | factor
20
21
             : '(' expr ')'
    factor
22
              | VAR
23
              | NUMBER
24
25
    VAR
              : [a-zA-Z][a-zA-Z_0-9]*;
              : '0'|[1-9][0-9]*;
27
    MULTI_COMMENT : '/*' .*? '*/' -> skip;
    LINE_COMMENT : '//' ~('\n')* -> skip;
29
    WS
                   : [ \t\n\r]
                                    -> skip;
```

Figure 8.1: 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 8.1 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 8.2 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.

```
n := read();
s := 0;
i := 0;
while (i < n * n) {
    i := i + 1;
    s := s + i;
}
print(s);</pre>
```

Figure 8.2: Ein Programm zur Berechnung der Summe  $\sum_{i=0}^{n^2} i$ .

Ähnlich wie bei unserem Programm zum symbolischen Differenzieren werden wir die einzelnen Befehle als geschachtelte Listen darstellen.

Abbildung 8.3 zeigt die Implementierung des Parsers mit dem Werkzeug ANTLR.

- 1. Das Start-Symbol der Grammatik ist die Variable program. Beim Parsen dieser Variable gibt der Parser eine Liste von Befehlen in der Variable stmnt\_list zurück. Dazu initialisieren wir mit Hilfe des Schlüsselwortes "@init" die Variable SL als leere Liste. Anschließend wird jeder Befehl, der erkannt wird, an diese Liste angehängt. Schließlich wird diese Liste in der Attribut-Variable stmnt\_list gespeichert.
- 2. Die syntaktische Variable statement beschreibt die verschiedenen Befehle, die in unserer einfachen Sprache unterstützt werden.
  - (a) Die einfachsten Befehle sind die Zuweisungen. Diese haben die Form:

```
v := e
```

Hierbei ist v eine Variable und e ist ein arithmetischer Ausdruck. Eine solche Zuweisung wird durch das geschachtelte Tupel

$$(':=',v,e)$$

dargestellt.

(b) Der Befehl zum Einlesen einer Zahl hat die Form:

```
v := read();
```

Die Aufgabe dieses Befehl ist es, den vom Benutzer eingegebenen Text in eine Zahl umzuwandeln, die dann in der Variablen v abgespeichert wird. Der Befehl wird durch das geschachtelte Tupel

dargestellt.

(c) Der Befehl zur Ausgabe eines Wertes hat die Form:

```
print(e);
```

Die Aufgabe dieses Befehl ist es, den Wert des Ausdrucks e zu berechnen und auszugeben. Dieser Befehl

```
grammar Simple;
    program returns [stmnt_list]
        @init {SL = []}
        : (s = statement {SL.append($s.stmnt)})+ {$stmnt_list = SL}
    statement returns [stmnt]
        @init {SL = []}
        : v = VAR ':=' e = expr ';'
                                         {$stmnt = (':=', $v.text, $e.result)}
10
        | v = VAR ':=' 'read' '(' ')' ';' {$stmnt = ('read', $v.text)}
11
        12
        | 'if' '(' b = boolExpr ')' '{' (1 = statement {SL.append($1.stmnt) })* '}'
          {\$stmnt = ('if', \$b.result, SL)}
14
        | 'while' '(' b = boolExpr ')' '{' (1 = statement {SL.append($1.stmnt) })* '}'
          {\$stmnt = ('while', \$b.result, \SL)}
16
18
    boolExpr returns [result]
        : l = expr '==' r = expr {$result = ('==', $1.result, $r.result)}
20
        | l = expr '<' r = expr {$result = ('<', $1.result, $r.result)}
22
    expr returns [result]
24
        : e = expr '+' p = product {$result = ('+', $e.result, $p.result)}
        | e = expr '-' p = product {$result = ('-', $e.result, $p.result)}
26
        | p = product
                                   {\$result = \$p.result}
27
28
29
    product returns [result]
        : p = product '*' f = factor {$result = ('*', $p.result, $f.result)}
31
        | p = product '/' f = factor {$result = ('/', $p.result, $f.result)}
        | f = factor
                                     {\$result = \$f.result}
33
34
35
    factor returns [result]
        : '(' expr ')' {$result = $expr.result}
37
        | v = VAR
                       {\$result = \$v.text}
        | n = NUMBER {\text{$result = int(\$n.text)}}
39
```

Figure 8.3: ANTLR-Spezifikation der Grammatik.

```
wird durch das geschachtelte Tupel (\mbox{'print'},v) dargestellt. 
 (d) Ein Test hat die Syntax:
```

**if** ( *b* ) { *statements* }

Hierbei ist b ein Ausdruck, dessen Auswertung True oder False ergibt und statements ist eine Liste von

Befehlen, die ausgeführt werden, falls b den Wert True hat. Dieser Befehl wird durch das geschachtelte Tupel

```
('if', b, statements)
```

dargestellt.

(e) Eine Schleife hat die Syntax:

```
while (b) { statements }
```

Hierbei ist b ein Ausdruck, dessen Auswertung True oder False ergibt und statements ist eine Liste von Befehlen, die ausgeführt werden, solange b den Wert True hat. Dieser Befehl wird durch das geschachtelte Tupel

```
('while', b, statements)
```

dargestellt.

- 3. Die syntaktische Variable boolExpr beschreibt einen Ausdruck, der einen Boole'schen Wert annimmt. Es gibt zwei Möglichkeiten einen solchen Wert zu erzeugen.
  - (a) Ein Ausdruck der Form

$$l == r$$

testet, ob die Auswertungen von l und r die selben Werte ergeben. Dieser Befehl Ausdruck wird durch das geschachtelte Tupel

$$('==',l,r)$$

dargestellt.

(b) Ein Ausdruck der Form

```
l < r
```

testet, ob die Auswertung von l einen Wert ergibt, der kleiner ist als der Wert, der sich bei der Auswertung von r ergibt. Dieser Befehl Ausdruck wird durch das geschachtelte Tupel

dargestellt.

4. In analoger Weise beschreiben die Variablen expr, product und factor arithmetische Ausdrücke. Da wir dies bereits hinlänglich früher diskutiert haben, gehen wir an dieser Stelle nicht weiter auf die Grammatik-Regeln ein, durch die diese Variablen definiert werden.

```
41
42  VAR : [a-zA-Z][a-zA-Z_0-9]*;
43  NUMBER : '0'|[1-9][0-9]*;
44
45  MULTI_COMMENT : '/*' .*? '*/' -> skip;
46  LINE_COMMENT : '//' ~('\n')* -> skip;
47  WS : [\t\n\r] -> skip;
```

Figure 8.4: ANTLR-Spezifikation der verschiedenen Token.

Die Spezifikation der Token ist in Abbildung 8.4 gezeigt. Der Scanner unterscheidet im Wesentlichen zwischen Variablen und Zahlen. Variablen beginnen mit einem großen oder kleinen Buchstaben, auf den dann zusätzlich Ziffern und der Unterstrich folgen können. Folgen von Ziffern werden als Zahlen interpretiert. Enthält eine solche Folge mehr als ein Zeichen, so darf die erste Ziffer nicht 0 sein. Darüber hinaus entfernt der Scanner Whitespace

und Kommentare. Außerdem haben wir bei der Spezifikation von mehrzeiligen Kommentaren die sogenannte *non-greedy*-Version des Operators "\*" benutzt. Die non-greedy-Version des Operators "\*" wird als "\*?" geschrieben und matched sowenig wie möglich. Daher steht der reguläre Ausdruck

```
'/*' .*? '*/'
```

für einen String, der mit der Zeichenkette "/\*", mit der Zeichenkette "\*/" endet und außerdem so kurz wie möglich ist. Dadurch werden in einer Zeile der Form

```
/* Hugo */ i := i + 1; /* Anton */
```

zwei getrennte Kommentare erkannt.

```
def main(file):
    with open(file, 'r') as handle:
        program_text = handle.read()
    input_stream = antlr4.InputStream(program_text)
    lexer = SimpleLexer(input_stream)
    token_stream = antlr4.CommonTokenStream(lexer)
    parser = SimpleParser(token_stream)
    result = parser.program()
    execute_list(result.stmnt_list)
```

Figure 8.5: Die Funktion InputReader.

```
def execute_list(Statement_List, Values={}):
        for stmnt in Statement_List:
             execute(stmnt, Values)
    def execute(stmnt, Values):
        op = stmnt[0]
6
        if op == ':=':
             _, var, value = stmnt
            Values[var] = evaluate(value, Values)
        elif op == 'read':
10
             _, var = stmnt
11
            Values[var] = int(input())
12
        elif op == 'print':
             _, expr = stmnt
14
            print(evaluate(expr, Values))
        elif op == 'if':
16
             _, test, SL = stmnt
             if evaluate(test, Values):
18
                 execute_list(SL, Values)
        elif op == 'while':
20
             _, test, SL = stmnt
21
             while evaluate(test, Values):
22
                 execute_list(SL, Values)
23
        else:
24
             assert False, f'{stmnt} unexpected'
```

Figure 8.6: Die Funktion execute.

```
def evaluate(expr, Values):
         if isinstance(expr, int):
             return expr
        if isinstance(expr, str):
             return Values[expr]
        op = expr[0]
        if op == '==':
             _, lhs, rhs = expr
             return evaluate(lhs, Values) == evaluate(rhs, Values)
         if op == '<':
10
11
             _{-}, lhs, rhs = expr
             return evaluate(lhs, Values) < evaluate(rhs, Values)</pre>
12
         if op == '+':
             _, lhs, rhs = expr
14
             return evaluate(lhs, Values) + evaluate(rhs, Values)
         if op == '-':
16
             _, lhs, rhs = expr
             return evaluate(lhs, Values) - evaluate(rhs, Values)
18
         if op == '*':
             _, lhs, rhs = expr
20
             return evaluate(lhs, Values) * evaluate(rhs, Values)
        if op == '/':
22
             _{,} lhs, rhs = expr
             return evaluate(lhs, Values) / evaluate(rhs, Values)
24
         assert False, f'{stmnt} unexpected'
```

Figure 8.7: Die Funktion evaluate.

### Aufgabe 22:

- (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.
- (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 9

# 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

$$L_{\mathsf{square}} = \left\{ ww \mid w \in \Sigma^* \right\}$$

definiert wird, für das Alphabet  $\Sigma = \{a, b\}$  keine kontextfreie Sprache ist. Bevor wir das Pumping-Lemma für kontextfreie Sprachen beweisen, zeigen wir, wie sich nutzlose Symbole aus einer Grammatik entfernen lassen.

### 9.1 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 21 (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.

<sup>&</sup>lt;sup>1</sup> 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

$$A \rightarrow \alpha$$

und sind alle syntaktischen Variablen, die in dem Wort  $\alpha$  auftreten, bereits als erzeugende Variablen erkannt, so ist auch die syntaktische Variable A erzeugend.

**Beispiel**: Es sei  $G = \langle \{S, A, B, C\}, \{x\}, R, S \rangle$  und die Menge der Regeln R sei wie folgt gegeben:

$$S \rightarrow ABC \mid A$$

 $A \rightarrow AA \mid x$ 

 $B \rightarrow AC$ 

 $C \rightarrow BA$ .

Aufgrund der Regel

$$A \to \mathbf{x}$$

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.  $\Box$ 

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  $G = \langle \{S, A, B\}, \{x, y\}, R, S \rangle$ , deren Regeln durch

$$S \rightarrow A v$$
,

$$A \rightarrow AA \mid x \text{ und}$$

$$B \rightarrow Ay$$

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  $\alpha$  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  $G = \langle \{S, A, B, C\}, \{x, y\}, R, S \rangle$ , deren Regeln durch

$$\begin{array}{cccc} S & \rightarrow & BC \mid A, \\ A & \rightarrow & AA \mid \mathbf{x}, \\ B & \rightarrow & \mathbf{y} & \text{und} \\ C & \rightarrow & CC \end{array}$$

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

$$A \Rightarrow^* x$$
,  $S \Rightarrow^* x$  und  $B \Rightarrow^* y$ .

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.

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.

### 9.2 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 9.2 auf Seite 98 zeigt einen Parse-Baum für den String "2\*3+4", der mit Hilfe der in Abbildung 9.1 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:

```
expr 
ightarrow expr "+" product \ | expr "-" product \ | product \ | product \ | product "*" factor \ | product "/" factor \ | factor \ | factor \ | "(" expr ")" \ | "2" | "3" | "4"
```

Figure 9.1: Grammatik für arithmetische Ausdrücke.

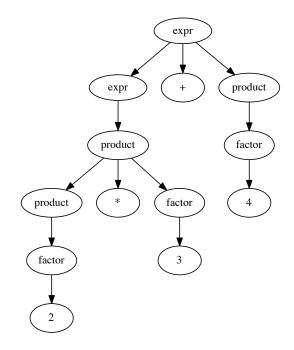


Figure 9.2: Ein Parse-Baum für den String "2\*3+4".

```
[ [expr, expr, product, product, factor, "2" ],
    [expr, expr, product, "*" ],
    [expr, expr, product, factor, "3" ],
    [expr, "+" ],
    [expr, product, factor, "4" ]
].
```

 $\Diamond$ 

Die nun folgende Definition der Funktion parseTree() formalisiert die Berechnung des Parse-Baums.

#### Definition 22 (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  $\mathit{parseTree}(A \Rightarrow^* w) = \big[ \, [A, \varepsilon] \, \big].$
- 3. Falls  $A \Rightarrow^* w$  gilt, weil

$$(A \to B_1 B_2 \cdots B_m) \in R$$
,  $B_i \Rightarrow^* w_i$  für alle  $i = 1, \cdots, m$  und  $w = w_1 w_2 \cdots w_m$ 

gilt, so definieren wir (unter Benutzung der SETLX-Notation für die Definition von Listen)

$$parseTree(A \Rightarrow^* w) = [A] + l : i \in [1, \dots, m], l \in parseTree(B_i \Rightarrow^* w_i)].$$

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

$$parseTree(B_i \Rightarrow^* B_i) := [[B_i]].$$

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 23 (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

$$parseTree(A \Rightarrow^* w)$$

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

$$A \to t_1 t_2 \cdots t_m$$
 mit  $w = t_1 t_2 \cdots t_m$  und  $t_i \in T$  für alle  $i = 1, \cdots, m$ 

in der Grammatik G geben. Es gilt dann

$$parseTree(A \Rightarrow^* w) = [[A, t_1], [A, t_2], \cdots, [A, t_m]].$$

Daraus folgt

$$|w| = |t_1 t_2 \cdots t_m| = m < b = b^1 = b^{2-1}$$

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

$$parseTree(B_i \Rightarrow^* w_i)$$

für alle  $i=1,\cdots,m$  höchstens die Länge n. Nach Induktions-Voraussetzung wissen wir also, dass

$$|w_i| \leq b^{n-1}$$
 für alle  $i = 1, \dots, m$ 

gilt. Daher haben wir

$$|w| = |w_1| + \dots + |w_m| \le b^{n-1} + \dots + b^{n-1} = m \cdot b^{n-1} \le b \cdot b^{n-1} = b^n = b^{(n+1)-1}.$$

## 9.3 Das Pumping-Lemma für kontextfreie Sprachen

**Satz 24 (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.  $\forall h \in \mathbb{N} : uv^h w x^h y \in L$ .

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 k hat. Wir definieren

$$n := b^{k+1}$$

Sei nun  $s \in L$  mit  $|s| \ge n$ . Wir wählen einen Parse-Baum  $\tau$  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  $\tau$  betrachten wir die Listen aus

$$\tau = parseTree(S \Rightarrow^* s).$$

Falls alle Listen hier eine Länge kleiner-gleich k+1 hätten, so würde aus den Beschränktheits-Lemma 23 folgen, dass

$$|s| \le b^{(k+1)-1} = b^k < b^{k+1} = n$$
, also  $|s| < n$ 

gilt, im Widerspruch zu der Voraussetzung  $|s| \geq n$ . Also muss es in  $\tau$  eine Liste geben, die mindestens die Länge k+2 hat. Wir wählen die längste Liste unter den Listen in  $\tau$  aus. Diese Liste hat dann die Form

$$[A_1, \cdots, A_l, t]$$
 mit  $A_i \in V$  für alle  $i \in \{1, \cdots, l\}, t \in T$ , sowie  $l \ge k + 1$ .

Wegen  $l \ge k+1$  können nicht alle Variablen  $A_1, \dots, A_l$  voneinander verschieden sein, denn es gibt ja nur insgesamt k verschiedene syntaktische Variablen. Wir finden daher in der Menge  $\{l, l-1, \dots, l-k\}$  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

$$S \Rightarrow^* uA_i y \Rightarrow^* uvA_i xy \Rightarrow^* uvwxy = s.$$

Insbesondere gilt also

$$S \Rightarrow^* uAy$$
,  $A \Rightarrow^* vAx$ , und  $A \Rightarrow^* w$ .

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  $uv^hwx^hy \in L$  für beliebige  $h \in \mathbb{N}$ .

Wir müssen jetzt noch zeigen, dass  $vx \neq \varepsilon$  gilt. Die Ableitung

$$A \Rightarrow^* vAx$$

ist eigentlich die Ableitung

$$A_i \Rightarrow^* v A_i x$$

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  $\tau$  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

$$A = A_i \Rightarrow^* vA_i x \Rightarrow^* vwx.$$

Weil einerseits  $i \in \{l-k, l-(k-1), \cdots, l\}$  gilt und andererseits die der Ableitung  $A \Rightarrow^* vwx$  zugeordnete Liste aus  $\textit{parseTree}(S \Rightarrow^* w)$  von maximaler Länge war, wissen wir, dass die Länge der längsten Liste in

$$parseTree(A \Rightarrow^* vwx)$$

kleiner-gleich k+2 ist. Nach dem Beschränktheits-Lemma 23 folgt damit für die Länge von vwx die Abschätzung

$$|vwx| \le b^{(k+2)-1} = b^{k+1} = n.$$

**Bemerkung**: Das Pumping-Lemma wird in der deutschsprachigen Literatur gelegentlich als "Schleifensatz" bezeichnet. Es wurde 1961 von Bar-Hillel, Perles und Shamir [BHPS61] bewiesen.

## 9.4 Anwendungen des Pumping-Lemmas

Wir zeigen, wie mit Hilfe des Pumping-Lemmas der Nachweis erbracht werden kann, dass bestimmte Sprachen nicht kontextfrei sind.

### 9.4.1 Die Sprache $L = \{a^k b^k c^k \mid k \in \mathbb{N}\}$ ist nicht kontextfrei

Wir weisen nun nach, dass die Sprache

$$L = \{ \mathbf{a}^k \mathbf{b}^k \mathbf{c}^k \mid k \in \mathbb{N} \}$$

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.  $\forall i \in \mathbb{N} : uv^i w x^i y \in L$ .

Insbesondere können wir hier i=0 wählen und erhalten dann  $uwy \in L$ .

Wir definieren nun den String s wie folgt:

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

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:

$$count(vwx, c) = 0.$$

Da  $vx \neq \varepsilon$  ist, folgt

$$count(vx, a) + count(vx, b) > 0.$$

Wir nehmen zunächst an, dass  $count(vx, \mathbf{a}) > 0$  gilt, der Fall  $count(vx, \mathbf{b}) > 0$  ist analog zu behandeln. Dann erhalten wir einerseits

$$\begin{array}{lll} \textit{count}(uwy, \texttt{c}) & = & \textit{count}(s, \texttt{c}) - \textit{count}(vx, \texttt{c}) \\ & = & \textit{count}(s, \texttt{c}) - 0 \\ & = & \textit{count}(\texttt{a}^n \texttt{b}^n \texttt{c}^n, \texttt{c}) \\ & = & n \end{array}$$

Zählen wir nun die Häufigkeit, mit welcher der Buchstabe "a" in dem String uwy auftritt, so erhalten wir

$$\begin{array}{lll} \textit{count}(uwy, \texttt{a}) & = & \textit{count}(s, \texttt{a}) - \textit{count}(vx, \texttt{a}) \\ & < & \textit{count}(s, \texttt{a}) \\ & = & \textit{count}(\texttt{a}^n \texttt{b}^n \texttt{c}^n, \texttt{a}) \\ & = & n \end{array}$$

Damit haben wir dann aber

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.

0

Aufgabe 23: Zeigen Sie, dass die Sprache

$$L = \left\{ \left. \mathbf{a}^{k^2} \mid k \in \mathbb{N} \right. \right\}$$

nicht kontextfrei ist.

Hinweis: Argumentieren Sie über die Länge der betrachteten Strings.

**Lösung**: Wir führen den Beweis indirekt und nehmen an, dass die Sprache  $L_{\text{square}}$  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.  $\forall h \in \mathbb{N} : uv^h wx^h y \in L_{\text{square}}$ .

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

$$|s| = |a^{n^2}| = n^2 \ge n.$$

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

$$n^2 = c + d + e + f + g. (9.1)$$

2. Aus der Ungleichung |vwx| < n folgt

$$d+e+f \le n. (9.2)$$

3. Die Bedingung  $vx \neq \varepsilon$  liefert

$$d+f>0. (9.3)$$

4. Aus der Formel  $\forall h \in \mathbb{N} : uv^h wx^h y \in L_{\text{square}}$  folgt

$$\forall h \in \mathbb{N} : a^c a^{d \cdot h} a^e a^{f \cdot h} a^g \in L_{\text{square}}. \tag{9.4}$$

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  $L_{\text{square}}$  gibt es dann eine natürliche Zahl k, so dass gilt

$$c + 2 \cdot d + e + 2 \cdot f + g = k^2. \tag{9.5}$$

Addieren wir in Gleichung (9.1) auf beiden Seiten d + f, so erhalten wir insgesamt

$$n^2 + d + f = c + 2 \cdot d + e + 2 \cdot f + q = k^2$$
.

Wegen d + f > 0 folgt daraus

$$n < k. (9.6)$$

Andererseits haben wir

$$\begin{array}{lll} k^2 & = & c+2\cdot d+e+2\cdot f+g & \text{nach Gleichung (9.5)} \\ & = & (c+d+e+f+g)+(d+f) & \text{elementare Umformung} \\ & \leq & (c+d+e+f+g)+(d+e+f) & \text{denn } e \geq 0 \\ & \leq & (c+d+e+f+g)+n & \text{nach Ungleichung (9.2)} \\ & = & n^2+n & \text{nach Gleichung (9.1)} \\ & < & n^2+2\cdot n+1 & \text{da } n+1>0 \\ & = & (n+1)^2 & \text{elementare Umformung} \end{array}$$

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

$$k < n+1. (9.7)$$

Zusammen mit Ungleichung (9.6) liefert Ungleichung (9.7) 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 24**: Das Alphabet  $\Sigma$  sei durch die Festlegung  $\Sigma := \{a,b\}$  definiert. Zeigen Sie, dass die Sprache

$$L = \{ tt \mid t \in \Sigma^* \}$$

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  $|s| \ge n$  in fünf Teilstrings u, v, w, x, und y aufspalten lässt, so dass gilt:

- 1. s = uvwxy,
- 2.  $|vwx| \leq n$ ,
- 3.  $vx \neq \varepsilon$ ,
- $4. \ \forall h \in \mathbb{N} : uv^h w x^h y \in L.$

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

$$|s| = |a^n b^n a^n b^n| = 4 \cdot n \ge n.$$

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  $s=a^nb^na^nb^n$  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

$$r \sqsubseteq s$$
 g.d.w.  $r$  ist Teilstring von  $s$ .

Wollen wir zusätzlich die Position einschränken, an der r innerhalb von s auftritt, so schreiben wir

$$r \sqsubseteq_{i,j} s$$
 g.d.w.  $r \sqsubseteq s[i..j]$ .

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  $a^nb^na^nb^n$  gibt es aufgrund der Längenbeschränkung  $|vwx| \le n$  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  $a^nb^n$ .

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  $b^na^n$ .

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  $a^nb^n$ .

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  $s=a^nb^na^nb^n$  liegt und der String uwy aus s dadurch entsteht, dass wir v und x weglassen, hat der String uwy die Form

$$uwy = a^{n-k_1}b^{n-k_2}a^nb^n$$

und wegen  $vx \neq \varepsilon$  wissen wir, dass  $k_1 + k_2 > 0$  ist. Die Mitte des Strings uwy liegt daher innerhalb des Teilstrings  $a^nb^n$ . Wenn wir also

$$t't'=a^{n-k_1}b^{n-k_2}a^nb^n \quad \text{ für ein } t'\in \Sigma^*$$

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  $s=b^na^n$  liegt und der String uwy aus s dadurch entsteht, dass wir v und x weglassen, hat der String uwy die Form

$$uwu = a^n b^{n-k_1} a^{n-k_2} b^n$$

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

$$L^{c} = \left\{ s \in \Sigma^* \mid \neg (\exists w \in \Sigma^* : s = ww) \right\},\,$$

is context-free.

Aufgabe 25: Zeigen Sie, dass die Sprache

$$L = \{ \, \mathbf{a}^p \mid p \in \mathbb{P} \}$$

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.  $\forall h \in \mathbb{N} : uv^h wx^h y \in L$ .

Wir wählen nun eine Primzahl p, die größer als n+1 ist und setzen  $s=\mathtt{a}^p$ . 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  $\mathtt{a}^p$  der Form

$$a^p = uvwxy$$

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:

$$u = \mathbf{a}^c$$
,  $v = \mathbf{a}^d$ ,  $w = \mathbf{a}^e$ ,  $x = \mathbf{a}^f$  und  $y = \mathbf{a}^g$ .

Für die Zahlen c, d, e, f und g gilt dann Folgendes:

- 1. c + d + e + f + q = p,
- 2.  $d + f \neq 0$ .
- 3.  $d + e + f \le n$ ,
- 4.  $\forall h \in \mathbb{N} : c + h \cdot d + e + h \cdot f + q \in \mathbb{P}$ .

Setzen wir in der letzten Gleichung für h den Wert (c+e+g) ein, so erhalten wir

$$c + (c + e + q) \cdot d + e + (c + e + q) \cdot f + q \in P$$
.

Wegen

$$c + (c + e + q) \cdot d + e + (c + e + q) \cdot f + q = (c + e + q) \cdot (1 + d + f)$$

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

$$d+e+f \le n$$
,  $c+d+e+f+g=p$  und  $p>n+1$ 

wissen wir, dass

$$c + e + g \ge c + g = c + d + e + f + g - (d + e + f) = p - (d + e + f) \ge p - n > 1$$

ist, so dass auch der Faktor (c+e+q) 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.  $\Box$  **Aufgabe 26**: Zeigen Sie, dass die Sprache  $L = \{ a^{2^k} \mid k \in \mathbb{N} \}$  nicht kontextfrei ist.  $\diamondsuit$  **Aufgabe 27**: Zeigen Sie, dass die Sprache  $L = \{ a^k b^l c^{k \cdot l} \mid k, l \in \mathbb{N} \}$  nicht kontextfrei ist.  $\diamondsuit$  **Aufgabe 28**: 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.  $\diamondsuit$ 

## Chapter 10

# Earley-Parser\*

In diesem Kapitel stellen wir ein effizientes Verfahren vor, mit dem es möglich ist, für eine <u>beliebige</u> 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  $\operatorname{CYK}$ -Algorithmus bekannt, der unabhängig von John Cocke [CS70], Daniel H. Younger [You67] und Tadao Kasami [Kas65] entdeckt wurde. Der  $\operatorname{CYK}$ -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  $\operatorname{CYK}$ -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.

## 10.1 Der Algorithmus von Earley

Der zentrale Begriff des von Earley angegebenen Algorithmus ist der Begriff des Earley-Objekts, das wie folgt definiert ist.

**Definition 25 (Earley-Objekt)** Gegeben sei eine kontextfreie Grammatik  $G = \langle V, \Sigma, R, S \rangle$  und ein String  $s = x_1x_2 \cdots x_n \in \Sigma^*$  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. 
$$k \in \{0, 1, \dots, n\}$$
.

**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

$$\langle A \to \alpha \bullet \beta, k \rangle \in Q_j \quad \text{mit } j \ge k$$

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  $\alpha$  erkannt, es gilt also

$$\alpha \Rightarrow^* x_{k+1} \cdots x_i$$
.

3. Folglich versucht der Parser am Anfang des Teilstrings  $x_{j+1} \cdots x_n$  ein  $\beta$  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 := \{\langle \widehat{S} \to \bullet S, 0 \rangle\},\$$

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  $\beta$  erkannt. Folgt auf dieses  $\beta$  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  $\gamma$  erkennen. Daher wird in diesem Fall das Earley-Objekt

$$\langle A \to \beta a \bullet \gamma, k \rangle$$

dem Zustand  $Q_{i+1}$  hinzugefügt:

$$\langle A \to \beta \bullet a\gamma, k \rangle \in Q_j \land x_{j+1} = a \ \Rightarrow \ Q_{j+1} := Q_{j+1} \cup \big\{ \langle A \to \beta a \bullet \gamma, k \rangle \big\}.$$

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:

$$\langle A \to \beta \bullet C\delta, k \rangle \in Q_i \land (C \to \gamma) \in R \Rightarrow Q_i := Q_i \cup \{\langle C \to \bullet \gamma, j \rangle\}.$$

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:

$$\langle C \to \gamma \bullet, j \rangle \in Q_i \land \langle A \to \beta \bullet C \delta, k \rangle \in Q_i \Rightarrow Q_i := Q_i \cup \{ \langle A \to \beta C \bullet \delta, k \rangle \}.$$

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:

$$\begin{split} Q_0 &:= \big\{ \langle \widehat{S} \to \bullet S, 0 \rangle \big\}, \\ Q_i &:= \big\{ \big\} \quad \text{ für } i = 1, \cdots, n. \end{split}$$

- 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  $\varepsilon$ -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 10.1 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.

Figure 10.1: 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

$$\langle E \rightarrow \bullet \ E \text{"+"} P, 0 \rangle \quad \text{und} \quad \langle E \rightarrow \bullet P, 0 \rangle$$

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

$$\langle P \to \bullet \ P \ "*" F, 0 \rangle \quad \text{ und } \quad \langle P \to \bullet \ F, 0 \rangle.$$

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  $Q_0$ :

$$\langle F \to \bullet \text{``1"}, 0 \rangle$$
,  $\langle F \to \bullet \text{``2"}, 0 \rangle$ , und  $\langle F \to \bullet \text{``3"}, 0 \rangle$ .

Insgesamt enthält  $Q_0$  nun die folgenden Earley-Objekte:

- 1.  $\langle \widehat{S} \rightarrow \bullet E, 0 \rangle$ ,
- 2.  $\langle E \rightarrow \bullet E "+" P, 0 \rangle$
- 3.  $\langle E \rightarrow \bullet P, 0 \rangle$ ,
- 4.  $\langle P \rightarrow \bullet P "*" F, 0 \rangle$ ,
- 5.  $\langle P \rightarrow \bullet F, 0 \rangle$ ,
- 6.  $\langle F \rightarrow \bullet$  "1",  $0 \rangle$ ,
- 7.  $\langle F \rightarrow \bullet$  "2",  $0 \rangle$ ,
- 8.  $\langle F \rightarrow \bullet \text{"3"}, 0 \rangle$ .

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:

$$Q_1 = \{ \langle F \to \text{"1"} \bullet, 0 \rangle \}.$$

2. Nun setzen wir i=1 und wenden zunächst auf  $Q_1$  die Vervollständigungs-Operation an. Aufgrund des Earley-Objekts  $\langle F \to$  "1"  $\bullet$ ,  $0 \rangle$  in  $Q_1$  suchen wie in  $Q_0$  ein Earley-Objekt, bei dem die Markierung " $\bullet$ " 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.  $\langle P \rightarrow F \bullet, 0 \rangle$
- 2.  $\langle P \rightarrow P \bullet "*" F, 0 \rangle$ ,
- 3.  $\langle E \rightarrow P \bullet, 0 \rangle$ ,
- 4.  $\langle E \rightarrow E \bullet \text{"+"} P, 0 \rangle$ ,
- 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 "ullet" 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

$$\langle E \rightarrow E \bullet "+" P, 0 \rangle$$

enthält, fügen wir in  $Q_2$  das Earley-Objekt

$$\langle E \rightarrow E$$
 "+" •  $P, 0 \rangle$ 

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

$$\langle P \to \bullet \ F, 2 \rangle$$
 und  $\langle P \to \bullet \ P \text{"*"} F, 2 \rangle$ .

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.  $\langle F \rightarrow \bullet$  "1", 2 $\rangle$ ,
- 2.  $\langle F \rightarrow \bullet "2", 2 \rangle$ ,
- 3.  $\langle F \rightarrow \bullet "3", 2 \rangle$ .

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.  $\langle P \rightarrow P \bullet "*" F, 2 \rangle$ ,
- 2.  $\langle E \rightarrow E$  "+"  $P \bullet, 0 \rangle$ ,
- 3.  $\langle E \rightarrow E \bullet "+" P, 0 \rangle$
- 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.  $\langle F \rightarrow \bullet$  "1",  $4 \rangle$ ,
  - 2.  $\langle F \rightarrow \bullet "2", 4 \rangle$ ,
  - 3.  $\langle F \rightarrow \bullet "3", 4 \rangle$ .

Da das nächste Zeichen die Ziffer "3" ist, liefert die Lese-Operation für  $Q_5$ :

$$Q_5 = \langle F \rightarrow \text{"3"} \bullet, 4 \rangle \}.$$

- 6. Wir setzen i=5. Die Vervollständigungs-Operation liefert nacheinander die folgenden Earley-Objekte:
  - 1.  $\langle P \rightarrow P "*" F \bullet, 2 \rangle$ ,
  - 2.  $\langle E \rightarrow E$  "+"  $P \bullet, 0 \rangle$ ,
  - 3.  $\langle P \rightarrow P \bullet "*" F, 2 \rangle$ .
  - 4.  $\langle E \rightarrow E \bullet "+" P, 0 \rangle$

```
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 10.1 angegebenen Grammatik liegt. Benutzen Sie dazu den von Earley angegebenen Algorithmus.

## 10.2 Implementing Earley's Algorithm in SetIX

In this section we present a simple implementation of Earley's algorithm in  $\operatorname{Setl}X$ . 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.

#### 10.2.1 The class earleyItem

```
class earleyItem(variable, alpha, beta, index) {
        this.mVariable := variable;
2
        this.mAlpha
                        := alpha;
3
        this.mBeta
                        := beta;
        this.mIndex
                        := index;
        isComplete := procedure() {
             return mBeta == [];
        };
        sameVar := procedure(c) {
10
            return #mBeta > 0 && mBeta[1] == Var(c);
11
12
        myScan := procedure(t) {
13
             return #mBeta > 0 && mBeta[1] == Token("\'$t$\'");
14
        };
15
        nextVar := procedure() {
16
             if (#mBeta > 0) {
                 match (mBeta[1]) {
18
                     case Var(c): return c;
                 }
20
             }
        };
22
        moveDot := procedure() {
             return earleyItem(mVariable, mAlpha + [ mBeta[1]], mBeta[2..], mIndex);
24
        };
25
    }
26
```

Figure 10.2: The class earleyItem.

Figure 10.2 on page 113 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  $\alpha$ ,
- 3. mBeta represents  $\beta$ , 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

$$\langle A \to \alpha \bullet C \beta, k \rangle$$
,

i.e. it checks, whether the variable C that is given as argument to this method is following the " $\bullet$ ". In order to do this, we first have to check, whether there is anything following the " $\bullet$ ", i.e. whether  $\beta$  is not empty. Only then can we check whether the first item of  $\beta$  is indeed the variable C.

3. The method myScan(t) checks whether the Earley item has the form

$$\langle A \to \alpha \bullet t \beta, k \rangle$$
,

that is it checks whether the token t that is given as argument to this method follows the " $\bullet$ ".

4. The method nextVar() first checks whether the Earley item has the form

$$\langle A \to \alpha \bullet C\delta, k \rangle$$
,

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

$$\langle A \to \alpha \bullet X \delta, k \rangle$$

is transformed into

$$\langle A \to \alpha X \bullet \delta, k \rangle$$
.

Here, X can be either a variable or a token. The method moveDot should only be called for an Earley item  $\langle A \to \alpha \bullet \beta, k \rangle$  if we know that  $\beta \neq \varepsilon$ .

### 10.2.2 The class grammar

Next, we show how we represent a grammar. Figure 10.3 on page 115 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

$$A \to \beta$$

is represented as the pair  $[A, \beta]$ , where  $\beta$  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.

```
class grammar(rules) {
        this.mRules := rules;
        startItem := procedure() {
            return earleyItem("sHat", [], [ Var(startVar()) ], 0);
        };
        finishItem := procedure() {
            return earleyItem("sHat", [ Var(startVar()) ], [], 0);
        };
        startVar := procedure() {
10
            return mRules[1][1];
11
        };
12
    }
13
```

Figure 10.3: 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.

#### 10.2.3 The class earleyParser

We are finally ready to implement Earley's algorithm. It is implemented in the class earleyParser shown in Figure 10.4 on page 116. 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 = \mathtt{mStateList}[i+1].
```

Figure 10.4 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;
3
        this.mStateList := [ \{\}: i in \{1 ... \text{ #mString+1}\} ]; // Q_i == mStateList[i+1]
        this.mStateList[1] := mGrammar.startItem();
        earleyParse := procedure()
                                      { ··· };
                     := procedure(i) { ··· };
        complete
                     := procedure(i) { ··· };
        predict
                     := procedure(i) { ··· };
        myScan
10
11
```

Figure 10.4: 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 10.5 on page 116. 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  $Q_{i+1}$ .

Figure 10.5: The method *earleyParse()*.

Figure 10.6 on page 117 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 \land \langle A \to \beta \bullet C\delta, k \rangle \in Q_j \Rightarrow Q_i := Q_i \cup \{\langle A \to \beta C \bullet \delta, k \rangle\}
```

This formula is implemented as follows:

1. The variable i is the parameter i of the method complete.

```
complete := procedure(i) {
         change := false;
                := mStateList[i+1];
3
         do {
             added := false:
             newQi := {};
             for (item in qi | item.isComplete()) {
                 c := item.mVariable;
                 j := item.mIndex;
                 qj := mStateList[j+1];
10
                 for (cItem in qj) {
11
                      if (cItem.sameVar(c)) {
12
                          moved := cItem.moveDot();
                          newQi += { moved };
14
                     }
                 }
             }
             if (!(newQi <= qi)) {
18
                 change := added := true;
                 this.mStateList[i+1] += newQi;
20
                 qi := mStateList[i+1];
             }
22
         } while (added);
23
        return change;
24
    };
25
```

Figure 10.6: 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.

```
predict := procedure(i) {
         change := false;
2
                := mStateList[i+1];
3
         do {
             added := false;
             newQi := {};
             for (item in qi) {
                 c := item.nextVar();
                 if (c != om) {
                     for (rule in mGrammar.mRules | c == rule[1]) {
10
                         newQi += { earleyItem(c, [], rule[2], i) };
                     }
12
                 }
14
             if (!(newQi <= qi)) {
15
                 change := added := true;
                 this.mStateList[i+1] += newQi;
                 qi := mStateList[i+1];
18
             }
19
         } while (added);
         return change;
21
    };
22
```

Figure 10.7: Implementing the prediction operation via the method predict.

Figure 10.7 on page 116 shows the method predict. This method implements the prediction operation. In the previous section, this operation had been specified via the following formula:

$$\langle A \to \beta \bullet C\delta, k \rangle \in Q_i \land (C \to \gamma) \in R \Rightarrow Q_i := Q_i \cup \{\langle C \to \bullet \gamma, i \rangle\}$$

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  $Q_i$ .

```
myScan := procedure(i) {
    qi := mStateList[i+1];
    if (i < #mString) {
        a := mString[i+1];
        for (item in qi | item.myScan(a)) {
            this.mStateList[i+2] += { item.moveDot() };
        }
    }
}</pre>
```

Figure 10.8: Implementing the scan operation via the method myScan.

Figure 10.8 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 \land x_{i+1} = a \Rightarrow Q_{i+1} := Q_{i+1} \cup \{ \langle A \to \beta a \bullet \gamma, k \rangle \}.
```

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

$$\langle A \rightarrow \beta \ a \bullet \gamma, k \rangle$$

is added to the set  $Q_{i+1}$ . This Earley item can be conveniently computed using the method moveDot.

Finally, Figure 10.9 on page 120 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 10.1.

## 10.3 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  $s=x_1x_2\cdots x_n$  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

```
QO:
<expr -> (*) expr '+' prod, 0>
<expr -> (*) prod, 0>
<fact -> (*) '1', 0>
<fact -> (*) '2', 0>
<fact -> (*) '3', 0>
< (*) fact, 0>
< -> (*) prod '*' fact, 0>
<sHat -> (*) expr, 0>
Q1:
<fact -> '1' (*), 0>
<expr -> expr (*) '+' prod, 0>
<sHat -> expr (*), 0>
< +> fact (*), 0>
<expr -> prod (*), 0>
od -> prod (*) '*' fact, 0>
Q2:
<fact -> (*) '1', 2>
<fact -> (*) '2', 2>
<fact -> (*) '3', 2>
< (*) fact, 2>
< -> (*) prod '*' fact, 2>
<expr -> expr '+' (*) prod, 0>
Q3:
<fact -> '2' (*), 2>
<expr -> expr (*) '+' prod, 0>
<sHat -> expr (*), 0>
<expr -> expr '+' prod (*), 0>
< -> fact (*), 2>
< -> prod (*) '*' fact, 2>
<fact -> (*) '1', 4>
<fact -> (*) '2', 4>
<fact -> (*) '3', 4>
od -> prod '*' (*) fact, 2>
Q5:
<fact -> '3' (*), 4>
<expr -> expr (*) '+' prod, 0>
<sHat -> expr (*), 0>
<expr -> expr '+' prod (*), 0>
< -> prod (*) '*' fact, 2>
od -> prod '*' fact (*), 2>
```

Figure 10.9: Output of our Earleyd parser for the input "1+2\*3" and the grammar from Figure 10.1.

 $Q_n$ . 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 26** Es sei  $s=x_1x_2\cdots x_n\in T^*$  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,  $\alpha$ ,  $\beta$  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.:  $0, \dots, l \mapsto l+1$ . 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  $\langle A \to \alpha' \bullet C\beta, k \rangle \in Q_i$  folgt nach Induktions-Voraussetzung

$$\alpha' \Rightarrow^* x_{k+1} \cdots x_i$$

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:

$$\begin{array}{rcl} \alpha & = & \alpha'C \\ \Rightarrow^* & x_{k+1} \cdots x_j C \\ \Rightarrow & x_{k+1} \cdots x_j \delta \\ \Rightarrow^* & x_{k+1} \cdots x_j x_{j+1} \cdots x_i \\ = & x_{k+1} \cdots x_i \end{array}$$

Damit ist  $\alpha \Rightarrow^* x_{k+1} \cdots x_i$  nachgewiesen.

#### Korollar 27 (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 28 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

$$\delta \Rightarrow^* x_{j+1} \cdots x_h \tag{10.1}$$

und

$$\mu \Rightarrow^* x_{h+1} \cdots x_l$$
 (10.2)

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

$$\langle A \to \alpha x_{i+1} \cdots x_j \bullet D\mu \gamma, k \rangle \in Q_j.$$
 (10.3)

Auf dieses Earley-Objekt wenden wir die Vorhersage-Operation an und erhalten

$$\langle D \to \bullet \, \delta, j \rangle \in Q_j.$$
 (10.4)

Aus (10.4) und (10.1) folgt mit der Induktions-Voraussetzung

$$\langle D \to \delta \bullet, j \rangle \in Q_h.$$
 (10.5)

Aus (10.3) und (10.5) folgt mit der Vervollständigungs-Operation

$$\langle A \to \alpha x_{i+1} \cdots x_i D \bullet \mu \gamma, k \rangle \in Q_h.$$
 (10.6)

Aus (10.6) und (10.2) folgt mit der Induktions-Voraussetzung

$$\langle A \to \alpha x_{i+1} \cdots x_i D\mu \bullet \gamma, k \rangle \in Q_l.$$
 (10.7)

Wegen  $\beta = x_{i+1} \cdots x_i D\mu$  haben wir damit

$$\langle A \to \alpha \beta \bullet \gamma, k \rangle \in Q_l$$

gezeigt und das ist gerade die Behauptung.

#### Korollar 29 (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 k := n setzen.

### 10.4 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:

Wenn 
$$\langle A \to \alpha \bullet \beta, k \rangle \in Q_i$$
, dann  $k \in \{0, \dots, i\}$ .

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 10.7) 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.

 $\mathsf{Insgesamt} \ \mathsf{hat} \ \mathsf{ein} \ \mathsf{Aufruf} \ \mathsf{der} \ \mathsf{Methode} \ \mathsf{predict}() \ \mathsf{daher} \ \mathsf{die} \ \mathsf{Komplexit\"{a}t} \ \mathcal{O}(1) \cdot O(n) \cdot \mathcal{O}(1) = \mathcal{O}(n).$ 

- 2. Bei der Implementierung der Lese-Operation, die in Abbildung 10.8 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 10.6 auf Seite 117 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 11

# **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.

## 11.1 Bottom-Up-Parser

Die mit Anter 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

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:

Im ersten Schritt haben wir beispielsweise die Grammatik-Regel  $F \to$  "1" benutzt, um den String "1" durch F zu ersetzen und dabei dann den String "F + 2 \* 3" erhalten. Im zweiten Schritt haben wir die Regel  $P \to F$  benutzt, um F durch P zu ersetzen. Auf diese Art und Weise haben wir am Ende den ursprünglichen String "1 + 2 \* 3" auf E zurück geführt. Wir können an dieser Stelle zwei Beobachtungen machen:

- 1. Wir ersetzen bei unserem Vorgehen immer den am weitesten links stehenden Teilstring, der ersetzt werden kann, wenn wir den anfangs gegebenen String auf das Start-Symbol der Grammatik zurück führen wollen.
- 2. Schreiben wir die Ableitung, die wir rückwärts konstruiert haben, noch einmal in der richtigen Reihenfolge hin, so erhalten wir:

$$E \Rightarrow E + P$$

$$\Rightarrow E + P * F$$

$$\Rightarrow E + P * 3$$

$$\Rightarrow E + F * 3$$

$$\Rightarrow E + 2 * 3$$

$$\Rightarrow P + 2 * 3$$

$$\Rightarrow F + 2 * 3$$

$$\Rightarrow 1 + 2 * 3$$

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

$$E + P * 3$$
.

das wir oben im fünften Schritt erhalten haben. Hier könnten wir den Teilstring "P" mit Hilfe der Regel

$$E \to P$$

durch "E" ersetzen. Dann würden wir den String

$$E + E * 3$$

erhalten. Die einzigen Reduktionen, die wir jetzt noch durchführen können, führen über die Zwischenergebnisse E + E \* F und E + E \* P zu dem String

$$E + E * E$$

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.

#### 11.2 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

$$P = \langle Q, q_0, action, goto \rangle$$

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

$$\textit{Action} := \big\{ \langle \mathtt{shift}, q \rangle \mid q \in Q \big\} \cup \big\{ \langle \mathtt{reduce}, r \rangle \mid r \in R \big\} \cup \big\{ \mathtt{accept} \big\} \cup \big\{ \mathtt{error} \big\},$$

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:

$$action: Q \times T \rightarrow Action.$$

4. goto ist eine Funktion, die jedem Zustand  $q \in Q$  und jeder syntaktischen Variablen  $v \in V$  einen neuen Zustand aus Q zuordnet:

$$goto: Q \times V \rightarrow Q.$$

Ein Shift-Reduce-Parser arbeitet nun mit den folgenden Daten-Strukturen.

1. Einem Stack *states*, auf dem Zustände aus der Menge Q abgelegt werden:

$$states \in Stack\langle Q \rangle$$
.

2. Einem Stack symbols, auf dem Grammatik-Symbole, also Terminale und Variablen-Symbole abgelegt werden:

$$symbols \in Stack\langle T \cup V \rangle$$
.

Zur Vereinfachung der folgenden Überlegungen nehmen wir an, dass die Menge T der Terminale das spezielle Symbol " $\mathrm{EoF}$ " enthält und dass dieses Symbol das Ende des zu parsenden Strings spezifiziert aber sonst in dem String nicht auftritt.

Figure 11.1 on page 129 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;
3
        parseSR := procedure(t1) {
             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];
10
                 t := tl[index];
11
                 p := mActionTable[q,t];
12
                 match (p) {
                 case om:
14
                      return false;
                 case Shift(s):
16
                      symbols := symbols + [t];
                      states := states + [s];
18
                      index
                              += 1;
                 case Rule(head, body):
20
                               := #body;
                      symbols := symbols[.. -(n+1)];
22
                      states := states[.. -(n+1)];
23
                      symbols := symbols + [head];
                               := states[-1];
25
                      states := states + [ mGotoTable[state, head] ];
26
                 case Accept():
27
                      return true;
                 }
29
             }
30
        };
31
    }
32
```

Figure 11.1: 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,
  - ullet 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  $action(q,t) = \mathtt{error}$ . 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  $\operatorname{SETLX}$  as the term  $\operatorname{Shift}(s)$ . 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  $\mathtt{Rule}(A,[X_1,\cdots,X_n])$ . In this case, we use the grammar rule

$$r = (A \to X_1 \cdots X_n)$$

to reduce the symbol stack. The SETLX variable *head* represents the left hand side A of this rule, while the list  $[X_1, \dots, X_n]$  is represented by the SETLX variable *body*.

In this case, it can be shown that the symbols  $X_1, \dots, X_n$  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 11.1 and 11.2 show these functions for the grammar given in Figure 11.2. 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 11.2. The coding of the functions *action* and *goto* is shown in the Figures 11.3, 11.4, and 11.5 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 
ightarrow expr "+" product \ | expr "-" product \ | product \ | product \ | product "*" factor \ | product "/" factor \ | factor \ | factor \ | NUMBER
```

Figure 11.2: A grammar for arithmetical expressions.

State	EOF	+	_	*	/	(	)	Number
$s_0$						$\langle \mathit{shft}, s_5 \rangle$		$\langle \mathit{shft}, s_2 \rangle$
$s_1$	$\langle \mathit{rdc}, r_6 \rangle$	$\langle rdc, r_6 \rangle$	$\langle rdc, r_6 \rangle$	$\langle \mathit{rdc}, r_6 \rangle$	$\langle \mathit{rdc}, r_6 \rangle$		$\langle \mathit{rdc}, r_6 \rangle$	
$s_2$	$\langle \mathit{rdc}, r_8 \rangle$	$\langle \mathit{rdc}, r_8 \rangle$	$\langle rdc, r_8 \rangle$	$\langle rdc, r_8 \rangle$	$\langle rdc, r_8 \rangle$		$\langle \mathit{rdc}, r_8 \rangle$	
$s_3$	$\langle \mathit{rdc}, r_3 \rangle$	$\langle \mathit{rdc}, r_3 \rangle$	$\langle \mathit{rdc}, r_3 \rangle$	$\langle \mathit{shft}, s_{12} \rangle$	$\langle shft, s_{11} \rangle$		$\langle \mathit{rdc}, r_3 \rangle$	
$s_4$	accept	$\langle \mathit{shft}, s_8  angle$	$\langle \mathit{shft}, s_9 \rangle$					
$s_5$						$\langle \mathit{shft}, s_5  angle$		$\langle \mathit{shft}, s_2 \rangle$
$s_6$		$\langle \mathit{shft}, s_8  angle$	$\langle \mathit{shft}, s_9  angle$				$\langle \mathit{shft}, s_7  angle$	
$s_7$	$\langle rdc, r_7 \rangle$	$\langle \mathit{rdc}, r_7 \rangle$	$\langle rdc, r_7 \rangle$	$\langle rdc, r_7 \rangle$	$\langle rdc, r_7 \rangle$		$\langle rdc, r_7 \rangle$	
$s_8$						$\langle \mathit{shft}, s_5  angle$		$\langle shft, s_2 \rangle$
$s_9$						$\langle \mathit{shft}, s_5  angle$		$\langle shft, s_2 \rangle$
$s_{10}$	$\langle \mathit{rdc}, r_2 \rangle$	$\langle \mathit{rdc}, r_2 \rangle$	$\langle rdc, r_2 \rangle$	$\langle \mathit{shft}, s_{12} \rangle$	$\langle shft, s_{11} \rangle$		$\langle \mathit{rdc}, r_2 \rangle$	
$s_{11}$						$\langle \mathit{shft}, s_5  angle$		$\langle \mathit{shft}, s_2 \rangle$
$s_{12}$						$\langle \mathit{shft}, s_5  angle$		$\langle \mathit{shft}, s_2 \rangle$
$s_{13}$	$\langle rdc, r_4 \rangle$	$\langle rdc, r_4 \rangle$	$\langle rdc, r_4 \rangle$	$\langle rdc, r_4 \rangle$	$\langle rdc, r_4 \rangle$		$\langle rdc, r_4 \rangle$	
$s_{14}$	$\langle \mathit{rdc}, r_5 \rangle$	$\langle \mathit{rdc}, r_5 \rangle$	$\langle \mathit{rdc}, r_5 \rangle$	$\langle \mathit{rdc}, r_5 \rangle$	$\langle rdc, r_5 \rangle$		$\langle \mathit{rdc}, r_5 \rangle$	
$s_{15}$	$\langle rdc, r_1 \rangle$	$\langle rdc, r_1 \rangle$	$\langle rdc, r_1 \rangle$	$\langle shft, s_{12} \rangle$	$\langle shft, s_{11} \rangle$		$\langle rdc, r_1 \rangle$	

Table 11.1: 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 11.2: 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");
11
12
    actionTable["s1", "EOF"] := r6;
                                       actionTable["s2", "EOF"] := r8;
13
    actionTable["s1", "+" ] := r6;
                                       actionTable["s2", "+" ] := r8;
14
    actionTable["s1", "-" ] := r6; actionTable["s2", "-" ] := r8;
15
                                       actionTable["s2", "*" ] := r8;
    actionTable["s1", "*" ] := r6;
16
    actionTable["s1", "/" ] := r6;
                                       actionTable["s2", "/"
                                                              ] := r8;
17
                                       actionTable["s2", "(" ] := r8;
    actionTable["s1", ")" ] := r6;
18
                                       actionTable["s2", ")" ] := r8;
19
20
    actionTable["s3", "EOF"] := r3;
21
    actionTable["s3", "+" ] := r3;
22
    actionTable["s3", "-" ] := r3;
    actionTable["s3", "*" ] := Shift("s12");
24
    actionTable["s3", "/" ] := Shift("s11");
    actionTable["s3", ")" ] := r3;
26
    actionTable["s4", "EOF"] := Accept();
28
    actionTable["s4", "+" ] := Shift("s8");
    actionTable["s4", "-" ] := Shift("s9");
30
31
    actionTable["s5", "(" ] := Shift("s5");
32
    actionTable["s5", "int"] := Shift("s2");
33
34
    actionTable["s6", "+" ] := Shift("s8");
35
    actionTable["s6", "-" ] := Shift("s9");
36
    actionTable["s6", ")" ] := Shift("s7");
37
    actionTable["s7", "EOF"] := r7;
39
    actionTable["s7", "+" ] := r7;
    actionTable["s7", "-" ] := r7;
41
    actionTable["s7", "*" ] := r7;
    actionTable["s7", "/" ] := r7;
43
    actionTable["s7", ")" ] := r7;
```

Figure 11.3: Action table coded in SetlX, first part.

```
actionTable["s8", "(" ] := Shift("s5");
    actionTable["s8", "int"] := Shift("s2");
    actionTable["s9", "(" ] := Shift("s5");
    actionTable["s9", "int"] := Shift("s2");
    actionTable["s10", "EOF"] := r2;
    actionTable["s10", "+"] := r2;
    actionTable["s10", "-"] := r2;
    actionTable["s10", "*"] := Shift("s12");
10
    actionTable["s10", "/"] := Shift("s11");
11
    actionTable["s10", ")" ] := r2;
12
    actionTable["s11", "(" ] := Shift("s5");
14
    actionTable["s11", "int"] := Shift("s2");
16
    actionTable["s12", "(" ] := Shift("s5");
    actionTable["s12", "int"] := Shift("s2");
18
    actionTable["s13", "EOF"] := r4;
20
    actionTable["s13", "+" ] := r4;
21
    actionTable["s13", "-" ] := r4;
22
    actionTable["s13", "*" ] := r4;
23
    actionTable["s13", "/" ] := r4;
^{24}
    actionTable["s13", ")" ] := r4;
25
26
    actionTable["s14", "E0F"] := r5;
27
    actionTable["s14", "+" ] := r5;
    actionTable["s14", "-" ] := r5;
29
    actionTable["s14", "*" ] := r5;
    actionTable["s14", "/" ] := r5;
31
    actionTable["s14", ")" ] := r5;
33
   actionTable["s15", "E0F"] := r1;
   actionTable["s15", "+" ] := r1;
35
    actionTable["s15", "-" ] := r1;
   actionTable["s15", "*" ] := Shift("s12");
37
    actionTable["s15", "/" ] := Shift("s11");
    actionTable["s15", ")" ] := r1;
```

Figure 11.4: Action table coded in SETLX, second part.

```
gotoTable := {};
    gotoTable["s0", "E"] := "s4";
    gotoTable["s0", "P"] := "s3";
    gotoTable["s0", "F"] := "s1";
    gotoTable["s5", "E"] := "s6";
    gotoTable["s5", "P"] := "s3";
    gotoTable["s5", "F"] := "s1";
10
    gotoTable["s8", "P"] := "s15";
11
    gotoTable["s8", "F"] := "s1";
^{12}
13
    gotoTable["s9", "P"] := "s10";
14
    gotoTable["s9", "F"] := "s1";
15
    gotoTable["s11", "F"] := "s14";
17
    gotoTable["s12", "F"] := "s13";
```

Figure 11.5: Goto table coded in  $\operatorname{Setl}X$ .

#### 11.3 SLR-Parser

In diesem Abschnitt zeigen wir, wie wir für eine gegebene kontextfreie Grammatik G die im letzten Abschnitt verwendeten Funktionen

$$action: Q \times T \rightarrow Action$$
 and  $goto: Q \times V \rightarrow Q$ 

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 M wird hier unglücklicherweise der inhaltsleere Begriff "M verwendet.

**Definition 30 (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  $\beta$  gesehen hat und als nächstes versucht,  $\gamma$  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 11.2 auf Seite 131 gezeigten Grammatik aus, wobei wir diese Grammatik noch um ein neues Start-Symbol  $\widehat{s}$  und die Regel

$$\widehat{s} \to expr Eof$$

erweitern. Der Start-Zustand enthält offenbar die markierte Regel

$$\widehat{s} \to \varepsilon \bullet \exp \operatorname{Eof}$$
.

denn am Anfang versuchen wir ja, das Start-Symbol  $\widehat{s}$  herzuleiten. Die Komponente  $\varepsilon$  drückt aus, dass wir bisher noch nichts verarbeitet haben. Neben dieser markierten Regel muss der Start-Zustand dann außerdem die markierten Regeln

- 1.  $expr \rightarrow \varepsilon \bullet expr$  "+" product,
- 2.  $expr \rightarrow \varepsilon \bullet expr$  "-" product und
- 3.  $expr \rightarrow \varepsilon \bullet product$

enthalten, denn es könnte ja beispielsweise sein, dass wir die Regel

verwenden müssen, um die gesuchte expr herzuleiten. Genauso gut könnte es natürlich sein, dass wir stattdessen die Regel

$$expr \rightarrow product$$

benutzen müssen. Das erklärt, warum wir die markierte Regel

$$expr \rightarrow \varepsilon \bullet product$$

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

$$expr \rightarrow \varepsilon \bullet product$$

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 → 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.  $factor \rightarrow \bullet \text{ Number}$ .

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 31 (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

$$a \to \beta \bullet c \delta$$

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

$$c o ullet \gamma$$

ein Element der Menge K. Als Formel schreibt sich dies wie folgt:

$$(a \to \beta \bullet c \delta) \in \mathcal{K} \land (c \to \gamma) \in R \Rightarrow (c \to \bullet \gamma) \in \mathcal{K}$$

Die so definierte Menge  $\mathcal K$  ist eindeutig bestimmt und wird im Folgenden mit  $\mathit{closure}(\mathcal M)$  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

$$a \to \beta \bullet c \delta$$

aus der Menge  $\mathcal{K}$ , für die c eine syntaktische Variable ist und fügen dann für alle Regeln der Form  $c \to \gamma$  die neue markierte Regel

$$c \to \bullet \gamma$$

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 11.2 auf Seite 131 gezeigten Grammatik aus und betrachten die Menge

$$\mathcal{M} := \{ \textit{product} \rightarrow \textit{product "*"} \bullet \textit{factor} \}$$

Für die Menge  $closure(\mathcal{M})$  finden wir dann

$$\begin{array}{lll} \textit{closure}(\mathcal{M}) & = & \left\{ & \textit{product} \rightarrow \textit{product "*"} \bullet \textit{factor}, \\ & \textit{factor} \rightarrow \bullet \text{ "(" expr ")"}, \\ & \textit{factor} \rightarrow \bullet \text{ NUMBER} \\ & \left. \right\}. \end{array} \right.$$

Unser Ziel ist es, für eine gegebene kontextfreie Grammatik  $G = \langle V, T, R, s \rangle$  einen Shift-Reduce-Parser

$$P = \langle Q, q_0, action, goto \rangle$$

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  $\Gamma$  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  $closure(\mathcal{M}) = \mathcal{M}$  gilt. Wir definieren daher

$$Q:=\left\{\mathcal{M}\in 2^{\Gamma}\mid \mathit{closure}(\mathcal{M})=\mathcal{M}\right\}.$$

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  $\mathrm{EoF}$  zu der Grammatik

$$\widehat{G} = \langle V \cup \{\widehat{s}\}, T \cup \{\text{Eof}\}, R \cup \{\widehat{s} \to s \text{ Eof}\}, \widehat{s} \rangle.$$

Das Token EoF steht dabei als Abkürzung für <u>eend of file</u>. 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:

$$q_0 := \operatorname{closure} \left( \left\{ \widehat{s} \to \bullet s \operatorname{Eof} \right\} \right).$$

Als nächstes konstruieren wir die Funktion goto(). Die Definition lautet:

$$goto(\mathcal{M},c) := closure\Big(\big\{a o \beta \, c ullet \delta \mid (a o \beta ullet c \, \delta) \in \mathcal{M}\big\}\Big).$$

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  $\beta$  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  $goto(\mathcal{M},c)$ . Für die gleich folgende Definition der Funktion  $action(\mathcal{M},t)$  ist es nützlich, die Definition der Funktion  $action(\mathcal{M},t)$  ist es nützlich  $action(\mathcal{M},t)$  ist es nützlich actio

$$goto(\mathcal{M},t) := closure\Big(\big\{a \to \beta\,t \bullet \delta \mid (a \to \beta \bullet t\,\delta) \in \mathcal{M}\big\}\Big).$$

Als Letztes spezifizieren wir, wie die Funktion  $action(\mathcal{M},t)$  für eine Menge von markierten Regeln  $\mathcal{M}$  und ein Token t berechnet wird. Bei der Definition von  $action(\mathcal{M},t)$  unterscheiden wir vier Fälle.

1. Falls  $\mathcal{M}$  eine markierte Regel der Form  $a \to \beta \bullet t \delta$  enthält, setzen wir

$$action(\mathcal{M}, t) := \langle \mathtt{shift}, goto(\mathcal{M}, t) \rangle$$
,

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  $\beta$  erkannt. Ist nun das nächste Token im Eingabe-String tatsächlich das Token t, so kann der Parser dieses t lesen und geht dabei von dem Zustand  $a \to \beta \bullet t\delta$  in den Zustand  $a \to \beta t \bullet \delta$  über, der von der Funktion  $goto(\mathcal{M},t)$  berechnet wird. Insgesamt haben wir also

$$action(\mathcal{M},t) := \langle \mathtt{shift}, goto(\mathcal{M},t) \rangle \quad \mathsf{falls} \quad (a \to \beta \bullet t \, \delta) \in \mathcal{M}.$$

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

$$action(\mathcal{M}, t) := \langle \mathtt{reduce}, a \to \beta \rangle$$
,

denn in diesem Fall versucht der Parser ein a mit Hilfe der Regel  $a \to \beta$  zu erkennen und hat bereits  $\beta$  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  $t \in Follow(a)$ , so kann der Parser die Regel  $a \to \beta$  anwenden und den Symbol-Stack mit dieser Regel reduzieren. Wir haben dann

$$action(\mathcal{M}, t) := \langle \mathtt{reduce}, a \to \beta \rangle$$
 falls  $(a \to \beta \bullet) \in \mathcal{M}, \ a \neq \widehat{s} \ \mathsf{und} \ t \in \mathit{Follow}(a) \ \mathsf{gilt}.$ 

3. Falls  $\mathcal M$  die markierte Regel  $\widehat s \to s \bullet \mathrm{EoF}$  enthält und wir den zu parsenden String vollständig gelesen haben, setzen wir

$$action(\mathcal{M}, \text{Eof}) := \text{accept},$$

denn in diesem Fall versucht der Parser,  $\widehat{s}$  mit Hilfe der Regel  $\widehat{s} \to s \, \mathrm{EoF}$  zu erkennen und hat also bereits s erkannt. Ist nun das nächste Token im Eingabe-String das Datei-Ende-Zeichen  $\mathrm{EoF}$ , so liegt der zu parsende String in der durch die Grammatik G spezifizierten Sprache L(G). Wir haben daher

$$action(\mathcal{M}, \text{Eof}) := \text{accept}, \quad \text{falls } (\widehat{s} \to s \bullet \text{Eof}) \in \mathcal{M}.$$

4. In den restlichen Fällen setzen wir

$$action(\mathcal{M},t) := error.$$

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

$$c \to \delta \bullet \quad \text{mit } t \in follow(c).$$

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

$$c_1 \to \gamma_1 \bullet$$
 und  $c_2 \to \gamma_2 \bullet$ 

enthält und wenn gleichzeitig  $t \in Follow(c_1) \cap Follow(c_2)$  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 11.2 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

$$start \rightarrow expr Eof$$

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:

$$Q := \left\{ \mathcal{M} \in 2^{\Gamma} \mid \mathit{closure}(\mathcal{M}) = \mathcal{M} \right\}.$$

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:

$$s := start, e := expr, p := product, f := factor, N := NUMBER und  $s := EoF$ .$$

Wir beginnen mit dem Start-Zustand:

$$\begin{array}{lll} 1. & s_0 & := & \mathit{closure} \big( \{ & s \to \bullet \, e \, \$ & \} \big) \\ & = & \big\{ & s \to \bullet \, e \, \$, \\ & & e \to \bullet \, e \, \text{``+''} \, p, \, e \to \bullet \, e \, \text{`'-''} \, p, \, e \to \bullet \, p, \\ & & p \to \bullet \, p \, \text{``*''} \, f, \, p \to \bullet \, p \, \text{`'/''} \, f, \, p \to \bullet \, f, \\ & & f \to \bullet \, \text{``("e ")"} \, , \, f \to \bullet \, N \end{array} \right. \end{array}$$

2. 
$$s_1 := goto(s_0, f)$$
  
=  $closure(\{p \rightarrow f \bullet\})$   
=  $\{p \rightarrow f \bullet\}.$ 

3. 
$$s_2 := goto(s_0, N)$$
  
=  $closure(\{f \rightarrow N \bullet\})$   
=  $\{f \rightarrow N \bullet\}.$ 

4. 
$$s_3 := goto(s_0, p)$$
  
 $= closure(\{p \rightarrow p \bullet \text{ "*" } f, p \rightarrow p \bullet \text{ "/" } f, e \rightarrow p \bullet \})$   
 $= \{p \rightarrow p \bullet \text{ "*" } f, p \rightarrow p \bullet \text{ "/" } f, e \rightarrow p \bullet \}.$ 

5. 
$$s_4 := goto(s_0, e)$$
  
 $= closure(\{s \rightarrow e \bullet \$, e \rightarrow e \bullet \text{"+" } p, e \rightarrow e \bullet \text{"-" } p \})$   
 $= \{s \rightarrow e \bullet \$, e \rightarrow e \bullet \text{"+" } p, e \rightarrow e \bullet \text{"-" } p \}.$ 

6. 
$$s_5 := goto(s_0, \text{ "(")})$$
  
 $= closure(\{f \to \text{ "("} \bullet e \text{ ")"}\})$   
 $= \{f \to \text{ "("} \bullet e \text{ ")"}$   
 $e \to \bullet e \text{ "+" } p, \ e \to \bullet e \text{ "-" } p, \ e \to \bullet p,$   
 $p \to \bullet p \text{ "*" } f, \ p \to \bullet p \text{ "/" } f, \ p \to \bullet f,$   
 $f \to \bullet \text{ "("e ")" }, \ f \to \bullet N$  }.

7. 
$$s_6 := goto(s_5, e)$$
  
=  $closure(\{ f \rightarrow \text{ "(" } e \bullet \text{ ")" }, e \rightarrow e \bullet \text{ "+" } p, e \rightarrow e \bullet \text{ "-" } p \})$   
=  $\{ f \rightarrow \text{ "(" } e \bullet \text{ ")" }, e \rightarrow e \bullet \text{ "+" } p, e \rightarrow e \bullet \text{ "-" } p. \}$ 

8. 
$$s_7 := goto(s_6, ")")$$
  
=  $closure(\{f \to "("e")" \bullet \})$   
=  $\{f \to "("e")" \bullet \}.$ 

9. 
$$s_{8} := goto(s_{4}, "+")$$
 $= closure(\{e \rightarrow e "+" \bullet p\})$ 
 $= \{e \rightarrow e "+" \bullet p\}$ 
 $p \rightarrow \bullet p "*" f, p \rightarrow \bullet p "/" f, p \rightarrow \bullet f,$ 
 $f \rightarrow \bullet "("e ")", f \rightarrow \bullet N$  }.

10.  $s_{9} := goto(s_{4}, "-")$ 
 $= closure(\{e \rightarrow e "-" \bullet p\})$ 
 $= \{e \rightarrow e "-" \bullet p\}$ 
 $p \rightarrow \bullet p "*" f, p \rightarrow \bullet p "/" f, p \rightarrow \bullet f,$ 
 $f \rightarrow \bullet "("e ")", f \rightarrow \bullet N$  }.

11.  $s_{10} := goto(s_{9}, p)$ 
 $= closure(\{e \rightarrow e "-" p \bullet, p \rightarrow p \bullet "*" f, p \rightarrow p \bullet "/" f\}.$ 

12.  $s_{11} := goto(s_{3}, "/")$ 
 $= closure(\{p \rightarrow p "/" \bullet f\})$ 
 $= \{p \rightarrow p "/" \bullet f, f \rightarrow \bullet "("e ")", f \rightarrow \bullet N\}.$ 

13.  $s_{12} := goto(s_{3}, "*")$ 
 $= closure(\{p \rightarrow p "*" \bullet f\})$ 
 $= \{p \rightarrow p "*" \bullet f, f \rightarrow \bullet "("e ")", f \rightarrow \bullet N\}.$ 

14.  $s_{13} := goto(s_{12}, f)$ 
 $= closure(\{p \rightarrow p "*" f \bullet \})$ 
 $= \{p \rightarrow p "*" f \bullet \}.$ 

15.  $s_{14} := goto(s_{11}, f)$ 
 $= closure(\{p \rightarrow p "/" f \bullet \})$ 
 $= \{p \rightarrow p "/" f \bullet \}.$ 

16.  $s_{15} := goto(s_{8}, p)$ 
 $= closure(\{e \rightarrow e "+" p \bullet, p \rightarrow p \bullet "*" f, p \rightarrow p \bullet "/" f \}).$ 

Weitere Rechnungen führen nicht mehr auf neue Zustände. Berechnen wir beispielsweise  $goto(s_8, "("), so finden wir$ 

$$\begin{array}{ll} & \textit{goto}(s_8, \text{ "(")}) \\ = & \textit{closure}(\{f \rightarrow \text{ "("} \bullet e \text{ ")"} \}) \\ = & \{ f \rightarrow \text{ "("} \bullet e \text{ ")"} \\ & e \rightarrow \bullet e \text{ "+"} p, \ e \rightarrow \bullet e \text{ "-"} p, \ e \rightarrow \bullet p, \\ & p \rightarrow \bullet p \text{ "*"} f, \ p \rightarrow \bullet p \text{ "/"} f, \ p \rightarrow \bullet f, \\ & f \rightarrow \bullet \text{ "("} e \text{ ")"} , \ f \rightarrow \bullet N \end{array} \right. \} \\ = & s_5. \end{array}$$

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

$$p \rightarrow p \bullet$$
 "\*"  $f$ 

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

$$e \rightarrow e$$
 "+"  $p \bullet$ .

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

$$Follow(e) = \{ "+", "-", ")", "$" \},$$

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-Konflikte auftreten.

Als nächstes berechnen wir die Funktion action. Wir betrachten exemplarisch zwei Fälle.

1. Als erstes berechnen wir  $action(s_1, "+")$ . Es gilt

$$\begin{array}{lll} \mathit{action}(s_1, \text{ "+" }) & = & \mathit{action}(\{p \to f \bullet\}, \text{ "+" }) \\ & = & \langle \mathit{reduce}, p \to f \rangle, \end{array}$$

denn wir haben "+"  $\in$  Follow(P).

2. Als nächstes berechnen wir  $action(s_4, "+")$ . Es gilt

Würden wir diese Rechnungen fortführen, so würden wir die Tabelle 11.1 erhalten, denn wir haben die Namen der Zustände so gewählt, dass diese mit den Namen der entsprechenden Zustände in den Tabellen 11.1 und 11.2 übereinstimmen.

**Aufgabe 29**: Berechnen Sie die Menge der SLR-Zustände für die in Abbildung 11.6 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. ♦

Figure 11.6: Eine Grammatik für Boole'sche Ausdrücke in konjunktiver Normalform.

#### 11.3.1 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 11.7 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

lesen.

 $(1+2)*3 \quad \text{ als auch als } \quad 1+(2*3)$ 

Figure 11.7: Eine Grammatik mit Shift-Reduce-Konflikten.

Wir berechnen zunächst den Start-Zustand  $s_0$ .

$$\begin{array}{lll} s_0 & = & \mathit{closure}\big(\{s \to \bullet \, e \, \$\}\big) \\ & = & \big\{s \to \bullet \, e \, \$, \, \, e \to \bullet \, e \, \text{ "+"} \, \, e, \, \, e \to \bullet \, e \, \text{ "*"} \, \, e, \, \, e \to \bullet \, N \, \, \big\}. \end{array}$$

Als nächstes berechnen wir  $s_1 := goto(s_0, e)$ :

$$\begin{array}{lll} s_1 &=& \mathit{goto}(s_0,e) \\ &=& \mathit{closure}\big(\{s \rightarrow e \bullet \$\ e \rightarrow e \bullet \text{``+''}\ e,\ e \rightarrow e \bullet \text{``*''}\ e\,\}\big) \\ &=& \{s \rightarrow e \bullet \$,\ e \rightarrow e \bullet \text{``+''}\ e,\ e \rightarrow e \bullet \text{``*''}\ e\,\}\big) \end{array}$$

Nun berechnen wir  $s_2 := goto(s_1, "+")$ :

$$\begin{array}{lll} s_2 &=& \mathit{goto}(s_1, \text{``+''}) \\ &=& \mathit{closure}\big(\{e \rightarrow e \text{ ``+''} \bullet e, \; \}\big) \\ &=& \big\{e \rightarrow e \text{ ``+''} \bullet e, \; e \rightarrow \bullet e \text{ ``+''} \; e, \; e \rightarrow \bullet e \text{ ``*''} \; e, \; e \rightarrow \bullet N \; \big\}\big) \end{array}$$

Als nächstes berechnen wir  $s_3 := goto(s_2, e)$ :

```
\begin{array}{lll} s_3 &=& \gcd(s_2,e) \\ &=& \operatorname{closure} \bigl( \{e \rightarrow e \text{ "+" } e \bullet, \ e \rightarrow e \bullet \text{ "+" } e, \ e \rightarrow e \bullet \text{ "*" } e \} \bigr) \\ &=& \left\{ e \rightarrow e \text{ "+" } e \bullet, \ e \rightarrow e \bullet \text{ "+" } e, \ e \rightarrow e \bullet \text{ "*" } e \, \right\} ) \end{array}
```

Hier tritt bei der Berechnung von  $action(s_3, "*")$  ein Shift-Reduce-Konflikt auf, denn einerseits verlangt die markierte Regel

$$e \rightarrow e \bullet$$
 "\*"  $e$ ,

dass das Token "\*" auf den Stack geschoben wird, andererseits haben wir

$$Follow(e) = \{ "+", "*", "$" \},$$

so dass, falls das nächste zu lesende Token den Wert "\*" hat, der Symbol-Stack mit der Regel

$$e \rightarrow e$$
 "+"  $e \bullet$ ,

reduziert werden sollte.

**Aufgabe 30**: Bei der in Abbildung 11.7 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.

Figure 11.8: 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 11.8 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.

$$\begin{array}{lll} s_0 & = & \mathit{closure}\big(\{\widehat{s} \rightarrow \bullet \, s \, \$\}\big) \\ & = & \left\{\widehat{s} \rightarrow \bullet \, s \, \$, \, \, s \rightarrow \bullet \, a \, \text{``x''} \, \, a \, \text{``y''} \, \, , \, \, s \rightarrow \bullet \, b \, \text{``y''} \, \, b \, \text{``x''} \, \, , \, \, a \rightarrow \bullet \, \varepsilon, \, b \rightarrow \bullet \, \varepsilon \, \right\} \\ & = & \left\{\widehat{s} \rightarrow \bullet \, s \, \$, \, \, s \rightarrow \bullet \, a \, \, \text{``x''} \, \, a \, \, \text{``y''} \, \, , \, \, s \rightarrow \bullet \, b \, \, \text{``y''} \, \, b \, \, \text{``x''} \, \, , \, \, a \rightarrow \varepsilon \bullet, \, b \rightarrow \varepsilon \bullet \right\}, \end{array}$$

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

$$a \to \bullet \varepsilon$$
 und  $b \to \varepsilon \bullet$ .

Dieser Konflikt tritt bei der Berechnung von

$$action(s_0, "x")$$

auf, denn wir haben

$$Follow(a) = \{ \text{"x"}, \text{"y"} \} = Follow(b).$$

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 Follow(a) als auch der Menge Follow(b).

Es ist interessant zu bemerken, dass die obige Grammatik die LL(1)-Eigenschaft hat, denn es gilt

$$First(a "x" \ a "y") = \{ "x" \}, \quad First(b "y" \ b "x") = \{ "y" \}.$$

und daraus folgt sofort

$$First(a \text{ "x" } a \text{ "y" }) \cap First(b \text{ "y" } b \text{ "x" }) = \{\text{"x"}\} \cap \{\text{"y"}\} = \{\}.$$

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.

#### 11.4 Kanonische LR-Parser

Der Reduce-Reduce-Konflikt, der in der in Abbildung 11.8 gezeigten Grammatik auftritt, kann wie folgt gelöst werden: In dem Zustand

$$\begin{array}{lll} s_0 & = & \mathit{closure}\big(\{\widehat{s} \rightarrow \bullet \, s \, \$\}\big) \\ & = & \left\{\widehat{s} \rightarrow \bullet \, s \, \$, \; s \rightarrow \bullet \, a \; \text{``x''} \; a \; \text{``y''} \; , \; s \rightarrow \bullet \, b \; \text{``y''} \; b \; \text{``x''} \; , \; a \rightarrow \varepsilon \bullet, \; b \rightarrow \varepsilon \bullet \right. \end{array}$$

kommen die markierten Regeln  $a o \varepsilon ullet$  und  $b o \varepsilon ullet$  von der Berechnung des Abschlusses der Regeln

$$s \to \bullet \ a \text{ "x" } a \text{ "y"} \quad \text{ und } \quad s \to \bullet \ b \text{ "y" } b \text{ "x" }.$$

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 Follow(a) bzw. Follow(b) enthalten ist, denn jetzt berücksichtigen wir den Kontext, in dem die syntaktische Variable auftaucht. Damit können wir die Funktion  $action(s_0, "x")$  und  $action(s_0, "y")$  wie folgt definieren:

$$action(s_0, \text{"x"}) = \langle \text{reduce}, a \to \varepsilon \rangle$$
 und  $action(s_0, \text{"y"}) = \langle \text{reduce}, b \to \varepsilon \rangle$ .

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 32 (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  $\beta$  erkannt. Damit die Regel  $a \to \beta \gamma$  angewendet werden kann, muss nun  $\gamma$  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 33 (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 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 e.m.R.

```
c \to \bullet \gamma : \bigcup \{ \textit{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  $\alpha$  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  $\delta$  gegeben, der in der Regel  $a \to \beta \bullet c \, \delta$ : L auf das c folgt. Möglicherweise leitet  $\delta$  den leeren String  $\varepsilon$  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  $\mathcal K:=\operatorname{closure}(\mathcal M)$  iterativ erfolgen. Abbildung 11.9 zeigt die Berechnung von  $\operatorname{closure}(\mathcal M)$ . Der wesentliche Unterschied gegenüber der früheren Berechnung von  $\operatorname{closure}()$  ist, dass wir bei den e.m.R.s, die wir für eine Variable c mit in  $\operatorname{closure}(\mathcal M)$  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.

```
procedure closure(\mathcal{M}) {
\mathcal{K} := \mathcal{M};
\mathcal{K}^{-} := \{\};
\text{while } (\mathcal{K}^{-} \neq \mathcal{K}) \text{ } \{
\mathcal{K}^{-} := \mathcal{K};
\mathcal{K} := \mathcal{K} \cup \{(c \to \bullet \gamma : \bigcup \{\mathit{First}(\delta t) \mid t \in L\}) \mid (a \to \beta \bullet c \delta : L) \in \mathcal{K} \land (c \to \gamma) \in R\};
\mathcal{K} := \mathcal{K} \cup \{(c \to \bullet \gamma : \bigcup \{\mathit{First}(\delta t) \mid t \in L\}) \mid (a \to \beta \bullet c \delta : L) \in \mathcal{K} \land (c \to \gamma) \in R\};
\mathcal{K} := \mathcal{K} \cup \{(c \to \bullet \gamma : \bigcup \{\mathit{First}(\delta t) \mid t \in L\}) \mid (a \to \beta \bullet c \delta : L) \in \mathcal{K} \land (c \to \gamma) \in R\};
\mathcal{K} := \mathcal{K} \cup \{(c \to \bullet \gamma : \bigcup \{\mathit{First}(\delta t) \mid t \in L\}) \mid (a \to \beta \bullet c \delta : L) \in \mathcal{K} \land (c \to \gamma) \in R\};
\mathcal{K} := \mathcal{K} \cup \{(c \to \bullet \gamma : \bigcup \{\mathit{First}(\delta t) \mid t \in L\}) \mid (a \to \beta \bullet c \delta : L) \in \mathcal{K} \land (c \to \gamma) \in R\};
```

Figure 11.9: Berechnung von  $closure(\mathcal{M})$ 

**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  $\delta$  den leeren String  $\varepsilon$  ableiten kann oder nicht, denn es gilt

$$\bigcup \{\mathit{First}(\delta\,t) \mid t \in L\} = \left\{ \begin{array}{ll} \mathit{First}(\delta) \cup L & \mathsf{falls}\ \delta \Rightarrow^* \varepsilon; \\ \mathit{First}(\delta) & \mathsf{sonst.} \end{array} \right.$$

Die Berechnung von  $goto(\mathcal{M},t)$  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  $\mathrm{EoF}$  hinzu. Dann hat der Start-Zustand die Form

$$q_0 := \operatorname{closure}(\{\widehat{s} \to \bullet s : \operatorname{Eof}\}),$$

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.  $(a \to \beta \bullet t \, \delta : L) \in \mathcal{M} \implies \operatorname{action}(\mathcal{M}, t) := \langle \operatorname{shift}, \operatorname{goto}(\mathcal{M}, t) \rangle$ .
- 2.  $(a \to \beta \bullet : L) \in \mathcal{M} \land a \neq \hat{s} \land t \in L \implies \operatorname{action}(\mathcal{M}, t) := \langle \operatorname{reduce}, a \to \beta \rangle$ .
- 3.  $(\widehat{s} \to s \bullet : \text{Eof}) \in \mathcal{M} \implies action(\mathcal{M}, \text{Eof}) := \text{accept}.$
- 4. Sonst:  $action(\mathcal{M}, t) := error.$

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  $action(\mathcal{M},t)$  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  $t \in Follow(c)$  gilt und die Menge  $L_2$  ist in der Regel kleiner als die Menge 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  $Follow(a) \cap Follow(c)$  gibt und die Follow-Mengen sind oft größer als die Mengen  $L_1$  und  $L_2$ .

**Beispiel**: Wir greifen das Beispiel der in Abbildung 11.8 gezeigten Grammatik wieder auf und berechnen zunächst die Menge aller Zustände. Um die Schreibweise zu vereinfachen, schreiben wir an Stelle von "EoF" kürzer "\$".

```
\begin{array}{lll} 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\}. \end{array}
```

$$\begin{array}{lll} \textbf{2.} & s_1 & := & \textit{goto}(s_0, a) \\ & = & \textit{closure}\big(\big\{s \rightarrow a \bullet \text{``x''} a \text{``y''} : \$\big\}\big) \\ & = & \big\{s \rightarrow a \bullet \text{``x''} a \text{``y''} : \$\big\}. \end{array}$$

3. 
$$s_2 := goto(s_0, s)$$
  
 $= closure(\{\widehat{s} \rightarrow s \bullet : \$\})$   
 $= \{\widehat{s} \rightarrow s \bullet : \$\}.$ 

4. 
$$s_3 := goto(s_0, b)$$
  
 $= closure(\{s \rightarrow b \bullet \text{"y"}b\text{"x"}:\$\})$   
 $= \{s \rightarrow b \bullet \text{"y"}b\text{"x"}:\$\}.$ 

5. 
$$s_4 := goto(s_3, \text{"y"})$$
  
 $= closure(\{s \rightarrow b \text{"y"} \bullet b \text{"x"} : \$\})$   
 $= \{s \rightarrow b \text{"y"} \bullet b \text{"x"} : \$, b \rightarrow \bullet : \text{"x"}\}$ 

6. 
$$s_5 := goto(s_4, b)$$
  
 $= closure(\{s \rightarrow b \text{"y"} b \bullet \text{"x"} : \$\})$   
 $= \{s \rightarrow b \text{"y"} b \bullet \text{"x"} : \$\}.$ 

7. 
$$s_6 := goto(s_5, "x")$$
  
  $= closure(\{s \to b "y" b "x" \bullet : \$\})$   
  $= \{s \to b "y" b "x" \bullet : \$\}.$   
8.  $s_7 := goto(s_1, "x")$   
  $= closure(\{s \to a "x" \bullet a "y" : \$\})$   
  $= \{s \to a "x" \bullet a "y" : \$, a \to \bullet : "y"\}.$   
9.  $s_8 := goto(s_7, a)$   
  $= closure(\{s \to a "x" a \bullet "y" : \$\})$   
  $= \{s \to a "x" a \bullet "y" : \$\}.$   
10.  $s_9 := goto(s_8, "y")$   
  $= closure(\{s \to a "x" a "y" \bullet : \$\})$   
  $= \{s \to a "x" a "y" \bullet : \$\}.$ 

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

$$Follow(a) \cap Follow(b) = \{\text{"x"}, \text{"y"}\} \neq \{\}$$

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 31: Berechnen Sie die Menge der Zustände eines LR-Parsers für die folgende Grammatik:

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 Ir-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".

#### 11.5 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 11.10 gezeigt Grammatik, die ich dem *Drachenbuch* [ASUL06] entnommen habe. (Das "Drachenbuch" ist das Standardwerk im Bereich Compilerbau.)

Figure 11.10: Eine Grammatik aus dem Drachenbuch.

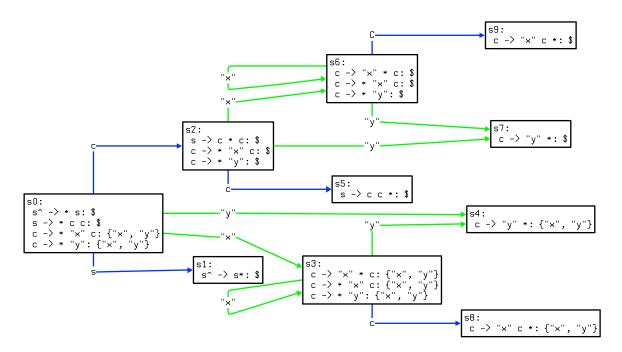


Figure 11.11: LR-Goto-Graph für die Grammatik aus Abbildung 11.10.

Abbildung 11.11 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

$$s_6 = \Big\{s \rightarrow \text{``x''} \bullet c: \text{``\$''}, c \rightarrow \bullet \text{``x''} \ c: \text{``\$''}, c \rightarrow \bullet \text{``y''} \ : \text{``\$''} \Big\},$$

und andererseits haben wir

$$s_3 = \Big\{s \rightarrow \text{``x''} \bullet c : \{\text{``x''}, \text{``y''}\}, c \rightarrow \bullet \text{``x''} \ c : \{\text{``x''}, \text{``y''}\}, c \rightarrow \bullet \text{``y''} \ : \{\text{``x''}, \text{``y''}\}\Big\}.$$

Offenbar entsteht die Menge  $s_3$  aus der Menge  $s_6$  indem überall "\$" durch die Menge {"x", "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

$$goto(s_3, c) = s_8$$
 und und  $goto(s_6, c) = s_9$ 

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 {"x", "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 11.12 gezeigten Goto-Graphen.

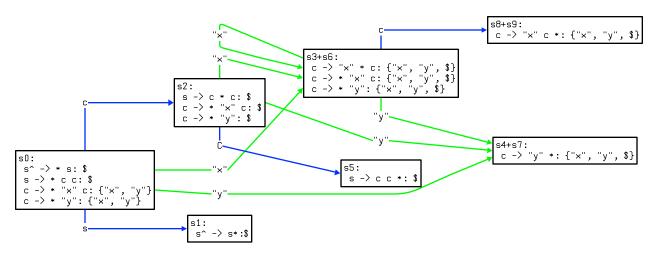


Figure 11.12: Der LALR-Goto-Graph für die Grammatik aus Abbildung 11.10.

Um die Beobachtungen, die wir bei der Betrachtung der in Abbildung 11.10 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:

$$core(\mathcal{M}) := \{ a \to \beta \bullet \gamma \mid (a \to \beta \bullet \gamma : L) \in \mathcal{M} \}.$$

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

$$goto(\mathcal{M},x) := closure\Big(\big\{a \to \beta \, x \bullet \gamma : L \mid (a \to \beta \bullet x \, \gamma : L) \in \mathcal{M}\big\}\Big).$$

definiert. Offenbar spielt die Menge der Folge-Token bei der Berechnung von  $goto(\mathcal{M}, x)$  keine Rolle, formal gilt für zwei e.m.R.-Mengen  $\mathcal{M}_1$  und  $\mathcal{M}_2$  und ein Symbol x die Formel:

$$core(\mathcal{M}_1) = core(\mathcal{M}_2) \Rightarrow core(goto(\mathcal{M}_1, x)) = core(goto(\mathcal{M}_2, x)).$$

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

$$\mathcal{M} \uplus \mathcal{N} := \{ a \to \beta \bullet \gamma : K \cup L \mid (a \to \beta \bullet \gamma : K) \in \mathcal{M} \ \land \ (a \to \beta \bullet \gamma : L) \in \mathcal{N} \}.$$

Diese Definition verallgemeinern wir zu einer Operation  $\biguplus$ , die auf einer Menge von Mengen von e.m.R.s definiert ist: Ist  $\Im$  eine Menge von Mengen von e.m.R.s, die alle den gleichen Kern haben, gilt also

$$\mathfrak{I} = \{\mathcal{M}_1, \cdots, \mathcal{M}_k\}$$
 mit  $core(\mathcal{M}_i) = core(\mathcal{M}_j)$  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  $\Delta$  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  $\Delta$  gegeben, deren Elemente den gleichen Kern haben:

$$\mathfrak{Q}:=\left\{\biguplus \mathfrak{I} \mid \mathfrak{I} \in 2^{\Delta} \wedge \forall \mathcal{M}, \mathcal{N} \in \mathfrak{I} : \mathit{core}(\mathcal{M})=\mathit{core}(\mathcal{N}) \wedge \mathsf{und} \ \mathfrak{I} \ \mathsf{maximal}\right\}.$$

Die Forderung " $\mathfrak I$  maximal" drückt in der obigen Definition aus, dass in  $\mathfrak I$  tatsächlich <u>alle</u> Mengen aus  $\Delta$  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  $goto(\mathcal{M},X)$  ändern muss, wenn  $\mathcal{M}$  ein Element der Menge  $\mathfrak Q$  der LALR-Zustände ist. Zur Berechnung von  $goto(\mathcal{M},X)$  berechnen wir zunächst die Menge

$$closure \Big( \big\{ A \to \alpha X \bullet \beta : L \mid (A \to \alpha \bullet X\beta : L) \in \mathcal{M} \big\} \Big).$$

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

$$\Big\{q\in\mathfrak{Q}\mid \mathit{core}(q)=\mathit{core}\big(\mathit{closure}\big(\big\{a\to\beta\,x\bullet\gamma:L\mid (a\to\beta\bullet x\,\gamma:L)\in\mathcal{M}\big\}\big)\big)\Big\}$$

genau ein Element und dieses Element ist der Wert von  $goto(\mathcal{M}, X)$ . Folglich können wir

$$\mathit{goto}(\mathcal{M},X) := \mathit{arb}\Big(\Big\{q \in \mathfrak{Q} \mid \mathit{core}(q) = \mathit{core}\Big(\mathit{closure}\Big(\big\{a \to \beta \, X \bullet \gamma : L \mid (a \to \beta \bullet X \, \gamma : L) \in \mathcal{M}\big\}\Big)\Big)\Big\}\Big)$$

setzen. Die hier verwendete Funktion arb() dient dazu, ein beliebiges Element aus einer Menge zu extrahieren. Da die Menge, aus der hier das Element extrahiert wird, genau ein Element enthält, ist  $goto(\mathcal{M},x)$  wohldefiniert. Die Berechnung des Ausdrucks  $action(\mathcal{M},t)$  ändert sich gegenüber der Berechnung für einen LR-Parser nicht.

### 11.6 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:

$$SLR$$
- $Sprache \subseteq LALR$ - $Sprache \subseteq kanonische LR$ - $Sprache$  (\*)

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.

### 11.6.1 SLR-Sprache $\subseteq$ 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 144 gesehen, dass die Grammatik

$$s \rightarrow a$$
 "x"  $a$  "y"  $\mid b$  "y"  $b$  "x" ,  $\quad a \rightarrow \varepsilon, \quad b \rightarrow \varepsilon$ 

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.

#### 11.6.2 LALR-Sprache ⊆ 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 11.13 zeigt eine solche Grammatik, die ich dem Drachenbuch entnommen habe.

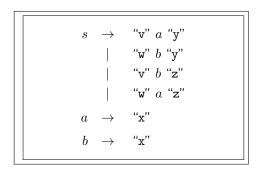


Figure 11.13: Eine kanonische LR-Grammatik, die keine LALR-Grammatik ist.

Wir berechnen zunächst die Menge der Zustände eines kanonischen LR-Parsers für diese Grammatik. Wir erhalten dabei die folgende Mengen von erweiterten markierten Regeln:

3. 
$$s_2 = goto(s_0, \text{ "v"}) = \{ s \rightarrow \text{ "v"} \bullet b \text{"z"} : \$, s \rightarrow \text{ "v"} \bullet a \text{"y"} : \$, a \rightarrow \bullet \text{"x"} : \text{"y"}, b \rightarrow \bullet \text{"x"} : \text{"z"} \}$$

4. 
$$s_3 = goto(s_0, \text{ "w"}) = \{ s \rightarrow \text{ "w"} \bullet a \text{"z"} : \$, s \rightarrow \text{ "w"} \bullet b \text{"y"} : \$, a \rightarrow \bullet \text{"x"} : \text{"z"}, b \rightarrow \bullet \text{"x"} : \text{"y"} \},$$

5. 
$$s_4 = goto(s_2, \text{ "x"}) = \{a \to \text{ "x"} \bullet : \text{"y"}, b \to \text{ "x"} \bullet : \text{"z"}\},$$

6. 
$$s_5 = goto(s_3, \text{ "x"}) = \{a \to \text{ "x"} \bullet : \text{ "z"}, b \to \text{ "x"} \bullet : \text{ "y"} \},$$

7. 
$$s_6 = goto(s_2, a) = \{s \to \text{"v"} a \bullet \text{"y"} : \$\},$$

8. 
$$s_7 = goto(s_6, "y") = \{s \rightarrow "v" a "y" \bullet : \$\},$$

9. 
$$s_8 = goto(s_2, b) = \{s \to \text{"v"} b \bullet \text{"z"} : \$\},$$

10. 
$$s_9 = goto(s_8, \text{"z"}) = \{s \to \text{"v"} b \text{"z"} \bullet : \$\},$$

11. 
$$s_{10} = goto(s_3, a) = \{s \rightarrow \text{``w''} a \bullet \text{``z''} : \$\},$$

12. 
$$s_{11} = goto(s_{10}, \text{ "z"}) = \{s \to \text{"w"} a \text{"z"} \bullet : \$\},$$

13. 
$$s_{12} = goto(s_3, b) = \{s \to \text{``w''} b \bullet \text{``y''} : \$\},$$

14. 
$$s_{13} = goto(s_{12}, \text{ "y"}) = \{s \rightarrow \text{"w"} b \text{"y"} \bullet : \$\}.$$

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

$$a \rightarrow$$
 "x" als auch mit  $b \rightarrow$  "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

$$core(s_4) = \{a \rightarrow \text{"x"} \bullet, b \rightarrow \text{"x"} \bullet\} = core(s_5).$$

Bei der Berechnung der LALR-Zustände werden diese beiden Zustände zu einem Zustand  $s_{\{4,5\}}$  zusammengefasst. Dieser neue Zustand hat die Form

$$s_{\{4,5\}} = \big\{A \rightarrow \text{``x"} \bullet : \{\text{``y"}, \text{``z"}\}, \ B \rightarrow \text{``x"} \bullet : \{\text{``y"}, \text{``z"}\}\big\}.$$

Hier gibt es offensichtlich einen Reduce-Reduce-Konflikt, denn einerseits haben wir

$$action(s_{\{4,5\}}, \text{"y"}) = \langle reduce, A \rightarrow \text{"x"} \rangle,$$

andererseits gilt aber auch

$$action(s_{\{4.5\}}, \text{"y"}) = \langle reduce, B \rightarrow \text{"x"} \rangle.$$

**Aufgabe 32**: 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.

#### 11.6.3 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 12

# 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.

# 12.1 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 12.1 auf Seite 156 zeigt eine *Cup*-Spezifikation, mit deren Hilfe arithmetische Ausdrücke ausgewertet werden können. In dieser *Cup*-Spezifikation sind die Schlüsselwörter unterstrichen.

- 1. Die gezeigte  $\mathrm{Cup} ext{-}\mathsf{Spezifikation}$  enthält keine Paket-Deklarationen.
- 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 12.4 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.

```
import java_cup.runtime.*;
    class ExprParser;
                        SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
    terminal
    terminal
                        LPAREN, RPAREN;
    terminal Integer
                        NUMBER;
    nonterminal
                        expr_list, expr_part;
    nonterminal Integer expr, prod, fact;
10
11
    start with expr_list;
12
    expr_list ::= expr_list expr_part
13
                | expr_part
14
15
16
    expr_part ::= expr:e {: System.out.println("result = " + e); :} SEMI
17
18
19
    expr ::= expr:e PLUS prod:p {: RESULT = e + p; :}
20
           expr:e MINUS prod:p {: RESULT = e - p; :}
                                   {: \underline{RESULT} = p;}
             prod:p
22
23
24
    prod ::= prod:p TIMES fact:f {: RESULT = p * f; :}
25
           prod:p DIVIDE fact:f {: RESULT = p / f; :}
26
                            fact:f {: RESULT = p % f; :}
             prod:p MOD
27
             fact:f
                                    {: RESULT = f;
28
29
30
    fact ::= LPAREN expr:e RPAREN {: RESULT = e;
31
           | MINUS fact:e
                                    {: RESULT = - e; :}
32
           NUMBER:n
                                    \{: RESULT = n;
33
34
```

Figure 12.1:  $\mathrm{Cup} ext{-}\mathsf{Spezifikation}$  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, \dots, t_n;
```

Durch diese Deklaration werden die Symbole  $t_1, \dots, t_n$  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, \dots, t_n;
```

Hierbei spezifiziert type den Typ, der den Terminalen  $t_1, \dots, t_n$  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 160 in Abbildung 12.4 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 \ ":}" \ \vdots \ "{:" \ body_n \ "{:" \ action_n \ ":}"}
```

#### Dabei gilt

- 1. var ist die syntaktische Variable, die von dieser Regel erzeugt wird.
- 2. body<sub>i</sub> ist der Rumpf der i-ten Grammatik-Regel, der aus einer Liste von Terminalen und syntaktischen Variablen besteht.
- 3. action<sub>i</sub> 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 Antle 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 Antle-Grammatiken. Der Grund dafür ist, dass bei Cup zum Scannen ein separates Werkzeug, nämlich *JFlex*, zum Scannen verwendet wird. Demgegenüber wird bei Antle der Scanner zusammen mit dem Parser in derselben Datei spezifiziert und das Werkzeug Antle 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 12.1 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 $0
```

Unter Windows können Sie sich stattdessen eine entsprechende .bat-Datei anlegen. Danach können Sie  $\mathrm{CuP}$  einfacher mit dem Befehl

```
cup calc.cup
```

aufrufen. Dieser Befehl erzeugt verschiedene Java-Dateien.

- 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.
- 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 12.2 auf Seite 159 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 12.3 auf Seite 159 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
3
      public static final int DIVIDE = 6;
      public static final int NUMBER = 10;
                                        = 7;
      public static final int MOD
      public static final int SEMI
                                        = 2;
                                        = 0;
      public static final int EOF
      public static final int PLUS
                                        = 3;
      public static final int error
                                        = 1;
10
      public static final int RPAREN = 9;
11
      public static final int TIMES
12
      public static final int LPAREN = 8;
      public static final String[] terminalNames = new String[] {
14
           "EOF",
           "error",
16
           "SEMI",
           "PLUS",
18
           "MINUS",
           "TIMES",
20
           "DIVIDE",
           "MOD",
22
           "LPAREN",
23
           "RPAREN",
24
           "NUMBER"
25
      };
26
    }
27
```

Figure 12.2: Die Klasse ExprParserSym.

```
import java_cup.runtime.*;
    import java.io.*;
2
    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();
11
            } catch (Exception e) {}
12
        }
13
    }
```

Figure 12.3: Die Klasse Calculator.

## 12.2 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 12.4 auf Seite 160 zeigt einen solchen Scanner, den wir jetzt im Detail diskutieren.

```
import java_cup.runtime.*;
2
    %%
    %char
    %line
    %column
    %cupsym ExprParserSym
    %cup
    %unicode
10
11
    %{
12
        private Symbol symbol(int type) {
13
             return new Symbol(type, yychar, yychar + yylength());
14
15
        private Symbol symbol(int type, Object value) {
17
             return new Symbol(type, yychar, yychar + yylength(), value);
19
    %}
20
21
    %eofval{
22
        return new Symbol(ExprParserSym.EOF);
23
    %eofval}
25
    %%
26
27
                   { return symbol( ExprParserSym.SEMI
28
                   { return symbol( ExprParserSym.PLUS
                                                            ); }
29
    "-"
                   { return symbol( ExprParserSym.MINUS
30
                   { return symbol( ExprParserSym.TIMES
    11 / 11
                   { return symbol( ExprParserSym.DIVIDE ); }
32
    "%"
                   { return symbol( ExprParserSym.MOD
33
    "("
                   { return symbol( ExprParserSym.LPAREN ); }
34
    ")"
                   { return symbol( ExprParserSym.RPAREN ); }
36
    0|[1-9][0-9]* { return symbol(ExprParserSym.NUMBER, new Integer(yytext())); }
37
38
                   { /* skip white space */ }
    [ \t \v \n \]
40
                   { throw new Error("Illegal character '" + yytext() +
    [^]
41
                                       "' at line " + yyline +
42
                                       ", column " + yycolumn); }
43
```

Figure 12.4: 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 Zeichen 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.
  - 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 12.2 auf Seite 159 zeigt diese
  - 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())

von Cup erzeugte Klasse.

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 12.3 gesehen, wie diese Klasse an den Parser angebunden wird.

## 12.3 Shift/Reduce und Reduce/Reduce-Konflikte

```
import java_cup.runtime.*;
                        PLUS, MINUS, TIMES, DIVIDE, MOD;
    terminal
3
    terminal
                        UMINUS, LPAREN, RPAREN;
    terminal Integer
                        NUMBER;
    nonterminal Integer expr;
9
    expr ::= expr PLUS
                          expr
             expr MINUS
                          expr
10
             expr TIMES
                          expr
11
             expr DIVIDE expr
12
             expr MOD
                          expr
             NUMBER
14
             MINUS expr
             LPAREN expr RPAREN
16
```

Figure 12.5: Cup-Spezifikation eines Parsers für arithmetische Ausdrücke

Wir betrachten nun ein weiteres Beispiel. Abbildung 12.5 zeigt eine  $\mathrm{Cup} ext{-}$ 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 12.5 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

$$R_1 := \Big( \mathsf{expr} o \mathsf{expr} \ ext{``-''} \ \mathsf{expr} ullet$$

und der markierten Regel

$$R_2 := \left( \mathsf{expr} o \, \mathsf{expr} \,ullet \, ext{"+" expr} 
ight)$$

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 ,
  [expr ::= expr (*) MINUS expr ,
  [expr ::= expr (*) DIVIDE expr ,
  [expr ::= expr (*) DIVIDE expr ,
  [expr ::= expr (*) TIMES expr ,
  [expr ::= expr (*) TIMES expr ,
  [expr ::= expr (*) PLUS expr ,
  [expr ::= expr (*) PLUS expr ,
  [expr ::= expr (*) PLUS expr ,
  [expr ::= expr MINUS expr (*) ,
  [expr ::= expr MINUS expr (*) ,
  [expr ::= expr MINUS expr (*) ]
}
```

Damit können wir jetzt den Shift/Reduce-Konflikt interpretieren: Im Zustand 12 ist der Parser entweder dabei, die Eingabe mit der Regel

$$expr \rightarrow expr$$
 "-"  $expr$ 

zu reduzieren, oder der Parser ist gerade dabei, die rechte Seite der Regel

$$expr o expr$$
 "+"  $expr$ 

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  $\mathrm{Cup}$  erzeugte Parser immer dafür, das Token auf den Stack zu schieben.

# 12.4 Operator-Präzendenzen

Es ist mit Cup möglich, Shift/Reduce-Konflikte durch die Angabe von *Operator-Präzedenzen* aufzulösen. Abbildung 12.6 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 " $^{\circ}$ ", 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 (4^3)^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 12.6 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

```
import java_cup.runtime.*;
    class ExprParser;
    terminal
                     SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD, POW;
                     UMINUS, LPAREN, RPAREN;
    terminal
    terminal Double NUMBER;
                       expr_list, expr_part;
    nonterminal
    nonterminal Double expr;
10
11
    precedence left
                      PLUS, MINUS;
12
    precedence left
                      TIMES, DIVIDE, MOD;
13
    precedence right UMINUS, POW;
14
15
    start with expr_list;
16
17
    expr_list ::= expr_list expr_part
18
               | expr_part
19
20
21
    expr_part ::= expr:e {: System.out.println("result = " + e); :} SEMI
22
23
    expr ::= expr:e1 PLUS expr:e2 {: RESULT = e1 + e2; :}
25
          | expr:e1 MINUS expr:e2 {: RESULT = e1 - e2; :}
          | expr:e1 TIMES expr:e2 {: RESULT = e1 * e2; :}
27
          expr:e1 DIVIDE expr:e2 {: RESULT = e1 / e2; :}
            expr:e1 MOD
                            expr:e2 {: RESULT = e1 % e2; :}
29
             expr:e1 POW
                             expr:e2 {: RESULT = Math.pow(e1, e2); :}
          | NUMBER:n
                                    \{: RESULT = n;
                                                         :}
31
                                                          :} %prec UMINUS
             MINUS expr:e
                                     {: RESULT = - e;}
32
             LPAREN expr:e RPAREN {: RESULT = e;
                                                          :}
33
34
```

Figure 12.6: Auflösung der Shift/Reduce-Konflikte durch Operator-Präzedenzen.

Wie erläutern nun, wie diese Bindungsstärken benutzt werden, um Shift/Reduce-Konflikte aufzulösen.  $\mathrm{Cup}$  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
$E \rightarrow E$ "+" $E$	1
$E \rightarrow E$ "-" $E$	1
$E \rightarrow E$ "*" $E$	2
$E \rightarrow E$ "/" $E$	2
$E \rightarrow E$ "^" $E$	3
$E \rightarrow$ "(" $E$ ")"	_
$E \to N$	_

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

$$r_1 = (a \to \beta \bullet o \delta : L_1)$$
 und  $r_2 = (c \to \gamma \bullet : L_2)$  mit  $o \in L_2$ 

vorkommen, so gibt es bei der Berechnung von

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:

$$action(s, o) = \langle shift, goto(s, o) \rangle$$
.

Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

mit den Grammatik-Regeln

$$E \rightarrow E$$
 "+"  $E \mid E$  "\*"  $E \mid \text{Number}$ 

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:

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

$$action(s, o) = \langle reduce, r_2 \rangle.$$

Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

1\*2+3

mit den Grammatik-Regeln

$$E \rightarrow E$$
 "+"  $E \mid E$  "\*"  $E \mid \text{Number}$ 

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:

$$\left\{ \begin{array}{cccc} E & \to & E \bullet \text{ "*"} E : \{\$, \text{ "*"}, \text{ "+"} \}, \\ E & \to & E \bullet \text{ "+"} E : \{\$, \text{ "*"}, \text{ "+"} \}, \\ E & \to & E \text{ "*"} E \bullet : \{\$, \text{ "*"}, \text{ "+"} \} \end{array} \right.$$

Wenn in diesem Zustand als nächstes Zeichen ein "+" gelesen wird, so soll der bisher gelesene String "1\*2" mit der Regel  $E \to E$  "\*" 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

$$action(s, o) = \langle reduce, r_2 \rangle.$$

Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

mit den Grammatik-Regeln

$$E \rightarrow E$$
 "+"  $E \mid E$  "-"  $E \mid \text{Number}$ 

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:

Wenn in diesem Zustand als nächstes Zeichen ein "-" gelesen wird, so soll der bisher gelesene String "1-2" mit der Regel  $E \to E$  "-" E reduziert werden, denn wir wollen von der Zahl 1 ja zunächst die Zahl 2 subtrahieren.

4.  $p(o) = p(c \rightarrow \gamma)$  und der Operator o ist rechts-assoziativ: In diesem Fall wird o auf den Stack geschoben:

$$action(s, o) = \langle shift, goto(s, o) \rangle.$$

Wenn wir diesen Fall verstehen wollen, reicht es aus, den String

mit den Grammatik-Regeln

$$E \to E \hat{\ } E \mid \text{Number}$$

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:

$$action(s, o) = error.$$

Diesen Fall verstehen Sie, wenn Sie versuchen, einen String der Form

mit den Grammatik-Regeln

$$E \rightarrow E$$
 "<"  $E \mid E$  "+"  $E \mid \text{Number}$ 

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  $p(c \rightarrow \gamma)$  ist undefiniert.

In diesem Fall erzeugt  $\mathrm{CuP}$  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  $\mathrm{Cup}$  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  $\mathrm{Cup}$  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 12.6 gezeigte Grammatik erzeugt wird.

1. Der Zustand Nummer 14 hat die in Abbildung 12.7 gezeigte Form. Hier gibt es unter anderem einen Shift/Reduce-Konflikt zwischen den beiden markierten Regeln

$$E \to E \bullet$$
 "+"  $E$  und  $E \to E$  "\*"  $E \bullet$ ,

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

$$E \to E \bullet$$
 "\*"  $E$  und  $E \to E$  "\*"  $E \bullet$ .

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.

 Der Zustand Nummer 18 hat die in Abbildung 12.8 gezeigte Form. Zunächst gibt es hier einen Shift/Reduce-Konflikt zwischen den Regeln

$$E \to E \bullet$$
 "^"  $E$  und  $E \to E$  "^"  $E \bullet$ ,

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

$$E \to E \bullet$$
 "+"  $E$  und  $E \to E$  "^"  $E \bullet$ ,

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 }]
10
    From state #14
11
     [term 2:REDUCE(with prod 7)] [term 3:REDUCE(with prod 7)]
12
     [term 4:REDUCE(with prod 7)] [term 5:REDUCE(with prod 7)]
13
     [term 6:REDUCE(with prod 7)] [term 7:REDUCE(with prod 7)]
14
     [term 8:SHIFT(to state 9)] [term 11:REDUCE(with prod 7)]
```

Figure 12.7: 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 }]
    }
10
    From state #18
11
     [term 2:REDUCE(with prod 10)] [term 3:REDUCE(with prod 10)]
12
     [term 4:REDUCE(with prod 10)] [term 5:REDUCE(with prod 10)]
     [term 6:REDUCE(with prod 10)] [term 7:REDUCE(with prod 10)]
14
     [term 8:SHIFT(to state 9)] [term 11:REDUCE(with prod 10)]
```

Figure 12.8: Der Zustand Nummer 18.

**Aufgabe 33**: 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.

### 12.5 Das Dangling-Else-Problem

```
class Dangling;
    terminal LPAREN, RPAREN, IF, ELSE, WHILE, LBRACE, RBRACE, ASSIGN, SEMI;
    terminal EQUAL, ID;
    nonterminal stmnt, stmntList, expr;
    start with stmnt;
9
    stmnt ::= IF LPAREN expr RPAREN stmnt
10
            | IF LPAREN expr RPAREN stmnt ELSE stmnt
11
            | WHILE LPAREN expr RPAREN stmnt
12
              LBRACE stmntList RBRACE
13
               ID ASSIGN expr SEMI
            15
    stmntList ::= stmntList stmnt
17
                  /* epsilon */
18
19
20
    expr ::= ID EQUAL ID
21
            | ID
22
23
```

Figure 12.9: 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 12.9 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 12.10 ausschnittsweise gezeigten Shift/Reduce-Konflikt.

Der Konflikt entsteht bei der Berechnung von action ("state #10", "else") zwischen den beiden markierten Regeln

```
stmnt \rightarrow "if" "(" expr ")" stmnt \bullet und stmnt \rightarrow "if" "(" expr ")" stmnt \bullet "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 12.9 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:

```
Warning : *** Shift/Reduce conflict found in state #10
between stmnt ::= IF LPAREN expr RPAREN stmnt (*)
and stmnt ::= IF LPAREN expr RPAREN stmnt (*) ELSE stmnt
under symbol ELSE
Resolved in favor of shifting.

lalr_state [10]: {
   [stmnt ::= IF LPAREN expr RPAREN stmnt (*) , {EOF IF ELSE WHILE LBRACE RBRACE ID }]
   [stmnt ::= IF LPAREN expr RPAREN stmnt (*) ELSE stmnt , {EOF IF ELSE WHILE LBRACE RBRACE ID }]
}
```

Figure 12.10: 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 12.9 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 12.11 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.

```
stmnt ::= matchedStmnt
              unmatchedStmnt
            1
2
    matchedStmnt ::= IF LPAREN expr RPAREN matchedStmnt ELSE matchedStmnt
                   | WHILE LPAREN expr RPAREN matchedStmnt
6
                      LBRACE stmntList RBRACE
                      ID ASSIGN expr SEMI
10
    unmatchedStmnt ::= IF LPAREN expr RPAREN stmnt
11
                     IF LPAREN expr RPAREN matchedStmnt ELSE unmatchedStmnt
12
                     WHILE LPAREN expr RPAREN unmatchedStmnt
13
15
    stmntList ::= stmntList stmnt
                  /* epsilon */
17
18
19
    expr ::= ID EQUAL ID
            | ID
21
            ;
22
```

Figure 12.11: 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 12.12 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 12.10 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.

```
precedence nonassoc IF;
    precedence nonassoc ELSE;
    start with stmnt;
    stmnt ::= IF LPAREN expr RPAREN stmnt
                                                         %prec IF
            | IF LPAREN expr RPAREN stmnt ELSE stmnt
              WHILE LPAREN expr RPAREN stmnt
              LBRACE stmntList RBRACE
9
               ID ASSIGN expr SEMI
10
11
12
    stmntList ::= stmntList stmnt
13
                  /* epsilon */
14
16
    expr ::= ID EQUAL ID
17
            | ID
18
19
            ;
```

Figure 12.12: 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 12.13 auf Seite 173 zeigt die Grammatik no-dangling.cup, bei der durch die Verwendung von geschweiften Klammern erreicht wurde, dass keine Mehrdeutigkeit mehr auftritt.

# 12.6 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 12.5 in der Grammatik

```
stmnt ::= IF LPAREN expr RPAREN block
1
            | IF LPAREN expr RPAREN block ELSE block
2
              WHILE LPAREN expr RPAREN stmnt
3
              block
               ID ASSIGN expr SEMI
            1
    block ::= LBRACE stmntList RBRACE
9
10
    stmntList ::= stmntList stmnt
11
                | /* epsilon */
12
13
14
    expr ::= ID EQUAL ID
            | ID
16
17
            ;
```

Figure 12.13: 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.

#### 12.6.1 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 12.14 zeigt die Grammatik lookAhead.cup<sup>1</sup>, 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 \rightarrow \text{"X"} \bullet : \text{"U"}, c \rightarrow \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 \rightarrow b "U" "V"
```

verwendet werden und folglich ist das "X" als ein b zu interpretieren. Ist das zweite Token hinter dem "X" hingegen

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.

Figure 12.14: Eine eindeutige Grammatik ohne die LR(1)-Eigenschaft.

ein "W", so ist die zu verwendende Regel

```
a \rightarrow c "U" "W"
```

und folglich ist das "X" als c zu lesen.

Figure 12.15: Eine zu 12.14 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 \rightarrow \text{``X'' ''U''} \bullet : \text{``V''}, c \rightarrow \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 \rightarrow \text{"X" "U"}.
```

ist das nächste Token hingegen der Buchstabe "W", so nehmen wir stattdessen die Regel

```
c \rightarrow \text{``X'' ``U''} \bullet : \text{``W''}.
```

Abbildung 12.15 zeigt die entsprechend modifizierte Grammatik solved.cup,

#### 12.6.2 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 12.16 gezeigte Grammatik mysterious.cup habe ich dem *Bison*-Handbuch entnommen.

Ubersetzen wir diese Grammatik mit CUP, so erhalten wir unter anderem den folgenden Zustand:

```
lalr_state [1]: {
```

```
ID, COMMA, COLON;
    terminal
    nonterminal def, param_spec, return_spec, type, name_list, name;
    def
         ::= param_spec return_spec COMMA
5
          ;
6
    param_spec
         ::= type
          | name_list COLON type
9
10
11
    return_spec
         ::= type
12
          | name COLON type
13
14
15
     type
         ::= ID
16
17
    name
18
         ::= ID
19
20
21
    name_list
         ::= name
22
          1
             name COMMA name_list
23
24
```

Figure 12.16: 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 12.17 gezeigt sind.

Analysieren wir die Zustände, so stellen wir fest, dass beim Ubergang 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 = goto(s_0, ID).
```

2. Der Zustand  $s_8$  entsteht, wenn das Token ID im Zustand  $s_2$  gelesen wird, wir haben

```
s_8 = goto(s_2, ID).
```

Die Idee zur Auflösung des Konflikts ist, dass wir den Zustand  $s_2$  so ändern, dass die Kerne von  $s_7 = goto(s_0, ID)$  und  $s_8 = goto(s_0, ID)$  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 \rightarrow <*> def: [\$],
1
            def -> <*> param_spec return_spec ',': [$],
2
            param_spec -> <*> name_list ':' type: [ID],
3
            param_spec -> <*> type: [ID],
            name_list -> <*> name: [':'],
            name_list -> <*> name ',' name_list: [':'],
            name -> <*> ID: [',', ':'],
            type -> <*> ID: [ID]
9
    s2 = { def -> param_spec <*> return_spec ',': [$],
10
            return_spec -> <*> name ':' type: [','],
11
            return_spec -> <*> type: [','],
12
            name -> <*> ID: [':'],
13
            type -> <*> ID: [',']
14
    s7 = \{ name \rightarrow ID < *>: [',', ':'], 
16
            type -> ID <*>: [ID]
17
18
    s8 = { name -> ID <*>: [':'],
19
            type -> ID <*>: [',']
20
          }
```

Figure 12.17: LR-Zustände der in Abbildung 12.16 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 12.18 gezeigten Grammatik  $mysterious\_solved.cup$  ab, indem wir eine zusätzliche Grammatik-Regel

```
return\_spec 	o 	exttt{ID} 	exttt{ 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 12.19 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.

## 12.7 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 11.7 auf Seite 143 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 12.4 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 ','
2
    param_spec
3
         : type
         | name_list ':' type
    return_spec
         : type
         | name ':' type
9
         | ID BOGUS
                          // this never happens
10
11
    type: ID
12
13
    name: ID
14
    name_list
16
17
         : name
         | name ',' name_list
18
```

Figure 12.18: Auflösung des mysteriösen Reduce/Reduce-Konflikts.

```
s0 = \{ S \rightarrow <*> def: [\$],
            def -> <*> param_spec return_spec ',': [$],
2
            name -> <*> ID: [',', ':'],
3
            name_list -> <*> name: [':'],
            name_list -> <*> name ',' name_list: [':'],
            param_spec -> <*> name_list ':' type: [ID],
            param_spec -> <*> type: [ID],
            type -> <*> ID: [ID]
9
    s2 = { def -> param_spec <*> return_spec ',': [$],
10
            name -> <*> ID: [':'],
11
            return_spec -> <*> ID BOGUS: [','],
12
            return_spec -> <*> name ':' type: [','],
13
            return_spec -> <*> type: [','],
14
            type -> <*> ID: [',']
15
16
    s7 = \{ name \rightarrow ID <*>: [',', ':'], \}
17
            type -> ID <*>: [ID]
18
19
    s8 = \{ name -> ID <*>: [':'],
20
            return_spec -> ID <*> BOGUS: [','],
21
            type -> ID <*>: [',']
22
          }
23
```

Figure 12.19: Einige Zustände der in Abbildung 12.18 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 12.20 gezeigte Grammatik lambda.cup. Diese Grammatik beschreibt drei Arten von Ausdrücken:

1.  $\lambda$ -Ausdrücke haben syntaktisch die Form

```
[x_1, \cdots, x_n] \mapsto e.
```

Ein solcher Ausdruck steht für eine Funktion, die n Argumente  $x_1, \dots, x_n$  verarbeitet und als Ergebnis den Ausdruck e zurück liefert, wobei der Ausdruck e im Allgemeinen von den Parametern  $x_1, \dots, x_n$  abhängen wird. Ein konkretes Beispiel wäre etwa der  $\lambda$ -Ausdruck

$$[x, y] \mapsto [x, y, x].$$

- 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.

```
terminal MAPSTO, LBRACKET, RBRACKET, COMMA, IDENTIFIER;
    nonterminal expr, exprList, lambdaDefinition, identifierList;
3
    expr ::= lambdaDefinition
             IDENTIFIER
             LBRACKET exprList RBRACKET
    lambdaDefinition
9
         ::= LBRACKET identifierList RBRACKET MAPSTO expr
10
          ;
11
    identifierList
12
          ::= IDENTIFIER COMMA identifierList
13
          | IDENTIFIER
14
15
    exprList
16
         ::= expr COMMA exprList
17
              expr
18
```

Figure 12.20: Eine Cup-Grammatik für  $\lambda$ -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 12.21 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 	o 	ext{IDENTIFIER}
```

reduziert werden darf, wenn das Look-Ahead-Token ein Komma ist. Um zu entscheiden, ob der Parser versucht eine Liste von IDENTIFIERn 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 	o 	ext{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.

```
lalr_state [6]: {
    [identifierList ::= IDENTIFIER (*) COMMA identifierList , {RBRACKET }]
    [expr ::= IDENTIFIER (*) , {RBRACKET COMMA }]
    [identifierList ::= IDENTIFIER (*) , {RBRACKET }]
}
```

Figure 12.21: Der Zustand mit der Nummer 6 zu der Grammatik aus Abbildung 12.20

Wir können den Shift/Reduce-Konflikt lösen, indem wir die Grammatik wie in der in Abbildung 12.22 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.

```
MAPSTO, LBRACKET, RBRACKET, COMMA, IDENTIFIER;
    terminal
1
2
    nonterminal expr, exprList, lambdaDefinition;
3
    expr ::= lambdaDefinition
             IDENTIFIER
             LBRACKET exprList RBRACKET
9
    lambdaDefinition
         ::= LBRACKET exprList RBRACKET MAPSTO expr
11
12
13
    exprList
14
         ::= expr COMMA exprList
15
          expr
16
17
```

Figure 12.22: Die verallgemeinerte Grammatik für  $\lambda$ -Ausdrücke.

## Chapter 13

# 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.
- 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.

### 13.1 Eine Beispielsprache

Wir stellen jetzt die Sprache  $\mathrm{TTL}$  (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  $\mathrm{TTL}$  ist sehr einfach gehalten, damit wir uns auf die wesentlichen Ideen der Typ-Überprüfung konzentrieren können. Daher können wir in  $\mathrm{TTL}$  auch keine wirklichen Programme schreiben, sondern einzig und allein überprüfen, ob Ausdrücke wohlgetypt sind.

```
type list(X) := nil + cons(X, list(X));

signature concat: list(T) * list(T) -> list(T);
signature x: int;
signature y: int;
signature z: int;
concat(nil, nil): list(int);
concat(cons(x, nil), cons(y, cons(z, nil))): list(int);
```

Figure 13.1: Ein TTL-Beispiel-Programm

Abbildung 13.1 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 list(X). Das X ist hier der Typ-Parameter, der später durch einen konkreten Typ wie z.B. int, string oder list(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 List(X):

```
\mathit{nil} \in \mathit{list}(X)
```

2. Induktions-Schritt: Falls a ein Element der Menge X und l ein Element der Menge list(X) ist, dann ist der Term cons(a, l) ebenfalls ein Element der Menge list(X):

```
a \in X \land l \in \mathit{list}(X) \to \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  $\mathtt{list}(T)$  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

```
concat(nil, nil) und concat(cons(x, nil), cons(y, cons(z, nil)))
```

angegeben und es wird behauptet, dass diese den Typ list(int) haben. Die Aufgabe der Typ-Uberprüfung besteht darin, diese Aussage zu verifizieren.

```
grammar ttl;
               : typeDef* signature* typedTerm*
5
    typeDef
               : 'type' FCT ':=' type ('+' type)* ';'
               | 'type' FCT '(' PARAM (', ' PARAM)* ')' ':=' type ('+' type)* ';'
               : FCT '(' type (',' type)* ')'
    type
               | FCT
11
               | PARAM
12
13
14
    signature : 'signature' FCT ':' type ('*' type)* '->' type ';'
15
               | 'signature' FCT ':' type ';'
16
17
18
               : FCT '(' term (',' term)* ')'
    term
19
               | FCT
20
21
22
    typedTerm : term ':' type ';'
23
24
25
    PARAM
             : ('A'..'Z')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
26
             : ('a'...'z')('a'...'z'|'A'...'Z'|'__'|'0'...'9')*;
    FCT
27
```

Figure 13.2: Eine EBNF-Grammatik für die getypte Beispielsprache

Die genaue Syntax der Sprache TTL wird durch die in Abbildung 13.2 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  $\mathtt{list}(X)$ . 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.

## 13.2 Grundlegende Begriffe

Als erstes definieren wir den Begriff der Signatur eines Funktions-Zeichens. Die *Signatur* eines Funktionszeichens legt fest,

 $\Diamond$ 

- 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

$$f: \sigma_1 \times \cdots \times \sigma_n \to \rho$$

an. Damit spezifizieren wir:

- 1. Die Funktion f erwartet n Argumente.
- 2. Das *i*-te Argument hat den Typ  $\sigma_i$ .
- 3. Das von der Funktion berechnete Ergebnis hat den Typ  $\varrho$ .

Als nächstes definieren wir, was wir unter einem Typ verstehen wollen. Anschaulich sind das Ausdrücke wie

$$map(K, V)$$
, double, oder  $list(int)$ .

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

$$\mathbb{K} = \{ \textit{map}, \textit{list}, \textit{int}, \textit{double} \} \quad \text{und} \quad \mathbb{P} = \{ K, V \}$$

setzen. Zusätzlich muss noch eine Funktion

$$arity: \mathbb{K} \to \mathbb{N}$$

gegeben sein, die für jeden Typ-Konstruktor festlegt, wieviel Argumente er erwartet. In dem obigen Beispiel hätten wir

$$arity(map) = 2$$
,  $arity(list) = 1$ ,  $arity(int) = 0$  und  $arity(double) = 0$ .

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 arity(c) = 0, so ist auch c ein Typ:

$$c \in \mathbb{K} \land \operatorname{arity}(c) = 0 \Rightarrow c \in \mathcal{T}.$$

3. Ist f ein n-stelliger Typ-Konstruktor und sind  $\tau_1, \dots, \tau_n$  Typen, so ist auch  $f(\tau_1, \dots, \tau_n)$  ein Typ:

$$f \in \mathbb{K} \land arity(f) = n \land n > 0 \land \tau_1 \in \mathcal{T} \land \cdots \land \tau_n \in \mathcal{T} \Rightarrow f(\tau_1, \cdots, \tau_n) \in \mathcal{T}.$$

**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,
- list(double),
- 4. list(V),
- 5. map(K, list(double)).

**Definition 34 (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  $i \in \{1, \dots, n\}$ ,
- 2.  $\tau_i$  is a type for all  $i \in \{1, \dots, n\}$ ,
- 3. the type parameters occurring in  $\sigma$  are pairwise distinct, that is we have

$$i \neq j \rightarrow X_i \neq X_j$$
 for all  $i, j \in \{1, \dots, n\}$ .

If  $\sigma = \{\langle X_1, \tau_1 \rangle, \cdots, \langle X_n, \tau_n \rangle\}$  is a parameter substitution, then  $\sigma$  is written as

$$\sigma = [X_1 \mapsto \tau_1, \cdots, X_n \mapsto \tau_n].$$

In this case, the domain of  $\sigma$  is defined as

$$dom(\sigma) = \{X_1, \cdots, X_n\}.$$

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  $\tau$  is a type and  $\vartheta = [X_1 \mapsto \tau_1, \cdots, X_n \mapsto \tau_n]$  is a substitution, then  $\tau\vartheta$  is the type that we get if we replace all occurrences of  $X_i$  in  $\tau$  by  $\tau_i$ , The formal definition follows.

#### **Definition 35 (Application of a Substitution)**

If  $\tau$  is a type and  $\vartheta = [X_1 \mapsto \tau_1, \cdots, X_n \mapsto \tau_n]$  is a substitution, then the application of  $\vartheta$  to  $\tau$  (written  $\tau\vartheta$ ) is defined by induction on  $\tau$ :

- 1. If  $\tau$  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  $Y \not\in \{X_1, \cdots, X_n\}$ . Then

$$Y\vartheta := Y$$
.

2. Otherwise  $\tau$  must have the form  $\tau = f(\sigma_1, \dots, \sigma_m)$ . 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  $\vartheta$  as

$$\vartheta := [X_1 \mapsto \mathtt{double}, X_2 \mapsto \mathtt{list}(\mathtt{int})].$$

Then we have the following:

- 1.  $X_3\vartheta=X_3$ ,
- 2.  $list(X_2)\vartheta = list(list(int)),$
- 3.  $map(X_1, set(X_2))\vartheta = map(double, set(list(int)))$ .

Next, we show how substitutions can be composed.

#### Definition 36 (Composition of Substitutions) If

$$\vartheta = \begin{bmatrix} X_1 \mapsto \sigma_1, \cdots, X_m \mapsto \sigma_m \end{bmatrix} \quad \text{ and } \quad \eta = \begin{bmatrix} Y_1 \mapsto \tau_1, \cdots, Y_n \mapsto \tau_n \end{bmatrix}$$

are substitutions such that  $dom(\vartheta) \cap dom(\eta) = \{\}$ , then we define the composition  $\vartheta \eta$  of  $\vartheta$  and  $\eta$  as

$$\vartheta \eta := [X_1 \mapsto \sigma_1 \eta, \cdots, X_m \mapsto \sigma_m \eta, Y_1 \mapsto \tau_1, \cdots, Y_n \mapsto \tau_n].$$

Example: Define

$$\vartheta := [X_1 \mapsto \mathtt{double}, \, X_2 \mapsto \mathtt{list}(\mathtt{X}_3)]$$
 and

$$\eta := [X_3 \mapsto \mathtt{map}(\mathtt{int},\mathtt{double}), X_4 \mapsto \mathtt{char}].$$

Then we have

$$\vartheta \eta = \big[ X_1 \mapsto \mathtt{double}, \ X_2 \mapsto \mathtt{list}(\mathtt{map}(\mathtt{int},\mathtt{double})), \ X_3 \mapsto \mathtt{map}(\mathtt{int},\mathtt{double}), \ X_4 \mapsto \mathtt{char} \big]. \qquad \diamond$$

**Exercise 34**: How would we have to change the definition of the composition of  $\vartheta$  and  $\eta$  if we drop the condition  $dom(\vartheta) \cap dom(\eta) = \{\}$ ?

The following theorem is a consequence of the previous definition.

#### Theorem 37 (Associativity of Composition)

If  $\tau$  is a type and  $\vartheta$  and  $\eta$  are substitutions such that  $dom(\vartheta) \cap dom(\eta) = \{\}$  holds, then we have

$$(\tau \vartheta) \eta = \tau (\vartheta \eta).$$

This theorem is proved by an easy induction on  $\tau$ .

Next, we formalize the notion of a term t being of type  $\tau$ .

#### **Definition 38** $(t:\tau)$

For a term t and a type  $\tau$  the relation t:  $\tau$  (read: t has type  $\tau$ ) is defined by induction on t.

1. If c is a nullary function symbol with signature  $c:\tau$  and if  $\vartheta$  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.  $\vartheta$  is a substitution such that we have  $t_i : \sigma_i \vartheta$ .

Then 
$$f(t_1, \dots, t_n) : \varrho \vartheta$$
.

**Examples**: The following examples assume that the types and signatures are given as in Figure 13.1 on page 181.

1. We have nil: list(int), because nil has the signature

Therefore, defining

$$\vartheta = [X \mapsto \mathtt{int}]$$

gives  $list(X)\vartheta = list(int)$  showing the claim.

2. Next, we show

The signature of concat is

$$concat : list(T) \times list(T) \rightarrow list(T).$$

Define

$$\vartheta = [T \mapsto \mathtt{int}].$$

We have already seen that

holds. Because of  $list(T)\vartheta = list(int)$  this shows the claim.

### 13.3 A Type Checking Algorithm

Assume a term  $t=f(t_1,\cdots,t_n)$  and a type  $\tau$  is given and we want to check whether  $t:\tau$  holds. Furthermore, assume that the signature of f is given as

$$f: \sigma_1 \times \cdots \times \sigma_n \to \rho$$
.

According to the definition of  $t:\tau$  the term t has type  $\tau$  iff there is a parameter substitution  $\vartheta$  such that  $\varrho\vartheta=\tau$  and  $t_i:\sigma_i\vartheta$ . Therefore

$$f(t_1, \dots, t_n) : \tau \iff \exists \vartheta \in \text{SUBST} : (\varrho \vartheta = \tau \land \forall i \in \{1, \dots, n\} : t_i : \sigma_i \vartheta)$$

The problem of type checking is to compute the substitution  $\vartheta$  or to show that  $\vartheta$  can not exist.

**Examples**: We discuss the previous examples (that is nil: list(int) and concat(nil, nil): list(int)) again using the formula

$$f(t_1, \dots, t_n) : \tau \iff \exists \vartheta \in \text{SUBST} : (\varrho \vartheta = \tau \land \forall i \in \{1, \dots, n\} : t_i : \sigma_i \vartheta).$$

1. Lets prove nil: list(int) again. We have the signature

Therefore, we have

$$nil: \mathtt{list}(\mathtt{int}) \Leftrightarrow \exists \vartheta \in \mathtt{SUBST}: (\mathtt{list}(X)\vartheta = \mathtt{list}(\mathtt{int})).$$

Obviously, we can define

$$\vartheta = [X \mapsto \mathtt{int}].$$

Then, the equation  $list(X)\vartheta = list(int)$  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

$$concat : list(T) \times list(T) \rightarrow list(T).$$

Therefore, our claim is equivalent to

$$\Leftrightarrow \exists \vartheta \in \text{SUBST} : (\text{list}(T)\vartheta = \text{list}(\text{int}) \land \textit{nil} : \textit{list}(T)\vartheta \land \textit{nil} : \textit{list}(T)\vartheta).$$

The equation  $list(T)\vartheta = list(int)$  is solved by the substitution

$$\vartheta = [T \mapsto \mathtt{int}]$$

and therefore the correctness of

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

$$typeEqs: Term \times Type \rightarrow set(Equation).$$

For a term t and a type  $\tau$ 

$$typeEqs(t, \tau)$$

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:

$$\left(f:\sigma_1\times\cdots\times\sigma_n\to\varrho\right)\implies \textit{typeEqs}\big(f(t_1,\cdots,t_n),\tau\big):=\{\varrho=\tau\}\cup\bigcup_{i=1}^n\textit{typeEqs}(t_i,\sigma_i).$$

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* 

$$\{arrho= au\}\cupigcup_{i=1}^n ext{typeEqs}(t_i,\sigma_i)$$

has a solution  $\vartheta$ . We formalize the notion of a *type equation* next.

#### **Definition 39 (Type Equation)** If $\sigma$ and $\tau$ 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  $\vartheta$  solves a type equation  $\sigma \doteq \tau$  if and only if  $\sigma \vartheta = \tau \vartheta$ . A substitution  $\vartheta$  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  $\varrho$  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  $\vartheta$  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  $\vartheta$  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

$$\Big\langle E \cup \big\{ X \doteq \tau \big\}, \vartheta \Big\rangle \quad \leadsto \quad \Big\langle E[X \mapsto \tau], \vartheta \big[ X \mapsto \tau \big] \Big\rangle.$$

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  $\vartheta$  by transforming  $\vartheta$  into the new substitution  $\vartheta[X \mapsto \tau]$ . 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

$$\langle E \cup \{f(\sigma_1, \cdots, \sigma_n) \doteq f(\tau_1, \cdots, \tau_n)\}, \vartheta \rangle \rightsquigarrow \langle E \cup \{\sigma_1 \doteq \tau_1, \cdots, \sigma_n \doteq \tau_n\}, \vartheta \rangle.$$

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

$$\langle E \cup \{c \doteq c\}, \vartheta \rangle \rightsquigarrow \langle E, \vartheta \rangle.$$

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  $E \cup \{f(\sigma_1, \cdots, \sigma_m) \doteq g(\tau_1, \cdots, \tau_n)\}$  has no solution if f and g are different. Therefore, we have

$$\langle E \cup \{f(\sigma_1, \dots, \sigma_m) \doteq g(\tau_1, \dots, \tau_n)\}, \vartheta \rangle \rightsquigarrow \Omega \quad \text{if } f \neq g.$$

Here  $\Omega$  denotes unsolvability. If the solution of  $\textit{typeEqs}(t,\tau)$  yields  $\Omega$ , 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

$$map(X_2, X_3) \doteq map(char, list(int)).$$

We proceed as follows:

$$\begin{split} & \left\langle \left\{ \operatorname{map}(X_2, X_3) \doteq \operatorname{map}(\operatorname{char}, \operatorname{list}(\operatorname{int})) \right\}, \left[ \right] \right\rangle \\ \leadsto & \left\langle \left\{ X_2 \doteq \operatorname{char}, \ X_3 \doteq \operatorname{list}(\operatorname{int}) \right\}, \left[ \right] \right\rangle \\ \leadsto & \left\langle \left\{ X_3 \doteq \operatorname{list}(\operatorname{int}) \right\}, \left[ X_2 \mapsto \operatorname{char} \right] \right\rangle \\ \leadsto & \left\langle \left\{ \right\}, \left[ X_2 \mapsto \operatorname{char}, \ X_3 \mapsto \operatorname{list}(\operatorname{int}) \right] \right\rangle \end{split}$$

In this case, the algorithm is successful and the resulting substitution

$$[X_2 \mapsto \mathtt{char}, \ X_3 \mapsto \mathtt{list}(\mathtt{int})]$$

is a solution of the type equation given above.

### 13.4 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 13.2 auf Seite 182 präsentiert. Die Abbildungen 13.3, 13.4 und 13.5 auf den folgenden Seiten zeigen die *JavaCup*-Spezifikation eines Parsers für diese Grammatik. Der zugehörige Scanner ist in Abbildungen 13.7 auf Seite 193 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 13.6 gezeigten Spezifikation erzeugt.

Die Klasse MartelliMontanari enthält die in Abbildung 13.8 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:

$$\langle E \cup \{X \doteq \tau\}, \vartheta \rangle \quad \leadsto \quad \langle E[X \mapsto \tau], \vartheta[X \mapsto \tau] \rangle.$$

Der Typ-Parameter, der in der obigen Formel mit X bezeichnet wird, trägt im Programm den Namen var und der Typ  $\tau$  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

$$\langle E \cup \{f(\sigma_1, \cdots, \sigma_m) \doteq g(\tau_1, \cdots, \tau_n)\}, \vartheta \rangle \rightsquigarrow \Omega.$$

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

$$\langle E \cup \{f(\sigma_1, \dots, \sigma_n) \doteq f(\tau_1, \dots, \tau_n)\}, \vartheta \rangle \rightsquigarrow \langle E \cup \{\sigma_1 \doteq \tau_1, \dots, \sigma_n \doteq \tau_n\}, \vartheta \rangle.$$

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  $\tau$ , falls die Funktion f eine Signatur

$$f: \sigma_1 \times \cdots \times \sigma_n \to \varrho$$

hat und es darüber hinaus eine Parameter-Substitution  $\vartheta$  gibt, so dass  $\tau = \varrho \vartheta$  gilt und weiterhin die Terme  $s_i$  vom Typ  $\sigma_i \vartheta$  sind:

$$f(s_1, \dots, s_n) : \tau \iff \exists \vartheta \in \text{SUBST} : \rho \vartheta = \tau \land s_1 : \sigma_1 \vartheta \land \dots \land s_n : \sigma_n \vartheta.$$

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.

```
import java_cup.runtime.*;
    import java.util.*;
                        TYPE, SIGNATURE, LEFT_PAR, RIGHT_PAR;
    terminal
    terminal
                        COMMA, COLON, SEMICOLON, ASSIGN, ARROW, PLUS, TIMES;
    terminal String
                        FUNCTION, PARAMETER;
    nonterminal Program
                                  program;
    nonterminal Term
                                  term;
    nonterminal List<Term>
                                  termList;
10
    nonterminal List<Parameter> varList;
11
    nonterminal Type
                                  type;
12
    nonterminal List<Type>
                                  typeList;
13
    nonterminal List<Type>
                                  typeSum;
14
    nonterminal TypeDef
                                  typeDef;
    nonterminal List<TypeDef>
                                  typeDefList;
16
    nonterminal Signature
                                  signature;
17
    nonterminal List<Signature> signatures;
18
    nonterminal List<Type>
                                  argTypes;
    nonterminal TypedTerm
                                  typedTerm;
20
    nonterminal List<TypedTerm> typedTerms;
21
22
                 ::= typeDefList:typDefs signatures:signList typedTerms:termList
23
                     {: RESULT = new Program(typDefs, signList, termList); :}
24
25
    typeDefList ::= typeDefList:1 typeDef:t {: 1.add(t); RESULT = 1; :}
26
                  | typeDef:t
27
                     {: List<TypeDef> l = new LinkedList<TypeDef>();
28
                        1.add(t); RESULT = 1;
29
                     :}
30
31
                 ::= TYPE FUNCTION:f ASSIGN typeSum:s SEMICOLON
    typeDef
32
                     {: RESULT = new SimpleTypeDef(f, s); :}
33
                     TYPE FUNCTION:f LEFT_PAR varList:a RIGHT_PAR ASSIGN
34
                          typeSum:s SEMICOLON
35
                     {: RESULT = new ParamTypeDef(f, a, s); :}
37
                 ::= FUNCTION:f LEFT_PAR typeList:t RIGHT_PAR
                                  {: RESULT = new CompositeType(f, t); :}
39
                     FUNCTION:f {: RESULT = new CompositeType(f);
                                                                         :}
40
                  1
                     PARAMETER:v {: RESULT = new Parameter(v);
                                                                         : }
41
42
                 ::= typeList:1 COMMA type:t {: 1.add(t); RESULT = 1; :}
    typeList
43
                     type:t {: List<Type> 1 = new LinkedList<Type>();
44
                                1.add(t);
45
                                RESULT = 1;
46
                             :}
47
                  ;
48
```

Figure 13.3: JavaCup-Spezifikation der TTL-Grammatik, 1. Teil

```
::= typeSum:1 PLUS type:t {: 1.add(t); RESULT = 1; :}
    typeSum
                  | type:t
2
                     {:
3
                        List<Type> 1 = new LinkedList<Type>();
                         1.add(t);
                        RESULT = 1;
                     :}
                 ::= SIGNATURE FUNCTION:f COLON argTypes:a
    signature
9
                                ARROW type:t SEMICOLON
10
                     {: RESULT = new Signature(f, a, t); :}
11
                     SIGNATURE FUNCTION: f COLON type:t SEMICOLON
12
                     {: List a = new LinkedList<Type>();
                         RESULT = new Signature(f, a, t);
14
                     :}
16
                 ::= signatures:1 signature:s {: l.add(s); RESULT = 1; :}
    signatures
                  | signature:s
18
                     {: List<Signature> l = new LinkedList();
                         1.add(s);
20
                         RESULT = 1;
                     : }
22
23
                 ::= argTypes:1 TIMES type:t {: l.add(t); RESULT = 1; :}
    argTypes
24
                     type:t
25
                     {: List<Type> l = new LinkedList();
26
                         1.add(t); RESULT = 1;
27
                     :}
29
    varList
                 ::= varList:l COMMA PARAMETER:v
30
                                  {: l.add(new Parameter(v)); RESULT = 1; :}
31
                     PARAMETER:v {: List<Parameter> l = new LinkedList();
                                     1.add(new Parameter(v));
33
                                     RESULT = 1;
34
                                  :}
35
                 ::= FUNCTION:f LEFT_PAR termList:l RIGHT_PAR
    term
37
                                 \{: RESULT = new Term(f, 1); :\}
38
                     FUNCTION:f {: List<Term> l = new LinkedList<Term>();
39
                                    RESULT = new Term(f, 1);
40
                                 :}
41
42
                 ::= termList:1 COMMA term:t
    termList
43
                             {: 1.add(t); RESULT = 1;
44
                     term:t {: List<Term> l = new LinkedList();
45
                                1.add(t); RESULT = 1;
46
                             :}
47
                  ;
48
```

Figure 13.4: JavaCup-Spezifikation der TTL-Grammatik, 2. Teil

Figure 13.5: JavaCup-Spezifikation der TTL-Grammatik, 3. Teil

```
Program = Program(List<TypeDef>
                                        typeDefs,
                       List<Signature> signatures,
2
                       List<TypedTerm> typedTerms);
3
    TypeDef = SimpleTypeDef(String name, List<Type> typeSum)
            + ParamTypeDef(String
                                         name,
                            List<String> parameters,
                            List<Type> typeSum);
    Type = Parameter(String name)
10
         + CompositeType(String name, List<Type> argTypes);
11
12
    substitute: Type * Parameter * Type -> Type;
13
14
    Signature = Signature(String name, List<Type> argList, Type result);
15
16
    Term = Term(String function, List<Term> termList);
17
18
    typeEqs: Term * Type * Map<String, Signature> -> List<Equation>;
19
20
    TypedTerm = TypedTerm(Term term, Type type);
21
22
    Substitution = Substitution(List<Parameter> variables, List<Type> types);
23
24
    Equation = Equation(Type lhs, Type rhs);
25
26
    substitute: Equation * Parameter * Term -> Equation;
27
28
    MartelliMontanari =
29
        MartelliMontanari(List<Equation> equations, Substitution theta);
30
```

Figure 13.6: Definition der benötigten Java-Klassen mit Hilfe von EP.

2. Ansonsten speichern wir in Zeile 8 die Argument-Typen  $\sigma_1, \dots, \sigma_n$  in der Variablen argTypes. Die Signatur von f muss natürlich genau so viele Argument-Typen haben, wie das Funktions-Zeichen f Argumente hat.

```
import java_cup.runtime.*;
    %%
    %char
    %line
    %column
    %cup
    %{
10
        private Symbol symbol(int type) {
11
            return new Symbol(type, yychar, yychar + yylength());
12
        }
14
        private Symbol symbol(int type, Object value) {
            return new Symbol(type, yychar, yychar + yylength(), value);
16
17
    %}
18
    %%
20
21
                        { return symbol( sym.PLUS
22
    "*"
                        { return symbol( sym.TIMES
    "("
                        { return symbol( sym.LEFT_PAR ); }
24
    ")"
                        { return symbol( sym.RIGHT_PAR ); }
                        { return symbol( sym.COMMA
                                                        ); }
                        { return symbol( sym.COLON
    ":"
                        { return symbol( sym.SEMICOLON ); }
28
    ":="
                        { return symbol( sym.ASSIGN
29
    "->"
                        { return symbol( sym.ARROW
                                                         ); }
    "type"
                        { return symbol( sym.TYPE
                                                        ); }
31
                        { return symbol( sym.SIGNATURE ); }
    "signature"
33
    [a-z][a-zA-Z_0-9]* { return symbol(sym.FUNCTION, yytext()); }
    [A-Z][a-zA-Z_0-0]* { return symbol(sym.PARAMETER, yytext()); }
35
    [ \t \n]
                        { /* skip white space */ }
37
                        { /* skip comments
    "//" [^\n]*
39
    [^] { throw new Error("Illegal character '" + yytext() +
40
                           "' at line " + yyline + ", column " + yycolumn); }
41
```

Figure 13.7: 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

$$ig(f:\sigma_1 imes\cdots imes\sigma_n oarrhoig)\implies ext{typeEqs}ig(f(t_1,\cdots,t_n), auig):=\{arrho= au\}\cupigcup_{i=1}^n ext{typeEqs}(t_i,\sigma_i).$$

Dazu erstellen wir in Zeile 17 zuächst die Typ-Gleichungen

$$\varrho \doteq \tau$$

```
public Substitution solve() {
        while (mEquations.size() != 0) {
            Equation eq = mEquations.remove(0);
            Type lhs = eq.getLhs();
            Type rhs = eq.getRhs();
             if (lhs instanceof Parameter) {
                 Parameter var = (Parameter) lhs;
                 List<Equation> newEquations = new LinkedList<Equation>();
                 for (Equation equation: mEquations) {
                     Equation neq = equation.substitute(var, rhs);
10
                     newEquations.add(neq);
11
                 }
12
                 mEquations = newEquations;
                            = mTheta.substitute(var, rhs);
                 mTheta
14
            } else if (lhs instanceof CompositeType) {
                 CompositeType compLhs = (CompositeType) lhs;
16
                 CompositeType compRhs = (CompositeType) rhs;
                 if (!compLhs.getName().equals(compRhs.getName())) {
18
                     // different type constructors, no solution
                     System.err.println("Error: different type constructors\n");
20
                     return null;
                 }
22
                 List<Type> lhsArgs = compLhs.getArgTypes();
                 List<Type> rhsArgs = compRhs.getArgTypes();
                 for (int i = 0; i < lhsArgs.size(); ++i) {</pre>
25
                     Type sigmaLhs = lhsArgs.get(i);
26
                     Type sigmaRhs = rhsArgs.get(i);
27
                     mEquations.add(0, new Equation(sigmaLhs, sigmaRhs));
29
            }
30
        }
31
        return mTheta;
    }
33
```

Figure 13.8: 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 für i=1,\cdots,n
```

ergeben. Als Ergebnis erhalten wir eine Menge von Typ-Gleichungen, die genau dann lösbar sind, wenn der Term  $f(s_1, \dots, s_n)$  den Typ  $\tau$  hat.

Als letztes diskutieren wir die Klasse Program. Abbildung 13.10 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

```
public List<Equation> typeEqs(Type tau, Map<String, Signature> map) {
        Signature sign = map.get(mFunction);
        if (sign == null) {
            System.err.println("The function " + mFunction +
                                " has not been declared!");
            throw new Error("Undeclared function in " + myString());
        }
        List<Type> argTypes = sign.getArgList();
        if (argTypes.size() != mTermList.size()) {
            System.err.println("Wrong number of parameters for function " +
10
                                mFunction);
11
            System.err.println("expected: " + argTypes.size());
12
            System.err.println("found:
                                            " + mTermList.size());
            throw new Error("Wrong number of parameters in " + myString());
14
        }
        List<Equation> result = new LinkedList<Equation>();
16
        Equation eq = new Equation(sign.getResult(), tau);
        result.add(eq);
18
        for (int i = 0; i < mTermList.size(); ++i) {</pre>
            Term argI = mTermList.get(i);
20
            Type sigmaI = argTypes.get(i);
            result.addAll( argI.typeEqs(sigmaI, map) );
22
        }
        return result;
24
    }
25
```

Figure 13.9: 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:

```
    nil: List(T),
    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 au von den Funktionen  $f_1, \cdots, f_k$  erzeugt. Die i-te Funktion  $f_i$  hat die Signatur

$$f_i: \sigma_1^{(i)} \times \cdots \times \sigma_{n_i}^{(i)} \to \tau.$$

```
public Program(List<TypeDef>
                                     typeDefs,
                    List<Signature> signatures,
                    List<TypedTerm> typedTerms)
3
    {
                     = typeDefs;
5
        mTypeDefs
        mSignatures = signatures;
        mTypedTerms = typedTerms;
        mSignatureMap = new TreeMap<String, Signature>();
        for (Signature s: mSignatures) {
             mSignatureMap.put(s.getName(), s);
10
11
        for (TypeDef td: mTypeDefs) {
12
             if (td instanceof SimpleTypeDef) {
                 SimpleTypeDef std = (SimpleTypeDef) td;
14
                 Type rho = new CompositeType(std.getName(), new LinkedList<Type>());
                 for (Type tau: std.getTypeSum()) {
16
                     CompositeType c = (CompositeType) tau;
                     String name = c.getName();
18
                     Signature s = new Signature(name, c.getArgTypes(), rho);
                     mSignatureMap.put(name, s);
20
                 }
             } else {
22
                 ParamTypeDef ctd = (ParamTypeDef) td;
                 List<Type> paramList = new LinkedList<Type>();
                 for (Parameter v : ctd.getParameters()) {
25
                     paramList.add(v);
26
27
                 Type rho = new CompositeType(ctd.getName(), paramList);
                 for (Type tau: ctd.getTypeSum()) {
29
                     CompositeType c = (CompositeType) tau;
30
                     String name = c.getName();
31
                     Signature s = new Signature(name, c.getArgTypes(), rho);
                     mSignatureMap.put(name, s);
33
                 }
34
             }
35
        }
    }
37
```

Figure 13.10: Der Konstruktor der Klasse Program.

Diese Signatur wird in Zeile 19 aus dem Ergebnis-Typ  $\tau$  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 13.11 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  $\tau$  die Menge der Typ-Gleichungen  $typeEqs(t,\tau)$  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  $\tau$ . Andernfalls wird eine Fehlermeldung ausgegeben.

```
public void typeCheck() {
        for (TypedTerm tt: mTypedTerms) {
2
            System.out.println("\nChecking " + tt.myString());
3
                     = tt.getTerm();
            Term t
4
            Type tau = tt.getType();
            List<Equation> typeEquations = t.typeEqs(tau, mSignatureMap);
            MartelliMontanari mm = new MartelliMontanari(typeEquations);
            Substitution theta = mm.solve();
            if (theta != null) {
                System.out.println(tt.myString() + " has been verified!");
                System.out.println(theta);
11
            } else {
                System.out.println(tt.myString() + " type ERROR!!!");
13
            }
        }
15
    }
16
```

Figure 13.11: Die Methode typeCheck() in der Klasse Program.

## 13.5 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

```
\mathtt{String} \subseteq \mathtt{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 parametrischen Polymorphismus. Die Behandlung von Inklusions-Polymorphismus ist besonders dann interessant, wenn dieser zusammen mit parametrischen Polymorphismus auftritt. Wir zeigen exemplarisch ein Beispiel in Abbildung parametrischen Polymorphismus

- 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.

```
public class TypeSurprise {

public static void main(String[] args) {
    String[] x = { "a", "b", "c" };

Object[] y = x;
    y[1] = new Integer(2);
    System.out.println(x[1]);
}
```

Figure 13.12: 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 14

## **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.

#### 14.1 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 14.1 on page 199 shows the program Hello.jas. We discuss this program line by line.

```
.class public Hello
    .super java/lang/Object
    .method public <init>()V
        aload 0
        invokenonvirtual java/lang/Object/<init>()V
        return
    .end method
    .method public static main([Ljava/lang/String;)V
10
        .limit locals 1
11
        .limit stack 2
12
                       java/lang/System/out Ljava/io/PrintStream;
        getstatic
13
                       "Hello World!"
14
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
        return
16
    .end method
17
```

Figure 14.1: 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 14.1 on page 199 corresponds to the Java code shown in Figure 14.2 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.

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Figure 14.2: 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<sup>1</sup>. 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.

<sup>&</sup>lt;sup>1</sup> 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 $0
```

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.

#### 14.2 Assembler Instructions

We proceed to discuss some of the  $J_{VM}$  instructions. Since there are more than 160  $J_{VM}$  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  $J_{VM}$ . In the  $J_{VM}$ , 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 14.3 on page 202 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.

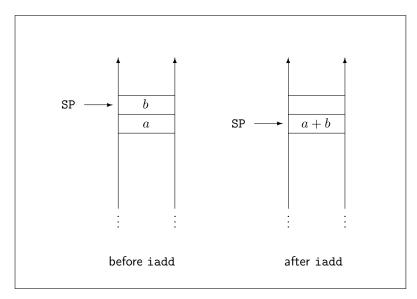


Figure 14.3: The effect of iadd.

- 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 14.4 on page 203 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.

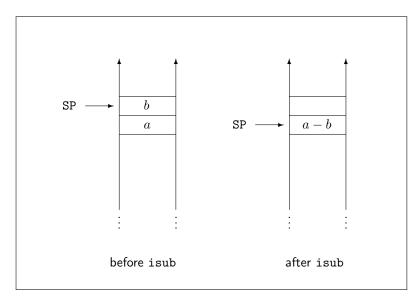


Figure 14.4: The effect of isub.

- 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 14.5 on page 203 pictures this command.

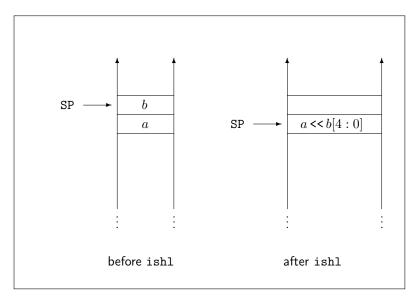


Figure 14.5: The effect of ishl.

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.

#### 14.2.1 Instructions to Manipulate the Stack

1. The instruction "dup" duplicates the value that is on top of the stack. Figure 14.6 on page 204 pictures this command.

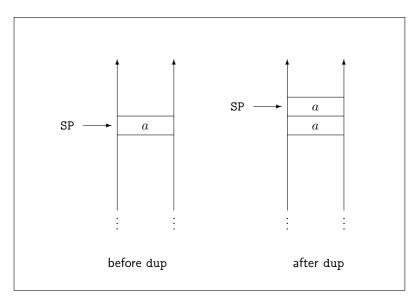


Figure 14.6: The effect of dup.

2. The instruction "pop" removes the value that is on top of the stack. Figure 14.7 on page 204 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.

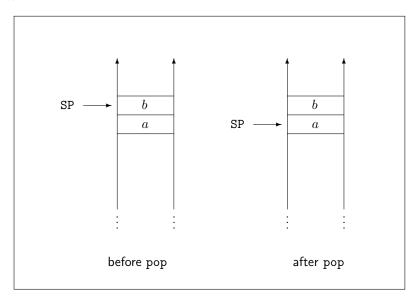


Figure 14.7: The effect of pop.

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 14.8 on page 205 pictures this command.

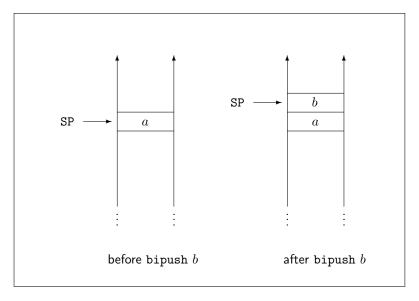


Figure 14.8: The effect of bipush b.

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 14.9 on page 206 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 14.10 on page 206 pictures this command.
- 8. The instruction "1dc 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 "1dc 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 14.9: The effect of iload v.



Figure 14.10: The effect of istore  $\boldsymbol{v}.$ 

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  $a_1, \dots, a_n$  is to be called, then first the arguments  $a_1, \dots, a_n$  have to be pushed onto the stack. When the method m is done and has computed its result r, the arguments  $a_1, \dots, a_n$  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.

## 14.3 An Example Program

Figure 14.11 shows the C program sum.c that computes the sum

```
\sum_{i=1}^{6^2} i.
```

The function  $\operatorname{sum}(n)$  computes the sum  $\sum_{i=1}^n i$  and the function main calls this function with the argument  $6 \cdot 6$ . Figure 14.12 on page 209 shows how this program can be translated into assembler program Sum. jas. We discuss the implementation of this program next.

```
#import "stdio.h"
2
    int sum(int n) {
        int s;
        s = 0;
        while (n != 0) {
             s = s + n;
             n = n - 1;
        };
        return s;
    }
11
    int main() {
12
        printf("%d\n", sum(6*6));
13
        return 0;
14
    }
15
```

Figure 14.11: 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.
- 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 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.

```
.class public Sum
     .super java/lang/Object
     .method public <init>()V
         aload 0
         invokenonvirtual java/lang/Object/<init>()V
        return
     .end method
     .method public static main([Ljava/lang/String;)V
10
         .limit locals 1
11
         .limit stack
12
         getstatic
                        java/lang/System/out Ljava/io/PrintStream;
         ldc
14
         dup
         imul
16
         invokestatic Sum/sum(I)I
         invokevirtual java/io/PrintStream/println(I)V
18
        return
     .end method
20
     .method public static sum(I)I
22
         .limit locals 2
23
         .limit stack 2
24
         ldc
25
                                       ; s = 0;
         istore 1
26
    loop:
27
         iload 0
                                       ; n
         ifeq
                finish
                                       ; if (n == 0) goto finish;
29
         iload 1
                                       ; s
30
         iload
                                       ; n
31
         iadd
                                       ; s = s + n;
         istore 1
33
         iload
34
                                       ; n
         ldc
35
         isub
         istore
                 0
                                       ; n = n - 1;
37
         goto
                 loop
    finish:
39
         iload
40
         ireturn
                                       ; return s;
41
     .end method
42
```

Figure 14.12: 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

14.4. Disassembler\* Chapter 14. Assembler

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 35**: Implement an assembler program that computes the factorial function. Test your program by printing n! for  $n = 1, \dots, 10$ .

#### 14.4 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 <u>javap</u> 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 14.13 shows the class *Java* class <u>Sum</u> 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.

```
public class Sum {
    public static void main(String[] args) {
        System.out.println(sum(6 * 6));
}

static int sum(int n) {
    int s = 0;
    for (int i = 0; i <= n; ++i) {
        s += i;
    }
    return s;
}
</pre>
```

Figure 14.13: 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 14.14. 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

14.4. Disassembler\* Chapter 14. Assembler

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:
                             #2
                                           // Field java/lang/System.out:Ljava/io/PrintStream;
           0: getstatic
11
           3: bipush
                             36
           5: invokestatic #3
                                           // Method sum:(I)I
13
           8: invokevirtual #4
                                           // Method java/io/PrintStream.println:(I)V
          11: return
15
16
      static int sum(int);
17
        Code:
18
           0: iconst_0
19
           1: istore_1
           2: iconst_0
21
           3: istore_2
           4: iload_2
           5: iload_0
24
           6: if_icmpgt
                             19
           9: iload_1
26
          10: iload_2
          11: iadd
28
          12: istore_1
          13: iinc
                             2, 1
30
                             4
          16: goto
          19: iload_1
32
          20: ireturn
33
    }
34
```

Figure 14.14: The output of "javap -c Sum.class".

## Chapter 15

# **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.
- 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.

## 15.1 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 15.1 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 8 einen Interpreter entwickelt haben. Die in Abbildung 15.1 gezeigte Grammatik ist mehrdeutig:

```
program \rightarrow function^*
 function \rightarrow
               "int" ID "(" paramList ")" "{" decl* stmnt* "}"
paramList \rightarrow ("int" ID ("," "int" ID)*)?
               "int" ID ";"
     decl \rightarrow
    stmnt \rightarrow "{"stmnt""}"
                 ID "=" expr ";"
                 "if" "(" boolExpr ")" stmnt
                 "if" "(" boolExpr ")" stmnt "else" stmnt
                 "while" "(" boolExpr ")" stmnt
                 "return" expr ";"
                 expr ";"
 boolExpr \rightarrow expr("==" | "!=" | "<=" | ">=" | "<" | ">") expr
                 "!" boolExpr
                boolExpr ("&&" | "||") boolExpr
     expr \rightarrow expr("+" | "-" | "*" | "/") expr
                "(" expr ")"
                Number
                ID ("("(expr("," expr)*)? ")")?
```

Figure 15.1: Eine  $\operatorname{EbnF-Grammatik}$  für Integer- $\operatorname{C}$ 

1. Die Grammatik hat das Dangling-Else-Problem.

Da wir im Kapitel 12 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 15.2 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.

```
int sum(int n) {
         int s;
2
         s = 0;
3
         while (n != 0) {
             s = s + n:
             n = n - 1;
         return s;
    }
9
    int main() {
10
         int n;
11
         n = 6 * 6;
12
         println(sum(n));
13
    }
14
```

Figure 15.2: Ein einfaches INTEGER-C-Programm.

### 15.2 Developing the Scanner and the Parser

We construct the scanner using JFlex while we use CuP to implement the parser. Figure 15.3 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 15.4 on page 216. 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

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.

```
import java_cup.runtime.*;
    %char
    %line
    %column
    %cupsym IntegerCParserSym
    %cup
    %unicode
    %{ private Symbol symbol(int type) {
            return new Symbol(type, yychar, yychar + yylength());
10
11
        private Symbol symbol(int type, Object value) {
12
             return new Symbol(type, yychar, yychar + yylength(), value);
14
    %}
    %eofval{ return new Symbol(IntegerCParserSym.EOF);
16
17
    %%
18
    11411
                           { return symbol( IntegerCParserSym.PLUS
                                                                          ); }
    "-"
                           { return symbol( IntegerCParserSym.MINUS
                                                                          ); }
20
    "*"
                           { return symbol( IntegerCParserSym.TIMES
                                                                          ); }
21
    "/"
                           { return symbol( IntegerCParserSym.SLASH
22
                                                                          ); }
    "("
                           { return symbol( IntegerCParserSym.LPAREN
                                                                          ); }
    ")"
                           { return symbol( IntegerCParserSym.RPAREN
                                                                          ); }
24
    "{"
                           { return symbol( IntegerCParserSym.LBRACE
                                                                          ); }
    "}"
                           { return symbol( IntegerCParserSym.RBRACE
                                                                          ); }
                                                                          ); }
                           { return symbol( IntegerCParserSym.COMMA
    ":"
                           { return symbol( IntegerCParserSym.SEMICOLON ); }
28
    "="
                           { return symbol( IntegerCParserSym.ASSIGN
                                                                          ); }
29
    "=="
                           { return symbol( IntegerCParserSym.EQ
                                                                          ); }
    "!="
                           { return symbol( IntegerCParserSym.NE
                                                                          ); }
31
    11 < 11
                           { return symbol( IntegerCParserSym.LT
                                                                          ); }
    ">"
                           { return symbol( IntegerCParserSym.GT
                                                                          ); }
33
    "<="
                           { return symbol( IntegerCParserSym.LE
                                                                          ); }
    ">="
                           { return symbol( IntegerCParserSym.GE
                                                                          ); }
35
    " & & "
                           { return symbol( IntegerCParserSym.AND
                                                                          ); }
    "||"
                           { return symbol( IntegerCParserSym.OR
                                                                          ); }
37
    11 11 11
                           { return symbol( IntegerCParserSym.NOT
                                                                          ); }
    "int"
                           { return symbol( IntegerCParserSym.INT
                                                                          ); }
39
    "return"
                           { return symbol( IntegerCParserSym.RETURN
                                                                          ); }
    "if"
                           { return symbol( IntegerCParserSym.IF
                                                                          ); }
41
    "else"
                           { return symbol( IntegerCParserSym.ELSE
                                                                          ); }
42
    "while"
                           { return symbol( IntegerCParserSym.WHILE
                                                                          ); }
43
    [a-zA-Z][a-zA-Z_0-9]* { return symbol(IntegerCParserSym.ID,
                                                                       yytext());
44
    0|[1-9][0-9]*
                   { return symbol(IntegerCParserSym.NUMBER, new Integer(yytext())); }
45
                                                            */ }
    [ \t \v \n \r]
                           { /* skip white space
46
    "//" [^\n]*
                           { /* skip single line comments */ }
47
    "/*" ~"*/"
                           { /* skip multi line comments */ }
```

Figure 15.3: Der Scanner für Integer-C

```
Program = Program(List<Function> functionList);
    Function = Function(String
                                           name.
                         List<String>
                                           parameterList,
                         List < Declaration > mDeclarations,
5
                         List<Statement>
                                           body);
    Statement = Block(List<Statement> statementList)
              + Assign(String var, Expr expr)
9
              + IfThen(BoolExpr boolExpr, Statement statement)
10
              + IfThenElse(BoolExpr condition, Statement then, Statement else)
11
              + While(BoolExpr condition, Statement statement)
12
              + Return(Expr expr)
              + ExprStatement(Expr expr);
14
    Declaration = Declaration(String var);
16
17
    Expr = Sum(Expr lhs, Expr rhs)
                                                            // +
18
         + Difference(Expr lhs, Expr rhs)
                                                            // -
         + Product(Expr lhs, Expr rhs)
                                                            // *
20
         + Quotient(Expr lhs, Expr rhs)
                                                            11 1
         + MyNumber(Integer number)
22
         + Variable(String name)
         + FunctionCall(String name, List<Expr> args);
24
25
                                                            // ==
    BoolExpr = Equation(Expr lhs, Expr rhs)
26
                                                            // !=
             + Inequation(Expr lhs, Expr rhs)
27
             + LessOrEqual(Expr lhs, Expr rhs)
                                                            // <=
28
             + GreaterOrEqual(Expr lhs, Expr rhs)
                                                            // >=
29
                                                            // <
             + LessThan(Expr lhs, Expr rhs)
             + GreaterThan(Expr lhs, Expr rhs)
                                                            // >
31
                                                            //!
             + Negation(BoolExpr expr)
             + Conjunction(BoolExpr lhs, BoolExpr rhs)
                                                            // &&
33
             + Disjunction(BoolExpr lhs, BoolExpr rhs);
                                                            // 11
```

Figure 15.4: Spezifikation der Klassen zur Darstellung des Syntax-Baums.

#### 3. Next, the equation

tells us that there is an abstract class Statement and that the classes Block, Assign, ..., 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 statements 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. If Then 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. If Then Else 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 15.1 on page 213 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  $\mathrm{Cup}$  specification of our parser. For reasons of space, we have split the  $\mathrm{Cup}$  grammar into three parts.

- 1. Figure 15.5 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 15.6 and 15.7 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.

```
import java_cup.runtime.*;
    import java.util.*;
    class IntegerCParser;
                       COMMA, PLUS, MINUS, TIMES, SLASH, LPAREN, RPAREN, LBRACE, RBRACE;
    terminal
                       ASSIGN, EQ, LT, GT, LE, GE, NE, AND, OR, NOT;
    terminal
                       IF, ELSE, WHILE, RETURN, SEMICOLON;
    terminal
    terminal
                       INT;
    terminal String
                       ID;
10
    terminal Integer
                       NUMBER;
11
    nonterminal Program
                                   program;
13
    nonterminal List<Function>
                                  functionList;
    nonterminal Function
15
                                   function;
    nonterminal List<String>
                                   paramList, neParamList;
    nonterminal Declaration
                                   declaration;
17
    nonterminal List<Declaration> declarations;
    nonterminal Statement
                                   stmnt;
19
    nonterminal List<Statement>
                                   stmntList;
    nonterminal Expr
21
                                   expr;
    nonterminal List<Expr>
                                   exprList, neExprList;
    nonterminal BoolExpr
                                   boolExpr;
23
24
    precedence left
                        OR;
25
    precedence left
26
                        AND;
    precedence right
                        NOT;
27
    precedence left
                        PLUS, MINUS;
28
    precedence left
                        TIMES, SLASH;
```

Figure 15.5: Declaration of terminals, syntactical variables, and operator precedences.

```
start with program;
31
32
    program ::= functionList:1 {: RESULT = new Program(1); :} ;
33
    functionList ::= /* epsilon */ {: RESULT = new LinkedList<Function>(); :}
34
                  functionList:1 function:f {: 1.add(f); RESULT = 1; :}
35
36
37
    function ::= INT ID:f LPAREN paramList:p RPAREN
38
                 LBRACE declarations:d stmntList:l RBRACE
39
                 {: RESULT = new Function(f, p, d, 1); :}
40
41
42
    paramList ::= /* epsilon */ {: RESULT = new LinkedList<String>(); :}
               | neParamList:1 {: RESULT = 1;
                                                                        :}
44
45
46
    neParamList ::= INT ID:v {: RESULT = new LinkedList<String>(); RESULT.add(v); :}
                 neParamList:1 COMMA INT ID:v {: RESULT = 1; RESULT.add(v);
48
49
50
    declaration ::= INT ID:v SEMICOLON {: RESULT = new Declaration(v); :} ;
51
52
    declarations ::= /* epsilon */ {: RESULT = new LinkedList<Declaration>(); :}
53
                  declarations:1 declaration:d {: RESULT = 1; RESULT.add(d); :}
54
                  ;
55
56
    stmnt ::= LBRACE stmntList:1 RBRACE
                                                   {: RESULT = new Block(1);
57
           | ID:v ASSIGN expr:e SEMICOLON
                                                   {: RESULT = new Assign(v, e); :}
              IF LPAREN boolExpr:b RPAREN stmnt:s {: RESULT = new IfThen(b, s); :}
59
              IF LPAREN boolExpr:b RPAREN stmnt:t ELSE stmnt:e
                  {: RESULT = new IfThenElse(b, t, e); :}
61
             WHILE LPAREN boolExpr:b RPAREN stmnt:s
                  {: RESULT = new While(b, s); :}
63
              RETURN expr:e SEMICOLON {: RESULT = new Return(e);
           expr:e SEMICOLON
                                        {: RESULT = new ExprStatement(e); :}
65
67
    stmntList ::= /* epsilon */
                                   {: RESULT = new LinkedList<Statement>(); :}
68
               stmntList:l stmnt:s {: l.add(s); RESULT = 1;
                                                                                 :}
69
70
```

Figure 15.6: The first part of the CUP grammar.

```
boolExpr ::= expr:l EQ expr:r
                                            {: RESULT = new Equation(
                                                                            1, r); :}
71
              | expr:l NE expr:r
                                          {: RESULT = new Inequation(
                                                                            1, r); :}
72
              | expr:l LE expr:r
                                           {: RESULT = new LessOrEqual(
                                                                            1, r); :}
73
              | expr:l GE expr:r
                                           {: RESULT = new GreaterOrEqual(1, r); :}
74
                 expr:l LT expr:r
                                            {: RESULT = new LessThan(
                                                                            1, r); :}
75
                                                                            1, r); :}
              | expr:l GT expr:r
                                           {: RESULT = new GreaterThan(
76
              | NOT boolExpr:e
                                            {: RESULT = new Negation(
                                                                            e ); :}
77
              1
                 boolExpr:1 AND boolExpr:r {: RESULT = new Conjunction(
                                                                            1, r); :}
78
                 boolExpr:l OR boolExpr:r {: RESULT = new Disjunction(
                                                                            1, r); :}
79
80
81
    expr ::= expr:l PLUS expr:r
                                             {: RESULT = new Sum(
                                                                          1, r);
                                                                                   :}
82
                                         {: RESULT = new Difference(1, r);
{: RESULT = new Product( 1, r);
          | expr:1 MINUS expr:r
                                                                                   :}
83
            expr:l TIMES expr:r
                                                                                   :}
84
             expr:1 SLASH expr:r
                                             {: RESULT = new Quotient( 1, r);
                                                                                   :}
          LPAREN expr:e RPAREN
                                             {: RESULT = e;
                                                                                   :}
          1
86
                                                                                   :}
          | NUMBER:n
                                             {: RESULT = new MyNumber(n);
             ID:v
                                             {: RESULT = new Variable(v);
                                                                                   :}
88
             ID:n LPAREN exprList:1 RPAREN {: RESULT = new FunctionCall(n, 1); :}
          90
91
    exprList ::= /* epsilon */ {: RESULT = new LinkedList<Expr>(); :}
92
                 neExprList:1 {: RESULT = 1;
                                                                     :}
93
94
95
    neExprList ::= expr:e {: RESULT = new LinkedList<Expr>(); RESULT.add(e); :}
96
                neExprList:1 COMMA expr:e {: RESULT = 1; RESULT.add(e);
97
98
```

Figure 15.7: The last part of the CUP grammar.

## 15.3 Darstellung der Assembler-Befehle

Die Abbildungen 15.8 und 15.9 zeigen eine mögliche Übersetzung des Programms zur Berechnung der Summe  $\sum_{i=1}^n i$  aus Abbildungen 15.2 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 15.10 zeigt die Spezifikation dieser Klassen.

```
.class public MySum
     .super java/lang/Object
2
3
     .method public <init>()V
4
         aload 0
         invokenonvirtual java/lang/Object/<init>()V
        return
     .end method
     .method public static main([Ljava/lang/String;)V
10
         .limit locals 1
11
         .limit stack 2
12
         ldc 6
13
         ldc 6
         imul
15
         istore 0
         getstatic java/lang/System/out Ljava/io/PrintStream;
17
         iload 0
18
         invokestatic MySum/sum(I)I
19
         invokevirtual java/io/PrintStream/println(I)V
20
         bipush 42
21
        pop
22
        return
23
     .end method
24
```

Figure 15.8: Eine Übersetzung des Programms aus Abbildung 15.2 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 "1" 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 15.11 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 "1" vor- und der Doppelpunkt ":" nachgestellt, damit die Ausgabe der Syntax des *Jasmin*-Assemblers entspricht.

 Neben den eigentlichen Assembler-Kommandos zeigt Abbildung 15.10 noch die Definition von Klassen wie beispielsweise der Klasse MAIN, deren Implementierung in Abbildung 15.12 gezeigt wird. Diese Klasse dient nur dazu, die Direktive

<sup>&</sup>quot;.method public static main([Ljava/lang/String;)V"

```
.method public static sum(I)I
1
         .limit locals 2
         .limit stack 2
3
         ldc 0
         istore 1
      11:
         iload 0
         ldc 0
         if_icmpne 13
         bipush 0
10
         goto 14
11
      13:
12
         bipush 1
14
         ifeq 12
         iload 1
16
         iload 0
         iadd
18
         istore 1
         iload 0
20
         ldc 1
         isub
22
23
         istore 0
         goto 11
24
      12:
25
         iload 1
26
         ireturn
27
     .end method
```

Figure 15.9: Übersetzung des Programms aus Abbildung 15.2 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)
3
                  + IADD()
                  + ISUB()
                  + IMUL()
                  + IDIV()
                  + IAND()
                  + IOR()
                  + POP()
10
                  + BIPUSH(Integer number)
11
                  + ILOAD(String var)
12
                  + ISTORE(String var)
                  + INVOKE(String name)
14
                  + IRETURN()
                  + LABEL(Integer label)
16
                  + GOTO(Integer label)
                  + IFEQ(Integer label)
18
                  + IFLT(Integer label)
                  + LDC(Integer number)
20
                  + LIMIT(String what, Integer bound)
                  + MAIN()
22
                  + NEWLINE()
23
                  + POP()
24
                  + PRINTLN()
                  + RETURN();
26
```

Figure 15.10: 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 15.11: Die Klasse LABEL.

## 15.4 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.

```
package Assembler;

public class MAIN extends AssemblerCmd {

public MAIN() {}

public String toString() {
 return ".method public static main([Ljava/lang/String;)V";
}
}
```

Figure 15.12: Die Klasse MAIN.

## 15.4.1 Übersetzung arithmetischer Ausdrücke

Die Ubersetzung 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 15.13 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 Ubersetzung der verschiedenen arithmetischen Ausdrücke der Reihe nach.

```
public abstract class Expr {
    public abstract List<AssemblerCmd> compile(Map<String, Integer> symbolTable);
    public abstract Integer stackSize(); // maximum size of stack needed
}
```

Figure 15.13: Die Klasse Expr.

#### Übersetzung einer Variablen

Um eine Variable v auszuwerten, laden wir diese Variable mit dem Kommando

```
{\tt iload}\ v
```

auf den Stack. Daher hat die Klasse Variable die in Abbildung 15.14 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.

```
public class Variable extends Expr {
        private String mName;
        public Variable(String name) {
            mName = name;
        }
        public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
            List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
            AssemblerCmd iload = new ILOAD(symbolTable.get(mName));
            result.add(iload);
10
            return result;
12
        public Integer stackSize() {
13
            return 1;
14
        }
    }
16
```

Figure 15.14: Die Klasse Variable.

#### Übersetzung einer Konstanten

Eine Konstante c kann mit Hilfe des Befehls

 ${\tt ldc}\ c$ 

auf den Stack geladen werden.

```
public class MyNumber extends Expr {
        private Integer mNumber;
2
        public MyNumber(Integer number) {
            mNumber = number;
        public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
            List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
            AssemblerCmd ldc = new LDC(mNumber);
            result.add(ldc);
            return result;
11
        }
12
        public Integer stackSize() {
13
            return 1;
14
        }
15
    }
16
```

Figure 15.15: Die Klasse MyNumber.

Abbildung 15.15 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 1dc 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

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:

```
compile(Ihs "+" rhs) = Ihs.compile() + rhs.compile() + [iadd],
```

wobei der Operator "+" auf der rechten Seite dieser Gleichung der Verkettung von Listen dient. Abbildung 15.16 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 Ihs einen Stack der Größe Ihs.stackSize(). Nachdem Ihs ausgewertet worden ist, verbleibt aber nur der Wert von Ihs auf dem Stack.
- 2. Falls wir für die Auswertung von rhs weniger Platz auf dem Stack benötigen als für die Auswertung von Ihs, dann reicht insgesamt der Stack aus, der für die Auswertung von Ihs allokiert worden ist, denn das Ergebnis der Auswertung von Ihs benötigt nur eine Speicherstelle und da die Auswertung von rhs nach Voraussetzung weniger Platz auf dem Stack benötigt als die Auswertung von Ihs 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

```
rhs.stackSize() + 1
```

haben, denn wir müssen zusätzlich ja noch das Ergebnis der Auswertung von Ihs speichern.

Insgesamt sehen wir, dass die Höhe des Stacks durch die Formel

```
max(\mathit{lhs}.stackSize(), \mathit{rhs}.stackSize() + 1)
```

gegeben ist. Die Übersetzung von Ausdrücken der Form

```
lhs - rhs, lhs * rhs und lhs/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:

```
compile(f(e_1, \dots, e_n)) = compile(e_1) + \dots + compile(e_n) + [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

```
public class Sum extends Expr {
        private Expr mLhs;
        private Expr mRhs;
3
        public Sum(Expr lhs, Expr rhs) {
            mLhs = lhs;
            mRhs = rhs;
        }
        public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
            List<AssemblerCmd> result = mLhs.compile(symbolTable);
10
            result.addAll(mRhs.compile(symbolTable));
11
            result.add(new IADD());
12
            return result;
14
        public Integer stackSize() {
            return Math.max(mLhs.stackSize(), mRhs.stackSize() + 1);
16
    }
18
```

Figure 15.16: Die Klasse Sum.

abzulegen. Philosophische Betrachtungen, die über den Rahmen der Vorlesung hinausgehen, legen nahe, hierfür den Wert 42 zu wählen.

Abbildung 15.17 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.

```
public class FunctionCall extends Expr {
        private String
                            mName;
        private List<Expr> mArgs;
3
        public FunctionCall(String name, List<Expr> args) {
            mName = name;
            mArgs = args;
        }
        public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
            List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
10
            if (mName.equals("println")) {
11
                 AssemblerCmd getStatic =
12
                     new GETSTATIC("java/lang/System/out Ljava/io/PrintStream;");
                 result.add(getStatic);
14
                 for (Expr arg: mArgs) {
                     result.addAll(arg.compile(symbolTable));
                 }
                 AssemblerCmd println = new PRINTLN();
18
                 AssemblerCmd bipush = new BIPUSH(42);
                result.add(println);
20
                result.add(bipush);
                return result;
22
            }
            for (Expr arg: mArgs) {
                 result.addAll(arg.compile(symbolTable));
25
26
            String descr = Compiler.sClassName + "/" + mName + "(";
27
            for (int i = 0; i < mArgs.size(); ++i) {</pre>
                 descr += "I";
29
            descr += ")I";
31
            AssemblerCmd invoke = new INVOKE(descr);
            result.add(invoke);
33
            return result;
35
        public Integer stackSize() {
            Integer biggest = 0;
37
            for (int i = 0; i < mArgs.size(); ++i) {
                 biggest = Math.max(biggest, i + mArgs.get(i).stackSize());
            }
            if (mName.equals("println")) {
41
                 ++biggest;
42
            }
43
            return Math.max(biggest, 1);
44
        }
45
    }
46
```

Figure 15.17: Die Klasse FunctionCall.

### 15.4.2 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:

```
public class Equation extends BoolExpr {
        private Expr mLhs;
2
        private Expr mRhs;
3
        public Equation(Expr lhs, Expr rhs) {
             mLhs = lhs;
            mRhs = rhs;
        public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
             List<AssemblerCmd> result = mLhs.compile(symbolTable);
             result.addAll(mRhs.compile(symbolTable));
11
             LABEL
                          trueLabel = new LABEL();
             LABEL
                          nextLabel = new LABEL();
13
             AssemblerCmd if_icmpeq = new IF_ICMPEQ(trueLabel.getLabel());
14
                                    = new BIPUSH(0);
             AssemblerCmd bipush0
             AssemblerCmd gotoNext
                                    = new GOTO(nextLabel.getLabel());
16
             AssemblerCmd bipush1
                                     = new BIPUSH(1);
17
             result.add(if_icmpeq);
             result.add(bipush0);
             result.add(gotoNext);
20
             result.add(trueLabel);
21
             result.add(bipush1);
22
             result.add(nextLabel);
             return result;
24
        }
25
        public Integer stackSize() {
26
             return Math.max(mLhs.stackSize(), mRhs.stackSize() + 1);
        }
28
    }
```

Figure 15.18: 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

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

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 \le rhs, lhs \le rhs, lhs \ge rhs, and lhs \le 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:

Abbildung 15.19 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

```
public class Conjunction extends BoolExpr {
        private BoolExpr mLhs;
        private BoolExpr mRhs;
3
        public Conjunction(BoolExpr lhs, BoolExpr rhs) {
            mLhs = lhs;
            mRhs = rhs;
        }
        public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
            List<AssemblerCmd> result = mLhs.compile(symbolTable);
10
            result.addAll(mRhs.compile(symbolTable));
11
            AssemblerCmd iand = new IAND();
12
            result.add(iand);
            return result;
14
        }
        public Integer stackSize() {
16
            return Math.max(mLhs.stackSize(), mRhs.stackSize() + 1);
18
    }
19
```

Figure 15.19: 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 Ubersetzung 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:

Abbildung 15.20 zeigt die Umsetzung dieser Idee.

#### 15.4.3 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.

```
public class Negation extends BoolExpr {
        private BoolExpr mExpr;
3
        public Negation(BoolExpr expr) {
            mExpr = expr;
        public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
            List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
            AssemblerCmd bipush1 = new BIPUSH(1);
            AssemblerCmd isub
                                  = new ISUB();
10
            result.add(bipush1);
11
            result.addAll(mExpr.compile(symbolTable));
12
            result.add(isub);
            return result;
14
        }
        public Integer stackSize() {
16
            return mExpr.stackSize() + 1;
17
18
    }
19
```

Figure 15.20: 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:

```
compile(x=expr) = expr.compile() + [istore x]
```

Die Idee wird in der in Abbildung 15.21 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:

Abbildung 15.22 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
```

```
public class Assign extends Statement {
        private String mVar;
        private Expr
                       mExpr;
3
        public Assign(String var, Expr expr) {
            mVar = var;
            mExpr = expr;
        }
        public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
            List<AssemblerCmd> result = mExpr.compile(symbolTable);
10
            AssemblerCmd
                                storeCmd = new ISTORE(symbolTable.get(mVar));
11
            result.add(storeCmd);
12
            return result;
14
        public Integer stackSize() {
            return mExpr.stackSize();
16
        }
17
    }
18
```

Figure 15.21: Die Klasse Assign.

```
public class ExprStatement extends Statement {
        private Expr mExpr;
        public ExprStatement(Expr expr) {
            mExpr = expr;
        public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
            List<AssemblerCmd> result = mExpr.compile(symbolTable);
                                popCmd = new POP();
            AssemblerCmd
            result.add(popCmd);
10
            return result;
11
        }
12
        public Integer stackSize() {
13
            return mExpr.stackSize();
14
        }
15
    }
16
```

Figure 15.22: 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:

Diese Spezifikation ist in der Abbildung 15.23 umgesetzt worden.

```
public class IfThen extends Statement {
        private BoolExpr mBoolExpr;
        private Statement mStatement;
3
        public IfThen(BoolExpr boolExpr, Statement statement) {
            mBoolExpr = boolExpr;
            mStatement = statement;
        }
        public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
            List<AssemblerCmd> result = mBoolExpr.compile(symbolTable);
10
            LABEL
                          elseLabel = new LABEL();
11
            AssemblerCmd ifeq
                                    = new IFEQ(elseLabel.getLabel());
12
            result.add(ifeq);
            result.addAll(mStatement.compile(symbolTable));
14
            result.add(elseLabel);
            return result;
16
        }
        public Integer stackSize() {
18
            return Math.max(mBoolExpr.stackSize(), mStatement.stackSize());
19
        }
20
    }
```

Figure 15.23: 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:

```
compile(if (expr) thenStmnt else elseStmnt) = expr.compile()
+ [ifeq else]
+ thenStmnt.compile()
+ [goto next]
+ [else:]
+ elseStmnt.compile()]
```

Diese Spezifikation ist in der Abbildung 15.24 umgesetzt worden.

#### Die Übersetzung einer Schleife

```
Die Übersetzung einer while-Schleife der Form
```

```
while (cond) statement
```

orientiert sich an der folgenden Spezifikation:

Die Umsetzung dieser Spezifikation sehen Sie in Abbildung 15.25.

```
public class IfThenElse extends Statement {
        private BoolExpr mExpr;
        private Statement mThen;
        private Statement mElse;
        public IfThenElse(BoolExpr expr, Statement thenStmnt, Statement elseStmnt) {
            mExpr = expr;
            mThen = thenStmnt;
            mElse = elseStmnt;
        }
10
        public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
11
            List<AssemblerCmd> result = mExpr.compile(symbolTable);
12
            LABEL
                          elseLabel = new LABEL();
            LABEL
                          nextLabel = new LABEL();
14
                                    = new IFEQ(elseLabel.getLabel());
            AssemblerCmd ifeq
            AssemblerCmd gotoNext = new GOTO(nextLabel.getLabel());
16
            result.add(ifeq);
            result.addAll(mThen.compile(symbolTable));
18
            result.add(gotoNext);
            result.add(elseLabel);
20
            result.addAll(mElse.compile(symbolTable));
            result.add(nextLabel);
22
            return result;
        }
24
        public Integer stackSize() {
            return Math.max(mExpr.stackSize(), Math.max(mThen.stackSize(), mElse.stackSize()));
26
        }
27
    }
28
```

Figure 15.24: 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(\{stmnt_1; \cdots stmnt_n; \}) = compile(stmnt_1) + \cdots + compile(stmnt_n).
```

Diese Idee ist in der Klasse Block realisiert worden. Abbildung 15.26 zeigt diese Klasse.

#### 15.4.4 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.
- Die Klasse Program repräsentiert das vollständige Programm.

```
public class While extends Statement {
        private BoolExpr mCondition;
        private Statement mStatement;
3
        public While(BoolExpr condition, Statement statement) {
            mCondition = condition;
            mStatement = statement;
        }
        public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
             List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
10
            LABEL
                          loopLabel = new LABEL();
11
            LABEL
                          nextLabel = new LABEL();
12
             AssemblerCmd ifeq
                                    = new IFEQ(nextLabel.getLabel());
             AssemblerCmd gotoLoop = new GOTO(loopLabel.getLabel());
14
            result.add(loopLabel);
            result.addAll(mCondition.compile(symbolTable));
16
            result.add(ifeq);
            result.addAll(mStatement.compile(symbolTable));
18
            result.add(gotoLoop);
            result.add(nextLabel);
20
            return result;
        }
22
        public Integer stackSize() {
23
            return Math.max(mCondition.stackSize(), mStatement.stackSize());
24
25
    }
26
```

Figure 15.25: Die Klasse While.

Wir beginnen mit der Diskussion der Klasse Function. Abbildung 15.27 zeigt die Klasse Function, allerdings ohne die Implementierung der Methode *compile()*, die wir aus Platzgründen in die Abbildung 15.28 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.

```
public class Block extends Statement {
        private List<Statement> mStatementList;
        public Block(List<Statement> statementList) {
            mStatementList = statementList;
        public List<AssemblerCmd> compile(Map<String, Integer> symbolTable) {
            List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
            for (Statement stmnt: mStatementList) {
                result.addAll(stmnt.compile(symbolTable));
11
            return result;
12
        public Integer stackSize() {
14
            Integer biggest = 0;
            for (Statement stmnt: mStatementList) {
16
                biggest = Math.max(biggest, stmnt.stackSize());
18
            return biggest;
        }
20
    }
```

Figure 15.26: Die Klasse Block.

```
public class Function {
        private String
                                   mName;
        private List<String>
                                   mParameterList;
        private List<Declaration> mDeclarations;
        private List<Statement>
                                   mBody;
        private Integer
                                   mLocals; // number of local variables
        public Function(String
                                            name,
                         List<String>
                                            parameterList,
10
                         List<Declaration> declarations,
                         List<Statement>
                                            body)
        {
            mName
                            = name;
14
            mParameterList = parameterList;
15
            mDeclarations = declarations;
            mBody
                            = body;
17
            mLocals
                            = mParameterList.size() + mDeclarations.size();
19
        public List<AssemblerCmd> compile() { ... }
        public Integer stackSize() { ... }
21
    }
```

Figure 15.27: Die Klasse Function.

```
public List<AssemblerCmd> compile() {
        Map<String, Integer> symbolTable = new TreeMap();
        Integer count = 0;
3
        for (String var: mParameterList) {
            symbolTable.put(var, count);
            ++count;
        }
        for (Declaration decl: mDeclarations) {
            symbolTable.put(decl.getVar(), count);
            ++count;
10
        }
11
        Integer stackSize = 0;
12
        for (Statement stmnt: mBody) {
            stackSize = Math.max(stackSize, stmnt.stackSize());
14
        }
        List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
16
        AssemblerCmd nl = new NEWLINE();
        result.add(nl);
18
        if (mName.equals("main")) {
            AssemblerCmd main
                                      = new MAIN();
20
            AssemblerCmd limitLocals = new LIMIT("locals", mLocals);
            AssemblerCmd limitStack = new LIMIT("stack", stackSize);
22
            result.add(main);
            result.add(limitLocals);
            result.add(limitStack);
            for (Statement stmnt: mBody) {
26
                 result.addAll(stmnt.compile(symbolTable));
27
            }
            AssemblerCmd myReturn = new RETURN();
            AssemblerCmd endMain = new END_METHOD();
            result.add(myReturn);
            result.add(endMain);
        } else {
33
            AssemblerCmd method
                                      = new METHOD(mName, mParameterList.size());
            AssemblerCmd limitLocals = new LIMIT("locals", mLocals);
35
            AssemblerCmd limitStack = new LIMIT("stack", stackSize);
            result.add(method);
37
            result.add(limitLocals);
            result.add(limitStack);
39
            for (Statement stmnt: mBody) {
                 result.addAll(stmnt.compile(symbolTable));
41
            }
42
            AssemblerCmd endMethod = new END_METHOD();
43
            result.add(endMethod);
        }
45
        return result;
46
    }
47
```

Figure 15.28: Die Methode compile().

Die eigentliche Arbeit der Klasse Funktion wird in der Methode *compile()*, die in Abbildung 15.28 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:

Hier bezeichnet l die Anzahl der in der Funktion main verwendeten lokalen Variablen, s ist die maximale Höhe des Stacks und  $s_1, \dots, s_n$  bezeichnen die einzelnen Assemblerbefehle, die bei der Übersetzung des Rumpfes der Funktion erzeugt werden.

2. Andernfalls hat der erzeugte Code die folgende Form:

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  $s_1, \dots, s_n$  die Assemblerbefehle des Rumpfes der Funktion.

Abbildung 15.29 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.

```
public Integer stackSize() {
    Integer biggest = 0;
    for (Statement stmnt: mBody) {
        biggest = Math.max(biggest, stmnt.stackSize());
    }
    return biggest;
}
```

Figure 15.29: Computing the size of the stack

Zum Abschluss diskutieren wir die Klasse Program, die in Abbildung 15.30 gezeigt wird. Diese Klasse verwaltet in der Member-Variablen mFunctionList die Liste aller zu übersetzenden Funktionen.

Bei der Ubersetzung 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.

```
public class Program {
        private List<Function> mFunctionList;
3
        public Program(List<Function> functionList) {
            mFunctionList = functionList;
        public List<AssemblerCmd> compile() {
            List<AssemblerCmd> fctList = new LinkedList<AssemblerCmd>();
             int indexMain = mFunctionList.size() - 1;
             Function main = mFunctionList.get(indexMain);
10
             fctList.addAll(main.compile());
11
             for (int i = 0; i < indexMain; ++i) {</pre>
12
                 Function f = mFunctionList.get(i);
                 fctList.addAll(f.compile());
14
             }
            return fctList;
16
        }
17
    }
18
```

Figure 15.30: 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 15.2 gezeigte Funktion zur Berechnung der Summe  $\sum_{i=1}^n i$  mit dem Compiler, so erhalten wir die in den Abbildungen 15.8 und 15.9 gezeigten Assembler-Datei. Vergleichen wir dieses Programm mit dem in Abbildung 15.8 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 36: 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 1:* 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*<sup>TM</sup> Language Specification. Sun Microsystems, 1.0 edition, August 1996. appeared also as book with same title in Addison-Wesleys 'The Java Series'.
- [HFA<sup>+</sup>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.

BIBLIOGRAPHY BIBLIOGRAPHY

[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

$2^M$ , 9	Grammatik-Regel, 58
L(F), 26, 31	
$L^st$ , 9	immediate token definition, 16
$L^n$ , 8	input alphabet, 25
$L_1 \cdot L_2$ , 8	
#M, 11	Kleene closure, 9
$\Sigma$ , 25	kontextfreie Grammatik, 58
$\Sigma^*$ , 5	
Ø, 10	length of a string, 5
~, 31 ~, 31	Nfa, 30
PLY, 15	Nicht-Terminal, 57, 58
det(F), 33	
$RegExp_{\Sigma}$ , 9	non-deterministic $F_{SM}$ , 30
$\varepsilon$ , 5, 10	palindrome, 61
	Parse-Baum, 64
$\varepsilon$ transition, 30 $\varepsilon$ -closure, 31	parser generator, 78
	power set, 9
<i>n</i> -th power of a language, 8	•
$r_1 \doteq r_2$ , 12	prime number, 6
ANTLR, 4, 78	product, 8
Ascii-Alphabet, 4	Pumping Lemma for regular languages, 51
DFA, 30	pumping lemma for regular languages, 51
Fsm, 25	regular expression, definition, 9
Ply, 4, 16	regular language, 5, 47
Alleham California	
Ableitungs-Schritt, 59	reversal of a language, 49
accepted language, 26	reversal of a string, 49
accepting state, 24	scanner, 15
accepting states, set of, 25	scanner states, 19
alphabet, 4	separable, 44
17.40	•
complement of a language, 47, 49	separates, 44
complete, finite state machine, 26	set difference, 49
concatenation, 5	start state, 24, 25
configuration (of an NFA), 31	Start-Symbol, 59
context-free language, 5	string, 5
1 1	symbol, 4
dead state, 26	symbolic differentiation, 85
death of an FSM, 25	syntaktische Kategorie, 58
deterministic finite automaton, 30	syntaktische Variable, 57, 58
and alarma of manufacture 10	T
equivalence of regular expressions, 12	Terminal, 57
erreichbar, 43	Terminale, 58
finite state machine, 25	token, 15
formal language, 4, 5	transition function, 25
	universal language 6
functional token definitions, 16	universal language, 6