Formal Languages and Their Applications

An Introduction via PLY

— 2024 —

Prof. Dr. Karl Stroetmann

December 6, 2024

These lecture notes, along with their LaTeX source files and any associated programs, are available on the following GitHub repository:

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

Given the ever-changing nature of the field of computer science, these notes are subject to periodic updates. If you have `git` installed on your computer, you can clone the repository using the following command:

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

To update to the most recent version of these notes, execute the command

```
git pull
```

from the repository's root directory.

Since these notes are frequently updated, they may contain typos or errors. If you find any, please consider submitting a pull request or contacting me via Discord or email at karl.stroetmann@dhbw-mannheim.de.

Contents

1 Introduction and Motivation **4**
- 1.1 Basic Definitions . . . 4
- 1.2 Overview . . . 8
- 1.3 Literature . . . 8
- 1.4 Check your Understanding . . . 9

2 Regular Expressions **10**
- 2.1 Preliminary Definitions . . . 10
- 2.2 The Formal Definition of Regular Expressions . . . 11
- 2.3 Algebraic Simplification of Regular Expressions . . . 14
- 2.4 Check your Understanding . . . 15

3 Building Scanners with Ply **16**
- 3.1 The Structure of a PLY Scanner Specification . . . 17
- 3.2 The Syntax of Regular Expressions in *Python* . . . 19
- 3.3 A Complex Example: Evaluating an Exam . . . 20
- 3.4 Scanner States . . . 20
- 3.5 Check your Understanding . . . 24

4 Finite State Machines **25**
- 4.1 Deterministic Finite State Machines . . . 25
- 4.2 Non-Deterministic Finite State Machines . . . 29
- 4.3 Equivalence of Deterministic and Non-Deterministic FSMs . . . 32
  - 4.3.1 Implementation . . . 37
- 4.4 From Regular Expressions to Non-Deterministic Finite State Machines . . . 37
  - 4.4.1 Implementation . . . 40
- 4.5 Translating a Deterministic FSM into a Regular Expression . . . 40
  - 4.5.1 Implementation . . . 43
- 4.6 Minimization of Finite State Machines . . . 43
  - 4.6.1 Implementation . . . 46
- 4.7 Conclusion . . . 46
- 4.8 Check your Understanding . . . 47

5 The Theory of Regular Languages **48**
- 5.1 Closure Properties of Regular Languages . . . 48
- 5.2 Recognizing Empty Languages . . . 51
- 5.3 Equivalence of Regular Expressions . . . 52
- 5.4 Limits of Regular Languages . . . 52
- 5.5 Check your Understanding . . . 57

6 Context-Free Languages **58**
6.1 Context-Free Grammars . . . 58
6.1.1 Derivations . . . 61
6.1.2 Parse Trees . . . 64
6.1.3 Ambiguous Grammars . . . 65
6.2 Top-Down Parser . . . 66
6.2.1 Rewriting a Grammar to Eliminate Left Recursion . . . 66
6.2.2 Implementing a Top Down Parser in *Python* . . . 69
6.2.3 Implementing a Recursive Descent Parser that Uses an EBNF Grammar . . . 72
6.3 Check your Understanding . . . 74

7 Using Ply as a Parser Generator **75**
7.1 A Symbolic Calculator . . . 75
7.2 Symbolic Differentiation . . . 79
7.3 Implementing a Simple Interpreter . . . 80
7.4 Check your Understanding . . . 92

8 Earley Parser **93**
8.1 The Earley Algorithm . . . 93
8.2 Implementing Earley's Algorithm in *Python* . . . 98
8.3 Check Your Understandig . . . 98

9 Bottom-up Parsers **99**
9.1 Bottom-Up-Parser . . . 99
9.2 Shift-Reduce-Parser . . . 101
9.3 SLR-Parser . . . 108
9.3.1 The Functions *First* and *Follow* . . . 111
9.3.2 Computing the Function *action* . . . 114
9.3.3 Shift/Reduce and Reduce/Reduce Conflicts . . . 118
9.4 Canonical LR Parsers . . . 120
9.5 LALR-Parser . . . 123
9.6 Comparison of SLR, LR, and LALR Parsers . . . 126
9.6.1 SLR Language $\subsetneq$ LALR Language . . . 126
9.6.2 LALR Language $\subsetneq$ Canonical LR Language . . . 127
9.6.3 Evaluation of the Different Methods . . . 128
9.7 Check your Understanding . . . 128

10 Advanced Features of Ply **130**
10.1 Shift-Reduce and Reduce-Reduce Conflicts . . . 130
10.2 Operator Precedence Declarations . . . 131
10.3 Resolving Shift-Reduce and Reduce-Reduce Conflicts . . . 135
10.3.1 Look-Ahead-Konflikte . . . 135
10.3.2 Mysterious Reduce-Reduce Conflicts . . . 136

11 Assembler **138**
11.1 Introduction into JASMIN Assembler . . . 139
11.2 Assembler Instructions . . . 141
11.2.1 Instructions to Manipulate the Stack . . . 143
11.3 An Example Program . . . 147
11.4 Disassembler* . . . 150

12 Development of a Simple Compiler **152**
12.1 The Programming Language *Integer*-C . . . 152
12.2 Developing the Scanner and the Parser . . . 153
12.3 Code Generation . . . 163
12.3.1 Translation of Arithmetic Expressions . . . 163
12.3.2 Translation of Boolean Expressions . . . 166
12.3.3 Translation of Statements . . . 170
12.3.4 Translation of a Function Definition . . . 174
12.3.5 Compiling a Program . . . 177

13 Register-Zuordnung **181**
13.1 Die Ershov-Zahlen . . . 181
13.2 Implementierung eines Compilers für den SRP . . . 182
13.2.1 Diskussion der verschiedenen Klassen . . . 183
13.3 Schlussbetrachtung . . . 189

Chapter 1

Introduction and Motivation

This lecture explores the theory of formal languages and delves into various applications, with a focus on the development of scanners, parsers, interpreters, and compilers. We also introduce a tool designed to aid in the creation of these components, specifically we will introduce PLY, which can be used both as a scanner generator and as a parser generator.

1.1 Basic Definitions

The cornerstone of this lecture is the concept of a *formal language*, essentially defined as a set of strings characterized by precise mathematical criteria. To establish this concept, we first introduce some foundational definitions.

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

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

The term symbol is occasionally used interchangeably with the term character. □

Examples:

(a) $\Sigma = \{0, 1\}$ is the alphabet for representing binary numbers.

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

(c) $\Sigma_{\text{ASCII}} = \{0, 1, \cdots, 127\}$ is known as the ASCII-alphabet. Here, numbers correspond to letters, digits, punctuation marks, and control characters. For instance, the set of number $\{65, \cdots, 90\}$ represents the letters $\{\texttt{A}, \cdots, \texttt{Z}\}$: The number 65 represents the letter "A", the number 66 represents the letter "B", and the number 90 represents the letter "Z". Table 1.1 on the next page shows the complete ASCII-alphabet. Note that the first 32 codes are non-printable control codes, for example the number 10 encodes a line feed control symbol, while the number 13 encodes a carriage return control symbol.

(d) $\Sigma_{\text{UNICODE}} = \{0, 1, \cdots, 154\,998\}$ represents version 16.0 of the UNICODE-alphabet. This extensive set accommodates a wide array of characters, including various scripts, symbols, and even emojis. Table 1.2 shows the Unicode encoding of the 120 most common Chinese characters. ◇

Definition 2 (Strings) For a given alphabet Σ, a string is a sequence of characters selected from Σ. In formal language theory, these sequences are written without brackets or separating commas. Thus, for $c_1, \cdots, c_n \in \Sigma$, we write

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

The empty string is represented by λ, so $\lambda =$ "". The set of all possible strings formed from the alphabet Σ is denoted as Σ^*. Strings are often enclosed in quotation marks for emphasis. □

Character	Code	Character	Code	Character	Code
NUL	0	SOH	1	STX	2
ETX	3	EOT	4	ENQ	5
ACK	6	BEL	7	BS	8
TAB	9	LF	10	VT	11
FF	12	CR	13	SO	14
SI	15	DLE	16	DC1	17
DC2	18	DC3	19	DC4	20
NAK	21	SYN	22	ETB	23
CAN	24	EM	25	SUB	26
ESC	27	FS	28	GS	29
RS	30	US	31	Space	32
!	33	"	34	#	35
$	36	%	37	&	38
'	39	(	40	)	41
*	42	+	43	,	44
-	45	.	46	/	47
0	48	1	49	2	50
3	51	4	52	5	53
6	54	7	55	8	56
9	57	:	58	;	59
<	60	=	61	>	62
?	63	@	64	A	65
B	66	C	67	D	68
E	69	F	70	G	71
H	72	I	73	J	74
K	75	L	76	M	77
N	78	O	79	P	80
Q	81	R	82	S	83
T	84	U	85	V	86
W	87	X	88	Y	89
Z	90	[	91	\	92
]	93	^	94	_	95
`	96	a	97	b	98
c	99	d	100	e	101
f	102	g	103	h	104
i	105	j	106	k	107
l	108	m	109	n	110
o	111	p	112	q	113
r	114	s	115	t	116
u	117	v	118	w	119
x	120	y	121	z	122
{	123	|	124	}	125
~	126	DEL	127		

Table 1.1: ASCII Encoding Table

Glyph	Unicode	Glyph	Unicode	Glyph	Unicode		
一	U+4E00	二	U+4E8C	三	U+4E09	四	U+56DB
五	U+4E94	六	U+516D	七	U+4E03	八	U+516B
九	U+4E5D	十	U+5341	人	U+4EBA	大	U+5927
天	U+5929	地	U+5730	日	U+65E5	月	U+6708
山	U+5C71	水	U+6C34	火	U+706B	木	U+6728
金	U+91D1	土	U+571F	子	U+5B50	女	U+5973
中	U+4E2D	国	U+56FD	王	U+738B	民	U+6C11
学	U+5B66	文	U+6587	本	U+672C	书	U+4E66
爱	U+7231	家	U+5BB6	友	U+53CB	生	U+751F
父	U+7236	母	U+6BCD	男	U+7537	女	U+5973
花	U+82B1	草	U+8349	狗	U+72D7	猫	U+732B
鸟	U+9E1F	鱼	U+9C7C	风	U+98CE	雨	U+96E8
云	U+4E91	雪	U+96EA	电	U+7535	光	U+5149
车	U+8F66	船	U+8239	米	U+7C73	茶	U+8336
饭	U+996D	面	U+9762	东	U+4E1C	西	U+897F
南	U+5357	北	U+5317	左	U+5DE6	右	U+53F3
高	U+9AD8	矮	U+77EE	长	U+957F	短	U+77ED
快	U+5FEB	慢	U+6162	早	U+65E9	晚	U+665A
你	U+4F60	好	U+597D	谢	U+8C22	是	U+662F
不	U+4E0D	我	U+6211	他	U+4ED6	她	U+5979
它	U+5B83	很	U+5F88	要	U+8981	有	U+6709
吃	U+5403	喝	U+559D	看	U+770B	听	U+542C
走	U+8D70	跑	U+8DD1	跳	U+8DF3	说	U+8BF4
问	U+95EE	答	U+7B54	学	U+5B66	校	U+6821
老	U+8001	师	U+5E08	生	U+751F	书	U+4E66

Table 1.2: UNICODE Encoding of some Chinese Characters

Examples:

(a) Let $\Sigma = \{0, 1\}$. If we define

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

then both w_1 and w_2 are strings over Σ. Hence, we can state:

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

(b) Consider $\Sigma = \{\mathsf{a}, \cdots, \mathsf{z}\}$. With

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

it follows that $w \in \Sigma^*$. ◇

The *length* of a string w is denoted by $|w|$ and represents the number of characters in w. We employ array notation for character extraction: given a string w and a natural number $i \leq |w|$, $w[i]$ specifies the i-th character in w. Character counting starts at 0, adhering to conventions in modern programming languages like C, *Java*, and *Python*.

Next, we introduce the concept of concatenation of two strings w_1 and w_2. Concatenation results in a new string w formed by appending w_2 to w_1. It is represented as $w_1 \cdot w_2$. In most books the concatenation of w_1 and w_2 is written without the dot as $w_1 w_2$.

Example: For $\Sigma = \{0, 1\}$ and $w_1 = \text{"01"}$ and $w_2 = \text{"10"}$, the concatenated strings are:

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

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

You may wonder what it means for a subset to be *precisely defined*. A subset L is precisely defined if for every $w \in \Sigma^*$ the notion $w \in L$ is unambiguously defined. For instance, English doesn't qualify as a formal language because it lacks a precise rule set for valid sentences.

This initial definition is intentionally broad. As we progress through the lecture, we will explore specialized versions, most notably regular languages and context-free languages. These are particularly relevant in computer science.

Examples:

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

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

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

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

   Let us define a function

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

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

   (a) $\mathit{value}(0) = 0$, $\mathit{value}(1) = 1$,
   (b) $|w| > 0 \to \mathit{value}(w0) = 2 \cdot \mathit{value}(w)$,
   (c) $|w| > 0 \to \mathit{value}(w1) = 2 \cdot \mathit{value}(w) + 1$.

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

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

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

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

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

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

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

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

   $$p \dagger x \dagger y$$

   such that

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

The above examples illustrate the expansive scope of what can be considered a formal language. While it's straightforward to identify strings in $L_{\mathbb{N}}$, it becomes substantially more challenging for languages like $L_{\mathbb{P}}$ or L_C. Moreover, since the halting problem is undecidable, no algorithm can definitively determine membership in L_u. Nevertheless, L_u is semi-decidable: if a string w belongs to L_u, we can eventually establish this fact.

1.2 Overview

The aim of this lecture is to cover a series of interconnected topics. While this overview will introduce some terms that may not be immediately clear, rest assured that these terms will be elaborated upon as the lecture progresses. The lecture is structured into three main parts:

1. The first part delves into the theory of regular languages.
   (a) We begin with an exploration of regular expressions. After formally defining this term, we will discuss its applications in *Python*.
   (b) We then demonstrate how the PLY tool can be utilized to create scanners.
   (c) Subsequently, we examine the implementation of regular expressions via finite state machines.
   (d) A discussion on how to verify the equivalence of regular expressions follows.
   (e) We conclude this section by discussing the limitations of regular languages, focusing particularly on the Pumping Lemma.
2. The second part is centered on context-free languages, which are instrumental in describing the syntax of programming languages.
   (a) We first introduce context-free grammars, which are employed to define context-free languages.
   (b) Next, we talk about using PLY as a parser generator.
   (c) Essential theory for understanding the parser generator PLY is then presented.
   (d) Lastly, we discuss the limitations of context-free languages.
3. The final part of this lecture covers interpreters and compilers.
   (a) We implement an interpreter for a toy language.
   (b) Then we discuss how to implement a compiler that translates another toy language into Java byte code.

1.3 Literature

Besides these lecture notes, there are three books I highly recommend:

(a) *Introduction to Automata Theory, Languages, and Computation* [HMU06] — Often considered the definitive text on formal languages, this book encompasses all the theoretical concepts discussed in this lecture, although we will only touch on a subset of its content.

(b) *Introduction to the Theory of Computation* [Sip12] — This book provides another accessible introduction to formal languages and extends to the theory of computability, which is beyond the scope of this lecture.

(c) *Compilers — Principles, Techniques and Tools* [ASUL06] — A standard reference in compiler theory, this book also covers a substantial portion of formal languages.

(d) The tool *ChatGPT o1* has reached a quality level that allows it to serve as a tutor for many concepts discussed in this course.

Nevertheless, I would like to warn you that quality of the free version of `ChatGPT` is rather poor and hence is useless when trying to understand scientific theories.

1.4 Check your Understanding

(a) Define the notion of an alphabet.

(b) Based on a given alphabet, define the concept of a string.

(c) Explain the concept of a formal language.

Chapter 2

Regular Expressions

Regular expressions are linguistic constructs designed to specify formal languages simple enough to be recognized by a finite state machine. These machines will be introduced in Chapter 4. At the core, regular expressions enable the specification of:

(a) Alternatives among different options,

(b) Concatenation of elements, and

(c) Repetition of elements.

Regular expressions form the backbone of many modern scripting languages. For instance, the initial surge in the popularity of the programming language *Perl* can be largely attributed to its efficient handling of regular expressions. Today, in most popular high-level programming languages (e.g. *Python*, `C++`, *Java*, `C#`, `JavaScript`, `PHP`, `Ruby`) regular expressions are integrated into the language. In `C` regular expressions are available via the POSIX library.

Moreover, various UNIX utilities, including `grep`, `sed`, and `awk`, are fundamentally based on regular expressions. Consequently, proficiency in regular expressions is indispensable for a budding computer scientist.

In this chapter, we will introduce a definition of regular expressions that is more concise than those typically found in programming languages. This streamlined definition will facilitate our theoretical examination of regular languages in Chapters 4 and 5. The syntax of regular expressions as used in *Python* will be covered in the subsequent chapter.

2.1 Preliminary Definitions

Before delving into the syntax and semantics of regular expressions, we need to introduce some preliminary definitions.

Definition 4 (Product of Languages) Given an alphabet Σ and formal languages $L_1, L_2 \subseteq \Sigma^*$, the product of L_1 and L_2 is denoted as $L_1 \cdot L_2$ and is defined as the set of all concatenations $w_1 \cdot w_2$ where $w_1 \in L_1$ and $w_2 \in L_2$. Formally,

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

Example: Suppose $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and L_1 and L_2 are given by

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

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

Definition 5 (Power of a Language) Let Σ be an alphabet, $L \subseteq \Sigma^*$ a formal language, and $n \in \mathbb{N}$. The n-th power of L, denoted as L^n, is defined inductively as follows:

B.C.: $n = 0$:

$$L^0 := \{\lambda\}, \quad \text{where } \lambda \text{ is the empty string.}$$

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

$L^{n+1} = L^n \cdot L$. ◇

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

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

(b) $L^1 = L$,

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

Definition 6 (Kleene Closure) Given an alphabet Σ and a formal language $L \subseteq \Sigma^*$, the Kleene closure of L, denoted as L^*, is the union of all L^n for $n \in \mathbb{N}$:

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

Note that $\lambda \in L^*$, so L^* is never empty, even when $L = \{\}$.

The Kleene closure is named after Stephen Cole Kleene (1909–1994), who invented regular expressions. ◇

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

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

where $\mathtt{a}^n$ represents a string of n consecutive a's. ◇

The previous example highlights that the Kleene closure of a finite language can be infinite. Specifically, L^* will be infinite whenever L contains a string different from λ.

Definition 7 (Power of a string, s^n) For a string s and a non-negative integer $n \in \mathbb{N}$, the expression s^n is defined inductively:

B.C.: $n = 0$:

$s^0 := \lambda$.

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

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

2.2 The Formal Definition of Regular Expressions

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

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

which interprets each regular expression r as a formal language $L(r) \subseteq 2^{\Sigma^*}$.[1]

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

Definition 8 (Regular Expressions, Stephen Cole Kleene) The set $\texttt{RegExp}_\Sigma$ of regular expressions on the alphabet Σ is defined inductively as follows:

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

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

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

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

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

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

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

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

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

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

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

   Observe that we identify characters with strings of length one.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Given the preceding definition, it is unclear whether we interpret the regular expression

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

as either

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

To remove this ambiguity, we assign operator precedences as follows:

1. The postfix operator "*" has the highest precedence.
2. The infix operator "$\cdot$" has a lower precedence than the postfix operator "*" but a higher precedence than the infix operator "+".
3. The infix operator "+" has the lowest precedence.

With these conventions, the expression

$$\mathrm{a} + \mathrm{b} \cdot \mathrm{c}^*$$

is interpreted as

$$\mathrm{a} + \big(\mathrm{b} \cdot (\mathrm{c}^*)\big).$$

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

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

1. $r_1 := (\mathtt{a} + \mathtt{b} + \mathtt{c}) \cdot (\mathtt{a} + \mathtt{b} + \mathtt{c})$
   This expression r_1 denotes the set of all strings from Σ^* that have length 2:
   $$L(r_1) = \big\{w \in \Sigma^* \;\big|\; |w| = 2\big\}.$$
2. $r_2 := (\mathtt{a} + \mathtt{b} + \mathtt{c}) \cdot (\mathtt{a} + \mathtt{b} + \mathtt{c})^*$
   This expression r_2 denotes the set of all strings from Σ^* that have at least length 1:
   $$L(r_2) = \big\{w \in \Sigma^* \;\big|\; |w| \geq 1\big\}.$$
3. $r_3 := (\mathtt{b} + \mathtt{c})^* \cdot \mathtt{a} \cdot (\mathtt{b} + \mathtt{c})^*$
   This expression r_3 denotes the set of all strings from Σ^* that have exactly one occurrence of the letter a:
   $$L(r_3) = \Big\{w \in \Sigma^* \;\Big|\; \#\big\{i \in \mathbb{N} \;\big|\; w[i] = \mathtt{a}\big\} = 1\Big\}.$$
4. $r_4 := (\mathtt{b} + \mathtt{c})^* \cdot \mathtt{a} \cdot (\mathtt{b} + \mathtt{c})^* + (\mathtt{a} + \mathtt{c})^* \cdot \mathtt{b} \cdot (\mathtt{a} + \mathtt{c})^*$
   This expression r_4 denotes the set of all strings from Σ^* that either contain exactly one occurrence of the letter a or exactly one occurrence of the letter b:
   $$L(r_4) = \Big\{w \in \Sigma^* \;\Big|\; \#\big\{i \in \mathbb{N} \;\big|\; w[i] = \mathtt{a}\big\} = 1\Big\} \cup \Big\{w \in \Sigma^* \;\Big|\; \#\big\{i \in \mathbb{N} \;\big|\; w[i] = \mathtt{b}\big\} = 1\Big\}. \qquad \diamond$$

Remark: The syntax of regular expressions given here is the same as the syntax used in [HMU06]. However, the syntax used for regular expressions in programming languages like *Python* differs. These differences will be discussed later. $\diamond$

Exercise 1:

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

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

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

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

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

2.3 Algebraic Simplification of Regular Expressions

Given two regular expressions r_1 and r_2, we use the notation

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

to indicate that r_1 and r_2 describe the same language. If the equation $r_1 \doteq r_2$ holds, then we refer to r_1 and r_2 as equivalent. The following algebraic laws hold:

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

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

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

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

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

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

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

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

This implies

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

The following equations are more or less obvious.

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

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

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

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

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

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

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

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

We have

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

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

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

holds. In order to prove the inclusion

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

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

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

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

$$w = u_1 \cdots u_n.$$

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

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

Combining these equations yields

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

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

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

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

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

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

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

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

2.4 Check your Understanding

(a) What is the definition of L^* for a given formal language L?

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

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

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

(e) Simplify the following regular expressions:

i. $\emptyset^*$

ii. ε^*

iii. $\varepsilon + r^* \cdot r$

iv. $(\varepsilon + r)^*$

Chapter 3

Building Scanners with Ply

After introducing the concept of regular expressions, we will now explore their practical applications. We examine the parser generator PLY, which is capable of generating both *Python scanners* and *Python parsers*. In this chapter, we will only discuss the generation of scanners via PLY. A scanner is a program designed to segment a given string into a sequence of *tokens*, where a token is a contiguous string of characters that logically group together. To illustrate, consider the input for a C-compiler, which is an ASCII-string representing a valid C program. The compiler initially organizes these characters into various tokens, including:

1. Keywords or reserved words, such as, for example, "`if`", "`while`", "`for`", and "`switch`".
2. Operator symbols, for example "`+`", "`+=`", "`<`", and "`<=`".
3. Parentheses, which include "`(`", "`[`", "`{`", and their corresponding closing symbols.
4. Constants, with C recognizing three types:
   (a) Numerical constants, such as the integer "42" or the floating-point number "`1.23e2`".
   (b) String constants enclosed in double quotes, like "`"hello"`".
   Note that the double quote character "`"`" is part of the string constant.
   (c) Single-character constants wrapped in single quotes, as in "`'a'`".
   Note that the single quote character "`'`" is part of the character constant.
5. Identifiers, which may serve as variable names, function names, or type names.
6. Comments, available in two forms: Single-line comments start with "`//`" and go to the end of the line, while multi-line comments begin with "`/*`" and end with "`*/`".
7. Whitespace characters, such as the blank character `' '`, the tab `'\t'`, the newline `'\n'`, and the carriage return `'\r'`.
   Both whitespace characters and comments are generally discarded by the scanner.

For a more tangible example, consider Figure 3.1 on page 17, which presents a C program that outputs the string "`Hello World!`" along with a newline character. The scanner processes this program into the subsequent list of tokens:

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

Note that the scanner omits all white space characters, except those that occur inside string constants, as they are only used to demarcate tokens. Furthermore, the comment has been removed by the scanner.

While it is feasible to manually create a scanner, using a scanner generator makes the task considerably simpler. We will explore one such utility in the following section.

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

Figure 3.1: A straightforward C program.

3.1 The Structure of a Ply Scanner Specification

In this section, we explore Python Lex-Yacc, commonly known as PLY. According to its official webpage,

> "PLY serves as a lex and yacc parsing toolkit for *Python*."

The toolkit was developed by David Beazley. In this section, we focus solely on its capabilities as a scanner generator. A more comprehensive discussion on its parser generation features will be given in Chapter 7.

To illustrate the core components of a PLY scanner specification, we consider a simple example that tokenizes arithmetic expressions. This example is adapted from the official PLY documentation. A PLY scanner specification generally consists of three key parts, as demonstrated in Figure 3.2:

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'\n+'` matches any positive number of newlines. Hence the counter `lineno` has to be incremented by the length of the string `t.value`.

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

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

"`' \t'`"

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

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

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

Executing the program depicted in Figure 3.2 yields the following output:

```
LexToken(NUMBER,3,1,0)
LexToken(PLUS,'+',1,2)
LexToken(NUMBER,4,1,4)
% ... (rest of the output)
LexToken(NUMBER,2,1,27)
```

The scanner returns tokens as instances of the `LexToken` class, which possess four distinct attributes:

1. The `type` attribute indicates the type of the token. It holds a string value that corresponds to one of the declared tokens.
2. The `value` attribute usually contains the recognized string. However, this attribute is mutable, allowing for transformations. For instance, the function `t_NUMBER` converts this string into an integer.
3. The `lineno` attribute specifies the line number where the token was identified.
4. The `lexpos` attribute serves as a counter, incremented for each character read.

Homework: Install PLY and verify that the previously presented example operates as expected. The program PLY can be installed via the following command provided the appropriate `conda` environment has already been activated:

```
conda install -c conda-forge ply
```

```
import ply.lex as lex

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

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

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

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

t_ignore  = ' \t'

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

lexer = lex.lex()

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

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

Figure 3.2: A simple scanner Specification for PLY.

3.2 The Syntax of Regular Expressions in Python

In the preceding chapter, we introduced regular expressions using a minimalistic syntax. While this simplicity is advantageous for the theoretical discussions in the upcoming chapter—where we explore the implementation of regular expressions via finite state machines—it can be limiting in practical applications. To address this, the *Python* module *re* offers various syntactic abbreviations that facilitate the use of complex regular expressions in a more concise manner. For an in-depth look at these features, I have authored a brief tutorial at

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Chapter-3/02-Regexp-Tutorial.ipynb

that focuses on the key aspects of the `re` module's regular expressions. For an interactive experience, it is advisable to consult this tutorial directly.

3.3 A Complex Example: Evaluating an Exam

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

```
1  Class: Algorithms and Complexity
2  Group: TINF22AI1
3  MaxPoints = 60
4
5  Exercise:      1. 2. 3. 4. 5. 6.
6  Jim Smith:     9 12 10  6  6  0
7  John Slow:     4  4  2  0  -  -
8  Susi Sorglos:  9 12 12  9  9  6
```

Figure 3.3: Results of an Exam

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

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

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Chapter-03/03-Exam-Evaluation.ipynb

3.4 Scanner States

Sometimes, regular expressions are not quite enough and it is beneficial for the scanner to have different states. The following example illustrates this. We will develop a program that is able to convert an Html file into a pure text file. This program is actually quite useful: Some years ago I had a student who was blind. If he read a web page, he would use his Braille display. For him, the Html markup was of no use so if the markup was removed, he could read web pages faster. In order to develop the program to remove Html tags, we have to use scanner states. The idea behind scanner states is that the scanner can use different regular expressions for different parts of the input. For example, the header of an Html file, i.e. the part that is between the `<head>` and `</head>` tags, can just be 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/Python/Chapter-03/04-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 formulas. In fact, these lecture notes have all been typeset using LaTeX. MathML is the part of Html that deals with the representation of mathematical formulas. 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 21 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 21. The program that you are going to develop should transform the LaTeX input file into an Html file. For your convenience, all these files are available in the github directory

Exercises/LaTeX2HTML.

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

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

Figure 3.4: An example LaTeX input file.

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

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

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

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

The area A of a circle is given as

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

while its circumference satisfies

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

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

In order to do this exercise, you have to understand a little bit about LaTeX and about MathML. In the following, we discuss those features of these two language that are needed in order to solve the given problem.

1. A LaTeX input file has the following structure:

(a) The first line list the type of the document. In our example, it reads

`\documentclass{article}`.

This line will be transformed into the following HTML:

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

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

(b) The next line has the form:

`\begin{document}`

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

`<body>`.

(c) After that, the LATEX file consists of text that contains mathematical formula.

(d) The LATEX input file finishes with a line of the form

`\end{document}`.

This line should be translated into the tags

`</body></html>`.

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

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

Here, I have used "···" to represent the mathematical content of the formula.

3. In LATEX, a formula that is displayed in its own line is started and ended with the string "$$". In MATHML, these formulas are called block formulas and are written as

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

Again, I have used "···" to represent the mathematical content of the formula.

4. While in LATEX a mathematical variable does not need any special markup, in MATHML a mathematical variable is written using the tags `<mi>` and `</mi>`. For example, the mathematical variable n is written as

`<mi>n</mi>`.

5. While in LATEX a number does not need any special markup, in MATHML a number is written using the tags `<mn>` and `</mn>`. For example, the number 3.14159 is written as

`<mn>3.14159</mn>`.

6. In LATEX the mathematical constant π is written using the command "\pi". In MATHML, we have to make use of the HTML entity "π" and hence we would write π as

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

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

`<mop>&sdot;</mop>`.

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

   <mop>+</mop>.

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

   x^{2}.

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

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

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

    \sqrt{a+b}.

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

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

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

    \frac{a+b}{2}.

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

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

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

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

    we have to write

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

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

    (a) The first argument is the operator, so in this case it is the entity "&sum".

    (b) The second argument initializes the indexing variable of the sum.

    (c) The third argument provides the upper bound.

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

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

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

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

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

3.5 Check your Understanding

(a) How are regular expressions defined in *Python*?

(b) Do you understand the structure of a PLY scanner specification?

(c) Are you able to use PLY to write a program that reads a given text, finds all numbers inside this text that are followed by a $ symbol and that then converts these numbers to their equivalent value in €?

Chapter 4

Finite State Machines

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

(a) The language described by a regular expression can be defined by a non-deterministic FSM.

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

(c) For every deterministic FSM there is an regular expression specifying the language recognized by the FSM.

4.1 Deterministic Finite State Machines

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

(a) Initialy, the FSM is in a state that is know as the start state.

(b) In every step of its computation, the FSM reads one character of the input string s. Every time a character is processed, the state of the FSM might change.

(c) Some states of the FSM are designated as accepting states. If the FSM has consumed all characters of the given input string s and the FSM has reached an accepting state, then the input string s is accepted and the FSM returns `True`. Otherwise the FSM returns `False` and the string s is rejected.

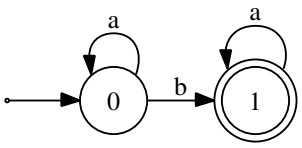

Figure 4.1: An FSM to recognize the language $L(\mathtt{a}^* \cdot \mathtt{b} \cdot \mathtt{a}^*)$.

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

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

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

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

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

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

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

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

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

Definition 9 (Fsm) A finite state machine (abbreviated as FSM) is a 5-tuple

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

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

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

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

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

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

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

where we have:

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

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

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

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

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

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

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

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

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

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

Definition 10 (Accepted Language, $L(F)$) For an FSM $F = \langle Q, \Sigma, \delta, q_0, A\rangle$ the language accepted by F is called $L(F)$ and is defined as

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

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

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

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

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

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

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

Proposition 11 For every FSM F there exists a complete FSM $\widehat{F}$ that accepts the same language as the FSM F, i.e. we have:

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

Proof: Assume F is given as

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

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

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

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

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

1. $\widehat{Q} := Q \cup \{☠\}$,

   the dead state is added to the set Q.

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

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

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

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

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

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

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

   because there is no escape from death.

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

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

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

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

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

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

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

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

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

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

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

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".

State 7: "baa".

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

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

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

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

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

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

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

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

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

The remaining transitions are found in a similar way. Figure 4.2 on page 29 shows the resulting FSM. We still have to explain how we have chosen the start state. When the computation starts, the finite state machine has not read any character. In particular, this implies that neither of the last three characters is the character "b". Hence we can use the state 0 as the start state of our FSM.

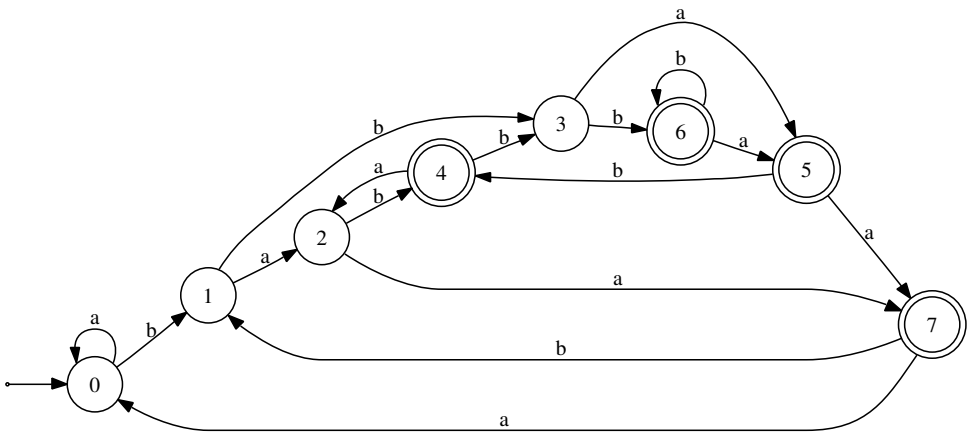

Figure 4.2: An FSM accepting $L\big((\mathtt{a} + \mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a} + \mathtt{b}) \cdot (\mathtt{a} + \mathtt{b})\big)$.

Remark: There is a nice tool available that can be used to better understand finite state machines. This tool is available at

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

4.2 Non-Deterministic Finite State Machines

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

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

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

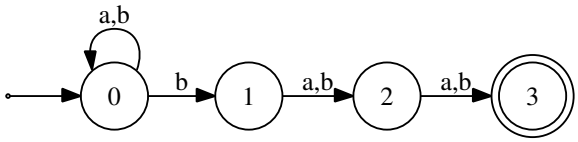

Figure 4.3: A non-deterministic finite state machine to recognize $L\big((\mathtt{a}+\mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a}+\mathtt{b}) \cdot (\mathtt{a}+\mathtt{b})\big)$.

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

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

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

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

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

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

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

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

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

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

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

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

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

It seems that the concept of a non-deterministic FSM is far more powerful than the concept of a deterministic FSM. After all, a non-deterministic FSM appears to have some form of clairvoyance for else it could not guess which states to choose. However, we will prove in the next section that both deterministic and non-deterministic FSMs have the same power to recognize languages: Every language recognized by a non-deterministic FSM is also recognized

by a deterministic FSM. In order to prove this claim, we have to formalize the notion of a non-deterministic FSM. The definition that follows is more general than the informal description of non-deterministic FSMs given so far, as we will allow the FSM to also have ε-transitions. A ε transition allows the FSM to switch its state without reading any character. For example, if there is an ε-transition from the state 1 into the state 2, we write

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

Definition 12 (NFA) A non-deterministic FSM (abbreviated as NFA for non-deterministic automaton) is a 5-tupel

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

such that the following holds:

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

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

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

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

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

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

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

$$q_1 \overset{c}{\mapsto} q_2.$$ □

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

Example: For the FSM F shown in Figure 4.3 on page 30 we have

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

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

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

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

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

$$\langle q, s\rangle$$

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

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

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

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

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

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

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

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

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

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

4.3 Equivalence of Deterministic and Non-Deterministic FSMs

In this section we show how a non-deterministic FSM

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

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

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

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

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

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

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

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

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

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

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

1. $\mathit{ec}(q_0) = \{q_0, q_1, q_2\}$,

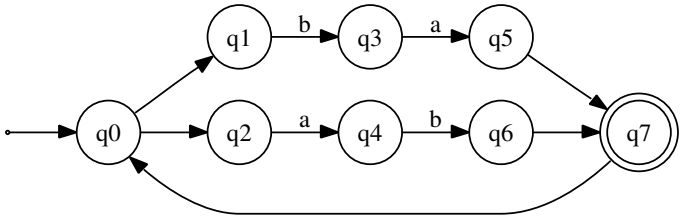

Figure 4.4: A non-deterministic FSM with ε-transitions.

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

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

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

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

This formula is to be read as follows:

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

(b) Then we take the union of all the sets $ec(q_2)$ where $q_2 \in \delta(q_1, c)$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

where the components of this tuple are defined as follows:

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

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

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

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

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

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

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

Exercise 6: Transform the non-deterministic FSM F that is shown in Figure 4.3 on page 30 into the deterministic FSM $\mathit{det}(F)$. ◇

Solution: We start by computing the set of states.

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

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

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

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

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

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

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

The transition function Δ is shown as a table:

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

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

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

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

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

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 from 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).

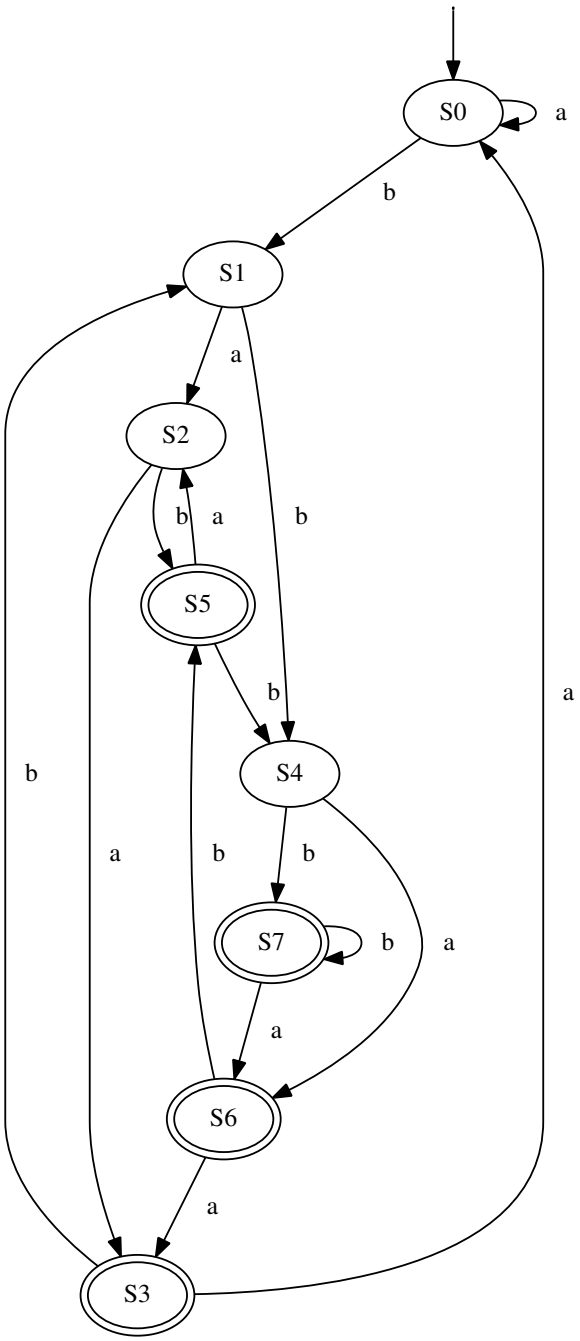

Figure 4.5: The deterministic FSM *det*(F).

Exercise 7: Transform the non-deterministic FSM F that is shown in Figure 4.4 on page 33 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/Chapter-04-05/01-NFA-2-DFA.ipynb

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

4.4 From Regular Expressions to Non-Deterministic Finite State Machines

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

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

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

1. $A(r)$ does not have a transition into its start state.
2. $A(r)$ has exactly one accepting state. Furthermore, there are no transitions out of this state.

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

1. The FSM $A(\emptyset)$ is defined as

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

   Note that this FSM has no transitions at all.

   

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

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

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

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

   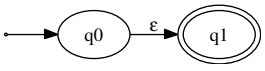

   Figure 4.7: The FSM $A(\varepsilon)$.

   Figure 4.7 shows the FSM $A(\varepsilon)$. We have that $L\big(A(\varepsilon)\big) = \{\lambda\}$, i.e. the FSM accepts only the empty string.

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

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

   Figure 4.8 shows $A(c)$. We have that $L\big(A(c)\big) = \{c\}$, i.e. the FSM accepts only the character c.

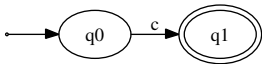

Figure 4.8: The FSM $A(c)$.

4. In order to define the FSM $A(r_1 \cdot r_2)$ for the concatenation $r_1 \cdot r_2$ we assume that we have already constructed finite state machines $A(r_1)$ and $A(r_2)$ such that $L\big(A(r_1)\big) = L(r_1)$ and $L\big(A(r_2)\big) = L(r_2)$. Furthermore, without loss of generality we assume that the states in the FSMs $A(r_1)$ and $A(r_2)$ are different. This can always be achieved by renaming the states of $A(r_2)$. Next, we assume that $A(r_1)$ and $A(r_2)$ have the following form:

   (a) $A(r_1) = \big\langle Q_1, \Sigma, \delta_1, q_1, \{q_2\}\big\rangle$,
   (b) $A(r_2) = \big\langle Q_2, \Sigma, \delta_2, q_3, \{q_4\}\big\rangle$,
   (c) $Q_1 \cap Q_2 = \{\}$.

   Then we can build the FSM $A(r_1 \cdot r_2)$ from $A(r_1)$ and $A(r_2)$ as follows:

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

   Here, the notation $\{\langle q_2, \varepsilon\rangle \mapsto q_3\} \cup \delta_1 \cup \delta_2$ specifies that $A(r_1 \cdot r_2)$ contains all transitions from both $A(r_1)$ and $A(r_2)$ and, furthermore, contains an ε-transition from q_2 to q_3. Formally, this transition function δ can be specified as follows:

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

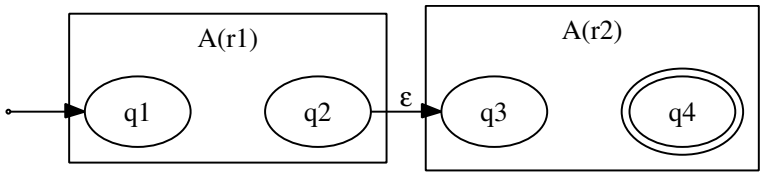

Figure 4.9: The FSM $A(r_1 \cdot r_2)$.

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

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

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

5. In order to define the FSM $A(r_1 + r_2)$ we assume that we have already constructed finite state machines $A(r_1)$ and $A(r_2)$ such that $L\big(A(r_1)\big) = L(r_1)$ and $L\big(A(r_2)\big) = L(r_2)$. Furthermore, without loss of generality we assume that the states in the FSMs $A(r_1)$ and $A(r_2)$ are different and that $A(r_1)$ and $A(r_2)$ have the following form:

   (a) $A(r_1) = \big\langle Q_1, \Sigma, \delta_1, q_1, \{q_3\}\big\rangle$,
   (b) $A(r_2) = \big\langle Q_2, \Sigma, \delta_2, q_2, \{q_4\}\big\rangle$,
   (c) $Q_1 \cap Q_2 = \{\}$.

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

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

   Figure 4.10 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:

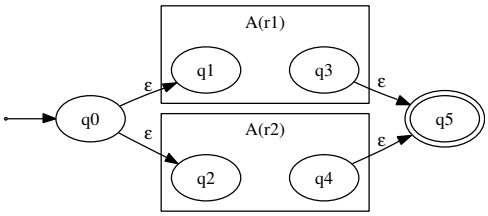

Figure 4.10: The FSM $A(r_1 + r_2)$.

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

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

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

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

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

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

6. In order to define the FSM $A(r^*)$ we assume that we have already constructed a finite state machine $A(r)$ such that $L\big(A(r)\big) = L(r)$ and that $A(r)$ has the form

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

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

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

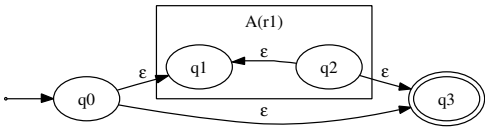

Figure 4.11: The FSM $A(r^*)$.

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

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

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

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

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

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

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

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

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

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

Consider what would happen if you would identify the two states q_0 and q_1 and the two states q_2 and q_3 in step 6 of the construction given above, while also identifying the two states q_2 and q_3 in step 4 of the construction of $A(r_1 \cdot r_2)$. ◇

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

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

4.4.1 Implementation

The *Jupyter notebook*

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Chapter-04-05/03-Regexp-2-NFA.ipynb

implements the theory discussed in this section.

4.5 Translating a Deterministic Fsm into a Regular Expression

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

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

We assume that the FSM F is given as follows:

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

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

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

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

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

To this end we define the ternary relation

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

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

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

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

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

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

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

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

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

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

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

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

B.C.: $k = 0$.

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

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

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

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

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

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

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

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

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

If this set is not empty we have

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

Otherwise we have

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

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

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

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

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

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

Therefore we define

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

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

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

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

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

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, and 2:

1. For $k = 0$ we have:
   (a) $r^{(0)}(0, 0) = \mathtt{a} + \varepsilon$,
   (b) $r^{(0)}(0, 1) = \mathtt{b}$,
   (c) $r^{(0)}(1, 0) = \emptyset$,
   (d) $r^{(0)}(1, 1) = \mathtt{a} + \varepsilon$.
2. For $k = 1$ we have:
   (a) For $r^{(1)}(0, 0)$ we have:
   $$\begin{array}{rcl} r^{(1)}(0,0) & = & r^{(0)}(0,0) + r^{(0)}(0,0) \cdot \big(r^{(0)}(0,0)\big)^* \cdot r^{(0)}(0,0) \\ & \doteq & r^{(0)}(0,0) \cdot \big(r^{(0)}(0,0)\big)^* \end{array}$$
   In the last step we used the fact that
   $$\begin{array}{rcl} r + r \cdot r^* \cdot r & \doteq & r \cdot (\varepsilon + r^* \cdot r) \\ & \doteq & r \cdot r^* \end{array}$$
   to simplify the result. If we substitute for $r^{(0)}(0, 0)$ the expression $\mathtt{a} + \varepsilon$ we get
   $$r^{(1)}(0, 0) \doteq (\mathtt{a} + \varepsilon) \cdot (\mathtt{a} + \varepsilon)^*.$$
   As we have $(\mathtt{a} + \varepsilon) \cdot (\mathtt{a} + \varepsilon)^* \doteq \mathtt{a}^*$ we have
   $$r^{(1)}(0, 0) \doteq \mathtt{a}^*.$$
   (b) For $r^{(1)}(0, 1)$ we have:
   $$\begin{array}{rcl} r^{(1)}(0,1) & \doteq & r^{(0)}(0,1) + r^{(0)}(0,0) \cdot \big(r^{(0)}(0,0)\big)^* \cdot r^{(0)}(0,1) \\ & \doteq & \mathtt{b} + (\mathtt{a} + \varepsilon) \cdot (\mathtt{a} + \varepsilon)^* \cdot \mathtt{b} \\ & \doteq & \mathtt{b} + \mathtt{a}^* \cdot \mathtt{b} \\ & \doteq & (\varepsilon + \mathtt{a}^*) \cdot \mathtt{b} \\ & \doteq & \mathtt{a}^* \cdot \mathtt{b} \end{array}$$
   (c) For $r^{(1)}(1, 0)$ we have:
   $$\begin{array}{rcl} r^{(1)}(1,0) & \doteq & r^{(0)}(1,0) + r^{(0)}(1,0) \cdot \big(r^{(0)}(0,0)\big)^* \cdot r^{(0)}(0,0) \\ & \doteq & \emptyset + \emptyset \cdot (\mathtt{a} + \varepsilon)^* \cdot (\mathtt{a} + \varepsilon) \\ & \doteq & \emptyset \end{array}$$

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

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

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

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

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

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

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

Hint: In order to avoid unnecessary computations, you should only compute those regular expression $r^{(k)}(p,q)$ that are needed in order to compute $r^{(3)}(q_0, q_2)$. $\diamond$

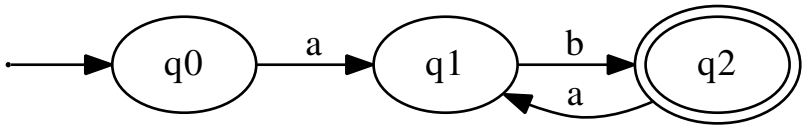

Figure 4.12: A deterministic finite state machine.

4.5.1 Implementation

The *Jupyter notebook*

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Chapter-04-05/05-DFA-2-RegExp.ipynb

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

4.6 Minimization of Finite State Machines

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

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

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

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

which accepts the same language as F, so we have

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

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

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

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

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

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

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

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

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

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

holds. In the following we assume that all states of the FSM F are reachable.

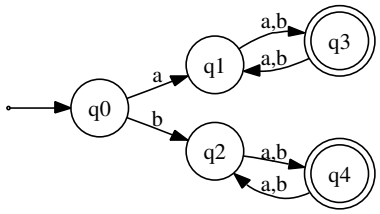

Figure 4.13: A finite state machine with equivalent states.

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

Definition 13 (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 the states p_1 and p_2. □

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

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

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

The question remains how we can determine which states are separable. One possibility is to create a set V with pairs of states. We add the pair $\langle p, q\rangle$ to the set V if we have discovered that p and q are separable. We recognize p

and q as separable if there is a letter $c \in \Sigma$ and two states s and t that are already known to be separable such that

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

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

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

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

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

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

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

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

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

Exercise 12: We consider the FSM shown in figure 4.13 and apply the algorithm described above to this FSM. We use a table for this. The columns and rows of this table are numbered with the different states. If we have recognized in the first step that the states i and j are separable, we insert a 1 in this table in the i-th row and the j-th column. Furthermore, since the states i and j can also be distinguished from the states j and i, we also insert a 1 in the j-th row and the i-th column.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

   (a) $q_1 \sim q_2$

   (b) $q_3 \sim q_4$

   Figure 4.14 shows the corresponding reduced finite $\textsc{Fsm}$.

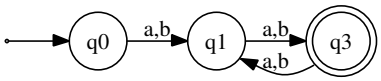

Figure 4.14: The reduced $\textsc{Fsm}$.

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

(a) Construct a non-deterministic $\textsc{Fsm}$ which recognizes this language.

(b) Transform this non-deterministic $\textsc{Fsm}$ into a deterministic $\textsc{Fsm}$.

(c) Minimize the number of states of this $\textsc{Fsm}$ with the algorithm given above.

4.6.1 Implementation

The *Jupyter notebook*

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Chapter-04-05/07-Minimize.ipynb

contains a program that takes a deterministic $\textsc{Fsm}$ F and returns an equivalent $\textsc{Fsm}$ 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 $\textsc{Fsm}$.

(c) Every non-deterministic $\textsc{Fsm}$ can be transformed into an equivalent deterministic $\textsc{Fsm}$.

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

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

4.8 Check your Understanding

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

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

(c) Given a regular expression r, can you describe the steps necessary to construct a non-deterministic $\textsc{Fsm}$ F that accepts the language specified by r?

(d) Given a deterministic $\textsc{Fsm}$ F can you describe how to construct a regular expression r that specifies the language accepted by F?

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

Chapter 5

The Theory of Regular Languages

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

$$L = L(r)$$

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

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

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

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

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

$$r_1 \doteq r_2$$

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

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

is not regular. To summarize, this chapter

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

5.1 Closure Properties of Regular Languages

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

Proposition 14 If L_1 and L_2 are regular languages, then the union of the sets L_1 and L_2, which is the set $L_1 \cup L_2$, is a regular language, too.

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

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

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

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

Since $L_1 \cup L_2$ is the language that is generated by the regular expression $r_1 + r_2$, it is a regular language. □

Proposition 15 If L_1 and L_2 are regular languages, then the intersection of the sets L_1 and L_2, which is the set $L_1 \cap L_2$, is a regular language, too.

Proof: While the proof of Proposition 14 follows directly from the definition of regular expressions, we have to do a little more work here. 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 DFA is complete. Let r_1 and r_2 be the regular expressions that define the languages L_1 and L_2, i.e. we have

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

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

$$L(F_1) = L_1(r_1) \quad \text{and} \quad L(F_2) = L_2(r_2).$$

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

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

holds. We define F as 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

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

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

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

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

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

Therefore we have shown that

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

and this completes the proof. □

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

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

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

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

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

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

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

Proposition 16 If L is a regular language with the alphabet Σ, then the complement of L, which is defined as the language $\Sigma^* \backslash L$, is regular.

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

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

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

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

i.e. the set of accepting states of $\overline{\texttt{det}(F)}$ is the complement of the set of accepting states of F. Then we have

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

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

Corollary 17 If L_1 and L_2 are regular languages over the common alphabet Σ, then the set difference $L_1 \backslash L_2$ is a regular language.

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

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

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

In summary, we have demonstrated that regular languages are closed under the operations of Boolean algebra on sets.

Exercise 14: Let Σ be a given alphabet. For a string $s = c_1 c_2 \ldots c_{n-1} c_n \in \Sigma^*$, the reversal of s is denoted by s^R and is defined as follows:

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

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

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

Subsequently, let us consider a language $L \subseteq \Sigma^*$ that is regular. Your task is to prove that L^R is also a regular language. ◇

5.2 Recognizing Empty Languages

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

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

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

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

1. $q_0 \in R$, because the start state is obviously reachable from the start state.
2. $p_1 \in R \wedge \delta(p_1, c) = p_2 \;\rightarrow\; p_2 \in R$,

   because if p_1 is reachable form q_0 and there is a transition from p_1 to p_2 while reading the character c, then p_2 is reachable from q_o.

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

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

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

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

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

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

5.3 Equivalence of Regular Expressions

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

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

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

Theorem 18 If r_1 and r_2 are regular expressions, then the question whether $r_1 \doteq r_2$ holds is decidable.

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

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

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

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

holds. We have seen in Chapter 4 how F_1 and F_2 can be constructed from r_1 and r_2. According to the corollary 17 the languages $L(r_1)\backslash L(r_2)$ and $L(r_2)\backslash L(r_1)$ are regular and we have seen how to construct FSMs $F_{1,2}$ and $F_{2,1}$ such that

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

holds. Hence we have

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

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

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

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Chapter-04-05/09-Equivalence.ipynb

implements this algorithm.

5.4 Limits of Regular Languages

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

Theorem 19 (Pumping Lemma for Regular Languages, Michael Rabin and Dana Scott 1959, [RS59])
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. $v \neq \lambda$,
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

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

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

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

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

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

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

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

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

$$q_0 \overset{c_1}{\longmapsto} q_1 \overset{c_2}{\longmapsto} q_2 \overset{c_3}{\longmapsto} \cdots \overset{c_m}{\longmapsto} q_m,$$

i.e. on reading the character c_{i+1} in the state q_i the FSM switches into the state q_{i+1} for all $i = 0, \cdots, m-1$. Since we have $s \in L$ we conclude that q_m must be an accepting state, i.e. $q_m \in A$. As $m \geq n$ and n is the total number of states of F, not all of the states

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

can be different. Because of

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

we know, that even in the list

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

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

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

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

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

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

1. Reading the string u changes the state of the FSM F from the start state q_0 to the state q_k, we have

$$q_0 \overset{u}{\longmapsto} q_k. \tag{5.1}$$

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

$$q_k \overset{v}{\longmapsto} q_k. \tag{5.2}$$

3. Reading the string w changes the state of the FSM F from the state $q_l = q_k$ to the accepting state q_m:

$$q_k \overset{w}{\longmapsto} q_m. \tag{5.3}$$

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

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

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

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

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

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

This can be condensed to

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

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

Proposition 20 The alphabet Σ is defined as $\Sigma = \{$ "a" , "b" $\}$. Define the language L as the set of all strings of the form $\mathtt{a}^k\mathtt{b}^k$ where k is some natural number:

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

Then the language L is not regular.

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

$$s = uvw$$

where, furthermore, we have that

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

holds. Let us define the string s as

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

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

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

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

$$v = \mathtt{a}^k \quad \text{for some } k \in \mathbb{N} \text{ such that } k > 0.$$

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

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

In order to facilitate our argument, we define the function

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

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

$$t \in L \Rightarrow \mathit{count}(t, \texttt{'a'}) = \mathit{count}(t, \texttt{'b'}).$$

On one hand we have:

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

But on the other hand we have

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

Therefore, we have

$$\mathit{count}(uw, \texttt{'a'}) < \mathit{count}(uw, \texttt{'b'})$$

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. □

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

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

Prove that the language L is not regular. ◇

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

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

$$L_{\text{square}} = \big\{\texttt{a}^m \;\big|\; \exists k \in \mathbb{N} : m = k^2\big\}$$

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

Solution: The proof employs the method of contradiction. We begin by assuming that L_{square} is a regular language. According to the Pumping Lemma, there exists a natural number $n > 0$ such that for any string $s \in L_{\text{square}}$ with $|s| \geq n$, the string can be decomposed into three parts u, v, and w satisfying the following conditions:

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

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

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

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

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

holds. Then we have the following.

(a) The partition $s = uvw$ has the form $a^{n^2} = a^x a^y a^z$ and hence we have

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

(b) The inequality $|uv| \leq n$ implies $x + y \leq n$, which implies

$$y \leq n. \tag{5.7}$$

(c) From the condition $v \neq \lambda$ we get

$$y > 0. \tag{5.8}$$

(d) The formula $\forall h \in \mathbb{N} : uv^h w \in L_{\text{square}}$ implies

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

In particular, this must hold for $h = 2$:

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

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

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

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

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

Because of $y > 0$ this implies

$$n < k. \tag{5.11}$$

On the other hand we have

$$\begin{array}{rcll} k^2 & = & x + 2 \cdot y + z & \text{according to (5.10)} \\ & = & x + y + z + y & \\ & \leq & x + y + z + n & \text{according to (5.7)} \\ & = & n^2 + n & \text{according to (5.6)} \\ & < & n^2 + 2 \cdot n + 1 & \text{since } n + 1 > 0 \\ & = & (n+1)^2 & \end{array}$$

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

$$k < n + 1. \tag{5.12}$$

The inequalities (5.11) and (5.12) imply

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

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

Exercise 17: Define $\Sigma := \{\mathtt{a}\}$. Prove that the language

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

is not regular. ◇

Proof: The proof is done by contradiction. Assume that L is regular. According to the Pumping Lemma there is a positive natural number n such that all strings $s \in L$ that have a length of at least n can be written as

$$s = uvw$$

where, furthermore, the substrings u, v, and w have the following properties:

1. $v \neq \lambda$,
2. $|uv| \leq n$, and
3. $\forall h \in \mathbb{N} : uv^h w \in L$.

Let us choose a prime number p such that $p \geq n + 2$. Since there are infinitely many prime numbers, this is always possible. Next, define $s := \mathsf{a}^p$. Then we have $|s| = p \geq n$ and we can use the Pumping Lemma to conclude that there must be substrings u, v, and w such that

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

holds. Hence the substrings u, v, and w can only contain the letter "a". Therefore there exist natural numbers x, y, and z such that

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

We can conclude that x, y, and z have the following properties:

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

   Here, $\mathbb{P}$ denotes the set of prime numbers.

If we choose to define $h := x + z$ in the last formula, we get

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

Because of $x + (x + z) \cdot y + z = (x + z) \cdot (1 + y)$ this leads to

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

This is impossible. As $y > 0$ the factor $1 + y$ is different from 1 and because of $x + y \leq n$, $x + y + z = p$, and $p \geq n + 2$ it follows that $z \geq 2$. Hence the factor $x + z$ is also greater than 1. But then the product $(x + z) \cdot (1 + y)$ is not a prime number and this contradiction shows that L can not be regular. □

Exercise 18: The language L_{power} is the set of all strings of the form a^n where n is a power of 2, i.e. we have

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

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

5.5 Check your Understanding

(a) What are the closure properties of regular languages?

(b) How can we check whether two regular expressions are equivalent?

(c) Give the exact wording of the Pumping Lemma.

(d) Can you prove that the language

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

is not regular?

Chapter 6

Context-Free Languages

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

6.1 Context-Free Grammars

Context-free languages are used to describe programming languages. Context-free languages are formal languages like regular languages, but they are much more expressive than regular languages. When we process a program, we not only want decide whether the program is syntactically correct, but we also want to understand the *structure* of the program. The process of *structuring* is also referred to as parsing and the program that does this structuring is called a parser. As input a parser usually does not receive the text of a program, but instead a sequence of so-called terminals, which are also called tokens. These tokens are generated by a scanner, which uses regular expressions to split the program text into single words, which we call tokens in this context.

The parser receives a sequence of tokens from the scanner and has the task of constructing a so-called syntax tree. For this purpose the parser uses a grammar which specifies how the input is to be structured. As an example, consider parsing arithmetic expressions. We define the set *arithExpr* of arithmetic expressions inductively. In order to correctly represent the structure of arithmetic expressions, we define simultaneously the sets *Product* and *Factor*. The set *Product* encompasses arithmetic expressions representing products and quotients, while the set *Factor* includes numbers and parenthesized expressions. Defining these additional sets is crucial for ensuring the correct precedence of operators. The basic building blocks of arithmetic expressions are variables, numbers, the operator symbols "+", "−", "*", "/", and the bracket symbols "(" and ")". Based on these symbols the inductive definition of the sets *Factor*, *Product* and *arithExpr* proceeds as follows:

1. Each number is a factor:

   $$C \in \textsc{Number} \Rightarrow C \in \mathit{factor}.$$

2. Each variable is a factor:

   $$V \in \textsc{Variable} \Rightarrow V \in \mathit{factor}.$$

3. If A is an arithmetic expression and we enclose this expression in parentheses we get an expression that we can use as a factor:

   $$A \in \mathit{arithExpr} \Rightarrow \text{ "(" } A \text{ ")" } \in \mathit{factor}.$$

A note on notation: In the preceding formula, A serves as a meta-variable representing an arbitrary arithmetic expression. The strings "(" and ")" should be taken literally and are thus enclosed in quotation marks. These quotation marks are not part of the arithmetic expression; they are used solely for notational clarity.

4. If F is a factor, then F is also a product:

$$F \in \textit{factor} \Rightarrow F \in \textit{product}.$$

5. If P is a product and if F is a factor, then the strings P "∗" F and P "/" F are also products:

$$P \in \textit{product} \land F \in \textit{factor} \Rightarrow P \text{“∗”} F \in \textit{product} \ \land \ P \text{“/”} F \in \textit{product}.$$

6. Each product is also an arithmetic expression

$$P \in \textit{product} \Rightarrow P \in \textit{arithExpr}.$$

7. If A is an arithmetic expression and P is a product, then the strings A "+" P and A "−" P are arithmetic expressions:

$$A \in \textit{arithExpr} \land P \in \textit{product} \Rightarrow A \text{“+”} P \in \textit{arithExpr} \ \land \ A \text{“−”} P \in \textit{arithExpr}.$$

The sets *factor*, *product*, and *arithExpr* are defined above through mutual recursion. This definition can be succinctly represented using what are known as grammar rules.

$$\begin{aligned}
\textit{arithExpr} &\to \textit{arithExpr} \text{ “+” } \textit{product} \\
\textit{arithExpr} &\to \textit{arithExpr} \text{ “−” } \textit{product} \\
\textit{arithExpr} &\to \textit{product} \\
\textit{product} &\to \textit{product} \text{ “∗” } \textit{factor} \\
\textit{product} &\to \textit{product} \text{ “/” } \textit{factor} \\
\textit{product} &\to \textit{factor} \\
\textit{factor} &\to \text{ “(” } \textit{arithExpr} \text{ “)”} \\
\textit{factor} &\to \textsc{Variable} \\
\textit{factor} &\to \textsc{Number}
\end{aligned}$$

Expressions on the left hand side of a grammar rule are known as syntactic variables or non-terminals, while all other expressions are termed terminals. We adopt the convention of writing syntactic variables in lowercase to align with the syntax used by the parser generator PLY, which we will explore later. However, it is worth noting that in much of the existing literature, the convention is reversed: syntactic variables are typically capitalized, and terminals are presented in lowercase. Additionally, syntactic variables may occasionally be referred to as syntactic categories.

In the example, *arithExpr*, *product*, and *factor* function as the syntactic variables. The remaining elements, namely NUMBER, VARIABLE, and the symbols "+", "−", "∗", "/", "(", and ")", are the terminals or tokens. These terminals are precisely the characters that are not found on the left side of any grammar rule. Terminals can be classified into two types:

1. Operator symbols and separators, like "/" and "(", are used in their literal sense.
2. Tokens such as NUMBER or VARIABLE carry associated values. For NUMBER, the value is numeric; for VARIABLE, it is a string representing the variable's name. To distinguish them from syntactic variables, these token types are always written in uppercase letters.

Grammar rules are often expressed in a notation more compact than that introduced previously. For the given example, the compact notation is as follows:

$$
\begin{array}{rcl}
\textit{arithExpr} & \to & \textit{arithExpr}\ \text{“+”}\ \textit{product} \\
& | & \textit{arithExpr}\ \text{“–”}\ \textit{product} \\
& | & \textit{product} \\
\textit{product} & \to & \textit{product}\ \text{“*”}\ \textit{factor} \\
& | & \textit{product}\ \text{“/”}\ \textit{factor} \\
& | & \textit{factor} \\
\textit{factor} & \to & \text{“(”}\ \textit{arithExpr}\ \text{“)”} \\
& | & \textsc{Number} \\
& | & \textsc{Variable}
\end{array}
$$

In this format, individual alternatives within a rule are demarcated by the metacharacter “|”. Building on the preceding example, we now introduce the formal definition of a context-free grammar.

Definition 21 (Context-Free Grammar) A context-free grammar G is a quadruple

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

where

1. V is a set called syntactic variables or non-terminals. In the preceding example, we have

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

2. T is a set referred to as terminals or tokens. The sets T and V are disjoint, hence

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

   For the aforementioned example, we have

   $$T = \{\textsc{Number}, \textsc{Variable}, \text{“+”}, \text{“–”}, \text{“*”}, \text{“/”}, \text{“(”}, \text{“)”}\}.$$

3. R is a set of grammar rules. Formally, a grammar rule is a pair $\langle A, \alpha \rangle$ where:

   (a) The first component, A, is a syntactic variable:

   $$A \in V.$$

   (b) The second component, α, is a string composed of syntactic variables and terminals:

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

   Consequently, we have

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

   A rule $\langle x, \alpha \rangle$ is typically written as

   $$x \to \alpha.$$

   In the example, the first rule is

   $$\textit{arithExpr} \to \textit{arithExpr}\ \text{“+”}\ \textit{product},$$

   which formally corresponds to the pair

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

4. S is a member of V designated as the start symbol. In the given example, *arithExpr* is the start symbol. ◇

When presenting a grammar, we typically enumerate the grammar rules and apply the following conventions to identify the start symbol and to differentiate between syntactic variables and terminals:

1. The start symbol is the syntactic variable appearing on the left side of the first rule.
2. Symbols enclosed within quotes are terminals.
3. Names beginning with an uppercase letter are terminals.
4. Names beginning with a lowercase letter are syntactic variables.

Example: The set of grammar rules below defines arithmetic expressions constructed from numbers, variables, and the binary operators "+" , "−" , "·" , and "/" .

$$
\begin{array}{rcl}
\textit{arithExpr} & \to & \textit{arithExpr}\ \text{“+”}\ \textit{product} \\
& | & \textit{arithExpr}\ \text{“−”}\ \textit{product} \\
& | & \textit{product} \\
\textit{product} & \to & \textit{product}\ \text{“·”}\ \textit{factor} \\
& | & \textit{product}\ \text{“/”}\ \textit{factor} \\
& | & \textit{factor} \\
\textit{factor} & \to & \text{“(”}\ \textit{arithExpr}\ \text{“)”} \\
& | & \textsc{Number} \\
& | & \textsc{Variable}
\end{array}
$$

In this case, the following observations can be made:

1. The start symbol is *arithExpr*, as it is the syntactic variable on the left in the first grammar rule.
2. *arithExpr*, *product*, and *factor* are recognized as syntactic variables because they start with a lowercase letter.
3. The symbols "+" , "−" , "·" , and "/" are identified as terminals since they are enclosed in quotes.
4. NUMBER and VARIABLE are terminals, indicated by their initial uppercase letters.

6.1.1 Derivations

Next, we aim to identify the language defined by a given grammar G. For this purpose, we first define the concept of a derivation step. Consider the following:

1. $G = \langle V, T, R, S \rangle$ is a grammar,
2. b is a syntactic variable in V,
3. $\alpha b \gamma$ is a string composed of terminals and syntactic variables from $(V \cup T)^*$, which includes the variable b, and
4. $(b \to \delta)$ is a rule in R.

Under these conditions, the string $\alpha b \gamma$ can undergo a derivation step to transform into the string $\alpha \delta \gamma$. This process involves substituting one occurrence of the syntactic variable b with the right-hand side δ of the rule $b \to \delta$. We denote this derivation step as

$$\alpha b \gamma \Rightarrow_G \alpha \delta \gamma.$$

When the grammar G is clear from the context, we may drop the subscript $_G$ and simply use $\Rightarrow$ instead of $\Rightarrow_G$.

The transitive and reflexive closure of the relation $\Rightarrow_G$ is indicated by $\Rightarrow_G^*$. To specify that the derivation of string w from the non-terminal b encompasses n derivation steps, we write:

We illustrate with an example:

$$\begin{aligned}
\textit{arithExpr} &\Rightarrow \textit{arithExpr}\ \text{“+”}\ \textit{product} \\
&\Rightarrow \textit{product}\ \text{“+”}\ \textit{product} \\
&\Rightarrow \textit{product}\ \text{“}\cdot\text{”}\ \textit{factor}\ \text{“+”}\ \textit{product} \\
&\Rightarrow \textit{factor}\ \text{“}\cdot\text{”}\ \textit{factor}\ \text{“+”}\ \textit{product} \\
&\Rightarrow \textsc{Number}\ \text{“}\cdot\text{”}\ \textit{factor}\ \text{“+”}\ \textit{product} \\
&\Rightarrow \textsc{Number}\ \text{“}\cdot\text{”}\ \textsc{Number}\ \text{“+”}\ \textit{product} \\
&\Rightarrow \textsc{Number}\ \text{“}\cdot\text{”}\ \textsc{Number}\ \text{“+”}\ \textit{factor} \\
&\Rightarrow \textsc{Number}\ \text{“}\cdot\text{”}\ \textsc{Number}\ \text{“+”}\ \textsc{Number}
\end{aligned}$$

Hence, we have demonstrated that

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

is valid. To be more precise, we could express this as

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

since the derivation comprises eight steps. By substituting the terminal $\textsc{Number}$ with distinct numerals, we demonstrate that the string

$$2 \cdot 3 + 4$$

is an arithmetic expression. Generally, the language $L(G)$ defined by a grammar G is the set of all strings that are exclusively composed of terminals and can be derived from the start symbol S of the grammar; that is, we define

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

Example: The language

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

is generated by the grammar

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

where the set of rules R is defined as:

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

Proof: We first establish that each string w in L can be derived from the start symbol s:

$$\text{If } w \in L, \text{ then } s \Rightarrow^* w.$$

Let $w_n = (^n)^n$. We use induction on $n \in \mathbb{N}$ to show that $w_n \in L(G)$.

B.C.: $n = 0$.

We have $w_0 = \lambda$. Since the grammar includes the rule $s \to \lambda$, it follows that

$$s \Rightarrow \lambda,$$

and thus $w_0 \in L(G)$.

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

The string w_{n+1} has the form $w_{n+1} = \text{“(”}\, w_n \,\text{“)”}$, where w_n is also in L. By the inductive hypothesis, there exists a derivation for w_n:

$$s \Rightarrow^* w_n.$$

Thus, we obtain the derivation

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

which confirms $w_{n+1} \in L(G)$.

Next, we prove that any string w derived from s belongs to L, using induction on the number n of derivation steps:

B.C.: $n = 1$.

The sole derivation from terminals that is one step long is

$$s \Rightarrow \lambda.$$

Hence, $w = \lambda$, and since $\lambda = (^0)^0$ belongs to L, we deduce $w \in L$.

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

For a derivation exceeding one step, it must begin as

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

This implies

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

By the inductive hypothesis, $v \in L$. Therefore, there exists a $k \in \mathbb{N}$ such that $v = (^k)^k$. Consequently,

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

□

Exercise 19: We define $\Sigma = \{\text{“A”}, \text{“B”}\}$ and define the language L as the set of words $w \in \Sigma^*$ in which the letters “A” and “B” occur with the same frequency:

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

Define a grammar G such that $L = L(G)$. ◇

Exercise 20: Define $\Sigma := \{\text{“A”}, \text{“B”}\}$. In the previous chapter, we have already defined the reversal of a string $w = c_1 c_2 \cdots c_{n-1} c_n \in \Sigma^*$ as the string

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

A string $w \in \Sigma^*$ is called a palindrome if the string is identical to its reversal, i.e. if

$$w = w^R$$

holds true. For example, the strings

$$w_1 = \texttt{ABABA} \quad \text{and} \quad w_2 = \texttt{ABBA}$$

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

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

(a) Prove that the language $L_{\text{palindrome}}$ is a context-free language.

(b) Prove that the language $L_{\text{palindrome}}$ is not regular. ◇

6.1.2 Parse Trees

Using a grammar G, we can not only tell whether a given string s is an element of the language $L(G)$ generated by the grammar, we can also structure the string by building a parse tree. If a grammar is

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

given, a parse tree for this grammar is a tree satisfying the following conditions:

1. The tree consists of two types of nodes:
   (a) The leaf nodes are those nodes that have no outgoing edges.
   (b) The inner nodes are all those nodes that have outgoing edges.
2. Each inner node is labeled with a variable.
3. Each leaf node is labeled with a terminal or with a variable.
4. If a leaf node is labeled with a variable a, then the grammar contains a rule of the form
   $$a \to \lambda.$$
5. If an inner node is labeled with a variable a and the children of of this node are labeled with the symbols X_1, X_2, $\cdots$, X_n, then the grammar G contains a rule of the form
   $$a \to X_1 X_2 \cdots X_n.$$

If we read the leave nodes of a parse tree from left to right, they yield a word that is derived from the grammar G. Figure 6.1 shows a parse tree for the word "2*3+4". It is derived using the grammar given above for arithmetic expressions. The only caveat here is that instead of the symbol "·" the parse tree uses the symbol "$*$".

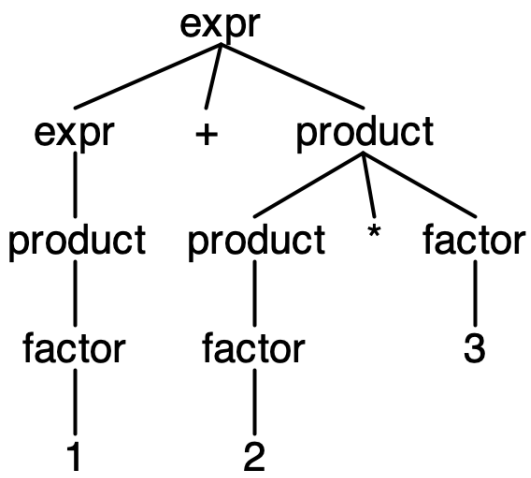

Figure 6.1: A parse tree for the string "2*3+4".

Since trees of the type shown in Figure 6.1 become too large very quickly we simplify these trees using the following rules:

1. Is n an interior node labeled with the variable A and among the children of this node there is exactly one child labeled with a terminal o then we remove this child and label the node n instead with the terminal o.
2. If an inner node has only one child, we replace that node with its child.

We call the tree obtained in this way the abstract syntax tree. Figure 6.2 shows the abstract syntax tree that results from the tree in Figure 6.1. The structure stored in this tree is exactly what we need to evaluate the arithmetic expression "2*3+4", because the tree shows us the order for evaluating the operators.

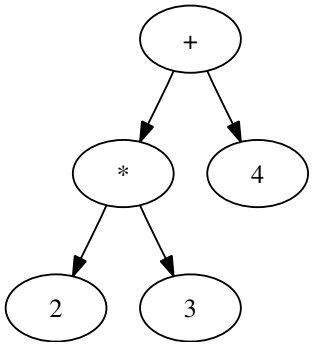

Figure 6.2: The abstract syntax tree for the string "2*3+4".

6.1.3 Ambiguous Grammars

The grammar given at the beginning of section 6.1 to describe arithmetic expressions seems very complicated because it uses three different syntactic categories: *arithExpr*, *product*, and *factor*. We introduce a simpler grammar G which describes the same language:

$$G = \big\langle \{\mathit{expr}\}, \{\textsc{Number}, \textsc{Variable}, \text{“+”}, \text{“−”}, \text{“·”}, \text{“/”}, \text{“(”}, \text{“)”}\}, R, \mathit{expr} \big\rangle,$$

The rules R are given as follows:

$$\begin{array}{lcl}
\mathit{expr} & \to & \mathit{expr}\ \text{“+”}\ \mathit{expr} \\
& | & \mathit{expr}\ \text{“−”}\ \mathit{expr} \\
& | & \mathit{expr}\ \text{“·”}\ \mathit{expr} \\
& | & \mathit{expr}\ \text{“/”}\ \mathit{expr} \\
& | & \text{“(”}\ \mathit{expr}\ \text{“)”} \\
& | & \textsc{Number} \\
& | & \textsc{Variable}
\end{array}$$

In order to show that the string "2 · 3+4" is in the language generated by this grammar, we give the following derivation:

$$\begin{array}{lcl}
\mathit{expr} & \Rightarrow & \mathit{expr}\ \text{“+”}\ \mathit{expr} \\
& \Rightarrow & \mathit{expr}\ \text{“·”}\ \mathit{expr}\ \text{“+”}\ \mathit{expr} \\
& \Rightarrow & 2\ \text{“·”}\ \mathit{expr}\ \text{“+”}\ \mathit{expr} \\
& \Rightarrow & 2\ \text{“·”}\ 3\ \text{“+”}\ \mathit{expr} \\
& \Rightarrow & 2\ \text{“·”}\ 3\ \text{“+”}\ 4
\end{array}$$

This derivation corresponds to the abstract syntax tree shown in Fig. 6.2 is shown. However, there is another derivation of the string "2 · 3+4" with this grammar:

$$\begin{array}{lcl}
\mathit{expr} & \Rightarrow & \mathit{expr}\ \text{“·”}\ \mathit{expr} \\
& \Rightarrow & \mathit{expr}\ \text{“·”}\ \mathit{expr}\ \text{“+”}\ \mathit{expr} \\
& \Rightarrow & 2\ \text{“·”}\ \mathit{expr}\ \text{“+”}\ \mathit{expr} \\
& \Rightarrow & 2\ \text{“·”}\ 3\ \text{“+”}\ \mathit{expr} \\
& \Rightarrow & 2\ \text{“·”}\ 3\ \text{“+”}\ 4
\end{array}$$

This derivation corresponds to the abstract syntax tree shown in Fig. 6.3. In this derivation, the string "2 · 3+4" is apparently taken to be a product, which contradicts the convention that the operator "·" binds stronger than the operator "+". If we were to evaluate the string using the last syntax tree, we would obviously get the wrong result! The reason for this problem is the fact that the last specified grammar is ambiguous. An ambiguous grammar is unsuitable

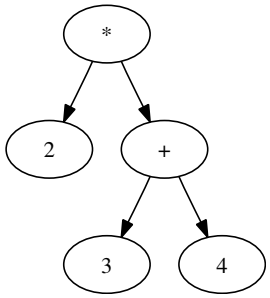

Figure 6.3: Another abstract syntax tree for the string "2 · 3+4". Here "·" is denoted as "∗".

for parsing. Unfortunately, the question of whether a given grammar is ambiguous is, in general, not decidable: It can be shown that this question is equivalent to the Post correspondence problem. Since Post's correspondence problem has been shown to be undecidable, the question whether a grammar is ambiguous is also unsolvable. Proofs of these claims can be found, for example, in the book by Hopcroft, Motwani, and Ullman [HMU06].

6.2 Top-Down Parser

In this section, we present a method that can be used to conveniently parse a whole range of grammars. The basic idea is simple: In order to parse a string w using a grammar rule of the form

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

we try to parse an X_1 first. In doing so, we decompose the string w into the form $w = w_1 r_1$ such that $w_1 \in L(X_1)$ holds. Then we try to find an X_2 in the residual string r_1, thus decomposing r_1 as $r_1 = w_2 r_2$ where $w_2 \in L(X_2)$ holds. Continuing this process, we end up with the string w being split as

$$w = w_1 w_2 \cdots w_n \quad \text{with } w_i \in L(X_i) \text{ for all } i = 1, \cdots, n.$$

Unfortunately, this procedure does not work when the grammar is left-recursive, that is, a rule has the form

$$a \to a\,\beta$$

because then to parse an a we would immediately try again to parse an a and thus we would be stuck in an infinite loop. There are two ways to deal with this kind of problem:

(a) We can rewrite the grammar so that it no longer is left-recursive.

(b) A simpler method is to extend the notion of a context-free grammar. We will use the notion of a so called extended Backus Naur form grammar (abbreviated as EBNF-grammar). Theoretically, the expressive power of EBNF grammars is the same as the expressive power of context-free grammars. In practice, however, it turns out that the construction of top-down parsers for EBNF grammars is easier, because in an EBNF grammar the left recursion can often be replaced by iteration.

In the rest of this chapter we will discuss these two procedures in more detail using the grammar for arithmetic expressions as an example.

6.2.1 Rewriting a Grammar to Eliminate Left Recursion

In the following, assume that a grammar $G = \langle V, T, R, S \rangle$ is given, $a \in V$ is a syntactic variable and the Greek letters β and γ stand for any strings consisting of syntactic variables and tokens, i.e. we have $\beta, \gamma \in (V \cup T)^*$. If a is defined by the two rules

$$\begin{array}{lcl} a & \to & a\beta \\ & \mid & \gamma \end{array}$$

then a derivation of a, where we always replace the syntactic variable a first, has the form

$$a \Rightarrow a\beta \Rightarrow a\beta\beta \Rightarrow a\beta\beta\beta \Rightarrow \cdots \Rightarrow a\beta^n \Rightarrow \gamma\beta^n.$$

Thus we see that the language $L(a)$ described by the syntactic variable a consists of all the strings that can be derived from the expression $\gamma\beta^n$:

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

This language can also be described by the following rules for a:

$$\begin{array}{lcl} a & \to & \gamma b \\ b & \to & \beta b \\ & \mid & \lambda \end{array}$$

Here we have introduced the auxiliary variable b. The derivations resulting from the syntactic variable b have the form

$$b \Rightarrow \beta b \Rightarrow \beta\beta b \Rightarrow \cdots \Rightarrow \beta^n b \Rightarrow \beta^n.$$

Hence the variable b describes the language

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

Thus it is clear that with the grammar rules given above we have

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

To remove the left recursion from the grammar shown in Figure 6.4 on page 68, we need to generalize the example given above. We now consider the general case and assume that a non-terminal a is defined by rules of the following form:

$$\begin{array}{lcl} a & \to & a\beta_1 \\ & \mid & a\beta_2 \\ & \vdots & \vdots \\ & \mid & a\beta_k \\ & \mid & \gamma_1 \\ & \vdots & \vdots \\ & \mid & \gamma_l \end{array}$$

We can reduce this case to the first case by introducing two auxiliary variables b and c:

$$\begin{array}{lcl} a & \to & ab \mid c \\ b & \to & \beta_1 \mid \cdots \mid \beta_k \\ c & \to & \gamma_1 \mid \cdots \mid \gamma_l \end{array}$$

Then we can rewrite this grammar by introducing a new auxiliary variable, let's call it l for list, and get

$$\begin{array}{lcl} a & \to & c\, l \\ l & \to & b\, l \mid \lambda. \end{array}$$

At this point, the auxiliary variables b and c can now be eliminated. This yields the following grammar rules:

$$\begin{array}{lcl} a & \to & \gamma_1\, l \mid \gamma_2\, l \mid \cdots \mid \gamma_l\, l \\ l & \to & \beta_1\, l \mid \beta_2\, l \mid \cdots \mid \beta_k\, l \mid \lambda \end{array}$$

If we apply this procedure to the grammar for arithmetic expressions shown in Figure 6.4, we obtain the grammar shown in Figure 6.5. The variables *exprRest* and *productRest* can be interpreted as follows:

1. *exprRest* describes a list of the form.

   op product $\cdots$ *op product*,

$$
\begin{array}{rcl}
\mathit{expr} & \to & \mathit{expr}\ \text{“+”}\ \mathit{product} \\
& | & \mathit{expr}\ \text{“−”}\ \mathit{product} \\
& | & \mathit{product} \\
\mathit{product} & \to & \mathit{product}\ \text{“·”}\ \mathit{factor} \\
& | & \mathit{product}\ \text{“/”}\ \mathit{factor} \\
& | & \mathit{factor} \\
\mathit{factor} & \to & \text{“(”}\ \mathit{expr}\ \text{“)”} \\
& | & \textsc{Number}
\end{array}
$$

Figure 6.4: Left-recursive grammar for arithmetic expressions.

$$
\begin{array}{rcl}
\mathit{expr} & \to & \mathit{product}\ \ \mathit{exprRest} \\
\mathit{exprRest} & \to & \text{“+”}\ \mathit{product}\ \ \mathit{exprRest} \\
& | & \text{“−”}\ \mathit{product}\ \ \mathit{exprRest} \\
& | & \lambda \\
\mathit{product} & \to & \mathit{factor}\ \ \mathit{productRest} \\
\mathit{productRest} & \to & \text{“·”}\ \mathit{factor}\ \ \mathit{productRest} \\
& | & \text{“/”}\ \mathit{factor}\ \ \mathit{productRest} \\
& | & \lambda \\
\mathit{factor} & \to & \text{“(”}\ \mathit{expr}\ \text{“)”} \\
& | & \textsc{Number}
\end{array}
$$

Figure 6.5: Grammar for arithmetic expressions without left-recursion.

where *op* ∈ { "+" , "−" }.

2. *productRest* describes a list of the form

   op factor ··· *op factor*,

   where *op* ∈ { "·" , "/" } holds.

Exercise 21:

(a) The following grammar describes regular expressions:

regExp	→	*regExp* "+" *regExp*
	|	*regExp regExp*
	|	*regExp* "*"
	|	"(" *regExp* ")"
	|	LETTER

This grammar uses only the syntactic variable {*regExp*} and the following Terminals

{"+", "*", "(", ")", LETTER}.

Since the grammar is ambiguous, this grammar is unsuitable for parsing. Transform this grammar into an unambiguous grammar where the postfix operator "*" binds more strongly than the concatenation of two regular expressions, while the "+" operator binds weaker than concatenation. Use the grammar for arithmetic expressions as a guide and introduce suitable new syntactic variables.

(b) Remove the left recursion from the grammar created in part (a) of this task. ◇

6.2.2 Implementing a Top Down Parser in Python

Now we are ready to implement a parser for recognizing arithmetic expressions. We will use the grammar that is shown in Figure 6.5 on page 68. Before we can implement the parser, we need a scanner. We will use a hand-coded scanner that is shown in Figure 6.6 on page 70. 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. In prticular, we would not be able to use comment inside the regular expression and we would not be able to format the regular expression using white space.

(b) The regular expression `lexSpec` contains 3 alternatives: white space, numbers, and operator symbols. White space is removed, while everything else is collected in the list `result`. Furthermore, the empty string that occurs at the end has to be removed in the same way as white space.

Figure 6.7 on page 71 shows an implementation of a recursive descent parser in PYTHON.

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

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

returns the result

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

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

```
def tokenize(s: str) -> list[str]:
    lexSpec = r'''[ \t]+        |  # blanks and tabs
                  [1-9][0-9]*|0 |  # numbers
                  [-+*/()]      |  # arithmetical operators and parentheses
              '''
    tokenList = re.findall(lexSpec, s, re.VERBOSE)
    result    = []
    for token in tokenList:
        if token == '' or token[0] in [' ', '\t']:        # skip blanks and tabs
            continue
        result += [ token ]
    return result
```

Figure 6.6: A scanner for arithmetic expressions.

(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 → *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.

In order to parse an arithmetic expression, the function first parses a *product* and then it tries to parse the remaining tokens as an *exprRest*. The function `parseExprRest` that is used to parse an *exprRest* needs two arguments:

(a) The first argument is the value of the product that has been parsed by the function `parseProduct`.

(b) The second argument is the list of tokens that can be used.

To understand the mechanics of `parseExpr`, consider the evaluation of

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

Here, the function `parseProduct` will return the result

```
(2, ['+', 3]),
```

where 2 is the result of parsing and evaluating the token list `[1, '*', 2]`, while `['+', 3]` is the part of the input token list that is not used by `parseProduct`. Next, the list `['+', 3]` needs to be parsed as the rest of an expression and then 3 needs to be added to 2.

```
def parse(s):
     TL           = tokenize(s)
     result, Rest = parseExpr(TL)
     assert Rest == [], f'Parse Error: could not parse {TL}'
     return result

def parseExpr(TL):
    product, Rest = parseProduct(TL)
    return parseExprRest(product, Rest)

def parseExprRest(Sum, TL):
    if TL == []:
        return Sum, []
    elif TL[0] == '+':
        product, Rest = parseProduct(TL[1:])
        return parseExprRest(Sum + product, Rest)
    elif TL[0] == '-':
        product, Rest = parseProduct(TL[1:])
        return parseExprRest(Sum - product, Rest)
    else:
        return Sum, TL

def parseProduct(TL):
    factor, Rest = parseFactor(TL)
    return parseProductRest(factor, Rest)

def parseProductRest(product, TL):
    if TL == []:
        return product, []
    elif TL[0] == '*':
        factor, Rest = parseFactor(TL[1:])
        return parseProductRest(product * factor, Rest)
    elif TL[0] == '/':
        factor, Rest = parseFactor(TL[1:])
        return parseProductRest(product / factor, Rest)
    else:
        return product, TL

def parseFactor(TL):
    if TL[0] == '(':
        expr, Rest = parseExpr(TL[1:])
        assert Rest[0] == ')', 'Parse Error: expected ")"'
        return expr, Rest[1:]
    else:
        return int(TL[0]), TL[1:]
```

Figure 6.7: A top down parser for arithmetic expressions.

(c) The function parseExprRest takes a number and a list of tokens. It implements the following grammar rules:

$$
\begin{array}{lcl}
\textit{exprRest} & \to & \text{“+”}\ \textit{product}\ \textit{exprRest} \\
& | & \text{“−”}\ \textit{product}\ \textit{exprRest} \\
& | & \lambda
\end{array}
$$

Therefore, it checks whether the first token is either “+” or “−”. If the token is “+”, it parses a *product*, adds the result of this product to the `sum` of values parsed already and proceeds to parse the rest of the tokens.

The case that the first token is “−” is similar to the previous case. If the next token is neither “+” nor “−”, then it could be either the token “)” or else it might be the case that the list of tokens is already exhausted. In either case, the rule

$$\textit{exprRest} \to \lambda$$

is used. Therefore, in that case we have not consumed any tokens and hence the input argument is already the result.

(d) The function `parseProduct` implements the rule

$$\textit{product} \to \textit{factor}\ \textit{exprRest}.$$

The implementation is similar to the implementation of *parseExpr*.

(e) The function `parseProductRest` implements the rules

$$
\begin{array}{lcl}
\textit{productRest} & \to & \text{“·”}\ \textit{factor}\ \textit{productRest} \\
& | & \text{“/”}\ \textit{factor}\ \textit{productRest} \\
& | & \lambda
\end{array}
$$

The implementation is similar to the implementation of *parseExprRest*.

(f) The function `parseFactor` implements the rules

$$
\begin{array}{lcl}
\textit{factor} & \to & \text{“(”}\ \textit{expr}\ \text{“)”} \\
& | & \textsc{Number}
\end{array}
$$

Therefore, we first check whether the next token is “(” because in that case, we have to use the first grammar rule, otherwise we use the second.

The parser shown in Figure 6.7 does not contain any error handling. Appropriate error handling will be discussed once we have covered the theory of top-down parsing.

Exercise 22: In Exercise 21 on page 69 you have developed a grammar for regular expressions that does not contain left recursion. Implement a top down parser for this grammar. The resulting grammar should return a nested tuple that represents a regular expression. ◇

6.2.3 Implementing a Recursive Descent Parser that Uses an EBNF Grammar

The previous solution to parse an arithmetical expression was not completely satisfying: The reason is that we did not really fix the problem of left recursion but rather cured the symptoms. The underlying reason for left recursion is that context free grammars are not that convenient to describe the structure of programming languages since a description of this structure needs both recursion and iteration, but context-free grammars provide no direct way to describe iteration. Rather, they simulate iteration via recursion. Let us therefore improve the convenience of context-free languages by admitting the regular expression operators “*”, “|”, and “?” on the right hand side of grammar rules. 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

$$
\begin{array}{rcl}
\text{expr} & \rightarrow & \text{product } \big(\text{('+' | '-') product}\big)^* \\
\text{product} & \rightarrow & \text{factor } \big(\text{('*' | '/') factor}\big)^* \\
\text{factor} & \rightarrow & \text{'(' expr ')'} \\
 & | & \texttt{NUMBER}
\end{array}
$$

Figure 6.8: EBNF grammar for arithmetical expressions.

parentheses. These new type of grammars are known as *extended Backus Naur form* grammars, which is abbreviated as EBNF grammars.

It can be shown that the languages described by EBNF grammars are still context-free languages. Therefore, these operators do not change the expressive power of context-free grammars. However, it is often much more convenient to describe a language using an EBNF grammar rather than using a context-free grammar. Figure 6.8 displays an EBNF grammar for arithmetical expressions.

Obviously, the grammar in Figure 6.8 is more concise than the context-free grammar shown in Figure 6.5 on page 68. For example, the first rule clearly expresses that an arithmetical expression is a list of products that are separated by the operators "+" and "−".
Figure 6.9 shows a recursive descent parser that implements this grammar.

1. The function `parseExpr` recognizes a `product` in line 2. The value of this `product` is stored in the variable `result` together with the list `Rest` of those tokens that have not been consumed yet. If the list `Rest` is not empty and the first token in this list is either the operator "+" or the operator "−", then the function `parseExpr` tries to recognize more products. These are added to or subtracted from the `result` computed so far in line 7 or 9. If there are no more products to be parsed, the `while` loop terminates and the function returns the `result` together with the list of the remaining tokens `Rest`.

2. The function `parseProduct` recognizes a `factor` in line 13. The value of this `factor` is stored in the variable `result` together with the list `Rest` of those tokens that have not been consumed yet. If the list `Rest` is not empty and the first token in this list is either the operator "*" or the operator "/", then the function `parseProduct` tries to recognize more factors. The `result` computed so far is multiplied with or divided by these factors in line 18 or 20. If there are no more products to be parsed, the `while` loop terminates and the function returns the `result` together with the list `Rest` of tokens that have not been consumed.

3. The function `parseFactor` recognizes a `factor`. This is either an expression in parentheses or a number.

   - If the first token is a an opening parenthesis, the function tries to parse an expression next. This expression has to be followed by a closing parenthesis. The tokens following this closing parenthesis are not consumed but rather are returned together with the result of evaluating the expression.
   - If the first token is a number, this number is returned together with the list of all those tokens that have not been consumed.

Exercise 23: In Exercise 21 on page 69 you have developed an EBNF grammar for regular expressions. Implement a top down parser for this grammar. The resulting grammar should return a nested tuple that represents a regular expression. ◇

Historical Notes The language ALGOL [Bac59, NBB+60] was the first programming language with a syntax that was based on an EBNF grammar.

```
def parseExpr(TL):
    result, Rest = parseProduct(TL)
    while len(Rest) > 1 and Rest[0] in {'+', '-'}:
        operator = Rest[0]
        arg, Rest = parseProduct(Rest[1:])
        if operator == '+':
            result += arg
        else:               # operator == '-':
            result -= arg
    return result, Rest

def parseProduct(TL):
    result, Rest = parseFactor(TL)
    while len(Rest) > 1 and Rest[0] in {'*', '/'}:
        operator = Rest[0]
        arg, Rest = parseFactor(Rest[1:])
        if operator == '*':
            result *= arg
        else:               # operator == '/':
            result /= arg
    return result, Rest

def parseFactor(TL):
    if TL[0] == '(':
        expr, Rest = parseExpr(TL[1:])
        assert Rest[0] == ')', "ERROR: ')' expected, got {Rest[0]}"
        return expr, Rest[1:]
    else:
        assert isinstance(TL[0], int), "ERROR: Number expected, got {TL[0]}"
        return TL[0], TL[1:]
```

Figure 6.9: A recursive descent parser for the grammar in Figure 6.8.

6.3 Check your Understanding

(a) Define the concept of a context-free grammar.

(b) Assume that G is a context-free grammar. How is the language $L(G)$ defined?

(c) What is the definition of a parse tree?

(d) How do we transform a parse tree into an abstract syntax tree?

(e) What are ambiguous grammars?

(f) How can a context free grammar that contains left recursion be transformed into an equivalent grammar that does not contain left recursion.

(g) How does a top-down parser work?

(h) Why do we have to eliminate left recursion from a grammar in order to build a top-down parser?

(i) Define the notion of an EBNF grammar.

Chapter 7

Using Ply as a Parser Generator

Most[1] modern programming languages can be parsed using a so called LALR-Parser. While we will discuss the theory of LALR languages later, this chapter introduces the parser generator PLY, which can be used to generate a parser for any language that has an LALR grammar. In Chapter 3 we have already seen how PLY can be used to generate a scanner. This chapter focuses on the parser-generating aspect of PLY. If you haven't done so already, you can install PLY via `anaconda` as follows:

```
pip install ply
```

In this chapter, we will discuss three example that use PLY as a parser generator.

1. First, we implement a symbolic calculator using PLY.
2. In our second example, we develop a program for symbolic differentiation. This example shows how PLY can be used to construct an abstract syntax tree.
3. In our final example we create an interpreter for a language that is a small fragment of the programming language `C`.

7.1 A Symbolic Calculator

Figure 7.1 on page 76 shows the grammar of a simple symbolic calculator, i.e. a calculator that supports the use of variables.

In order to generate a symbolic calculator that is based on this grammar we first need to implement a scanner. Figure 7.2 shows how to specify an appropriate scanner with PLY. As we have discussed scanner generation with PLY at length in Chapter 3 there is no need for further discussions here.

Figure 7.3 on page 78 shows how the grammar is implemented in PLY. We discuss it line by line.

1. Line 1 imports the module `ply.yacc`. This module contains the function `ply.yacc.yacc` which is responsible for computing the parse table. The name yacc is a homage to the Unix tool YACC, which is a popular parser generator for the language `C` and, furthermore, is part of the standard utilities of the Unix operating system.
2. Line 3 specifies that the syntactical variable `stmnt` is the start symbol of the grammar.
3. Line 5 – 7 define the function `p_stmnt_assign` which implements the grammar rule

   $$\textit{stmnt} \rightarrow \text{IDENTIFIER}\ \text{":="}\ \textit{expr}.$$

   Note that this grammar rule itself is represented by the document string of the function `p_stmnt_assign`. In general, if

   $$v \rightarrow \alpha$$

[1] The programming language `C++` is a noteable exception.

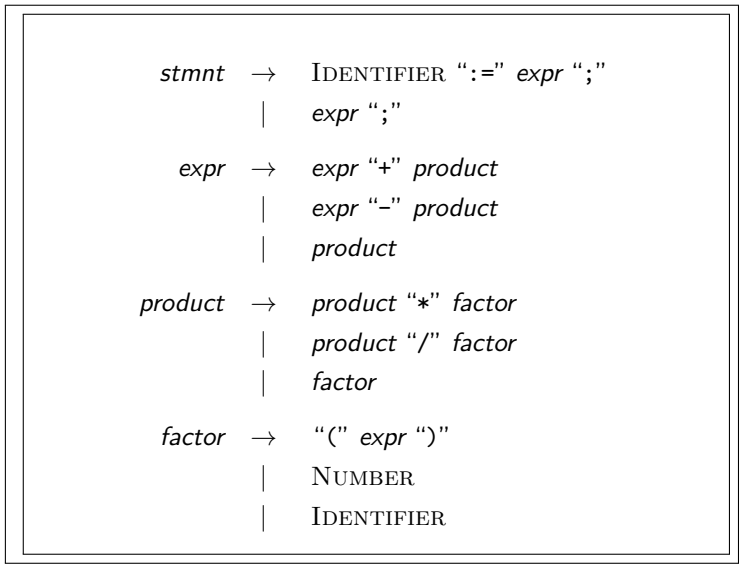

Figure 7.1: A grammar for a symbolic calculator.

is a grammar rule, then this grammar rule is represented by a function that has the name p_v_s. Here, the prefix "p_" specifies that the function implements a grammar rule (the p is short for parser), v should[2] be the name of the variable defined by this grammar rule, and s is a string chosen by the user to distinguish between different grammar rules for the same variable. Of course, s has to be chosen in a way such that the string p_v_s is a legal *Python* identifier.

The function always takes one argument p. This argument is a sequence of objects that can be indexed with array notation. If the grammar rule defining v has the form

$$v \to X_1 \cdots X_n,$$

then this sequence has a length of $n + 1$. If X_i is a token, then p[i] is the property with name value that is associated with this token. Often, this value is just a string, but it can also be a number. If X_i is a variable, then p[i] is the value that is returned when X_i is recognized. The value associated with the variable v is stored in the location p[0]. In the grammar rule shown in line 5–7, we have not assigned any value to p[0] and therefore there is no value associated with the syntactical variable stmnt that is defined by this grammar rule.

Note: Line 6 shows how a grammar rule is represented for PLY. A grammar rule of the form

$$v \to X_1 \cdots X_n$$

is represented as the string:

"v : X_1 $\cdots$ X_n"

It is very important that the character ":" is surrounded by space characters. Otherwise, the parser generator does not work but rather generates error messages that are difficult to understand.

The function p_stmnt_assign has the task of evaluating the expression that is on the right hand side of the assignment operator ":=". The result of this evaluation is then stored in the dictionary Names2Values. The key that is used is the name of the identifier to the left of the assignment operator.

4. The function in line 9 – 11 implements the grammar rule

   stmnt → *expr* ";" .

[2] This is just a convention. Technically, v can be any string that is a valid *Python* identifier.

```
import ply.lex as lex

tokens = [ 'NUMBER', 'IDENTIFIER', 'ASSIGN' ]

def t_NUMBER(t):
    r'0|[1-9][0-9]*(\.[0-9]+)?([eE][+-]?([1-9][0-9]*))?'
    t.value = float(t.value)
    return t

def t_IDENTIFIER(t):
    r'[a-zA-Z][a-zA-Z0-9_]*'
    return t

def t_ASSIGN_OP(t):
    r':='
    return t

literals = ['+', '-', '*', '/', '(', ')', ';']

t_ignore  = ' \t'

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

def t_error(t):
    c = t.value[0]
    n = t.lexer.lexpos
    l = t.lexer.lineno
    print(f"Illegal character '{c}' at character number {n} in line {l}.")
    t.lexer.skip(1)

lexer = lex.lex()
```

Figure 7.2: A scanner for the symbolic calculator.

The rule is implemented by evaluating the expression and then printing it.

5. The function `p_expr_plus` implements the grammar rule

   expr → *expr* "+" *prod*.

   It is implemented by evaluating the expression to the left of the operator "+", which is stored in `p[1]`, and the product to the right of this operator, which is stored in `p[3]`, and then adding the corresponding values. Finally, the resulting sum is stored in `p[0]` so that it is available later as the value of the expression that has been parsed.

   The remaining functions are similar to the ones that are discussed above.

```
import ply.yacc as yacc

start = 'stmnt'

def p_stmnt_assign(p):
    "stmnt : IDENTIFIER ASSIGN expr ';'"
    Names2Values[p[1]] = p[3]

def p_stmnt_expr(p):
    "stmnt : expr ';'"
    print(p[1])

def p_expr_plus(p):
    "expr : expr '+' prod"
    p[0] = p[1] + p[3]

def p_expr_minus(p):
    "expr : expr '-' prod"
    p[0] = p[1] - p[3]

def p_expr_prod(p):
    "expr : prod"
    p[0] = p[1]

def p_prod_mult(p):
    "prod : prod '*' factor"
    p[0] = p[1] * p[3]

def p_prod_div(p):
    "prod : prod '/' factor"
    p[0] = p[1] / p[3]

def p_prod_factor(p):
    "prod : factor"
    p[0] = p[1]

def p_factor_group(p):
    "factor : '(' expr ')'"
    p[0] = p[2]

def p_factor_number(p):
    "factor : NUMBER"
    p[0] = p[1]

def p_factor_id(p):
    "factor : IDENTIFIER"
    p[0] = Names2Values.get(p[1], float('nan'))
```

Figure 7.3: A parser for the symbolic calculator, part I.

```
1  def p_error(p):
2      if p:
3          print(f'Syntax error at {p.value} in line {p.lexer.lineno}.')
4      else:
5          print('Syntax error at end of input.')
6
7  parser = yacc.yacc()
8
9  Names2Values = {}
10
11 def main():
12     while True:
13         s = input('calc > ')
14         if s == '':
15             break
16         yacc.parse(s)
```

Figure 7.4: A parser for the symbolic calculator, part II.

Figure 7.4 on page 79 is discussed next.

1. Line 1 – 5 shows the function `p_error` which is used to print error messages in the case that the input can not be parsed because of a syntax error. The argument `p` is the token t that caused the entry `action`(s, t) in the action table to be undefined. If the syntax error happens at the end of the input, `p` has the value `None`.

2. Line 7 generates the parser.

3. Line 9 initializes the dictionary `Names2Values`. For every identifier x defined interactively, `Names2Values`$[x]$ is the value associated with x.

4. The function `main` is used as a driver for the parser. It reads a string `s` from the command line and tries to parse `s` using the function `yacc.parse`. The function `yacc.parse` is generated behind the scenes when the function `yacc.yacc` is invoked in line 7.

The Jupyter notebook `01-Calculator`, which is available at

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Chapter-07/01-Calculator.ipynb

implements the symbolic calculator discussed in this section.

7.2 Symbolic Differentiation

Our next example shows how we can compute an abstract syntax tree with the help of PLY. The Jupyter notebook `02-Differentiator`, which is available at

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Chapter-07/02-Differentiator.ipynb

implements a parser that generates an abstract syntax tree, which can then be used for symbolic differentiation.

7.3 Implementing a Simple Interpreter

Figure 7.5 show the grammar of a simple programming language. In this section, we will develop an interpreter for this language.

```
1   program
2       : /* epsilon */
3       | stmnt program
4
5   stmnt
6       : IF '(' bool_expr ')' stmnt
7       | WHILE '(' bool_expr ')' stmnt
8       | '{' stmnt_list '}'
9       | ID ':=' expr ';'
10      | expr ';'
11
12  bool_expr
13      : expr '==' expr
14      | expr '!=' expr
15      | expr '<=' expr
16      | expr '>=' expr
17      | expr '<'  expr
18      | expr '>'  expr
19
20  expr: expr '+' product
21      | expr '-' product
22      | product
23
24  product
25      : product '*' factor
26      | product '/' factor
27      | factor
28
29  factor
30      : '(' expr ')'
31      | NUMBER
32      | ID
33      | ID '(' expr_list ')'
34
35  expr_list
36      :
37      | expr ',' ne_expr_list
38
39  ne_expr_list
40      : expr
41      | expr ',' ne_expr_list
```

Figure 7.5: A grammar for a simple programming language.

The statements of the language that we want to implement include assignments, function calls, `if` statements, and `while` loops. The language supports boolean expression that can be build with the help of the comparison operators

==, !=, <=, >=, <, and >.

Arithmetic expression can use the arithmetic operators

+, +, -, *, and /.

In the base case, arithmetic expressions are build from numbers, variables, and function calls. Although the language supports the calling of predefined functions like `read` and `print`, it does not support user defined functions.

Figure 7.6 shows an example program that conforms to this grammar. This program first reads a number which is stored in the variable `n`. Subsequently, the sum

$$s := \sum_{i=1}^{n^2} i$$

is computed using a `while` loop. Finally, the sum is printed.

```
1  n := read();
2  s := 0;
3  i := 0;
4  while (i < n * n) {
5      i := i + 1;
6      s := s + i;
7  }
8  print(s);
```

Figure 7.6: A program to compute the sum $\sum_{i=0}^{n^2} i$.

Similar to our program for symbolic differentiation, we will represent the individual commands as nested tuples. The program shown in Figure 7.6 is represented by the nested tuple displayed in Figure 7.7. This nested tuple is nothing other than the abstract syntax tree corresponding to the original program. Note that lists of commands are represented as nested tuples that start with a ".".

```
1  ('.',
2      ('read', 'n'),
3      (':=', 's', 0),
4      (':=', 'i', 0),
5      ('while', ('<',  'i', ('*', 'n', 'n')),
6          ('.',
7              (':=', 'i', ('+', 'i', 1)),
8              (':=', 's', ('+', 's', 'i'))
9          )
10     ),
11     ('print', 's')
12 )
```

Figure 7.7: A nested tuple that represents the program from Figure 7.6.

Figure 7.8 on 82 show the implementation of the scanner. The scanner primarily differentiates between variables (token ID) and numbers (token NUMBER). Variables start with an uppercase or lowercase letter, followed by additional digits and the underscore. Furthermore, there are operator symbols for the comparison operators "==", "!=", "<=", ">=", "<", and ">", for the arithmetic operators "+", "-", "*", "/", the parentheses "(" and ")", and for the assignment operator ":=". Note that only those operators names that consist of more than two characters have to

```
tokens = [ 'NUMBER',    # r'0|[1-9][0-9]*'
           'ID',        # r'[a-zA-Z][a-zA-Z0-9_]*'
           'ASSIGN',    # r':='
           'EQ',        # r'=='
           'NE',        # r'!='
           'LE',        # r'<='
           'GE',        # r'>='
           'IF',        # r'if'
           'WHILE'      # r'while'
         ]

def t_COMMENT(t):
    r'(/\*(.|\n)*?\*/)|(//.*)'
    t.lexer.lineno += t.value.count('\n')
    pass

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

t_ASSIGN = r':='
t_EQ     = r'=='
t_NE     = r'!='
t_LE     = r'<='
t_GE     = r'>='

Keywords = { 'if': 'IF', 'while': 'WHILE' }

def t_ID(t):
    r'[a-zA-Z][a-zA-Z0-9_]*'
    t.type = Keywords.get(t.value, 'ID')
    return t

literals = ['+', '-', '*', '/', '%', '(', ')', '{', '}', ';', '<', '>', ',']

t_ignore  = ' \t\r'
```

Figure 7.8: Implementation of the scanner.

be declared as tokens, since the other symbols can be declared as `literals`. Finally, the language is equipped with two keywords "`if`" and "`while`". The scanner is pretty standard, but there are two things worth mentioning.

1. The scanner removes white space and comments. The language supports two kinds of comments:

   (a) The language supports multiline comments, i.e. text surrounded by "`/*`" "`*/`". These comments have the form

   ```
   /* ... */
   ```

   where "`...`" denotes arbitrary text not containing the substring "`*/`".
   We have to uses used the so-called *non-greedy* version of the Kleene operator "`*`" in the specification of multiline comments. The non-greedy version of the operator "`*`" is written as "`*?`" and matches as little as possible. Therefore, the regular expression

```
/\*(.|\n)*?\*/
```

represents a string that starts with the character sequence "/*", ends with the character sequence "*/", and is as short as possible. This ensures that in a line like

```
/* Hugo */ i := i + 1; /* Anton */
```

two separate comments are recognized.

(b) Furthermore, the language supports one line comments that start with the character sequence "//" and extend to the end of the line.

Note that the symbol COMMENT is not declared as a token. The reason is that it won't appear in the grammar rules later, as comments are discarded by the scanner.

2. The treatment of keywords requires special care. Since both "`if`" and "`while`" have the same syntactical form as variables, the function `t_ID` recognizes them. This is done with the help of the dictionary `Keywords`. If `t_ID` finds a string of characters, it first checks, whether this string has an associated token value in the dictionary `Keywords`. If there is a value associated with the string, the string is a keyword and the corresponding token is returned. On the other hand, if the string is not stored in the dictionary `Keywords`, the function `get` returns the token ID as a default.

The figures 7.9, 7.10, 7.11, and 7.12 show the implementation of the parser using PLY. 7.9 on page 84 implements the following grammar rules:

```
1  program
2      : /* epsilon */
3      | stmnt program
4
5  stmnt
6      : IF '(' bool_expr ')' stmnt
7      | WHILE '(' bool_expr ')' stmnt
8      | '{' stmnt_list '}'
9      | ID ':=' expr ';'
10     | expr ';'
```

The start symbol of the grammar is the variable `program`. When parsing this variable, the parser returns a nested tuple of statements. The first element of this tuple is a dot ".".

The syntactic variable `stmnt` describes the various commands that are supported in our simple language.

(a) A conditional test has the syntax:

`if (` b `)` *stmnt*

Here, b is an expression that evaluates to `True` or `False` and `stmnt` is a statement that is executed if b evaluates to `True`. This command is represented by the nested tuple

('if', b, *stmnt*).

Here, b is a nested tuple representing the Boolean expression.

(b) A loop has the syntax:

`while (` b `)` *stmnt*.

Here, b is an expression that evaluates to `True` or `False` and `stmnt` is a statement that is executed as long as b evaluates to `True`. This statement is represented by the nested tuple

('while', b, *stmnt*).

```
def p_program_one(p):
    "program : stmnt_list"
    p[0] = p[1]

def p_stmnt_list_empty(p):
    "stmnt_list : "
    p[0] = ('.',)

def p_stmnt_list_more(p):
    "stmnt_list : stmnt stmnt_list"
    p[0] = ('.', p[1]) + p[2][1:]

def p_stmnt_if(p):
    "stmnt : IF '(' bool_expr ')' stmnt"
    p[0] = ('if', p[3], p[5])

def p_stmnt_while(p):
    "stmnt : WHILE '(' bool_expr ')' stmnt"
    p[0] = ('while', p[3], p[5])

def p_stmnt_block(p):
    "stmnt : '{' stmnt_list '}'"
    p[0] = p[2]

def p_stmnt_assign(p):
    "stmnt : ID ASSIGN expr ';'"
    p[0] = (':=', p[1], p[3])

def p_stmnt_expr(p):
    "stmnt : expr ';'"
    p[0] = p[1]
```

Figure 7.9: Specification of statements.

(c) A block statement is a list of statements that is surrounded by curly braces.

(d) The simplest commands are the assignments. These have the form:

$$v \texttt{ := } e\texttt{;}$$

Here, v is a variable and e is an arithmetic expression. An assignment is represented by the nested tuple

$$(\texttt{':='}, v, e).$$

(e) Furthermore, every expression is a statement if it is followed by a semicolon. The reason to allow this is that some expressions have side effects. Syntactically, the string "`print(1+2)`" is an expression as it is a function call. By writing it as

```
print(1+2);
```

we can execute it as a statement.

```
def p_bool_expr_eq(p):
    "bool_expr : expr EQ expr"
    p[0] = ('==', p[1], p[3])

def p_bool_expr_ne(p):
    "bool_expr : expr NE expr"
    p[0] = ('!=', p[1], p[3])

def p_bool_expr_le(p):
    "bool_expr : expr LE expr"
    p[0] = ('<=', p[1], p[3])

def p_bool_expr_ge(p):
    "bool_expr : expr GE expr"
    p[0] = ('>=', p[1], p[3])

def p_bool_expr_lt(p):
    "bool_expr : expr '<' expr"
    p[0] = ('<', p[1], p[3])

def p_bool_expr_gt(p):
    "bool_expr : expr '>' expr"
    p[0] = ('>', p[1], p[3])

```

Figure 7.10: Specification of boolean expressions.

Figure 7.10 shows the parsing of boolean expressions. It implements the following grammar rule.

```
bool_expr
    : expr '==' expr
    | expr '!=' expr
    | expr '<=' expr
    | expr '>=' expr
    | expr '<'  expr
    | expr '>'  expr
```

The syntactic variable `boolExpr` describes an expression that takes a Boolean value.

1. An expression of the form

   $$l == r$$

   tests whether the evaluations of l and r yield the same values. This expression is represented by the nested tuple

   $$('==', l, r).$$

2. The other cases are similar.

Figure 7.11 on page 87 implements the following grammar rules:

```
expr: expr '+' product
    | expr '-' product
    | product

product
    : product '*' factor
    | product '/' factor
    | product '%' factor
    | factor

factor
    : '(' expr ')'
    | NUMBER
    | ID
    | ID '(' expr_list ')'
```

To understand the code shown in Figure 7.11 we have to understand the following:

1. An arithmetic expression of the form $a + b$ is represented as a triple of the form

   `('+', x, y)`

   where `x` and `y` are the tuples corresponding to the abstract syntax trees of a and b, respectively.

2. Arithmetic expressions using the operators "-", "*", "/", "%" are represented analogously.

3. An parenthesized expression, i.e. an expression of the form

   (e)

   is represented by the abstract tree that represents e. Therefore, the parentheses are just used for grouping and do not appear in the abstract syntax tree.

4. A number is represented by itself.

5. A variable is represented by its name, i.e. the variable `x` is represented by the string "`x`".

6. A function call of the form

   $f(a_1, \cdots a_n)$

   is represented by the tuple

   `('call',` f`,` a_1`, ⋯,` a_n`)`.

```
def p_expr_plus(p):
    "expr : expr '+' product"
    p[0] = ('+', p[1], p[3])

def p_expr_minus(p):
    "expr : expr '-' product"
    p[0] = ('-', p[1], p[3])

def p_expr_product(p):
    "expr : product"
    p[0] = p[1]

def p_product_times(p):
    "product : product '*' expr"
    p[0] = ('*', p[1], p[3])

def p_product_divide(p):
    "product : product '/' expr"
    p[0] = ('/', p[1], p[3])

def p_product_modulo(p):
    "product : product '%' factor"
    p[0] = ('%', p[1], p[3])

def p_product_factor(p):
    "product : factor"
    p[0] = p[1]

def p_factor_paren(p):
    "factor : '(' expr ')'"
    p[0] = p[2]

def p_factor_number(p):
    "factor : NUMBER"
    p[0] = p[1]

def p_factor_id(p):
    "factor : ID"
    p[0] = p[1]

def p_factor_fct_call(p):
    "factor : ID '(' expr_list ')'"
    p[0] = ('call', p[1]) + p[3][1:]
```

Figure 7.11: Specification of arithmetic expressions.

Figure 7.12 on page 88 implements the following grammar rules:

```
expr_list
    :
    | expr ',' ne_expr_list

ne_expr_list
    : expr
    | expr ',' ne_expr_list
```

```
def p_expr_list_empty(p):
    "expr_list : "
    p[0] = ('.',)

def p_expr_list_one(p):
    "expr_list : expr"
    p[0] = ('.', p[1])

def p_expr_list_more(p):
    "expr_list : expr ',' ne_expr_list"
    p[0] = ('.', p[1]) + p[3][1:]

def p_ne_expr_list_one(p):
    "ne_expr_list : expr"
    p[0] = ('.', p[1])

def p_ne_expr_list_more(p):
    "ne_expr_list : expr ',' ne_expr_list"
    p[0] = ('.', p[1]) + p[3][1:]
```

Figure 7.12: Specification of lists of expressions.

```
def main(file):
    with open(file, 'r') as handle:
        program = handle.read()
    stmnt = yacc.parse(program)
    Values = {}
    execute(stmnt, Values)
```

Figure 7.13: Specification of the function `main`.

Figure 7.13 shows the function `main`, which receives the name of a file containing a program in our simple programming language as input. This program is parsed and thus transformed into a tuple of statements. The function `execute` executes this tuple. In order to do so, it needs to know the values of all variables. Theses are stored in the dictionary `Values`, which initially is empty. Every time a variable is assigned, the variable will and the corresponding value will be stored in this dictionary.

Figure 7.14 shows the implementation of the function `execute`. This implementation consists mainly of a large case distinction based on the type of command to be executed.

```
def execute(stmnt: NestedTuple, Values: dict[str, Number]) -> None:
    match stmnt:
        case ('.', *SL):
            execute_tuple(tuple(SL), Values)
        case (':=', var, value):
            Values[var] = evaluate(value, Values)
        case ('call', 'print', expr):
            print(evaluate(expr, Values))
        case ('if', test, stmnt):
            if evaluate_bool(test, Values):
                execute(stmnt, Values)
        case ('while', test, stmnt):
            while evaluate_bool(test, Values):
                execute(stmnt, Values)
        case _:
            assert False, f'{stmnt} unexpected'

def execute_tuple(Statement_List, Values={}):
    for stmnt in Statement_List:
        execute(stmnt, Values)
```

Figure 7.14: The function `execute`.

1. First, we check if `stmnt` is a list of statements. A statement is a list of statements if the first element of the tuple representing this statement is the string `'.'`.

   In this case, the statements following the string `'.'` are executed via the auxiliary function `execute_tuple`.

2. If the statement is an assignment of the form

   ```
   (':=', var, value)
   ```

   then the value of the expression `value` is computed using the function `evaluate`. This value is then stored in the dictionary `Values` under the key `var`.

3. If the statement is an expression statement of the form

   ```
   ('print', expr)
   ```

   then the expression `expr` is first evaluated using the function `evaluate`. The resulting value is then printed.

4. If the command is of the form

   ```
   ('if', test, stmnt)
   ```

   then `test` is a nested tuple representing a Boolean expression, while `stmnt` is a statement. In this case, the expression `test` is first evaluated using the function `evaluate`. If this evaluation yields the value `True`, then `stmnt` is executed.

5. If the command is of the form

   ```
   ('while', test, stmnt)
   ```

   then `test` is a Boolean expression and `stmnt` is a statement. In this case, the expression `test` is first evaluated using the function `evaluate`. If this evaluation yields the value `True`, then `stmnt` is executed. Then the expression `test` is evaluated again. If the result is `False`, then the execution of the while loop finishes. Otherwise, `stmnt` is executed as long as the evaluation of `test` yields `True`.

```
def evaluate_bool(expr, Values):
    match expr:
        case ('==', lhs, rhs):
            return evaluate(lhs, Values) == evaluate(rhs, Values)
        case ('!=', lhs, rhs):
            return evaluate(lhs, Values) != evaluate(rhs, Values)
        case ('<=', lhs, rhs):
            return evaluate(lhs, Values) <= evaluate(rhs, Values)
        case ('>=', lhs, rhs):
            return evaluate(lhs, Values) >= evaluate(rhs, Values)
        case ('<', lhs, rhs):
            return evaluate(lhs, Values) <  evaluate(rhs, Values)
        case ('>', lhs, rhs):
            return evaluate(lhs, Values) >  evaluate(rhs, Values)
        case _:
            assert False, f'{expr} unexpected'
```

Figure 7.15: The evaluation of boolean expressions.

Figure 7.15 shows the implementation of the function `evaluate_bool`. This function receives a boolean expression and a dictionary storing the current values of the variables as input.

(a) If the expression to be evaluated is a Boolean expression of the form

```
('==', lhs, rhs)
```

we recursively evaluate the expressions `lhs` and `rhs` and return `True` if and only if both expressions yield the same value.

(b) If the expression to be evaluated is a Boolean expression of the form

```
('<', lhs, rhs)
```

we recursively evaluate the expressions `lhs` and `rhs` and return `True` if and only if the value obtained from the evaluation of `lhs` is less than the value obtained from the evaluation of `rhs`.

The remaining cases are similar.

Figure 7.16 shows the implementation of the function `evaluate`. This function receives an arithmetic expression and a dictionary storing the values of the variables as input.

(a) If the expression to be evaluated is a number, we return this number as the result.

(b) If the expression to be evaluated is a variable, we look up the value of this variable in the dictionary `Values` and return this value as the result.

(c) If the expression to be evaluated is a call to the function `read`, we prompt the user to enter a natural number. We then convert the string entered by the user into an integer.

(d) If the expression to be evaluated is a sum of the form

```
('+', lhs, rhs)
```

we recursively evaluate the expressions `lhs` and `rhs` and return the sum of these values.

(e) The evaluation of the arithmetic operators `'-'`, `'*'`, and `'/'` is analogous to the evaluation of the operator `'+'`.

```
def evaluate(expr, Values):
    match expr:
        case int():
            return expr
        case str():
            return Values[expr]
        case ('call', 'read'):
            return int(input('Please enter a natural number: '))
        case ('+', lhs, rhs):
            return evaluate(lhs, Values) + evaluate(rhs, Values)
        case ('-', lhs, rhs):
            return evaluate(lhs, Values) - evaluate(rhs, Values)
        case ('*', lhs, rhs):
            return evaluate(lhs, Values) * evaluate(rhs, Values)
        case ('/', lhs, rhs):
            return evaluate(lhs, Values) / evaluate(rhs, Values)
        case ('%', lhs, rhs):
            return evaluate(lhs, Values) % evaluate(rhs, Values)
        case _:
            assert False, f'{expr} unexpected'
```

Figure 7.16: The evaluation of arithmetic expressions.

The Jupyter notebook `03-Interpreter.ipynb`, which is available at

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Chapter-07/03-Interpreter.ipynb

implements an interpreter.

Exercise 24:

(a) Add `for` loops to the interpreter.

(b) Expand the interpreter to include logical operators "`&&`" for logical *and*, "`||`" for logical *or*, and "`!`" for *negation*. The operator "`!`" should bind the strongest and the operator "`||`" should bind weakest.

(c) Enhance the interpreter to allow for user-defined functions.

- A function should only have access to its parameters and those variables that are defined locally inside the function.
- A function should always return a value using a `return` statement. In order to facilitate that a function can contain multiple `return` statements and that a `return` statement can occur anywhere in the function, use an exception to communicate the return value and to transfer the control flow out of the function.

Figure 7.17 on page 92 shows a simple program that implements a function to compute the factorial function. Your interpreter should be able to execute this program. Use the Jupyter notebook `Interpreter-Frame.ipynb`, which is available at

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Chapter-07/Interpreter-Frame.ipynb

as a starting point for your solution. ◇

```
function factorial(n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}

for (i := 2; i <= 25; i := i + 1) {
    print(factorial(i));
}
```

Figure 7.17: A program to compute the factorial function.

7.4 Check your Understanding

(a) Are you able to implement a simple calculator using PLY?

(b) Would you be able to implement a PLY parser that reads propositional formulas and returns these formulas as nested tuples?

(c) Are you able to extend the interpreter that has been discussed in this chapter?

(d) Assume that we develop an interpreter that supports user defined functions. How can we accomodate the fact that a `return` statement can occur anywhere in a function?

Chapter 8

Earley Parser

In this Chapter we will discuss an efficient algorithm that takes two inputs:

1. A context-free grammar $G = \langle V, \Sigma, R, S \rangle$ and
2. a string $s \in \Sigma^*$.

The algorithm decides, whether $s \in L(G)$ holds, i.e. it checks, whether the string is a member of the language generated by th $\textsc{Cfg}$ G.

The algorithm that is presented next has been published in 1970 by Jay Earley [Ear70]. There is another algorithm solving the same problem, namely the Cocke-Younger-Kasami algorithm, which is also known as the $\textsc{Cyk}$ algorithm. It has been discovered independently by John Cocke [CS70], Daniel H. Younger [You67], and Tadao Kasami [Kas65]. The $\textsc{Cyk}$ algorithm can only be used when the grammar has a special form, namely it has to be in Chomsky normal form. As it is quite tedious to transform a given grammar into Chomsky normal form, this algorithm is not used in practical applications. In contrast, the Earley algorithm works for arbitrary context-free grammars and hence is used in practical applications. In the general case, the Earley algorithm has a complexity of $\mathcal{O}(n^3)$ where n is the length of the string that is to be parsed. However, if the grammar is not ambiguous, the complexity is only $\mathcal{O}(n^2)$. Skillful implementations of Earley's algorithm even achieve a linear runtime for many practically relevant grammars. For example, Earley's algorithm has a linear complexity for both $LL(k)$ grammars and also for $LR(1)$ grammars. We will discuss $LR(1)$ grammars in a later Chapter. On the contrary, the $\textsc{Cyk}$ algorithm always has the complexity $\mathcal{O}(n^3)$, which is prohibitive for practical applications.

This chapter is structured as follows:

(a) First, we sketch the theory of Earley's algorithm.

(b) Next, we show how this algorithm can be implemented in *Python*.

8.1 The Earley Algorithm

The central notion that is needed to understand Earley's algorithm is the notion of an Earley object. This notion is defined below.

Definition 22 (Earley Object) Assume that $G = \langle V, \Sigma, R, s \rangle$ is a context-free grammar and $w = x_1 x_2 \cdots x_n \in \Sigma^*$ is a string of length n. A pair of the form

$$\langle A \to \alpha \bullet \beta, k \rangle$$

is called an Earley object if and only if

(a) $(A \to \alpha\beta) \in R$ and

(b) $k \in \{0, 1, \cdots, n\}$. ◇

Explanation: An Earley describes a possible state of a parser. If a parser has to parse a string of the form $x_1 \cdots x_n$, then this parser will maintain $n+1$ sets of Earley objects. These sets are denoted as

$$Q_0, Q_1, \cdots, Q_n.$$

If $i \in \{0, 1, \cdots, n\}$, then the state Q_i contains those Earley objects that describe the state of the parser after it has parsed the tokens x_1, $\cdots$, x_i. The interpretation of

$$\langle a \to \beta \bullet \gamma, k\rangle \in Q_j \quad \text{where } j \geq k$$

is as follows:

1. The parser tries to use the grammar rule $a \to \beta\gamma$ to parse the variable a at the beginning of the substring $x_{k+1} \cdots x_n$.
2. The parser has already parsed β in the substring $x_{k+1} \cdots x_j$, we have
$$\beta \Rightarrow^* x_{k+1} \cdots x_j.$$
3. Hence, in order to parse the variable a the parser only needs to recognize γ at the beginning of the substring $x_{j+1} \cdots x_n$.

Earley's algorithm manages the sets Q_0, Q_1, $\cdots$, Q_n of Earley objects. The set Q_j contains those Earley objects that contain all those states the parser could be in when it has read the prefix $x_1 \cdots x_j$.

At the beginning of the algorithm we add a new start variable $\widehat{s}$ to the grammar. Furthermore the rule $\widehat{s} \to s$ is added to the grammar. Hence, the grammar $G = \langle V, \Sigma, R, s\rangle$ is transformed into the augmented grammar

$$\widehat{G} = \langle V \cup \{\widehat{s}\}, \Sigma, R \cup \{\widehat{s} \to s\}, \widehat{s}\rangle.$$

Next, the set Q_0 is defined as

$$Q_0 := \big\{\langle \widehat{S} \to \bullet S, 0\rangle\big\}.$$

The reason is that the parser should recognize the start symbol s an the beginning of the string $x_1 \cdots x_n$. The remaining sets Q_j are initially empty for $j = 1, \cdots, n$ leer. These sets are extended by the following three operations:

1. *Reading*

   If the set Q_j contains an Earley object of the form $\langle a \to \beta \bullet T\gamma, k\rangle$ and T is a terminal, then the parser tries to parse the right hand side of the grammar rule $a \to \beta T\gamma$ and, after reading $x_{k+1} \cdots x_j$ it has already recognized β. If this β is now followed by the token T, then in order to recognize a with the grammar rule $a \to \beta T\gamma$, the parser only has to recognize γ in the substring $x_{j+2} \cdots x_n$. Hence, in this case we add the Earley object

   $$\langle a \to \beta T \bullet \gamma, k\rangle$$

   to the set Q_{j+1}:

   $$\langle a \to \beta \bullet T\gamma, k\rangle \in Q_j \wedge x_{j+1} = T \;\Rightarrow\; Q_{j+1} := Q_{j+1} \cup \big\{\langle a \to \beta T \bullet \gamma, k\rangle\big\}.$$

2. *Prediction*

   If the set Q_j contains an Earley object of the form $\langle a \to \beta \bullet c\delta, k\rangle$ such that c is a variable, then the parser tries to recognize the substring $C\delta$ after having parsed the substring $x_{k+1} \cdots x_j$ as β. Hence the parser now has to recognize the variable c. Therefore, for every rule of the form $c \to \gamma$ that is contained in the grammar G the Earley object $\langle c \to \bullet\gamma, j\rangle$ is added to the set Q_j:

   $$\langle a \to \beta \bullet c\delta, k\rangle \in Q_j \wedge (c \to \gamma) \in R \;\Rightarrow\; Q_j := Q_j \cup \big\{\langle c \to \bullet\gamma, j\rangle\big\}.$$

3. *Completion*

   If the set Q_i contains an Earley object of the form $\langle c \to \gamma\bullet, j\rangle$ and the set Q_j contains an Earley object of the form $\langle a \to \beta \bullet c\delta, k\rangle$, then the parser has tried to parse the variable c after it had read the substring $x_1 \cdots x_j$ and after reading the substring $x_{j+1} \cdots x_i$ it has recognized the variable c. Therefore, the Earley object $\langle a \to \beta c \bullet \delta, k\rangle$ is now added to the set Q_i:

   $$\langle c \to \gamma\bullet, j\rangle \in Q_i \wedge \langle a \to \beta \bullet c\delta, k\rangle \in Q_j \;\Rightarrow\; Q_i := Q_i \cup \big\{\langle a \to \beta c \bullet \delta, k\rangle\big\}.$$

When trying to parse the string $w = x_1 \cdots x_n$ with the grammar $G = \langle V, \Sigma, R, s \rangle$ Earley's algorithm works as follows:

1. The sets Q_i are initiallized as follows:
$$Q_0 := \{\langle \widehat{s} \to \bullet s, 0 \rangle\},$$
$$Q_i := \{\} \quad \text{for } i = 1, \cdots, n.$$
2. After that we iterate from $i = 0$ to $i = n$ and perform the following operations:
   (a) The set Q_i is enlarged using completion until we find no further Earley objects.
   (b) Next we use prediction to enlarge the set Q_i. Again, this operation is performed until no new Earley objects are found.
   (c) If $i < n$, we read the next token and use it to initiallize Q_{i+1}.

   If the grammar G has ε-rules, i.e. if it has rules with empty right hand side of the form
   $$c \to \lambda,$$
   then it might happen that after applying prediction in the set Q_i we can apply immediately apply completion in Q_i. In this case prediction and completion have to be iterated until no further Earley objects are generated for Q_i.
3. If after termination of the algorithm the set Q_n contains the Earley object $\langle \widehat{s} \to s \bullet, 0 \rangle$, then parsing is successfull and we have shown that the string $w = x_1 \cdots x_n$ is an element of $L(G)$.

Example: Figure 8.1 shows a simplified grammar for arithmetic expressions, consisting only of the numbers "1", "2", and "3" and the two operator symbols "+" and "*". The set T of terminals for this grammar is therefore given by

$$T = \{ \text{ "1" }, \text{ "2" }, \text{ "3" }, \text{ "+" }, \text{ "*" } \}.$$

We demonstrate how the string "1+2*3" can be parsed using this grammar and Earley's algorithm. In the following representation, we will abbreviate the syntactic variable `expr` with the letter e, write p for `prod`, and use f as an abbreviation for `fact`.

```
1   expr : expr '+' prod
2        | prod
3        ;
4
5   prod : prod '*' fact
6        | fact
7        ;
8
9   fact : '1'
10       | '2'
11       | '3'
12       ;
```

Figure 8.1: Eine vereinfachte Grammatik für arithmetische Ausdrücke.

1. We initialize Q_0 as
$$Q_0 = \{\langle \widehat{s} \to \bullet\, e, 0 \rangle\}.$$
The set Q_1, Q_2, Q_3, Q_4 und Q_5 are initially empty, If we apply the completion operation to the set Q_0, we find no new Earley objects.

Next, we apply the prediction operation to the Earley object $\langle \widehat{S} \to \bullet\, e, 0\rangle$. This initially adds the following two Earley objects to the set Q_0:

$$\langle e \to \bullet\, e \text{“+”} p, 0\rangle \quad \text{and} \quad \langle e \to \bullet\, p, 0\rangle$$

We can apply the prediction operation once more to the Earley object $\langle e \to \bullet\, p, 0\rangle$, which results in two new Earley objects:

$$\langle p \to \bullet\, p \text{“*”} f, 0\rangle \quad \text{and} \quad \langle p \to \bullet\, f, 0\rangle$$

Applying the prediction operation to the Earley object $\langle p \to \bullet\, f, 0\rangle$, we finally add the following Earley objects to Q_0:

$$\langle f \to \bullet \text{“1”}, 0\rangle, \quad \langle f \to \bullet \text{“2”}, 0\rangle, \quad \text{and} \quad \langle f \to \bullet \text{“3”}, 0\rangle$$

In total, Q_0 now contains the following Earley objects:

(a) $\langle \widehat{s} \to \bullet\, e, 0\rangle$,
(b) $\langle e \to \bullet\, e \text{“+”} p, 0\rangle$,
(c) $\langle e \to \bullet\, p, 0\rangle$,
(d) $\langle p \to \bullet\, p \text{“*”} f, 0\rangle$,
(e) $\langle p \to \bullet\, f, 0\rangle$,
(f) $\langle f \to \bullet \text{“1”}, 0\rangle$,
(g) $\langle f \to \bullet \text{“2”}, 0\rangle$,
(h) $\langle f \to \bullet \text{“3”}, 0\rangle$.

Next, we apply the scanning operation to Q_0. Since the first character of the string to be parsed is “1,” the set Q_1 takes the following form:

$$Q_1 = \{\langle f \to \text{“1”}\bullet, 0\rangle\}$$

2. Now, we set $i = 1$ and first apply the completion operation to Q_1. Based on the Earley object $\langle f \to \text{“1”}\bullet, 0\rangle$ in Q_1, we look for an Earley object in Q_0 where the marker “•” is positioned before the variable F. We find the Earley object $\langle p \to \bullet\, f, 0\rangle$. Therefore, we add the following Earley object to Q_1:

   $$\langle p \to f\, \bullet, 0\rangle$$

   We can then apply the completion operation again. After repeated applications, Q_1 contains the following Earley objects:

   (a) $\langle p \to f\, \bullet, 0\rangle$,
   (b) $\langle p \to p\, \bullet \text{“*”} f, 0\rangle$,
   (c) $\langle e \to p\, \bullet, 0\rangle$,
   (d) $\langle e \to e\, \bullet \text{“+”} p, 0\rangle$,
   (e) $\langle \widehat{s} \to e\, \bullet, 0\rangle$.

   Next, we apply the prediction operation to these Earley objects. However, since the marker “•” in none of the Earley objects in Q_i precedes a variable, no new Earley objects are generated.

   Finally, we apply the scanning operation to Q_1. Since the character “+” is at position 2 in the string “1+2*3” and Q_1 contains the Earley object

   $$\langle e \to e\, \bullet \text{“+”} p, 0\rangle,$$

   we add the following Earley object to Q_2:

   $$\langle e \to e \text{“+”}\, \bullet\, p, 0\rangle$$

3. Next, we set $i = 2$ and first apply the completion operation to Q_2. At this point, we have

$$Q_2 = \{\langle e \to e \text{“+”} \bullet\ p, 0\rangle\}.$$

Since the marker "•" in the only Earley object present here is not at the end of the grammar rule, the completion operation does not yield any new Earley objects in this step.

Next, we apply the prediction operation to Q_2. Since the marker precedes the variable p, we first find the following two Earley objects:

$$\langle p \to \bullet\ f, 2\rangle \quad \text{and} \quad \langle p \to \bullet\ p \text{“*”}\ f, 2\rangle.$$

Since in the first Earley object the marker precedes the variable f, the prediction operation can be applied once more, yielding the following additional Earley objects:

(a) $\langle f \to \bullet \text{“1”}, 2\rangle$,
(b) $\langle f \to \bullet \text{“2”}, 2\rangle$,
(c) $\langle f \to \bullet \text{“3”}, 2\rangle$.

Finally, we apply the scanning operation to Q_2. Since the third character of the string "1+2*3" is the digit "2," Q_3 now takes the form

$$Q_3 = \{\langle f \to \text{“2”} \bullet, 2\rangle\}.$$

4. We set $i = 3$ and apply the completion operation to Q_3. This adds

$$\langle p \to f\ \bullet, 2\rangle$$

to Q_3. Here, we can apply the completion operation once more. Through iterative application of the completion operation, we additionally obtain the following Earley objects:

(a) $\langle p \to p \bullet \text{“*”}\ f, 2\rangle$,
(b) $\langle e \to e \text{“+”}\ p \bullet, 0\rangle$,
(c) $\langle e \to e\ \bullet \text{“+”} p, 0\rangle$,
(d) $\langle \widehat{s} \to e \bullet, 0\rangle$.

Finally, we apply the scanning operation. Since the next character to be read is the symbol "*," we get

$$Q_4 = \{\langle p \to p \text{“*”} \bullet\ f, 2\rangle\}.$$

5. We set $i = 4$. The completion operation does not yield any new Earley objects. The prediction operation provides the following Earley objects:

(a) $\langle f \to \bullet \text{“1”}, 4\rangle$,
(b) $\langle f \to \bullet \text{“2”}, 4\rangle$,
(c) $\langle f \to \bullet \text{“3”}, 4\rangle$.

Since the next character is the digit "3," the scanning operation for Q_5 results in:

$$Q_5 = \{\langle f \to \text{“3”} \bullet, 4\rangle\}.$$

6. We set $i = 5$. The completion operation sequentially provides the following Earley objects:

(a) $\langle p \to p \text{“*”}\ f \bullet, 2\rangle$,
(b) $\langle e \to e \text{“+”}\ p \bullet, 0\rangle$,

(c) $\langle p \to p \bullet \text{“*”} f, 2\rangle$,

(d) $\langle e \to e \bullet \text{“+”} p, 0\rangle$

(e) $\langle \widehat{s} \to e \bullet, 0\rangle$.

Since the set Q_5 contains the Earley object $\langle \widehat{s} \to e \bullet, 0\rangle$, we can conclude that the string "1+2*3" indeed belongs to the language generated by the given grammar.

Exercise 25: Use Earley's algorithm to show that the string "1*2+3" belongs to the language of the grammar shown in Figure 8.1. ◇

8.2 Implementing Earley's Algorithm in Python

The *Jupyter* notebook

https://github.com/karlstroetmann/Formal-Languages/blob/master/ANTLR4-Python/Earley-Parser/Earley-Parser.ipynb

contains an implementation of Earley's algorithm.

8.3 Check Your Understandig

(a) How is an Earley object defined, and how are the components of an Earley object interpreted?

(b) How does the scanning operation work in Earley's algorithm?

(c) How does the prediction operation work in Earley's algorithm?

(d) How does the completion operation work in Earley's algorithm?

(e) Outline Earley's algorithm for the case where the grammar contains no ε-rules.

(f) What is the complexity of Earley's algorithm in the general case?

(g) What is the complexity of Earley's algorithm in the case that the grammar is not ambiguous?

Chapter 9

Bottom-up Parsers

In constructing a parser, there are generally two approaches: top-down and bottom-up. We have already discussed the top-down approach. In this chapter, we will explain the bottom-up approach.

1. In the next section, we present the general concept underlying a bottom-up parser.
2. In the following section, we show how bottom-up parsers can be implemented and introduce shift-reduce parsers as one possible implementation.

   A *shift-reduce parser* operates with a table that specifies how the parser should process inputs in a given state.
3. We then develop the theory of how such a table can be meaningfully filled with information in the following section: First, we discuss the SLR parsers (simple LR parsers), which are the simplest class of shift-reduce parsers.
4. Unfortunately, the concept of SLR parsers is not powerful enough for practical use. Therefore, we refine this concept to obtain the class of canonical LR parsers.
5. As the tables for LR parsers often become large, we simplify these tables slightly and then obtain the concept of LALR parsers, which in terms of power lies between the concept of SLR parsers and that of LR parsers.

In the following chapter, we will discuss the parser generator PLY, which is an LALR parser.

9.1 Bottom-Up-Parser

The parsers that we have implemented in the Chapter 6 without the help of any tool are known as top-down parsers: starting from the start symbol of the grammar, they attempt to parse a given input by applying various grammar rules. In contrast, the parsers generated by PLY are bottom-up parsers. In such parsers, the idea is to start with the string to be parsed and group terminals based on the right-hand sides of the grammar rules.

As an example, consider parsing the string "1 + 2 * 3" using the grammar defined by the following rules:

$$
\begin{array}{lcl}
e & \to & e \text{ "+" } p \mid p \\
p & \to & p \text{ "*" } f \mid f \\
f & \to & \text{"1"} \mid \text{"2"} \mid \text{"3"}
\end{array}
$$

To parse this string, we look for substrings that correspond to the right-hand sides of grammar rules, scanning the string from left to right. This way, we try to construct a parse tree in reverse from bottom to top:

$$
\begin{array}{lcll}
\texttt{1 + 2 * 3} & \Leftarrow & f\ \texttt{+ 2 * 3} & (\text{rule: } f \to \text{“1”}) \\
& \Leftarrow & p\ \texttt{+ 2 * 3} & (\text{rule: } p \to f) \\
& \Leftarrow & e\ \texttt{+ 2 * 3} & (\text{rule: } e \to p) \\
& \Leftarrow & e\ \texttt{+}\ f\ \texttt{* 3} & (\text{rule: } f \to \text{“2”}) \\
& \Leftarrow & e\ \texttt{+}\ p\ \texttt{* 3} & (\text{rule: } p \to f) \\
& \Leftarrow & e\ \texttt{+}\ p\ \texttt{*}\ f & (\text{rule: } f \to \text{“3”}) \\
& \Leftarrow & e\ \texttt{+}\ p & (\text{rule: } p \to p\ \text{“*”}\ f) \\
& \Leftarrow & e & (\text{rule: } e \to e\ \text{“+”}\ P)
\end{array}
$$

In the first step, we used the grammar rule $f \to$ “1” to replace the string “1” with f, thus obtaining the string “f + 2 * 3”. In the second step, we used the rule $p \to f$ to replace f with p. In this manner, we ultimately traced the original string “1 + 2 * 3” back to e. At this point, we can make two observations:

(a) In our approach, we always replace the leftmost substring that can be replaced when we want to trace the initially given string back to the start symbol of the grammar.

(b) If we write down the derivation that we have constructed backwards, we obtain:

$$
\begin{array}{lcl}
e & \Rightarrow & e\ \texttt{+}\ p \\
& \Rightarrow & e\ \texttt{+}\ p\ \texttt{*}\ f \\
& \Rightarrow & e\ \texttt{+}\ p\ \texttt{* 3} \\
& \Rightarrow & e\ \texttt{+}\ f\ \texttt{* 3} \\
& \Rightarrow & e\ \texttt{+ 2 * 3} \\
& \Rightarrow & p\ \texttt{+ 2 * 3} \\
& \Rightarrow & f\ \texttt{+ 2 * 3} \\
& \Rightarrow & \texttt{1 + 2 * 3}
\end{array}
$$

We see here that in this derivation, the rightmost syntactic variable is always the one being replaced. Such a derivation is referred to as a rightmost derivation.

In contrast, the derivations produced by a top-down parser are exactly the opposite: there, the leftmost syntactic variable is always replaced. Therefore, the derivations created with such a parser are called leftmost derivations.

The above two observations are the reason why the parsers discussed in this chapter are referred to as LR Parsers. The L stands for left to right and describes the approach of always scanning the string from left to right, while the R stands for reverse rightmost derivation and indicates that such parsers construct a rightmost derivation in reverse.

In the implementation of an LR parser, two questions arise:

1. Which substring do we replace?
2. Which grammar rules should we use in the process?

Answering these questions is generally non-trivial. Although we always scan the strings from left to right, it is not immediately clear which substring we should replace, as the potentially replaceable substrings can overlap. Consider, for example, the intermediate result

$$e\ \texttt{+}\ p\ \texttt{* 3},$$

that we obtained in the fifth step above. Here, we could replace the substring “p” using the rule

$$e \to p$$

with “e”. Then, we would obtain the string

$$e\ \texttt{+}\ e\ \texttt{* 3}.$$

The only reductions we can now perform, via the intermediate results $e\ \texttt{+}\ e\ \texttt{*}\ f$ and $e\ \texttt{+}\ e\ \texttt{*}\ p$, lead to the string

$$e\ \texttt{+}\ e\ \texttt{*}\ e,$$

which then cannot be reduced any further with the grammar given above. The answers to the above questions, which substring we replace and which rule we use, require a fair amount of theory, which we will develop in the following sections.

9.2 Shift-Reduce-Parser

Shift-reduce parsing is one way to implement bottom up parsing. Assume a grammar $G = \langle V, T, R, S\rangle$ is given. A shift-reduce parser is defined as a 4-Tuple

$$P = \langle Q, q_0, \mathit{action}, \mathit{goto}\rangle$$

where

1. Q is the set of states of the shift-reduce parser.

   At first, states are purely abstract.

2. $q_0 \in Q$ is the start state.

3. *action* is a function taking two arguments. The first argument is a state $q \in Q$ and the second argument is a terminal $t \in T$. The result of this function is an element from the set

   $$\mathit{Action} := \big\{\langle \texttt{shift}, q\rangle \mid q \in Q\big\} \cup \big\{\langle \texttt{reduce}, r\rangle \mid r \in R\big\} \cup \big\{\texttt{accept}\big\} \cup \big\{\texttt{error}\big\}.$$

   Here `shift`, `reduce`, `accept`, and `error` are strings that serve to distinguish the different kinds of result of the function *action*. Therefore the signature of the function *action* is given as follows:

   $$\mathit{action} : Q \times T \to \mathit{Action}.$$

4. *goto* is a function that takes a state $q \in Q$ and a syntactical variable $v \in V$ and computes a new state. Therefore the signature of *goto* is as follows:

   $$\mathit{goto} : Q \times V \to Q.$$

To describe the algorithm implemented in a shift-reduce parser, we first describe the data structures that are used by this algorithm:

(a) *States* is a stack of states from the set Q:

$$\mathit{States} \in \mathit{Stack}[Q].$$

(b) *Symbols* is a stack of grammar symbols, i.e. this stack contains both terminals and syntactical variables:

$$\mathit{Symbols} \in \mathit{Stack}[T \cup V].$$

In order to simplify the exposition of shift-reduce parsing we assume that the set T of terminals contains the special symbol "$\textsc{Eof}$" (short for end of file). This symbol is assumed to occur at the end of the input string but does not occur elsewhere.

In order to understand how a shift-reduce parser works we introduce the notion of a parser configuration. A parser configuration is a triple of the form

$$\langle \mathit{States}, \mathit{Symbols}, \mathit{Tokens}\rangle$$

where *States* and *Symbols* are the aforementioned stacks of states and grammar symbols, while *Tokens* is the rest of the tokens from the input string that have not been processed. The stack *States* starts with the start state q_0, i.e. initially we have

$$\mathit{States} = [q_0].$$

The length of *States* is always the length of the stack *Symbols* plus one:

$$\texttt{length}(\mathit{States}) = \texttt{length}(\mathit{Symbols}) + 1.$$

If the input string that is to be parsed has the form

$$[t_1, \cdots, t_n]$$

and we have already reduced the first part $[t_1, \cdots, t_k]$ of the input string to produce the symbols

$$[X_1, \cdots, X_m],$$

while $[t_{k+1}, \cdots, t_n]$ is the part of the input string that still needs to be processed, then we have

$$\mathit{States} = [q_0, q_1 \cdots, q_m], \quad \mathit{Symbols} = [X_1, \cdots, X_m], \quad \text{and} \quad \mathit{Tokens} = [t_{k+1}, \cdots, t_n, \texttt{EOF}],$$

and the parser configuration $\langle \mathit{States}, \mathit{Symbols}, \mathit{Tokens} \rangle$ is written as

$$q_0, q_1, \cdots, q_m \mid X_1, \cdots, X_m \mid t_{k+1}, \cdots, t_n, \texttt{EOF}.$$

Shift-reduce parsing starts out with the configuration

$$q_0 \mid \mid t_1, \cdots, t_n, \texttt{EOF}.$$

Then, parsing proceeds iteratively. If the current configuration is

$$q_0, q_1, \cdots, q_m \mid X_1, \cdots, X_m \mid t_{k+1}, \cdots, t_n, \texttt{EOF},$$

then there is a case distinction according to the value of $\mathit{action}(q_m, t_{k+1})$.

(a) If $\mathit{action}(q_m, t_{k+1}) = \mathit{error}$, then we know that the given string $t_1 \cdots, t_n$ is not generated by the given grammar and parsing is aborted with an error message.

(b) If $\mathit{action}(q_m, t_{k+1}) = \mathit{accept}$, then we must have $k = n$ and hence $t_{k+1} = t_{n+1} = \texttt{EOF}$ and we also must have $X_1, \cdots, X_m = S$. In this case, we have reduced the string $t_1 \cdots t_n$ to the start symbol S of the given grammar and parsing finishes with success.

(c) If $\mathit{action}(q_m, t_{k+1}) = \langle \mathit{shift}, q \rangle$, then the current configuration is changed into the new configuration

$$q_0, q_1, \cdots, q_m, q \mid X_1, \cdots, X_m, t_{k+1} \mid t_{k+2}, \cdots, t_n, \texttt{EOF},$$

i.e. the next token t_{k+1} is moved from the unread input to the top of the symbol stack and the new state q is pushed onto the stack *States*.

(d) If $\mathit{action}(q_m, t_{k+1}) = \langle \mathit{reduce}, r \rangle$, where r is a grammar rule, then the grammar rule r must have the form

$$a \to X_{m-l} \cdots X_m,$$

i.e. the right hand side of the grammar rule matches the end of the stack *Symbols*. In this case, the symbols stack is reduced with this grammar rule, i.e. the symbols $X_{m-l}, \cdots, X_m$ are replaced by a. Furthermore, in the stack *States* the states $q_{m-l} \cdots q_m$ are replaced by the state $\mathit{goto}(q_{m-l-1}, a)$. Therefore, the configuration changes as follows:

$$q_0, q_1, \cdots, q_{m-l-1}, \mathit{goto}(q_{m-l-1}, a) \mid X_1, \cdots, X_{m-l-1}, a \mid t_{k+1}, \cdots, t_n, \texttt{EOF}.$$

```
1  class ShiftReduceParser():
2      def __init__(self, actionTable, gotoTable):
3          self.mActionTable = actionTable
4          self.mGotoTable   = gotoTable
```

Figure 9.1: Implementation of a shift-reduce parser in *Python*

The class `Shift-Reduce-Parser-Pure` that is shown in Figure 9.1 on page 102 displays the class `ShiftReduceParser`, which maintains two dictionaries.

(a) `mActionTable` stores the function $\mathit{action} : Q \times T \to \mathit{Action}$.

(b) `mGotoTable` stores the function $\mathit{goto} : Q \times V \to Q$.

Figure 9.2 on page 103 shows the implementation of the method *parse* that implements shift-reduce parsing. This method assumes that the function *action* is coded as a dictionary that is stored in the member variable `mActionTable`. The function *goto* is also represented as a dictionary. It is stored in the member variable `mGotoTable`. The method

```
def parse(self, TL):
    index   = 0      # points to next token
    Symbols = []     # stack of symbols
    States  = ['s0'] # stack of states, s0 is start state
    TL     += ['EOF']
    while True:
        q = States[-1]
        t = TL[index]
        # Below, an undefined table entry is interpreted as an error entry.
        match self.mActionTable.get((q, t), 'error'):
            case 'error':
                return False
            case 'accept':
                return True
            case 'shift', s:
                Symbols += [t]
                States  += [s]
                index   += 1
            case 'reduce', rule:
                head, body = rule
                n       = len(body)
                Symbols = Symbols[:-n]
                States  = States [:-n]
                Symbols = Symbols + [head]
                state   = States[-1]
                States += [ self.mGotoTable[state, head] ]
```

Figure 9.2: Implementation of a shift-reduce parser in *Python*

parse is called with one argument *TL*. This is the list of tokens that have to be parsed. We append the special token "EOF" at the end of this list. The invocation *parse*(*TL*) returns `True` if the token list *TL* can be parsed successfully and `false` otherwise. The implementation of *parse* works as follows:

1. The variable *index* points to the next token in the token list that is to be read. Therefore, this variable is initialized to 0.

2. The variable *Symbols* stores the stack of symbols. The top of this stack is at the end of this list. Initially, the stack of symbols is empty.

3. The variable *States* is the stack of states. The start state is assumed to be the state "s0". Therefore this stack is initialized to contain only this state.

4. The main loop of the parser

   - sets the variable q to the current state,
   - initializes t to the next token, and then
   - looks up the appropriate action in the action table.

   What happens next depends on this value of *action*(q, t).

   (a) *action*$(q, t) = error$.
   In this case the parser has found a syntax error and returns `False`.

(b) $\mathit{action}(q, t) = \texttt{accept}$.
In this case parsing is successful and therefore the function returns `True`.

(c) $\mathit{action}(q, t) = \langle \texttt{shift}, s \rangle$.
In this case, the token t is pushed onto the symbol stack in line 16, while the state s is pushed onto the stack of states. Furthermore, the variable *index* is incremented to point to the next unread token.

(d) $\mathit{action}(q, t) = \langle \texttt{reduce}, A \to X_1 \cdots X_n \rangle$.
In this case, we use the grammar rule

$$r = (A \to X_1 \cdots X_n)$$

to reduce the symbol stack. The variable *head* represents the left hand side A of this rule, while the list $[X_1, \cdots, X_n]$ is represented by the variable *body*.
In this case, it can be shown that the symbols X_1, $\cdots$, X_n are on top of the symbol stack. As we are going to reduce the symbol stack with the rule r, we remove these n symbols from the symbol stack and replace them with the variable A.
Furthermore, we have to remove n states from the stack of states. After that, we set the variable *state* to the state that is on top of the stack of states after we have removed n states. Next, the new state $\mathit{goto}(\mathit{state}, A)$ is put on top of the stack of states in line 26.

In order to make the function *parse* work we have to provide an implementation of the functions *action* and *goto*. The tables 9.1 and 9.2 show these functions for the grammar given in Figure 9.3. For this grammar, there are 16 different states, which have been baptized as s_0, s_1, $\cdots$, s_{15}. The tables use two different abbreviations:

1. $\langle \mathit{shft}, s_i \rangle$ is short for $\langle \texttt{shift}, s_i \rangle$.
2. $\langle \mathit{rdc}, r_i \rangle$ is short for $\langle \texttt{reduce}, r_i \rangle$, where r_i denothes the grammar rule number i. Here, we have numbered the rules as follows:
   (a) $r_1 = (\mathit{expr} \to \mathit{expr}$ "+" $\mathit{product})$
   (b) $r_2 = (\mathit{expr} \to \mathit{expr}$ "–" $\mathit{product})$
   (c) $r_3 = (\mathit{expr} \to \mathit{product})$
   (d) $r_4 = (\mathit{product} \to \mathit{product}$ "∗" $\mathit{factor})$
   (e) $r_5 = (\mathit{product} \to \mathit{product}$ "/" $\mathit{factor})$
   (f) $r_6 = (\mathit{product} \to \mathit{factor})$
   (g) $r_7 = (\mathit{factor} \to$ "(" expr ")" $)$
   (h) $r_8 = (\mathit{factor} \to \textsc{Number})$

The corresponding grammar is shown in Figure 9.3. The definition of the grammar rules and the coding of the functions *action* and *goto* is shown in the Figures 9.4, 9.6, and 9.5 on the following pages. Of course, at present we do not have any idea how the functions *action* and *goto* are computed. This requires some theory that will be presented in the next section.

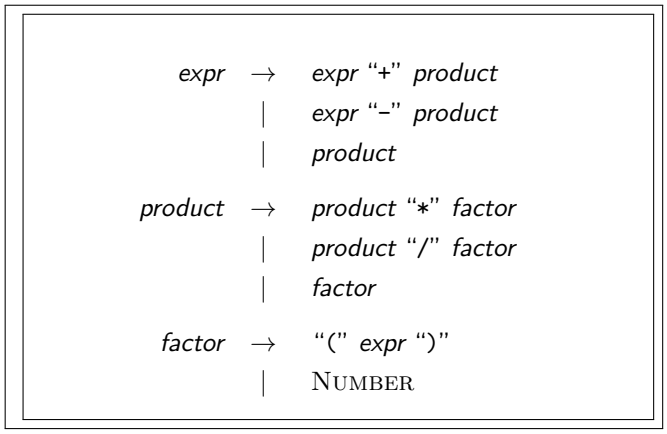

Figure 9.3: A grammar for arithmetical expressions.

State	EOF	+	-	*	/	(	)	NUMBER
s_0						$\langle shft, s_5\rangle$		$\langle shft, s_2\rangle$
s_1	$\langle rdc, r_6\rangle$	$\langle rdc, r_6\rangle$	$\langle rdc, r_6\rangle$	$\langle rdc, r_6\rangle$	$\langle rdc, r_6\rangle$		$\langle rdc, r_6\rangle$	
s_2	$\langle rdc, r_8\rangle$	$\langle rdc, r_8\rangle$	$\langle rdc, r_8\rangle$	$\langle rdc, r_8\rangle$	$\langle rdc, r_8\rangle$		$\langle rdc, r_8\rangle$	
s_3	$\langle rdc, r_3\rangle$	$\langle rdc, r_3\rangle$	$\langle rdc, r_3\rangle$	$\langle shft, s_{12}\rangle$	$\langle shft, s_{11}\rangle$		$\langle rdc, r_3\rangle$	
s_4	*accept*	$\langle shft, s_8\rangle$	$\langle shft, s_9\rangle$					
s_5						$\langle shft, s_5\rangle$		$\langle shft, s_2\rangle$
s_6		$\langle shft, s_8\rangle$	$\langle shft, s_9\rangle$				$\langle shft, s_7\rangle$	
s_7	$\langle rdc, r_7\rangle$	$\langle rdc, r_7\rangle$	$\langle rdc, r_7\rangle$	$\langle rdc, r_7\rangle$	$\langle rdc, r_7\rangle$		$\langle rdc, r_7\rangle$	
s_8						$\langle shft, s_5\rangle$		$\langle shft, s_2\rangle$
s_9						$\langle shft, s_5\rangle$		$\langle shft, s_2\rangle$
s_{10}	$\langle rdc, r_2\rangle$	$\langle rdc, r_2\rangle$	$\langle rdc, r_2\rangle$	$\langle shft, s_{12}\rangle$	$\langle shft, s_{11}\rangle$		$\langle rdc, r_2\rangle$	
s_{11}						$\langle shft, s_5\rangle$		$\langle shft, s_2\rangle$
s_{12}						$\langle shft, s_5\rangle$		$\langle shft, s_2\rangle$
s_{13}	$\langle rdc, r_4\rangle$	$\langle rdc, r_4\rangle$	$\langle rdc, r_4\rangle$	$\langle rdc, r_4\rangle$	$\langle rdc, r_4\rangle$		$\langle rdc, r_4\rangle$	
s_{14}	$\langle rdc, r_5\rangle$	$\langle rdc, r_5\rangle$	$\langle rdc, r_5\rangle$	$\langle rdc, r_5\rangle$	$\langle rdc, r_5\rangle$		$\langle rdc, r_5\rangle$	
s_{15}	$\langle rdc, r_1\rangle$	$\langle rdc, r_1\rangle$	$\langle rdc, r_1\rangle$	$\langle shft, s_{12}\rangle$	$\langle shft, s_{11}\rangle$		$\langle rdc, r_1\rangle$	

Table 9.1: The function *action*().

```
1  r1 = ('e', ('e', '+', 'p'))
2  r2 = ('e', ('e', '-', 'p'))
3  r3 = ('e', ('p'))
4
5  r4 = ('p', ('p', '*', 'f'))
6  r5 = ('p', ('p', '/', 'f'))
7  r6 = ('p', ('f'))
8
9  r7 = ('f', ('(', 'e', ')'))
10 r8 = ('f', ('NUMBER',))
```

Figure 9.4: Grammar rules coded in *Python*.

State	*expr*	*product*	*factor*
s_0	s_4	s_3	s_1
s_1			
s_2			
s_3			
s_4			
s_5	s_6	s_3	s_1
s_6			
s_7			
s_8		s_{15}	s_1
s_9		s_{10}	s_1
s_{10}			
s_{11}			s_{14}
s_{12}			s_{13}
s_{13}			
s_{14}			
s_{15}			

Table 9.2: The function *goto*().

```
1  gotoTable   := {};
2
3  gotoTable["s0", "e"] := "s4";
4  gotoTable["s0", "p"] := "s3";
5  gotoTable["s0", "f"] := "s1";
6
7  gotoTable["s5", "e"] := "s6";
8  gotoTable["s5", "p"] := "s3";
9  gotoTable["s5", "f"] := "s1";
10
11 gotoTable["s8", "p"] := "s15";
12 gotoTable["s8", "f"] := "s1";
13
14 gotoTable["s9", "p"] := "s10";
15 gotoTable["s9", "f"] := "s1";
16
17 gotoTable["s11", "f"] := "s14";
18 gotoTable["s12", "f"] := "s13";
```

Figure 9.5: Goto table coded in PYTHON.

```
actionTable = {}

actionTable['s0', '('     ] = ('shift', 's5');
actionTable['s0', 'NUMBER'] = ('shift', 's2');

actionTable['s1', 'EOF'] = ('reduce', r6);
actionTable['s1', '+'  ] = ('reduce', r6);
actionTable['s1', '-'  ] = ('reduce', r6);
actionTable['s1', '*'  ] = ('reduce', r6);
actionTable['s1', '/'  ] = ('reduce', r6);
actionTable['s1', ')'  ] = ('reduce', r6);

actionTable['s2', 'EOF'] = ('reduce', r8);
actionTable['s2', '+'  ] = ('reduce', r8);
actionTable['s2', '-'  ] = ('reduce', r8);
actionTable['s2', '*'  ] = ('reduce', r8);
actionTable['s2', '/'  ] = ('reduce', r8);
actionTable['s2', ')'  ] = ('reduce', r8);

actionTable['s3', 'EOF'] = ('reduce', r3);
actionTable['s3', '+'  ] = ('reduce', r3);
actionTable['s3', '-'  ] = ('reduce', r3);
actionTable['s3', '*'  ] = ('shift', 's12');
actionTable['s3', '/'  ] = ('shift', 's11');
actionTable['s3', ')'  ] = ('reduce', r3);

actionTable['s4', 'EOF'] = 'accept';
actionTable['s4', '+'  ] = ('shift', 's8');
actionTable['s4', '-'  ] = ('shift', 's9');

actionTable['s5', '('     ] = ('shift', 's5');
actionTable['s5', 'NUMBER'] = ('shift', 's2');

actionTable['s6', '+'  ] = ('shift', 's8');
actionTable['s6', '-'  ] = ('shift', 's9');
actionTable['s6', ')'  ] = ('shift', 's7');

actionTable['s7', 'EOF'] = ('reduce', r7);
actionTable['s7', '+'  ] = ('reduce', r7);
actionTable['s7', '-'  ] = ('reduce', r7);
actionTable['s7', '*'  ] = ('reduce', r7);
actionTable['s7', '/'  ] = ('reduce', r7);
actionTable['s7', ')'  ] = ('reduce', r7);
```

```
actionTable['s8', '('     ] = ('shift', 's5')
actionTable['s8', 'NUMBER'] = ('shift', 's2')

actionTable['s9', '('     ] = ('shift', 's5')
actionTable['s9', 'NUMBER'] = ('shift', 's2')

actionTable['s10', 'EOF'] = ('reduce', r2)
actionTable['s10', '+'  ] = ('reduce', r2)
actionTable['s10', '-'  ] = ('reduce', r2)
actionTable['s10', '*'  ] = ('shift', 's12')
actionTable['s10', '/'  ] = ('shift', 's11')
actionTable['s10', ')'  ] = ('reduce', r2)

actionTable['s11', '('     ] = ('shift', 's5')
actionTable['s11', 'NUMBER'] = ('shift', 's2')

actionTable['s12', '('     ] = ('shift', 's5')
actionTable['s12', 'NUMBER'] = ('shift', 's2')

actionTable['s13', 'EOF'] = ('reduce', r4)
actionTable['s13', '+'  ] = ('reduce', r4)
actionTable['s13', '-'  ] = ('reduce', r4)
actionTable['s13', '*'  ] = ('reduce', r4)
actionTable['s13', '/'  ] = ('reduce', r4)
actionTable['s13', ')'  ] = ('reduce', r4)

actionTable['s14', 'EOF'] = ('reduce', r5)
actionTable['s14', '+'  ] = ('reduce', r5)
actionTable['s14', '-'  ] = ('reduce', r5)
actionTable['s14', '*'  ] = ('reduce', r5)
actionTable['s14', '/'  ] = ('reduce', r5)
actionTable['s14', ')'  ] = ('reduce', r5)

actionTable['s15', 'EOF'] = ('reduce', r1)
actionTable['s15', '+'  ] = ('reduce', r1)
actionTable['s15', '-'  ] = ('reduce', r1)
actionTable['s15', '*'  ] = ('shift', 's12')
actionTable['s15', '/'  ] = ('shift', 's11')
actionTable['s15', ')'  ] = ('reduce', r1)
```

Figure 9.6: Action table coded in PYTHON.

9.3 SLR-Parser

In this section, we demonstrate how to compute the functions

$$\mathit{action} : Q \times T \to \mathit{Action} \quad \text{and} \quad \mathit{goto} : Q \times V \to Q$$

that have been used in the last section to build a shift-reduce parser for a given context-free grammar G. For this purpose, we first clarify what information the states contained in the set Q should hold. We will define these states in such a way that they contain information about which rule the shift-reduce parser is attempting to apply, which part of the right hand side of a grammar rule has already been recognized, and what are the symbols that are still expected. To this end, we define the concept of a marked rule. Unfortunately, the original English literature [Knu65] uses the rather meaningless term "item" instead of the notion of a *marked rule*.

Definition 23 (Marked Rule) A marked rule of a context-free grammar $G = \langle V, T, R, s \rangle$ is a triple

$$(a, \beta, \gamma),$$

such that

$$(a \to \beta\gamma) \in R.$$

A marked rule of the form $\langle a, \beta, \gamma \rangle$ is written as

$$a \to \beta \bullet \gamma.$$ ◇

The marked rule $a \to \beta \bullet \gamma$ indicates that the parser is trying to parse an a with the rule $a \to \beta\gamma$, having already seen β and is next attempting to recognize γ. The symbol $\bullet$ thus marks the part of the right side of the rule that we have already read. Now the idea is to represent a state of an SLR parser as a set of marked rules. To illustrate this idea, let's consider a specific example: we start with the grammar for arithmetic expressions shown in Figure 9.3 on page 105. We extend this grammar by a new start symbol $\widehat{s}$ and the rule

$$\widehat{s} \to \mathit{expr}\ \$$$

where the symbol "$" is used to denote EOF, i.e. the end of the input. The start state clearly contains the marked rule

$$\widehat{s} \to \lambda \bullet \mathit{expr}\ \$,$$

since at the beginning we are trying to derive the start symbol $\widehat{s}$. The component λ indicates that we have not yet processed anything. In addition to this marked rule, the start state must also contain the marked rules

1. $\mathit{expr} \to \lambda \bullet \mathit{expr}$ '+' $\mathit{product}$,
2. $\mathit{expr} \to \lambda \bullet \mathit{expr}$ '−' $\mathit{product}$ and
3. $\mathit{expr} \to \lambda \bullet \mathit{product}$

because, for example, it may be necessary to use the rule

$$\mathit{expr} \to \mathit{expr}\ \text{'+'}\ \mathit{product}$$

to derive the sought *expr*. Similarly, it could be that instead we need to use the rule

$$\mathit{expr} \to \mathit{product}$$

This explains why we must include the marked rule

$$\mathit{expr} \to \lambda \bullet \mathit{product}$$

in the start state, because at the beginning we cannot yet know which rule we will need, so the start state must contain all these rules. Once we have added the marked rule

$$\mathit{expr} \to \lambda \bullet \mathit{product}$$

to the start state, we see that we might need to read a *product* next. Therefore, the start state also contains the following marked rules[1]:

4. *product* $\to \bullet$ *product* '$*$' *factor*,
5. *product* $\to \bullet$ *product* '/' *factor*,
6. *product* $\to \bullet$ *factor*,

   Now, the sixth rule indicates that we might first read a *factor*. Therefore, we add the following two marked rules to the start state:

7. *factor* $\to \bullet$ '(' *expr* ')' ,
8. *factor* $\to \bullet$ NUMBER.

Overall, we see that the start state consists of a set of 8 marked rules. The system shown above, for deriving further rules from a given rule, is formalized in the concept of the closure of a set of marked rules.

Definition 24 (closure($\mathcal{M}$)) Let $\mathcal{M}$ be a set of marked rules. Then we define the closure of this set as the smallest set $\mathcal{K}$ of marked rules such that:

1. $\mathcal{M} \subseteq \mathcal{K}$,

   i.e. the closure includes the original set of rules.

2. If on the one hand

   $$a \to \beta \bullet c\,\delta$$

   is a marked rule from the set $\mathcal{K}$, where c is a syntactic variable, and on the other hand

   $$c \to \gamma$$

   is a grammar rule of the underlying grammar G, then the marked rule

   $$c \to \bullet\,\gamma$$

   is also an element of the set $\mathcal{K}$. Formally, this is written as follows:

   $$(a \to \beta \bullet c\,\delta) \in \mathcal{K} \;\wedge\; (c \to \gamma) \in R \;\Rightarrow\; (c \to \bullet\,\gamma) \in \mathcal{K}$$

The set $\mathcal{K}$ defined in this way is uniquely determined and is referred to as *closure*($\mathcal{M}$) in the following. ◇

Remark: If you recall the Earley algorithm, you will notice that the calculation of the closure is the same as the *prediction operation* in the Earley algorithm. ◇

For a given set $\mathcal{M}$ of marked rules, the computation of $\mathcal{K} := \mathit{closure}(\mathcal{M})$ can be done iteratively:

1. Initially, set $\mathcal{K} := \mathcal{M}$.
2. Then, find all rules of the form

   $$a \to \beta \bullet c\,\delta$$

   within the set $\mathcal{K}$, where c is a syntactic variable. Subsequently, for all rules of the form $c \to \gamma$, add the new marked rule

   $$c \to \bullet\,\gamma$$

   into the set $\mathcal{K}$. Repeat this step until no new rules are found.

[1] In order to be more concise, we drop the empty string λ in the following rules.

Example: Starting from the grammar for arithmetic expressions shown in Figure 9.3 on page 105, let's consider the set

$$\mathcal{M} := \{\mathit{product} \to \mathit{product}\ \text{'*'} \bullet \mathit{factor}\}$$

For the set $\mathit{closure}(\mathcal{M})$, we then find

$$\begin{array}{lcl} \mathit{closure}(\mathcal{M}) & = & \{\quad \mathit{product} \to \mathit{product}\ \text{'*'} \bullet \mathit{factor}, \\ & & \quad\ \mathit{factor} \to \bullet\ \text{'('}\ \mathit{expr}\ \text{')'}, \\ & & \quad\ \mathit{factor} \to \bullet\ \textsc{Number} \\ & & \}. \end{array}$$

◇

Our goal is to define a Shift-Reduce-Parser

$$P = \langle Q, q_0, \mathit{action}, \mathit{goto} \rangle$$

for a given context-free grammar $G = \langle V, T, R, s \rangle$. To achieve this goal, we first need to determine how to define the states in the set Q, as this will almost automatically define the remaining components. As mentioned earlier, we define the states as sets of marked rules. Let's first define

$$\Gamma := \{a \to \beta \bullet \gamma \mid (a \to \beta\gamma) \in R\}$$

as the set of all marked rules of the grammar. However, it is not sensible to allow arbitrary subsets of Γ as states: A subset $\mathcal{M} \subseteq \Gamma$ is only considered as a state if the set $\mathcal{M}$ is closed under the function $\mathit{closure}()$, meaning $\mathit{closure}(\mathcal{M}) = \mathcal{M}$. Therefore, we define

$$Q := \{\mathcal{M} \in 2^\Gamma \mid \mathit{closure}(\mathcal{M}) = \mathcal{M}\}.$$

The interpretation of the sets $\mathcal{M} \in Q$ is that a state $\mathcal{M}$ contains exactly those marked rules that can be applied in the situation described by the state.

To simplify the following constructions, we extend the grammar $G = \langle V, T, R, s \rangle$ by introducing a new start symbol $\widehat{s}$ and a new token \$ to the grammar

$$\widehat{G} = \Big\langle V \cup \{\widehat{s}\}, T \cup \{\$\}, R \cup \{\widehat{s} \to s\,\$\}, \widehat{s} \Big\rangle.$$

We refer to the grammar $\widehat{G}$ as the augmented grammar. The use of the augmented grammar enables the following definition of the start state. We set:

$$q_0 := \mathit{closure}\Big(\{\widehat{s} \to \bullet s\,\$\}\Big).$$

Next, we construct the function $\mathit{goto}()$. The definition is as follows:

$$\mathit{goto}(\mathcal{M}, c) := \mathit{closure}\Big(\{a \to \beta\,c \bullet \delta \mid (a \to \beta \bullet c\,\delta) \in \mathcal{M}\}\Big).$$

To understand this definition, assume that the parser is in a state where it tries to parse an a using the rule $a \to \beta\,c\,\delta$ and, furthermore, assume that the substring β has already been recognized. This state is described by the marked rule

$$a \to \beta \bullet c\,\delta$$

If a c is now recognized, the parser can transition from the state containing the rule $a \to \beta \bullet c\,\delta$ to a state containing the rule $a \to \beta\,c \bullet \delta$. Therefore, we get the above definition of the function $\mathit{goto}(\mathcal{M}, c)$. For the subsequent definition of the function $\mathit{action}(\mathcal{M}, t)$, it is useful to extend the definition of the function *goto* to terminals. For terminals $t \in T$ we set:

$$\mathit{goto}(\mathcal{M}, t) := \mathit{closure}\Big(\{a \to \beta\,t \bullet \delta \mid (a \to \beta \bullet t\,\delta) \in \mathcal{M}\}\Big).$$

Before we can compute the function *action*, we need to define two more functions, the functions *First* and *Follow*.

9.3.1 The Functions First and Follow

In this section, we define the two functions *First* and *Follow*. These functions are needed to compute the function *action*. We consider a given context-free grammar $G = \langle V, T, R, s \rangle$.

(a) For a syntactic variable a, *First*(a) calculates the set of all tokens with which a string w can begin, which is derived from the variable a, for which $a \Rightarrow_G^* w$ holds.

(b) For a syntactic variable a, *Follow*(a) calculates the set of tokens that can follow an a in a string derived from s, i.e., $t \in$ *Follow*(a) if there is a derivation of the following form:

$$s \Rightarrow_G^* \beta\, a\, t\, \gamma.$$

Here, β and γ are strings consisting of tokens and variables, while t denotes a single token.

Definition 25 (λ-generating) Let $G = \langle V, T, R, S \rangle$ be a context-free grammar and a be a syntactic variable, i.e., $a \in V$. The variable a is called λ-generating precisely when

$$a \Rightarrow^* \lambda$$

holds, which means that the empty word can be derived from the variable a. We write *nullable*(a) if the variable a is λ-generating. ◇

Examples:

(a) In the grammar shown in Figure 6.5 on page 68, the variables *exprRest* and *productRest* are λ-generating.

(b) Now let's consider a less obvious example. Suppose the grammar G contains the following rules:

$$S \to a\ b\ c$$
$$a \to \ \texttt{'X'}\ b \mid a\ \texttt{'Y'}\ \mid b\ c$$
$$b \to \ \texttt{'X'}\ b \mid a\ \texttt{'Y'}\ \mid c\ c$$
$$c \to a\ b\ c \mid \lambda$$

Initially, it's clear that the variable c is λ-generating. Then, due to the rule $b \to c\ c$, b is also λ-generating, and from the rule $a \to b\ c$, a becomes λ-generating as well. Finally, S is recognized as λ-generating, since the first rule states

$$S \to a\ b\ c$$

and here, all variables on the right side of the rule have already been proven to be λ-generating variables.

Definition 26 (First()) Let $G = \langle V, T, R, s \rangle$ be a context-free grammar and $a \in V$. We define *First*(a) as the set of all tokens t with which a word derived from a can begin:

$$\mathit{First}(a) := \{t \in T \mid \exists \gamma \in (V \cup T)^* : a \Rightarrow^* t\,\gamma\}.$$

The definition of the function *First*() can be extended to strings in $(V \cup T)^*$ as follows:

1. $\mathit{First}(\lambda) = \{\}$.
2. $\mathit{First}(t\beta) = \{t\}$ if $t \in T$.
3. $\mathit{First}(a\beta) = \begin{cases} \mathit{First}(a) \cup \mathit{First}(\beta) & \text{if } a \Rightarrow^* \lambda; \\ \mathit{First}(a) & \text{otherwise.} \end{cases}$

If a is a variable of G and the rules defining a are given as

$$a \to \alpha_1 \mid \cdots \mid \alpha_n,$$

then we have

$$\mathit{First}(a) = \bigcup_{i=1}^{n} \mathit{First}(\alpha_i).$$ ◇

Remark: Note that the definitions of the function *First*(a) for variables $a \in V$ and the function *First*(α) for strings $\alpha \in (V \cup T)^*$ are mutually recursive. The computation of *First*(a) is best done via a fixpoint computation: Start by setting *First*(a) := $\{\}$ for all variables $a \in V$ and then continue to iterate the equations defining *First*(a) until none of the sets *First*(a) changes any more. The next example clarifies this idea.

Example: We can iteratively compute the sets *First*(a) for the variables a of the grammar shown in Figure 6.5. It's best to compute the function *First*(a) for the individual variables a starting with those at the bottom of the hierarchy.

1. First, the rules

   $$\mathit{factor} \rightarrow \text{'('}\ \mathit{expr}\ \text{')'} \mid \textsc{Number}$$

   imply that every string derived from *factor* either starts with an opening parenthesis or a number:

   $$\mathit{First}(\mathit{factor}) = \{\ \text{'('}, \textsc{Number}\ \}.$$

2. Similarly, from the rules

   $$\mathit{productRest} \rightarrow \text{'*'}\ \mathit{factor}\ \ \mathit{productRest} \mid \text{'/'}\ \mathit{factor}\ \ \mathit{productRest} \mid \lambda$$

   we find that a *productRest* either starts with the character "*" or "/":

   $$\mathit{First}(\mathit{productRest}) = \{\ \text{'*'}, \text{'/'}\ \}.$$

3. The rule for the variable *product* is

   $$\mathit{product} \rightarrow \mathit{factor}\ \ \mathit{productRest}.$$

   Since the variable *factor* is not λ-generating, we see that the set *First*(*product*) matches the set *First*(*factor*):

   $$\mathit{First}(\mathit{product}) = \{\ \text{'('}, \textsc{Number}\ \}.$$

4. From the rules

   $$\mathit{exprRest} \rightarrow \text{'+'}\ \mathit{product}\ \ \mathit{exprRest} \mid \text{'-'}\ \mathit{product}\ \ \mathit{exprRest} \mid \lambda,$$

   we can compute *First*(*exprRest*) as follows:

   $$\mathit{First}(\mathit{exprRest}) = \{\ \text{'+'}, \text{'-'}\ \}.$$

5. Further, from the rule

   $$\mathit{expr} \rightarrow \mathit{product}\ \ \mathit{exprRest},$$

   and the fact that the variable *product* is not λ-generating, the set *First*(*expr*) matches the set *First*(*product*):

   $$\mathit{First}(\mathit{expr}) = \{\ \text{'('}, \textsc{Number}\ \}.$$

Since we have computed the sets *First*(a) in a clever order, we did not have to perform a proper fixpoint iteration in this example

Definition 27 (Follow()) Let $G = \langle V, T, R, s \rangle$ be a context-free grammar and let

$$\widehat{G} = \left\langle V \cup \{\widehat{s}\}, T \cup \{\$\}, R \cup \{\widehat{s} \rightarrow s\,\$\}, \widehat{s} \right\rangle$$

be the associated augmented grammar. For a variable $a \in V$ we define *Follow*(a) as the set of all tokens t that can follow the variable a in a derivation:

$$\mathit{Follow}(a) := \left\{ t \in T \cup \{\$\} \mid \exists \beta, \gamma \in (V \cup T \cup \{\$\})^* : \widehat{s} \Rightarrow^* \beta\, a\, t\, \gamma \right\}.$$

If the start symbol $\widehat{s}$ can somehow derive a string $\beta\, a\, t\, \gamma$ in which the token t follows the variable a, then t is an element of the set *Follow*(a). ◇

Example: Let's reexamine the grammar for arithmetic expressions shown in Figure 6.5.

1. Due to the newly added rule

   $$\widehat{S} \to \mathit{expr}\,\$$$

   the set *Follow*(*expr*) must contain the symbol $\$$. Furthermore, due to the rule

   $$\mathit{factor} \to \text{'('}\ \mathit{expr}\ \text{')'}$$

   the set *Follow*(*expr*) must also contain the symbol ')' . Thus, we have in total

   $$\mathit{Follow}(\mathit{expr}) = \{\$,\ \text{')'}\}.$$

2. Due to the rule

   $$\mathit{expr} \to \mathit{product}\ \ \mathit{exprRest}$$

   we know that all terminals that can follow an *expr* can also follow an *exprRest*. Thus, we already know that *Follow*(*exprRest*) contains the tokens $\$$ and ')' . Since *exprRest* otherwise only appears at the end of its defining rules, these are all the tokens that can follow *exprRest*, and we have

   $$\mathit{Follow}(\mathit{exprRest}) = \{\$,\ \text{')'}\}.$$

3. The rules

   $$\mathit{exprRest} \to \text{'+'}\ \mathit{product}\ \ \mathit{exprRest} \mid \text{'−'}\ \mathit{product}\ \ \mathit{exprRest}$$

   show that all elements from *First*(*exprRest*) can follow a *product*. However, that's not all: since the variable *exprRest* is λ-generating, additionally, all tokens that can follow *exprRest* can also follow *product*. Thus, we have in total

   $$\mathit{Follow}(\mathit{product}) = \{\ \text{'+'}\ ,\ \text{'−'}\ ,\$,\ \text{')'}\ \}.$$

4. The rule

   $$\mathit{product} \to \mathit{factor}\ \ \mathit{productRest}$$

   shows that all terminals that can follow a *product* can also follow a *productRest*. Since *productRest* otherwise only appears at the end of its defining rules, these are all the tokens that can follow *productRest*, and we have in total

   $$\mathit{Follow}(\mathit{productRest}) = \{\ \text{'+'}\ ,\ \text{'−'}\ ,\$,\ \text{')'}\ \}.$$

5. The rules

   $$\mathit{productRest} \to \text{'*'}\ \mathit{factor}\ \ \mathit{productRest} \mid \text{'/'}\ \mathit{factor}\ \ \mathit{productRest}$$

   show that all elements from *First*(*productRest*) can follow a *factor*. However, that's not all: since the variable *productRest* is λ-generating, additionally, all tokens that can follow *productRest* can also follow *factor*. Thus, we have in total

   $$\mathit{Follow}(\mathit{factor}) = \{\ \text{'*'}\ ,\ \text{'/'}\ ,\ \text{'+'}\ ,\ \text{'−'}\ ,\$,\ \text{')'}\ \}.$$ ◇

The last example demonstrates that the computation of the *nullable*() predicate and the computation of the sets *First*(*a*) and *Follow*(*a*) for a syntactic variable *a* are closely interconnected. Suppose we have a grammar rule:

$$a \to Y_1 Y_2 \cdots Y_k$$

Then the following relationships exist between the *nullable*() predicate and the *First*() and *Follow*() functions:

1. $\forall t \in T : \neg\, \mathit{nullable}(t)$.
2. $k = 0 \Rightarrow \mathit{nullable}(a)$.
3. $\big(\forall i \in \{1, \cdots, k\} : \mathit{nullable}(Y_i)\big) \Rightarrow \mathit{nullable}(a)$.

   Setting $k = 0$ here, we see that 2. is a special case of 3.
4. $\mathit{First}(Y_1) \subseteq \mathit{First}(a)$.

5. $\big(\forall j \in \{1, \cdots, i-1\} : \mathit{nullable}(Y_j)\big) \Rightarrow \mathit{First}(Y_i) \subseteq \mathit{First}(a)$.
   Setting $i = 1$ above, we see that 4. is a special case of 5.
6. $\mathit{Follow}(a) \subseteq \mathit{Follow}(Y_k)$.
7. $\big(\forall j \in \{i+1, \cdots, k\} : \mathit{nullable}(Y_j)\big) \Rightarrow \mathit{Follow}(a) \subseteq \mathit{Follow}(Y_i)$.
   Setting $i = k$ here, we see that 6. is a special case of 7.
8. $\forall i \in \{1, \cdots, k-1\} : \mathit{First}(Y_{i+1}) \subseteq \mathit{Follow}(Y_i)$.
9. $\big(\forall j \in \{i+1, \cdots, l-1\} : \mathit{nullable}(Y_j)\big) \Rightarrow \mathit{First}(Y_l) \subseteq \mathit{Follow}(Y_i)$.
   Setting $l = i + 1$ here, we see that 8. is a special case of 9.

Using these relationships, *nullable*(), *First*(), and *Follow*() can be computed iteratively through a fixpoint iteration:

1. Initially, the functions *First*(*a*) and *Follow*(*a*) for each syntactic variable *a* are initialized with the empty set. The predicate *nullable*(*a*) is set to `false` for each syntactic variable.
2. Subsequently, the rules outlined above are applied repeatedly until no further changes occur from their application.

9.3.2 Computing the Function action

Finally, we specify how the function $\mathit{action}(\mathcal{M}, t)$ is computed for a set of marked rules $\mathcal{M}$ and a token t. When defining $\mathit{action}(\mathcal{M}, t)$, we distinguish four cases.

1. If $\mathcal{M}$ contains a marked rule of the form $a \to \beta \bullet t\, \delta$ and $t \neq \$$, then we set

   $$\mathit{action}(\mathcal{M}, t) := \langle \texttt{shift}, \mathit{goto}(\mathcal{M}, t) \rangle,$$

   because in this case the parser is trying to recognize an a using the rule $a \to \beta\, t \delta$ and has already recognized β. If the next token in the input string is indeed the token t, the parser can read this t and transitions from the state $a \to \beta \bullet t\, \delta$ to the state $a \to \beta\, t \bullet \delta$, which is computed by the function $\mathit{goto}(\mathcal{M}, t)$. So, we have

   $$\mathit{action}(\mathcal{M}, t) := \langle \texttt{shift}, \mathit{goto}(\mathcal{M}, t) \rangle \quad \text{if} \quad (a \to \beta \bullet t\, \delta) \in \mathcal{M} \text{ and } t \neq \$.$$

2. If $\mathcal{M}$ contains a marked rule of the form $a \to \beta\bullet$ and additionally $t \in \mathit{Follow}(a)$, then we set

   $$\mathit{action}(\mathcal{M}, t) := \langle \texttt{reduce}, a \to \beta \rangle,$$

   because in this case the parser is trying to recognize an a using the rule $a \to \beta$ and has already recognized β. If the next token in the input string is the token t and t is a token that can follow a, i.e., $t \in \mathit{Follow}(a)$, then the parser can apply the rule $a \to \beta$ and reduce the symbol stack with this rule. Therefore,

   $$\mathit{action}(\mathcal{M}, t) := \langle \texttt{reduce}, a \to \beta \rangle \quad \text{if } (a \to \beta\bullet) \in \mathcal{M},\ a \neq \widehat{s} \text{ and } t \in \mathit{Follow}(a).$$

3. If $\mathcal{M}$ contains the marked rule $\widehat{s} \to s \bullet \$$ and we have completely read the string to be parsed, we set

   $$\mathit{action}(\mathcal{M}, \$) := \texttt{accept},$$

   because in this case the parser is trying to recognize $\widehat{s}$ using the rule $\widehat{s} \to s\, \$$ and has already recognized s. If the next token in the input string is the end-of-file character \$, then the string being parsed belongs to the language specified by the grammar G, $L(G)$. Therefore, we have

   $$\mathit{action}(\mathcal{M}, \$) := \texttt{accept}, \quad \text{if } (\widehat{s} \to s \bullet \$) \in \mathcal{M}.$$

4. In all other cases, we set

   $$\mathit{action}(\mathcal{M}, t) := \texttt{error}.$$

Conflicts can arise between the first two rules. We distinguish between two types of conflicts.

1. A shift-reduce conflict occurs when both the first and second cases apply. In this situation, the set $\mathcal{M}$ contains, on one hand, a marked rule of the form

   $$a \to \beta \bullet t\, \gamma,$$

   and on the other hand, $\mathcal{M}$ contains a rule of the form

   $$c \to \delta\bullet \quad \text{with } t \in \mathit{Follow}(c).$$

   If the next token is t, it is unclear whether this token should be pushed onto the symbol stack and the parser transition to a state with the marked rule $a \to \beta\, t \bullet \gamma$, or whether the symbol stack should be reduced using the rule $c \to \delta$ instead.

2. A reduce-reduce conflict exists if the set $\mathcal{M}$ contains two different marked rules of the form

   $$c_1 \to \gamma_1\bullet \quad \text{and} \quad c_2 \to \gamma_2\bullet$$

   and if at the same time $t \in \mathit{Follow}(c_1) \cap \mathit{Follow}(c_2)$, because then it is not clear which of the two rules the parser should apply when the next token to be read is t.

If either of these conflicts occurs, we say that the grammar is not an SLR grammar. Such a grammar cannot be parsed using an SLR parser. We will later provide examples of both types of conflicts, but first, we want to examine a grammar where no conflicts occur and actually compute the functions *goto*() and *action*() for it. We use the grammar for arithmetic expressions that is shown in Figure 9.3 as a basis.

Since the syntactic variable *expr* appears on the right side of grammar rules, we define *start* as a new start symbol and add the rule

$$\mathit{start} \to \mathit{expr}\,\$$$

to the grammar. This step corresponds to the previously discussed process of augmenting the grammar.

First, we compute the set of states Q. We had previously provided the following formula for this:

$$Q := \big\{\mathcal{M} \in 2^{\Gamma} \mid \mathit{closure}(\mathcal{M}) = \mathcal{M}\big\}.$$

However, this set also contains states that cannot be reached from the start state via the function *goto*(). Therefore, we only compute the states that can actually be reached from the start state using the function *goto*(). To avoid making the calculation too wordy, we introduce the following abbreviations:

$$s := \mathit{start},\ e := \mathit{expr},\ p := \mathit{product},\ f := \mathit{factor}, \text{ and } N := \textsc{Number}.$$

We begin with the start state:

1. $$\begin{aligned} s_0 &:= \mathit{closure}(\{\ \ s \to \bullet\, e\,\$\ \ \}) \\ &= \{\ \ s \to \bullet\, e\,\$, \\ &\quad\ e \to \bullet\, e\ \texttt{'+'}\ p,\ e \to \bullet\, e\ \texttt{'-'}\ p,\ e \to \bullet\, p, \\ &\quad\ p \to \bullet\, p\ \texttt{'*'}\ f,\ p \to \bullet\, p\ \texttt{'/'}\ f,\ p \to \bullet\, f, \\ &\quad\ f \to \bullet\, \texttt{'('}\, e\ \texttt{')'}\ ,\ f \to \bullet\, N \qquad\qquad \}. \end{aligned}$$

2. $$\begin{aligned} s_1 &:= \mathit{goto}(s_0, f) \\ &= \mathit{closure}(\{p \to f\bullet\}) \\ &= \{p \to f\bullet\}. \end{aligned}$$

3. $$\begin{aligned} s_2 &:= \mathit{goto}(s_0, N) \\ &= \mathit{closure}(\{f \to N\bullet\}) \\ &= \{f \to N\bullet\}. \end{aligned}$$

4. $$\begin{aligned} s_3 &:= \mathit{goto}(s_0, p) \\ &= \mathit{closure}(\{p \to p\bullet\ \texttt{'*'}\ f,\ p \to p\bullet\ \texttt{'/'}\ f,\ e \to p\bullet\}) \\ &= \{p \to p\bullet\ \texttt{'*'}\ f,\ p \to p\bullet\ \texttt{'/'}\ f, e \to p\bullet\}. \end{aligned}$$

5. $$\begin{aligned} s_4 &:= \mathit{goto}(s_0, e) \\ &= \mathit{closure}(\{s \to e\bullet\ \$,\ e \to e\bullet\ \texttt{'+'}\ p,\ e \to e\bullet\ \texttt{'-'}\ p\ \}) \\ &= \{\ s \to e\bullet\ \$,\ e \to e\bullet\ \texttt{'+'}\ p,\ e \to e\bullet\ \texttt{'-'}\ p\ \}. \end{aligned}$$

6. $s_5 := \mathit{goto}(s_0, \text{'('})$
$= \mathit{closure}(\{f \to \text{'('} \bullet e \text{')'}\})$
$= \{ f \to \text{'('} \bullet e \text{')'}$
$e \to \bullet e \text{'+'} p,\ e \to \bullet e \text{'-'} p,\ e \to \bullet p,$
$p \to \bullet p \text{'*'} f,\ p \to \bullet p \text{'/'} f,\ p \to \bullet f,$
$f \to \bullet \text{'('} e \text{')'},\ f \to \bullet N \quad \}.$

7. $s_6 := \mathit{goto}(s_5, e)$
$= \mathit{closure}(\{ f \to \text{'('} e \bullet \text{')'},\ e \to e \bullet \text{'+'} p,\ e \to e \bullet \text{'-'} p \})$
$= \{ f \to \text{'('} e \bullet \text{')'},\ e \to e \bullet \text{'+'} p,\ e \to e \bullet \text{'-'} p. \}$

8. $s_7 := \mathit{goto}(s_6, \text{')'})$
$= \mathit{closure}(\{ f \to \text{'('} e \text{')'} \bullet \})$
$= \{ f \to \text{'('} e \text{')'} \bullet \}.$

9. $s_8 := \mathit{goto}(s_4, \text{'+'})$
$= \mathit{closure}(\{e \to e \text{'+'} \bullet p\})$
$= \{ e \to e \text{'+'} \bullet p$
$p \to \bullet p \text{'*'} f,\ p \to \bullet p \text{'/'} f,\ p \to \bullet f,$
$f \to \bullet \text{'('} e \text{')'},\ f \to \bullet N \quad \}.$

10. $s_9 := \mathit{goto}(s_4, \text{'-'})$
$= \mathit{closure}(\{e \to e \text{'-'} \bullet p\})$
$= \{ e \to e \text{'-'} \bullet p$
$p \to \bullet p \text{'*'} f,\ p \to \bullet p \text{'/'} f,\ p \to \bullet f,$
$f \to \bullet \text{'('} e \text{')'},\ f \to \bullet N \quad \}.$

11. $s_{10} := \mathit{goto}(s_9, p)$
$= \mathit{closure}(\{e \to e \text{'-'} p \bullet,\ p \to p \bullet \text{'*'} f,\ p \to p \bullet \text{'/'} f \})$
$= \{ e \to e \text{'-'} p \bullet,\ p \to p \bullet \text{'*'} f,\ p \to p \bullet \text{'/'} f \}.$

12. $s_{11} := \mathit{goto}(s_3, \text{'/'})$
$= \mathit{closure}(\{p \to p \text{'/'} \bullet f \})$
$= \{ p \to p \text{'/'} \bullet f,\ f \to \bullet \text{'('} e \text{')'},\ f \to \bullet N \}.$

13. $s_{12} := \mathit{goto}(s_3, \text{'*'})$
$= \mathit{closure}(\{p \to p \text{'*'} \bullet f \})$
$= \{ p \to p \text{'*'} \bullet f,\ f \to \bullet \text{'('} e \text{')'},\ f \to \bullet N \}.$

14. $s_{13} := \mathit{goto}(s_{12}, f)$
$= \mathit{closure}(\{p \to p \text{'*'} f \bullet \})$
$= \{ p \to p \text{'*'} f \bullet \}.$

15. $s_{14} := \mathit{goto}(s_{11}, f)$
$= \mathit{closure}(\{p \to p \text{'/'} f \bullet \})$
$= \{ p \to p \text{'/'} f \bullet \}.$

16. $s_{15} := \mathit{goto}(s_8, p)$
$= \mathit{closure}(\{e \to e \text{'+'} p \bullet,\ p \to p \bullet \text{'*'} f,\ p \to p \bullet \text{'/'} f \})$
$= \{ e \to e \text{'+'} p \bullet,\ p \to p \bullet \text{'*'} f,\ p \to p \bullet \text{'/'} f \}.$

Further calculations do not yield any new states. For instance, when we compute $\mathit{goto}(s_8, \text{'('})$, we find:

$$
\begin{array}{lll}
 & \mathit{goto}(s_8,\ \text{'('}\ ) & \\
= & \mathit{closure}(\{f \to\ \text{'('}\ \bullet e\ \text{')'}\ \}) & \\
= & \{\ f \to\ \text{'('}\ \bullet e\ \text{')'} & \\
 & \quad e \to \bullet e\ \text{'+'}\ p,\ e \to \bullet e\ \text{'-'}\ p,\ e \to \bullet p, & \\
 & \quad p \to \bullet p\ \text{'*'}\ f,\ p \to \bullet p\ \text{'/'}\ f,\ p \to \bullet f, & \\
 & \quad f \to \bullet\ \text{'('}e\ \text{')'}\ ,\ f \to \bullet N & \} \\
= & s_5. &
\end{array}
$$

Therefore, the set of states for the Shift-Reduce parser is given by

$$Q := \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}\}$$

Next, we investigate whether there are any conflicts, and we look specifically at the set s_{15}. Due to the marked rule

$$p \to p \bullet\ \text{'*'}\ f$$

a shift must occur in state s_{15} if the next token is '*' . On the other hand, the state s_{15} includes the rule

$$e \to e\ \text{'+'}\ p \bullet .$$

This rule indicates that the symbol stack must be reduced with the grammar rule $e \to e$ '+' p if a character from the set *Follow*(e) appears in the input. If '*' were in *Follow*(e), then we would have a Shift-Reduce conflict. However,

$$\mathit{Follow}(e) = \{\ \text{'+'}\ ,\ \text{'-'}\ ,\ \text{')'}\ ,\ \text{'\$'}\ \},$$

and therefore '*' $\notin$ *Follow*(e), so there is no Shift-Reduce conflict here. An examination of the other sets shows that there are also no Shift-Reduce or Reduce-Reduce conflicts in any of them.

Next, we calculate the *action* function. We consider two cases as examples.

1. First, we compute *action*$(s_1,\ \text{'+'}\ )$. It holds that

$$
\begin{array}{lll}
\mathit{action}(s_1,\ \text{'+'}\ ) & = & \mathit{action}(\{p \to f\bullet\},\ \text{'+'}\ ) \\
 & = & \langle \mathit{reduce}, p \to f \rangle,
\end{array}
$$

   since we have '+' $\in$ *Follow*(p).

2. Next, we compute *action*$(s_4,\ \text{'+'}\ )$. It holds that

$$
\begin{array}{lll}
\mathit{action}(s_4,\ \text{'+'}\ ) & = & \mathit{action}(\{s \to e \bullet\ \$,\ e \to e \bullet\ \text{'+'}\ p,\ e \to e \bullet\ \text{'-'}\ p\ \},\ \text{'+'}\ ) \\
 & = & \langle \mathit{shift}, \mathit{closure}(\{e \to e\ \text{'+'}\ \bullet p\}) \rangle \\
 & = & \langle \mathit{shift}, s_8 \rangle.
\end{array}
$$

If we were to continue these calculations, we would obtain Table 9.1, as we have chosen the names of the states to match those of the corresponding states in Tables 9.1 and 9.2.

Exercise 26: Figure 9.7 shows a grammar for boolean expressions.

(a) Provide the sets *First*(v) for all syntactic variables v.

(b) Provide the sets *Follow*(v) for all syntactic variables v.

(c) Calculate the set of SLR states.

(d) Specify the function *action*.

(e) Specify the function *goto*.

Abbreviate the names of the syntactic variables and tokens with b, c, s, and I. ◇

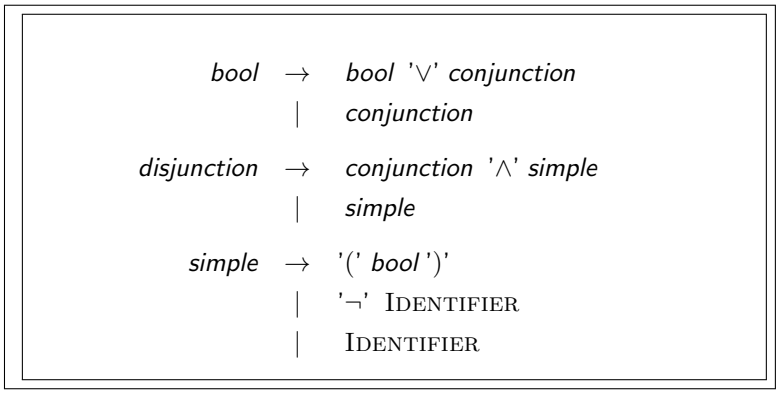

Figure 9.7: A grammar for Boolean expression in conjunctive normal form.

9.3.3 Shift/Reduce and Reduce/Reduce Conflicts

In this section, we examine shift-reduce and reduce-reduce conflicts in more detail and consider two examples. The first example shows a shift-reduce conflict. The grammar shown in Figure 9.8 is ambiguous because it does not specify whether the operator '+' binds stronger or weaker than the operator '∗'. If we interpret the non-terminal N as an abbreviation for NUMBER, this grammar allows us to interpret the expression $1 + 2 * 3$ either as

$$(1 + 2) * 3 \quad \text{or as} \quad 1 + (2 * 3).$$

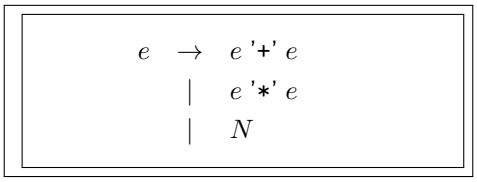

Figure 9.8: A grammar with shift/reduce conflicts.

We augment the grammar and calculate the start state s_0:

$$\begin{aligned} s_0 &= \mathit{closure}\big(\{s \to \bullet\, e\,\$\}\big) \\ &= \big\{s \to \bullet\, e\,\$,\ e \to \bullet\, e\text{'+'}e,\ e \to \bullet\, e\text{'}*\text{'}e,\ e \to \bullet\, N\ \big\}. \end{aligned}$$

Next, we calculate $s_1 := \mathit{goto}(s_0, e)$:

$$\begin{aligned} s_1 &= \mathit{goto}(s_0, e) \\ &= \mathit{closure}\big(\{s \to e \bullet\ \$,\ e \to e \bullet\ \text{'+'}\ e,\ e \to e \bullet\ \text{'}*\text{'}\ e\ \}\big) \\ &= \{s \to e \bullet\ \$,\ e \to e \bullet\ \text{'+'}\ e,\ e \to e \bullet\ \text{'}*\text{'}\ e\ \} \end{aligned}$$

Now we calculate $s_2 := \mathit{goto}(s_1, \text{'+'})$:

$$\begin{aligned} s_2 &= \mathit{goto}(s_1, \text{'+'}) \\ &= \mathit{closure}\big(\{e \to e\text{'+'} \bullet\, e,\ \}\big) \\ &= \{e \to e\ \text{'+'} \bullet\, e,\ e \to \bullet\, e\text{'+'}e,\ e \to \bullet\, e\text{'}*\text{'}e,\ e \to \bullet\, N\ \} \end{aligned}$$

Next, we calculate $s_3 := \mathit{goto}(s_2, e)$:

$$\begin{aligned} s_3 &= \mathit{goto}(s_2, e) \\ &= \mathit{closure}\big(\{e \to e\text{'+'}e\bullet,\ e \to e \bullet\ \text{'+'}\ e,\ e \to e \bullet\ \text{'}*\text{'}\ e\}\big) \\ &= \{e \to e\text{'+'}e\bullet,\ e \to e \bullet\ \text{'+'}\ e,\ e \to e \bullet\ \text{'}*\text{'}\ e\ \} \end{aligned}$$

Here, a shift-reduce conflict arises when calculating $\mathit{action}(s_3, \text{'}*\text{'})$, because on one hand, the marked rule

$$e \to e \bullet\ \text{'}*\text{'}\ e,$$

demands that the token '$*$' is pushed onto the stack, but on the other hand, we have

$$\mathit{Follow}(e) = \{\ \texttt{'+'},\ \texttt{'*'},\ \texttt{'\$'}\ \},$$

so that, if the next token to be read has the value '$*$', the symbol stack should be reduced with the rule

$$e \to e\,\texttt{'+'}\,e\,\bullet.$$

Remark: It is not surprising that we found a conflict in the grammar specified above, as this grammar is ambiguous. On the other hand, it can be shown that every SLR grammar must be unambiguous. Consequently, an ambiguous grammar is never an SLR grammar. However, the converse of this statement is not true, as we will see in the next example. ◇

$$\begin{array}{lcl} s & \to & a\ \texttt{'x'}\ a\ \texttt{'y'} \\ & \mid & b\ \texttt{'y'}\ b\ \texttt{'x'} \\ a & \to & \lambda \\ b & \to & \lambda \end{array}$$

Figure 9.9: A grammar containing a reduce/reduce conflict.

Next, we examine a grammar that is not an SLR grammar because it contains reduce-reduce conflicts. We consider the grammar shown in Figure 9.9. This grammar is unambiguous, as we have

$$L(s) = \{\ \texttt{'xy'},\ \texttt{'yx'}\ \}$$

and the string 'xy' can only be derived using the rule $s \to a\ \texttt{'x'}\ a\ \texttt{'y'}$, while the string 'yx' can only be generated using the rule $s \to b\ \texttt{'y'}\ b\ \texttt{'x'}$. To demonstrate that this grammar contains shift-reduce conflicts, we calculate the start state of an SLR parser for this grammar.

$$\begin{array}{lcl} s_0 & = & \mathit{closure}\big(\{\widehat{s} \to \bullet\, s\,\$\}\big) \\ & = & \big\{\widehat{s} \to \bullet\, s\,\$,\ s \to \bullet\, a\ \texttt{'x'}\ a\ \texttt{'y'}\ ,\ s \to \bullet\, b\ \texttt{'y'}\ b\ \texttt{'x'}\ ,\ a \to \bullet\,\lambda,\ b \to \bullet\,\lambda\ \big\} \\ & = & \big\{\widehat{s} \to \bullet\, s\,\$,\ s \to \bullet\, a\ \texttt{'x'}\ a\ \texttt{'y'}\ ,\ s \to \bullet\, b\ \texttt{'y'}\ b\ \texttt{'x'}\ ,\ a \to \lambda\bullet,\ b \to \lambda\bullet\ \big\}, \end{array}$$

since $a \to \bullet\,\lambda$ is the same as $a \to \lambda\bullet$. In this state, there is a reduce-reduce conflict between the two marked rules

$$a \to \lambda\,\bullet \quad \text{and} \quad b \to \lambda\,\bullet.$$

This conflict arises when calculating

$$\mathit{action}(s_0,\ \texttt{'x'}\ )$$

because we have

$$\mathit{Follow}(a) = \big\{\texttt{'x'}, \texttt{'y'}\big\} = \mathit{Follow}(b).$$

Thus, it is unclear which of these rules the parser should use to reduce the input in state s_0 when the next token read has the value 'x', as this token is both an element of the set $\mathit{Follow}(a)$ and the set $\mathit{Follow}(b)$.

Remark: As part of the resources provided with this lecture, the file

Formal-Languages/blob/master/Python/Chapter-09/SLR-Table-Generator.ipynb

contains a *Python* program that checks whether a given grammar is an SLR grammar. This program computes the states as well as the action table of a given grammar. ◇

9.4 Canonical LR Parsers

The reduce-reduce conflict that occurs in the grammar shown in Figure 9.9 can be resolved as follows: In the state

$$\begin{array}{lcl} s_0 & = & \mathit{closure}\big(\{\widehat{s} \to \bullet\, s\,\$\}\big) \\ & = & \big\{\widehat{s} \to \bullet\, s\,\$,\ s \to \bullet\, a\ \texttt{'x'}\ a\ \texttt{'y'}\ ,\ s \to \bullet\, b\ \texttt{'y'}\ b\ \texttt{'x'}\ ,\ a \to \lambda\bullet,\ b \to \lambda\bullet\ \big\} \end{array}$$

the marked rules $a \to \lambda\bullet$ and $b \to \lambda\bullet$ arise from the closure calculation of the rules

$$s \to \bullet\, a\ \texttt{'x'}\ a\ \texttt{'y'} \quad \text{and} \quad s \to \bullet\, b\ \texttt{'y'}\ b\ \texttt{'x'}\ .$$

For the first rule, it is clear that the first a must be followed by a 'x', and for the second rule, we see that the first b must be followed by a 'y'. This information goes beyond what is contained in the sets *Follow*(a) and *Follow*(b), as we now consider the context in which the syntactic variable appears. This allows us to define the function *action*$(s_0, \texttt{'x'})$ and *action*$(s_0, \texttt{'y'})$ as follows:

$$\mathit{action}(s_0, \texttt{'x'}) = \langle \texttt{reduce}, a \to \lambda\rangle \quad \text{and} \quad \mathit{action}(s_0, \texttt{'y'}) = \langle \texttt{reduce}, b \to \lambda\rangle.$$

This definition resolves the reduce-reduce conflict. The central idea is to include the context in which a rule appears in the closure calculation. To do this, we expand the definition of a marked rule next.

Definition 28 (Extended Marked Rule)
An extended marked rule (abbreviated: e.m.R.) of a grammar $G = \langle V, T, R, s\rangle$ is a quadruple

$$\langle a, \beta, \gamma, L\rangle,$$

where:

1. $(a \to \beta\gamma) \in R$.
2. $L \subseteq T$.

We write the extended marked rule $\langle a, \beta, \gamma, L\rangle$ as

$$a \to \beta \bullet \gamma : L.$$

If L consists of only one element t, that is if $L = \{t\}$, we omit the set brackets and write the rule as

$$a \to \beta \bullet \gamma : t.$$ ◇

Intuitively, we interpret the e.m.R. $a \to \beta \bullet \gamma : L$ as a state in which the following applies:

(a) The parser attempts to recognize an a using the grammar rule $a \to \beta\gamma$.

(b) So far, β has been recognized. For the rule $a \to \beta\gamma$ to be applied, γ must now be recognized.

(c) Additionally, we know that a token from the set L must follow the syntactic variable a.
Therefore, we refer to the set L as the set of Follow Tokens.

Working with extended marked rules is quite similar to working with marked rules, however, we need to modify the definitions of the functions *closure*, *goto*, and *action* slightly. We begin with the function *closure*.

Definition 29 (closure$(\mathcal{M})$) Let $\mathcal{M}$ be a set of extended marked rules. We define the closure of $\mathcal{M}$ as the smallest set $\mathcal{K}$ of marked rules for which the following holds:

1. $\mathcal{M} \subseteq \mathcal{K}$,
   thus the closure encompasses the original set of rules.
2. On one hand, if
   $$a \to \beta \bullet c\,\delta : L$$
   is an e.m.R. from the set $\mathcal{K}$, where c is a syntactic variable, and on the other hand, if

$$c \to \gamma$$

is a grammar rule of the underlying grammar G, then the e.m.R.

$$c \to \bullet\,\gamma : \bigcup\{\mathit{First}(\delta\, t) \mid t \in L\}$$

is also an element of the set $\mathcal{K}$. The function $\mathit{First}(\alpha)$ computes for a string $\alpha \in (T \cup V)^*$ the set of all tokens t with which a string that has been derived from α can begin.

The uniquely determined set $\mathcal{K}$ defined in this way is again denoted by $\mathit{closure}(\mathcal{M})$. ◇

Remark: Compared to the old definition, only the computation of the set of follow tokens has been added. The context in which the variable c, to be recognized by the rule $c \to \gamma$, appears is initially given by the string δ that follows c in the rule $a \to \beta \bullet c\,\delta : L$. Now it is possible that δ derives the empty string λ. In this case, the follow tokens from the set L also play a role, because if $\delta \Rightarrow^* \lambda$ is true, then a follow token t from the set L can also follow the c. ◇

For a given set of e.m.R.s $\mathcal{M}$, the computation of $\mathcal{K} := \mathit{closure}(\mathcal{M})$ can be done iteratively. Figure 9.10 shows the computation of $\mathit{closure}(\mathcal{M})$. The main difference compared to the earlier computation of *closure*() is that in the e.m.R.s that we include for a variable c in $\mathit{closure}(\mathcal{M})$, we consider the context in which c appears in the set of Follow Tokens. This makes it possible to describe the states of the parser more precisely than with marked rules.

```
1  function closure(M) {
2      K  := M;
3      K⁻ := {};
4      while (K⁻ ≠ K) {
5          K⁻ := K;
6          K  := K ∪ {(c → •γ : ⋃{First(δ t) | t ∈ L}) | (a → β • c δ : L) ∈ K ∧ (c → γ) ∈ R};
7      }
8      return K;
9  }
```

Figure 9.10: The computation of $\mathit{closure}(\mathcal{M})$.

Remark: The expression $\bigcup\{\mathit{First}(\delta\, t) \mid t \in L\}$ looks more complicated than it actually is. To compute this expression, it is useful to make a distinction based on whether δ can derive the empty string λ or not, because it holds that

$$\bigcup\{\mathit{First}(\delta\, t) \mid t \in L\} = \begin{cases} \mathit{First}(\delta) \cup L & \text{if } \delta \Rightarrow^* \lambda; \\ \mathit{First}(\delta) & \text{otherwise.} \end{cases}$$

The computation of $\mathit{goto}(\mathcal{M}, t)$ for a set $\mathcal{M}$ of extended rules and a symbol x changes from the computation in the case of simple marked rules only by appending the set of follow tokens, which itself remains unchanged:

$$\mathit{goto}(\mathcal{M}, x) := \mathit{closure}\Big(\big\{a \to \beta\, x \bullet \delta : L \mid (a \to \beta \bullet x\, \delta : L) \in \mathcal{M}\big\}\Big).$$

Similar to the theory of SLR parsers, we augment our grammar G by adding a new start variable $\widehat{s}$ to the set of variables and the new rule $\widehat{s} \to s$ to the set of rules. Further, we add the symbol \$ to the tokens. Then the start state has the form

$$q_0 := \mathit{closure}\big(\{\widehat{s} \to \bullet\, s : \$\}\big),$$

because the file end symbol "\$" must follow the start symbol. Finally, we show how the definition of the function *action*() must be changed. We specify the computation of this function through the following conditional equations.

1. $(a \to \beta \bullet t\,\delta : L) \in \mathcal{M} \;\Rightarrow\; \mathit{action}(\mathcal{M}, t) := \langle \texttt{shift}, \mathit{goto}(\mathcal{M}, t)\rangle$.
2. $(a \to \beta\bullet : L) \in \mathcal{M} \;\wedge\; a \neq \widehat{s} \;\wedge\; t \in L \;\Rightarrow\; \mathit{action}(\mathcal{M}, t) := \langle \texttt{reduce}, a \to \beta\rangle$.
3. $(\widehat{s} \to s\bullet : \$) \in \mathcal{M} \;\Rightarrow\; \mathit{action}(\mathcal{M}, \$) := \texttt{accept}$.

4. Otherwise: $\mathit{action}(\mathcal{M}, t) := \texttt{error}$.

If a conflict occurs in these equations, because both the condition of the first equation and the condition of the second equation are met, then we speak again of a shift-reduce conflict. A shift-reduce conflict thus occurs in the computation of $\mathit{action}(\mathcal{M}, t)$ when there are two e.m.R.s

$$(a \to \beta \bullet t\, \delta : L_1) \in \mathcal{M} \quad \text{and} \quad (c \to \gamma \bullet : L_2) \in \mathcal{M} \quad \text{with } t \in L_2$$

because then it is unclear whether in state $\mathcal{M}$ the token t should be pushed onto the stack, or whether instead the symbol stack should be reduced with the rule $c \to \gamma$.

Remark: Compared to an SLR parser, the possibility of shift-reduce conflicts is reduced, because in an SLR parser, a shift-reduce conflict already occurs if $t \in \mathit{Follow}(c)$ and the set L_2 is usually smaller than the set $\mathit{Follow}(c)$. ◇

A reduce-reduce conflict occurs when there are two e.m.R.s

$$(a \to \beta \bullet : L_1) \in \mathcal{M} \quad \text{and} \quad (c \to \delta \bullet : L_2) \in \mathcal{M} \quad \text{with} \quad L_1 \cap L_2 \neq \{\}$$

because then it is unclear which of these two rules should be used to reduce the symbol stack when the next token is an element of the intersection $L_1 \cap L_2$.

Remark: Compared to an SLR parser, the possibility of reduce-reduce conflicts is reduced, because in an SLR parser, a reduce-reduce conflict already occurs if there exists a t in the set $\mathit{Follow}(a) \cap \mathit{Follow}(c)$ and the *Follow* sets are often larger than the sets L_1 and L_2. ◇

Example: We revisit the example of the grammar shown in Figure 9.9 and first calculate the set of all states.

1. $s_0 := \mathit{closure}\Big(\{\widehat{s} \to \bullet\, s : \$\}\Big)$
   $= \{\widehat{s} \to \bullet\, s : \$, s \to \bullet\, a\texttt{'x'}a\texttt{'y'} : \$, s \to \bullet\, b\texttt{'y'}b\texttt{'x'} : \$, a \to \bullet : \texttt{'x'}, b \to \bullet : \texttt{'y'}\}$.

2. $s_1 := \mathit{goto}(s_0, a)$
   $= \mathit{closure}\big(\{s \to a \bullet \texttt{'x'}a\texttt{'y'} : \$\}\big)$
   $= \{s \to a \bullet \texttt{'x'}a\texttt{'y'} : \$\}$.

3. $s_2 := \mathit{goto}(s_0, s)$
   $= \mathit{closure}\big(\{\widehat{s} \to s\bullet : \$\}\big)$
   $= \{\widehat{s} \to s\bullet : \$\}$.

4. $s_3 := \mathit{goto}(s_0, b)$
   $= \mathit{closure}\big(\{s \to b \bullet \texttt{'y'}b\texttt{'x'} : \$\}\big)$
   $= \{s \to b \bullet \texttt{'y'}b\texttt{'x'} : \$\}$.

5. $s_4 := \mathit{goto}(s_3, \texttt{'y'})$
   $= \mathit{closure}\big(\{s \to b\texttt{'y'} \bullet b\texttt{'x'} : \$\}\big)$
   $= \{s \to b\texttt{'y'} \bullet b\texttt{'x'} : \$, b \to \bullet : \texttt{'x'}\}$.

6. $s_5 := \mathit{goto}(s_4, b)$
   $= \mathit{closure}\big(\{s \to b\texttt{'y'}b \bullet \texttt{'x'} : \$\}\big)$
   $= \{s \to b\texttt{'y'}b \bullet \texttt{'x'} : \$\}$.

7. $s_6 := \mathit{goto}(s_5, \texttt{'x'})$
   $= \mathit{closure}\big(\{s \to b\texttt{'y'}b\texttt{'x'}\bullet : \$\}\big)$
   $= \{s \to b\texttt{'y'}b\texttt{'x'}\bullet : \$\}$.

8. $s_7 := \mathit{goto}(s_1, \texttt{'x'})$
   $= \mathit{closure}\big(\{s \to a\texttt{'x'} \bullet a\texttt{'y'} : \$\}\big)$
   $= \{s \to a\texttt{'x'} \bullet a\texttt{'y'} : \$, a \to \bullet : \texttt{'y'}\}$.

9. $s_8 := \mathit{goto}(s_7, a)$
$= \mathit{closure}(\{s \to a\,\texttt{'x'}\,a \bullet \texttt{'y'} : \$\})$
$= \{s \to a\,\texttt{'x'}\,a \bullet \texttt{'y'} : \$\}.$

10. $s_9 := \mathit{goto}(s_8, \texttt{'y'})$
$= \mathit{closure}(\{s \to a\,\texttt{'x'}\,a\,\texttt{'y'} \bullet : \$\})$
$= \{s \to a\,\texttt{'x'}\,a\,\texttt{'y'} \bullet : \$\}.$

Next, we examine whether there are any conflicts in the states. In the last section, we found a Reduce-Reduce conflict in the start state s_0 between the two rules $a \to \lambda$ and $b \to \lambda$, because

$$\mathit{Follow}(a) \cap \mathit{Follow}(b) = \{\texttt{'x'}, \texttt{'y'}\} \neq \{\}$$

holds. This conflict has now disappeared, as there is no conflict between the e.m.R.s

$$a \to \bullet : \texttt{'x'} \quad \text{and} \quad b \to \bullet : \texttt{'y'}$$

due to 'x' $\neq$ 'y'. It is easy to see that no conflicts arise in the other states either.

Exercise 27: Calculate the set of states for an LR parser for the following grammar:

$$\begin{array}{lcl} e & \to & e\ \texttt{'+'}\ p \\ & \mid & p \\ p & \to & p\ \texttt{'*'}\ f \\ & \mid & f \\ f & \to & \texttt{'('}\ e\ \texttt{')'} \\ & \mid & N \end{array}$$

Additionally, investigate whether this grammar has shift-reduce conflicts or reduce-reduce conflicts.

Remark: As part of the resources provided with this lecture, the file

Python/Chapter-09/LR-Table-Generator.ipynb

contains a *Python* program that checks whether a given grammar qualifies as a canonical LR grammar. This program computes the LR-states as well as the action table for a given grammar. ◇

Remark: The theory of LR-parsing has been developed by Donald E. Knuth [Knu65]. His theory is described in the paper "On the translation of languages from left to right". ◇

9.5 LALR-Parser

The number of states in an LR parser is often significantly larger than the number of states in an SLR parser for the same grammar. For instance, an SLR parser for the C grammar manages with 349 states. However, since the language C is not an SLR language, creating an SLR parse table for C results in a number of conflicts, so that an SLR parser for the language C does not work. In contrast, an LR parser for the language C has 1572 states, as you can see here. In the seventies, when the available main memory of most computers was still modestly sized compared to today, LR parsers therefore had a size that was problematic in practice. A detailed analysis of the set of states of LR parsers showed that it is often possible to combine certain states. This can significantly reduce the number of states in most cases. We illustrate the concept with an example and consider the grammar shown in Figure 9.11, which I have taken from the Dragon Book [ASUL06]. (The "Dragon Book" is the standard work in the field of compiler construction.)

Figure 9.12 shows the so-called LR-goto-graph for this grammar. The nodes of this graph are the states. Looking at the LR-goto-graph, we find that the states s_6 and s_3 differ only in the sets of follow tokens, as on one hand, we have

$$
\begin{array}{rcl}
\widehat{s} & \to & s \\
s & \to & c\ c \\
c & \to & \texttt{'x'}\ c \\
 & | & \texttt{'y'}
\end{array}
$$

Figure 9.11: Eine Grammatik aus dem Drachenbuch.

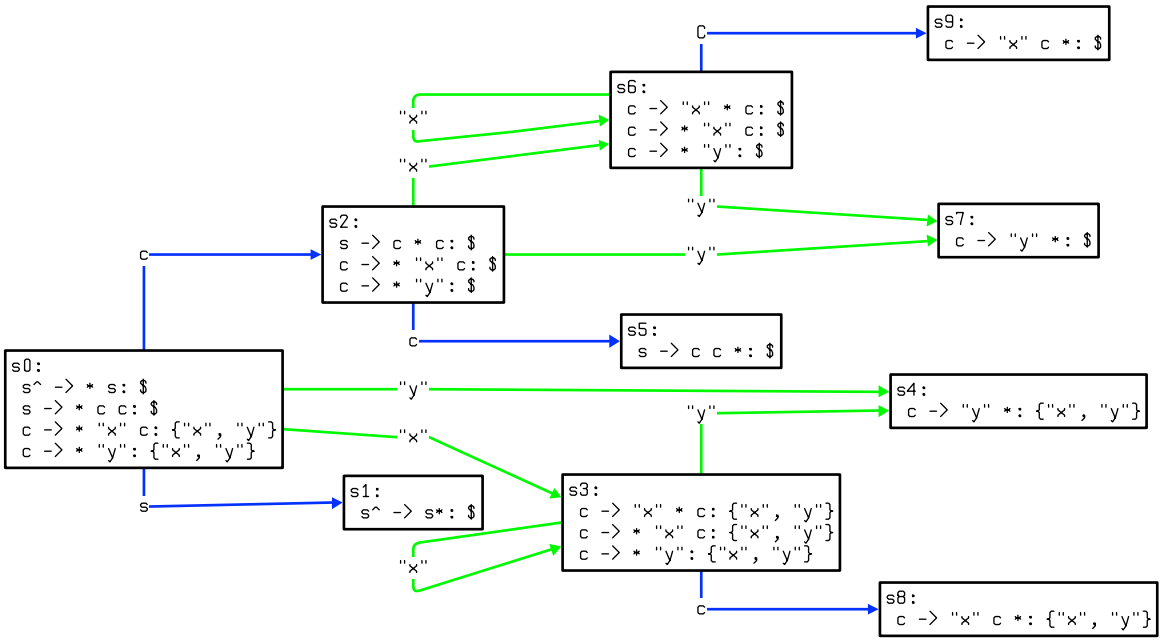

Figure 9.12: LR-goto-graph for the grammar of Figure 9.11.

$$s_6 = \Big\{ s \to \texttt{'x'} \bullet c : \texttt{'\$'}, c \to \bullet\ \texttt{'x'}\ c : \texttt{'\$'}, c \to \bullet\ \texttt{'y'}\ : \texttt{'\$'} \Big\},$$

and on the other hand, we have

$$s_3 = \Big\{ s \to \texttt{'x'} \bullet c : \{\texttt{'x'}, \texttt{'y'}\}, c \to \bullet\ \texttt{'x'}\ c : \{\texttt{'x'}, \texttt{'y'}\}, c \to \bullet\ \texttt{'y'}\ : \{\texttt{'x'}, \texttt{'y'}\} \Big\}.$$

Clearly, the set s_3 is derived from the set s_6 by replacing everywhere '\$' with the set $\{\texttt{'x'}, \texttt{'y'}\}$. Similarly, the set s_7 can be transformed into s_4, and s_9 into s_8. The crucial realization is that the function *goto*() is invariant under this kind of transformation, because in the definition of this function, the set of follow tokens is irrelevant. For example, we see that

$$\mathit{goto}(s_3, c) = s_8 \quad \text{and} \quad \mathit{goto}(s_6, c) = s_9$$

hold and that, on the other hand, the state s_9 changes into the state s_8 when we replace everywhere in s_9 the terminal '\$' with the set $\{\texttt{'x'}, \texttt{'y'}\}$. If we define the core of a set of extended marked rules by omitting the set of follow tokens in each rule, and then combine states with the same core, we obtain the goto-graph shown in Figure 9.13.

To generalize and formalize the observations made while examining the grammar shown in Figure 9.11, we define a function *core*(), which calculates the core of a set of e.m.R.s (extended marked rules) and thus transforms this set into a set of marked rules:

$$\mathit{core}(\mathcal{M}) := \{a \to \beta \bullet \gamma \mid (a \to \beta \bullet \gamma : L) \in \mathcal{M}\}.$$

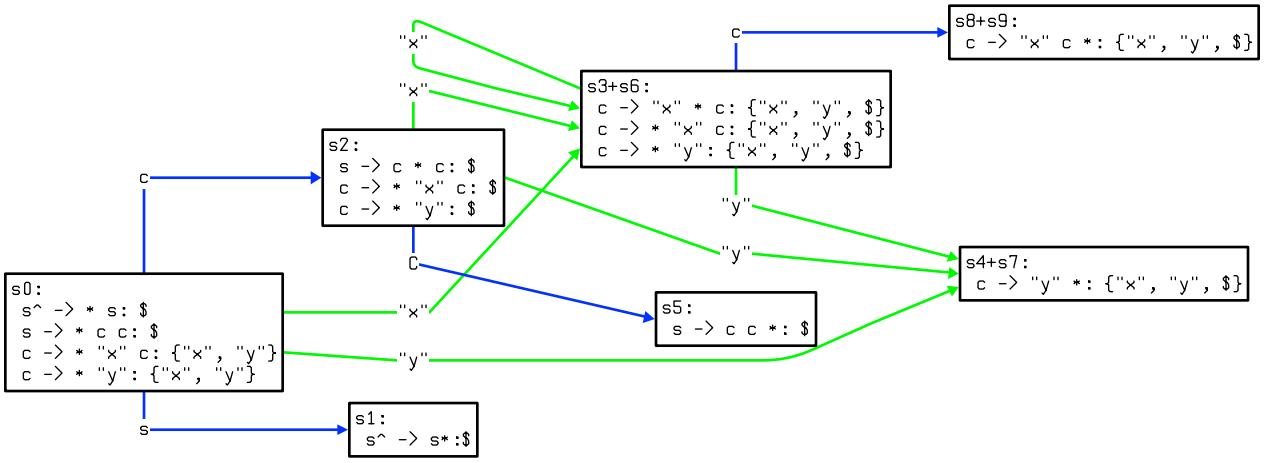

Figure 9.13: The LALR-goto-graph for the grammar from Figure 9.11.

Thus, the function *core*() simply removes the set of follow tokens from the e.m.R.s. We had defined the function *goto*() for a set $\mathcal{M}$ of extended marked rules and a symbol x as

$$\mathit{goto}(\mathcal{M}, x) := \mathit{closure}\Big(\big\{a \to \beta\, x \bullet \gamma : L \mid (a \to \beta \bullet x\, \gamma : L) \in \mathcal{M}\big\}\Big).$$

Clearly, the set of follow tokens plays no role in the calculation of $\mathit{goto}(\mathcal{M}, x)$, formally for two e.m.R.-sets $\mathcal{M}_1$ and $\mathcal{M}_2$ and a symbol x, the formula holds:

$$\mathit{core}(\mathcal{M}_1) = \mathit{core}(\mathcal{M}_2) \;\Rightarrow\; \mathit{core}(\mathit{goto}(\mathcal{M}_1, x)) = \mathit{core}(\mathit{goto}(\mathcal{M}_2, x)).$$

For two sets of e.m.R.s (extended marked rules) $\mathcal{M}$ and $\mathcal{N}$ that have the same core, we define the extended union $\mathcal{M} \uplus \mathcal{N}$ of $\mathcal{M}$ and $\mathcal{N}$ as

$$\mathcal{M} \uplus \mathcal{N} := \{a \to \beta \bullet \gamma : K \cup L \mid (a \to \beta \bullet \gamma : K) \in \mathcal{M} \;\wedge\; (a \to \beta \bullet \gamma : L) \in \mathcal{N}\}.$$

We generalize this definition to an operation $\biguplus$, defined on a set of sets of e.m.R.s: If $\mathfrak{I}$ is a set of sets of e.m.R.s, all having the same core, i.e.,

$$\mathfrak{I} = \{\mathcal{M}_1, \cdots, \mathcal{M}_k\} \quad \text{with} \quad \mathit{core}(\mathcal{M}_i) = \mathit{core}(\mathcal{M}_j) \quad \text{for all } i, j \in \{1, \cdots, k\},$$

then we define

$$\biguplus \mathfrak{I} := \mathcal{M}_1 \uplus \cdots \uplus \mathcal{M}_k.$$

Let Δ be the set of all states of an LR parser. Then, the set of states of the corresponding LALR parser is given by the extended union of all subsets of Δ whose elements have the same core:

$$\mathfrak{Q} := \Big\{ \biguplus \mathfrak{I} \mid \mathfrak{I} \in 2^{\Delta} \wedge \forall \mathcal{M}, \mathcal{N} \in \mathfrak{I} : \mathit{core}(\mathcal{M}) = \mathit{core}(\mathcal{N}) \wedge \text{and } \mathfrak{I} \text{ is maximal} \Big\}.$$

The requirement "$\mathfrak{I}$ maximal" in the above definition expresses that in $\mathfrak{I}$, indeed all sets from Δ are combined that have the same core. The thus defined set $\mathfrak{Q}$ is the set of LALR states.

Next, we consider how the computation of $\mathit{goto}(\mathcal{M}, X)$ must change if $\mathcal{M}$ is an element of the set $\mathfrak{Q}$ of LALR states. To compute $\mathit{goto}(\mathcal{M}, X)$, we first calculate the set

$$\mathit{closure}\Big(\big\{a \to \alpha X \bullet \beta : L \mid (a \to \alpha \bullet X \beta : L) \in \mathcal{M}\big\}\Big).$$

The problem is that this set is generally not an element of the set $\mathfrak{Q}$, as the states in $\mathfrak{Q}$ arise from the combination of several LR states. However, the states that are combined in the computation of $\mathfrak{Q}$ all have the same core. Therefore, the set

$$\Big\{q \in \mathfrak{Q} \mid \mathit{core}(q) = \mathit{core}\big(\mathit{closure}\big(\{a \to \beta\, x \bullet \gamma : L \mid (a \to \beta \bullet x\, \gamma : L) \in \mathcal{M}\}\big)\big)\Big\}$$

contains exactly one element and this element is the value of $\mathit{goto}(\mathcal{M}, X)$. Thus, we can set

$$\mathit{goto}(\mathcal{M}, X) := \mathit{arb}\Big(\Big\{q \in \mathfrak{Q} \mid \mathit{core}(q) = \mathit{core}\Big(\mathit{closure}\Big(\big\{a \to \beta\, X \bullet \gamma : L \mid (a \to \beta \bullet X\, \gamma : L) \in \mathcal{M}\big\}\Big)\Big)\Big\}\Big)$$

The function $\mathit{arb}()$ used here serves to extract an arbitrary element from a set. Since the set from which the element is extracted contains exactly one element, $\mathit{goto}(\mathcal{M}, x)$ is well-defined. The computation of the expression $\mathit{action}(\mathcal{M}, t)$ does not change compared to the computation for an LR parser.

9.6 Comparison of SLR, LR, and LALR Parsers

We now want to compare the different methods with which we have constructed Shift-Reduce parsers in this chapter. We call a language $\mathcal{L}$ an SLR language if $\mathcal{L}$ can be recognized by an SLR parser. The terms canonical LR language and LALR language are defined analogously. The following relationships exist between these languages:

$$\text{SLR language} \subsetneq \text{LALR language} \subsetneq \text{canonical LR language} \tag{$\star$}$$

These inclusions are easy to understand: In defining the LR parsers, we added sets of follow tokens to the marked rules. This made it possible to avoid Shift-Reduce and Reduce-Reduce conflicts in certain cases. As the state sets of canonical LR parsers can become very large, we then combined sets of extended marked rules that have the same core. This is how we obtained the LALR parsers. However, by combining rule sets, we can in some cases introduce Reduce-Reduce conflicts, so the set of LALR languages is a subset of the canonical LR languages.

We will show in the following subsections that the inclusions in $(\star)$ are proper.

9.6.1 SLR Language $\subsetneq$ LALR Language

The states of an LALR parser, as compared to those of an SLR parser, include sets of follow tokens. This makes LALR parsers at least as powerful as SLR parsers. We will now demonstrate that LALR parsers are indeed more powerful than SLR parsers. To support this claim, we present a grammar for which there exists an LALR parser, but no SLR parser. We had seen on page 119 that the grammar

$$\begin{array}{lcl} s & \to & a\ \texttt{'x'}\ a\ \texttt{'y'} \mid b\ \texttt{'y'}\ b\ \texttt{'x'} \\ a & \to & \lambda \\ b & \to & \lambda \end{array}$$

is not an SLR grammar. Later, we saw that this grammar can be parsed by a canonical LR parser. We will now show that this grammar can also be parsed by an LALR parser. To do this, we calculate the set of LALR states. This requires first calculating the set of canonical LR states. We had already carried out this calculation earlier and obtained the following states:

1. $s_0 = \big\{\widehat{s} \to \bullet\, s : \$, s \to \bullet\, a\texttt{'x'}a\texttt{'y'} : \$, s \to \bullet\, b\texttt{'y'}b\texttt{'x'} : \$, a \to \bullet\ : \texttt{'x'}, b \to \bullet\ : \texttt{'y'}\big\}$,
2. $s_1 = \big\{s \to a \bullet\, \texttt{'x'}a\texttt{'y'} : \$\big\}$,
3. $s_2 = \big\{\widehat{s} \to s\bullet : \$\big\}$,
4. $s_3 = \big\{s \to b \bullet\, \texttt{'y'}b\texttt{'x'} : \$\big\}$,
5. $s_4 = \big\{s \to b\texttt{'y'} \bullet\, b\texttt{'x'} : \$, b \to \bullet\ : \texttt{'x'}\big\}$,
6. $s_5 = \big\{s \to b\texttt{'y'}b \bullet\, \texttt{'x'} : \$\big\}$,
7. $s_6 = \big\{s \to b\texttt{'y'}b\texttt{'x'}\bullet : \$\big\}$,
8. $s_7 = \big\{s \to a\texttt{'x'} \bullet\, a\texttt{'y'} : \$, a \to \bullet\ : \texttt{'y'}\big\}$,
9. $s_8 = \big\{s \to a\texttt{'x'}a \bullet\, \texttt{'y'} : \$\big\}$,
10. $s_9 = \big\{s \to a\texttt{'x'}a\texttt{'y'}\bullet : \$\big\}$.

We observe that the cores of all the states listed here are different. Therefore, for this grammar, the set of states of the LALR parser coincides with the set of states of the canonical LR parser. This implies that there are no conflicts in the LALR states, because in the transition from canonical LR parsers to LALR parsers, we have only combined states with the same core, and the definition of the functions *goto*() and *action*() remained unchanged.

9.6.2 LALR Language $\subsetneq$ Canonical LR Language

We defined LALR parsers by combining different states of a canonical LR parser. Therefore, it is clear that canonical LR parsers are at least as powerful as LALR parsers. To demonstrate that canonical LR parsers are indeed more powerful than LALR parsers, we need a grammar for which a canonical LR parser can be generated, but not an LALR parser. Figure 9.14 shows such a grammar, which I have taken from the Dragon Book.

$$
\begin{array}{rcl}
s & \rightarrow & \texttt{'v'}\ a\ \texttt{'y'} \\
 & | & \texttt{'w'}\ b\ \texttt{'y'} \\
 & | & \texttt{'v'}\ b\ \texttt{'z'} \\
 & | & \texttt{'w'}\ a\ \texttt{'z'} \\
a & \rightarrow & \texttt{'x'} \\
b & \rightarrow & \texttt{'x'}
\end{array}
$$

Figure 9.14: A canonical LR grammar that is not an LALR grammar.

We first calculate the set of states for a canonical LR parser for this grammar. We obtain the following sets of extended marked rules:

1. $s_0 = \mathit{closure}(\widehat{s} \rightarrow \bullet\ s : \$) = \{\ \widehat{s} \rightarrow \bullet\ s : \$,\ s \rightarrow \bullet\texttt{'v'}a\texttt{'y'} : \$,\ s \rightarrow \bullet\texttt{'v'}b\texttt{'z'} : \$,\ s \rightarrow \bullet\texttt{'w'}a\texttt{'z'} : \$,\ s \rightarrow \bullet\texttt{'w'}b\texttt{'y'} : \$\ \},$
2. $s_1 = \mathit{goto}(s_0, s) = \{\widehat{s} \rightarrow s\bullet : \$\}$
3. $s_2 = \mathit{goto}(s_0,\ \texttt{'v'}\ ) = \{\ s \rightarrow \texttt{'v'} \bullet b\texttt{'z'} : \$,\ s \rightarrow \texttt{'v'} \bullet a\texttt{'y'} : \$,\ a \rightarrow \bullet\texttt{'x'} : \texttt{'y'},\ b \rightarrow \bullet\texttt{'x'} : \texttt{'z'}\ \},$
4. $s_3 = \mathit{goto}(s_0,\ \texttt{'w'}\ ) = \{\ s \rightarrow \texttt{'w'} \bullet a\texttt{'z'} : \$,\ s \rightarrow \texttt{'w'} \bullet b\texttt{'y'} : \$,\ a \rightarrow \bullet\texttt{'x'} : \texttt{'z'},\ b \rightarrow \bullet\texttt{'x'} : \texttt{'y'}\ \},$
5. $s_4 = \mathit{goto}(s_2,\ \texttt{'x'}\ ) = \{a \rightarrow \texttt{'x'}\bullet : \texttt{'y'},\ b \rightarrow \texttt{'x'}\bullet : \texttt{'z'}\},$
6. $s_5 = \mathit{goto}(s_3,\ \texttt{'x'}\ ) = \{a \rightarrow \texttt{'x'}\bullet : \texttt{'z'},\ b \rightarrow \texttt{'x'}\bullet : \texttt{'y'}\},$
7. $s_6 = \mathit{goto}(s_2, a) = \{s \rightarrow \texttt{'v'}a \bullet \texttt{'y'} : \$\},$
8. $s_7 = \mathit{goto}(s_6,\ \texttt{'y'}\ ) = \{s \rightarrow \texttt{'v'}a\texttt{'y'}\bullet : \$\},$
9. $s_8 = \mathit{goto}(s_2, b) = \{s \rightarrow \texttt{'v'}b \bullet \texttt{'z'} : \$\},$
10. $s_9 = \mathit{goto}(s_8,\ \texttt{'z'}\ ) = \{s \rightarrow \texttt{'v'}b\texttt{'z'}\bullet : \$\},$

11. $s_{10} = \mathit{goto}(s_3, a) = \{s \to \texttt{'w'}a \bullet \texttt{'z'} : \$\}$,
12. $s_{11} = \mathit{goto}(s_{10}, \texttt{'z'}) = \{s \to \texttt{'w'}a\texttt{'z'} \bullet : \$\}$,
13. $s_{12} = \mathit{goto}(s_3, b) = \{s \to \texttt{'w'}b \bullet \texttt{'y'} : \$\}$,
14. $s_{13} = \mathit{goto}(s_{12}, \texttt{'y'}) = \{s \to \texttt{'w'}b\texttt{'y'} \bullet : \$\}$.

The only states where conflicts could occur are the sets s_4 and s_5, as in these states, reductions with both rules

$$a \to \texttt{'x'} \quad \text{and} \quad b \to \texttt{'x'}$$

are potentially possible. However, since the sets of follow tokens have an empty intersection, there is actually no conflict, and the grammar is a canonical LR grammar.

Next, we calculate the LALR states for the grammar mentioned above. The only states that have a common core are s_4 and s_5, as

$$\mathit{core}(s_4) = \{a \to \texttt{'x'} \bullet,\ b \to \texttt{'x'} \bullet\} = \mathit{core}(s_5).$$

In calculating the LALR states, these two states are combined into one state $s_{\{4,5\}}$. This new state takes the form

$$s_{\{4,5\}} = \big\{A \to \texttt{'x'} \bullet : \{\texttt{'y'}, \texttt{'z'}\},\ B \to \texttt{'x'} \bullet : \{\texttt{'y'}, \texttt{'z'}\}\big\}.$$

Here, there is obviously a reduce-reduce conflict, as on one hand we have

$$\mathit{action}(s_{\{4,5\}}, \texttt{'y'}) = \langle \mathit{reduce}, A \to \texttt{'x'} \rangle,$$

but on the other hand, it also holds that

$$\mathit{action}(s_{\{4,5\}}, \texttt{'y'}) = \langle \mathit{reduce}, B \to \texttt{'x'} \rangle.$$

Exercise 28: Let $G = \langle V, T, R, s \rangle$ be an LR grammar and $\mathcal{N}$ be the set of LALR states of the grammar. Explain why there can be no shift-reduce conflicts in the set $\mathcal{N}$. ◇

Historical Notes The theory of LALR parsing is due to Franklin L. DeRemer [DeR71]. At the time of its invention, the space savings of LALR parsing in comparison to LR parsing were crucial.

9.6.3 Evaluation of the Different Methods

In practice, SLR parsers are not sufficient, as there are a number of practically relevant language constructs for which no SLR parser can be generated. Canonical LR parsers are much more powerful but often require significantly more states. LALR parsers represent a compromise here: on the one hand, LALR languages are almost as expressive as canonical LR languages, but on the other hand, the memory requirements of LALR parsers are in the same order of magnitude as those of SLR parsers. For example, the SLR parse table for the `C` language has a total of 349 states, the corresponding LR parse table has 1572 states, while the LALR parser manages with 350 states, thus only one state more than the SLR parser. In the main memories typically available today, however, canonical LR parsers can usually be accommodated without difficulty, so there is actually no compelling reason to use an LALR parser instead of an LR parser.

On the other hand, no one will want to program an LALR parser or a canonical LR parser by hand. Instead, you will later use a parser generator like *Bison* or *JavaCup*, which generates a parser for you. Bison is a parser generator for `C`, `C++` and also offers, albeit still experimental, support for *Java*, while *JavaCup* is limited to the language *Java*. If you use *JavaCup*, you have no choice, as this tool always generates an LALR parser. With *Bison*, from version 3.0, it is also possible to generate an LR parser.

9.7 Check your Understanding

(a) What is the definition of a shift-reduce parser and how is the set *Action* defined?

(b) How does a shift-reduce parser work?

If you are unsure about this question, you should run the notebook

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Chapter-09/Shift-Reduce-Parser.ipynb
and inspect its output.

(c) How is the concept of a marked rule defined? How is a marked rule interpreted?

(d) Can you compute the closure of a set of marked rules?

(e) Are you able to define the functions `First` and `Follow` for an SLR-parser?

(f) Are you able to compute the states of an SLR-parser?

(g) Given the set of all states of an SLR-parser, can you compute both the action table and the goto table?

(h) How is the concept of an extended marked rule defined?

(i) Can you compute the closure of a set of extended marked rules?

(j) Why are LR-parser-generators more powerful than SLR-parser-generators?

Chapter 10

Advanced Features of Ply

When we discussed the parser generator PLY in chapter 7, we were not able to discuss all the features of PLY, since the underlying theory, which is the theory of LALR parsing, had not been developed. Since we have aquainted ourselves with this theory in the pervious chapter, we are finally able to discuss some of the more advanced features of PLY.

1. First, we show how shift-reduce conflicts are presented in PLY.
2. Then we discuss how shift-reduce conflicts can be resolved with the help of precedence declarations.

10.1 Shift-Reduce and Reduce-Reduce Conflicts

In this section we show how shift-reduce and reduce-reduce conflicts are dealt with in PLY. Figure 10.1 on page 130 shows a grammar for arithmetical expressions that is ambiguous because it does not specify the precedence of the different arithmetical operators.

```
1  expr : expr '+' expr
2       | expr '*' expr
3       | NUMBER
```

Figure 10.1: An ambiguous grammar for arithmetical expressions.

This grammar does not specify whether the string

"1 + 2 * 3" is interpreted as "(1 + 2) * 3" or as "1 + (2 * 3)".

Since every LALR grammar is unambiguous, but the grammar shown in Figure 10.1 is ambiguous, it has to have at least one shift-reduce or reduce-reduce conflict. This grammar is part of the jupyter notebook

Formal-Languages/tree/master/Python/Chapter-10/01-Conflicts.ipynb.

When we try to generate a parser for this grammar using PLYs yacc command we get the message

```
WARNING: 4 shift/reduce conflicts.
```

The file `parser.out` that is generated by PLY shows how these conflicts are resolved. This file contains the LALR states created by the parser generator and for every state the possible actions are shown. Given the grammar shown above, PLY creates 6 different states. There are conflicts in two of these states. Figure 10.2 on page 131 shows state number 5 and its actions. We see that there are 2 shift-reduce conflicts in this state. Unfortunately, PLY only prints the marked rules defining these states, but it does not show the follow sets of these rules. We see that PLY resolves all conflicts in favour of shifting. The exclamation marks in the beginning of the line 13 and 14 are to be interpreted as negations and show those reduce actions that would have been possible in state 5, but are discarded in favour of the shift actions shown in line 10 and 11. Of course, in this example the shift action in line 10 is wrong

```
state 5

    (1) expr -> expr + expr .
    (1) expr -> expr . + expr
    (2) expr -> expr . * expr

  ! shift/reduce conflict for + resolved as shift
  ! shift/reduce conflict for * resolved as shift
    $end            reduce using rule 1 (expr -> expr + expr .)
    +               shift and go to state 3
    *               shift and go to state 4

  ! +               [ reduce using rule 1 (expr -> expr + expr .) ]
  ! *               [ reduce using rule 1 (expr -> expr + expr .) ]
```

Figure 10.2: An excerpt from the file `parse.out`.

because then the string

```
1 * 2 + 3
```

is interpreted as `1 * (2 + 3)` and not as `(1 * 2) + 3`.

10.2 Operator Precedence Declarations

expr → *expr* "+" *expr*
| *expr* "−" *expr*
| *expr* "∗" *expr*
| *expr* "/" *expr*
| *expr* "^" *expr*
| "(" *expr* ")"
| NUMBER

Figure 10.3: A grammar for a arithmetical expressions.

It is possible to resolve shift-reduce conflicts using operator precedence declarations. For example, for the grammar for arithmetical expressions shown in Figure 10.3 on page 131 we can use the following operator precedence declarations:

```
precedence = ( ('left', '+', '-'),   # precedence 1
               ('left', '*', '/'),   # precedence 2
               ('right', '^')        # precedence 3
             )
```

This declaration specifies that the operators "+" and "−" have a lower precedence than the operators "∗" and "/". Furthermore, it specifies that all these operators associate to the left. The operator "^" has the highest precedence and associates to the right as specified by the keyword "`right`". The jupyter notebook

Formal-Languages/tree/master/Python/Chapter-10/02-Conflicts-Resolved.ipynb.

shows this grammar. When we run this notebook, PLY doesn't give us a warning about any conflicts. If we inspect the generated file `parse.out`, the action table for the state number 11 has the form shown in Figure 10.4 on page 132.

```
1  state 11
2
3      (2) expr -> expr - expr .
4      (1) expr -> expr . + expr
5      (2) expr -> expr . - expr
6      (3) expr -> expr . * expr
7      (4) expr -> expr . / expr
8      (5) expr -> expr . ^ expr
9
10     +               reduce using rule 2 (expr -> expr - expr .)
11     -               reduce using rule 2 (expr -> expr - expr .)
12     $end            reduce using rule 2 (expr -> expr - expr .)
13     )               reduce using rule 2 (expr -> expr - expr .)
14     *               shift and go to state 6
15     /               shift and go to state 7
16     ^               shift and go to state 8
17
18   ! *               [ reduce using rule 2 (expr -> expr - expr .) ]
19   ! /               [ reduce using rule 2 (expr -> expr - expr .) ]
20   ! ^               [ reduce using rule 2 (expr -> expr - expr .) ]
21   ! +               [ shift and go to state 4 ]
22   ! -               [ shift and go to state 5 ]
```

Figure 10.4: An excerpt from the file `parse.out` when conflicts are resolved.

1. Since the operators "+" and "−" have the same precedence, we have

   $$\textit{action}(\texttt{state11}, \text{“+”}) = \langle \textit{reduce}, \textit{expr} \rightarrow \textit{expr}\ \text{“−”}\ \textit{expr} \rangle$$

   This way, the expression $1 - 2 + 3$ is parsed as $(1 - 2) + 3$ and not as $1 - (2 + 3)$ as it would if we would shift the operator "+" instead.

2. Since the operator "−" is left associative, we have

   $$\textit{action}(\texttt{state11}, \text{“−”}) = \langle \textit{reduce}, \textit{expr} \rightarrow \textit{expr}\ \text{“−”}\ \textit{expr} \rangle$$

   This way, the expression $1 - 2 - 3$ is parsed as $(1 - 2) - 3$.

3. Since the precedence of the operator "*" is higher than the precedence of the operator "−", we have

   $$\textit{action}(\texttt{state11}, \text{“*”}) = \langle \textit{shift}, \texttt{state6} \rangle$$

   This way, the expression $1 - 2 * 3$ is parsed as $1 - (2 * 3)$.

4. Since the precedence of the operator "/" is higher than the precedence of the operator "−", we have

   $$\textit{action}(\texttt{state11}, \text{“/”}) = \langle \textit{shift}, \texttt{state7} \rangle$$

   This way, the expression $1 - 2/3$ is parsed as $1 - (2/3)$.

Next, we explain in detail how PLY uses operator precedence relations to resolve shift-reduce conflicts.

1. First, PLY assigns a precedence level to every grammar rule. This precedence level is the precedence level of the last operator symbol occurring in the grammar rule. Most of the times, there is just one operator that determines the precedence of the grammar rule. In the grammar at hand the precedences of the rules would be as shown in the table below.

rule	precedence
expr → *expr* "+" *expr*	1
expr → *expr* "–" *expr*	1
expr → *expr* "*" *expr*	2
expr → *expr* "/" *expr*	2
expr → *expr* "^" *expr*	3
expr → "(" *expr* ")"	—
expr → NUMBER	—

   If a grammar rule does not contain an operator that has been given a precedence, then the precedence of the grammar rule remains undefined.

2. If s is a state that contains two e.m.R.s r_1 and r_2 such that

$$r_1 = (a \to \beta \bullet o\ \delta : L_1) \quad \text{and} \quad r_2 = (c \to \gamma \bullet : L_2) \quad \text{where} \quad o \in L_2,$$

   then there is a shift-reduce conflict when

$$\mathit{action}(s, o)$$

   is computed. Let us assume that the precedence of the operator o is $p(o)$ and the precedence of the rule r_2 is $p(r_2)$. Then there are six cases that depend on the relative values of $p(o)$ and $p(r_2)$ and on the associativity of the operator o.

   (a) $p(o) > p(r_2)$.
   In this case the precedence of the operator o is higher than the precedence of the rule r_2. Therefore the operator o is shifted:

$$\mathit{action}(s, o) = \langle \texttt{shift}, \mathit{goto}(s, o) \rangle.$$

   To understand this rule we just have to watch what happens when we parse

```
1+2*3
```

   using the grammar given above. After the part "1+2" has been read and the next token is the operator "*", the parser is in the following state:

$$\left\{ \begin{array}{lcl} \mathit{expr} & \to & \mathit{expr} \bullet \text{"+"}\ \mathit{expr} : \{\$, \text{"+"}, \text{"–"}\ \text{"*"}, \text{"/"}, \text{"^"}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{"–"}\ \mathit{expr} : \{\$, \text{"+"}, \text{"–"}\ \text{"*"}, \text{"/"}, \text{"^"}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{"*"}\ \mathit{expr} : \{\$, \text{"+"}, \text{"–"}\ \text{"*"}, \text{"/"}, \text{"^"}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{"/"}\ \mathit{expr} : \{\$, \text{"+"}, \text{"–"}\ \text{"*"}, \text{"/"}, \text{"^"}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{"^"}\ \mathit{expr} : \{\$, \text{"+"}, \text{"–"}\ \text{"*"}, \text{"/"}, \text{"^"}\}, \\ \mathit{expr} & \to & \mathit{expr}\text{"+"}\ \mathit{expr} \bullet : \{\$, \text{"+"}, \text{"–"}\ \text{"*"}, \text{"/"}, \text{"^"}\} \end{array} \right\}.$$

   When the parser next sees the token "*", then it must not reduce the symbol stack using the rule *expr* → *expr*"+"*expr*, because it has to multiply the numbers 2 and 3 first. Therefore, the token "*" has to be shifted.

   (b) $p(o) < p(r_2)$.
   Now the precedence of the operator that occurs in the rule r_2 is higher than the precedence of the operator o. Therefore the correct action is to reduce with the rule r_2:

$$\mathit{action}(s, o) = \langle \texttt{reduce}, r_2 \rangle.$$

   To see that this makes sense we discuss the parsing of the expression

```
1*2+3
```

with the grammar given previously. Assume the string "1*2" has already been read and the next token that is processed is the token "+". Then the state of the parser is as follows:

$$\left\{\begin{array}{lcl} \mathit{expr} & \to & \mathit{expr} \bullet \text{“+”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{“−”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{“*”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{“/”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{“^”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \text{“*”}\, \mathit{expr} \bullet : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\} \end{array}\right\}.$$

When the parser now sees the operator "+", it has to reduce the string "1*2" using the rule

$$\mathit{expr} \to \mathit{expr} \text{“*”}\, \mathit{expr},$$

as it has to multiply the numbers 1 and 2.

(c) $p(o) = p(r_2)$ and the operator o is left associative.
Then we reduce the symbol stack with the rule r_2, we have

$$\mathit{action}(s, o) = \langle \texttt{reduce}, r_2 \rangle.$$

To convince yourself that this is the right thing to do, inspect what happens when the string

```
1-2-3
```

is parsed with the grammar discussed previously. Assume that the string "1−2" has already be read and the next token is the operator "−". Then the state of the parser is as follows:

$$\left\{\begin{array}{lcl} \mathit{expr} & \to & \mathit{expr} \bullet \text{“+”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{“−”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{“*”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{“/”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{“^”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \text{“−”}\, \mathit{expr} \bullet : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\} \end{array}\right\}.$$

If the next token is the operator "−", then the parser has to reduce the symbol stack using the rule $\mathit{expr} \to \mathit{expr}$ "−" expr as it has to subtract 2 from 1. If it would shift instead it would compute $1 - (2 - 3)$ instead of computing $(1 - 2) - 3$.

(d) $p(o) = p(r_2)$ and the operator o associates to the right.
In this case the operator o is shifted

$$\mathit{action}(s, o) = \langle \texttt{shift}, \mathit{goto}(s, o) \rangle.$$

In order to understand this case, parse the string

```
2^3^4
```

with the grammar rules

$$\mathit{expr} \to \mathit{expr}\ \text{^}\ \mathit{expr} \mid \textsc{Number}.$$

Consider the situation when the string "1^2" has already been read and the next token is the exponentiation operator "^". The state of the parser is then as follows:

$$\left\{\begin{array}{lcl} \mathit{expr} & \to & \mathit{expr} \bullet \text{“+”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{“−”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{“*”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{“/”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{“^”}\, \mathit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\}, \\ \mathit{expr} & \to & \mathit{expr} \text{“^”}\, \mathit{expr} \bullet : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}, \text{“^”}\} \end{array}\right\}.$$

Here the token "^" has to be shifted since we first have to compute the expression "3^4".

(e) $p(o) = p(r_2)$ and the operator o has been declared to be non-associative.
In this case we have a syntax error:

$$\mathit{action}(s, o) = \mathit{error}.$$

To understand this case, try to parse a string of the form

```
1 < 1 < 1
```

using the grammar rules

$$\mathit{expr} \rightarrow \mathit{expr}\ \text{“<”}\ \mathit{expr} \mid \mathit{expr}\ \text{“+”}\ \mathit{expr} \mid \text{NUMBER}.$$

Once the string “`1 < 1`” has been read and the next token is the operator “<” the parser recognizes that there is an error. Therefore, PLY will resolve this shift-reduce conflict by putting an error entry into the action table.

(f) $p(o)$ is undefined or $p(r_2)$ is undefined.
In this case there is a shift-reduce conflict and PLY prints a warning message when generating the parser. The conflict is then resolved in favour of shifting.

10.3 Resolving Shift-Reduce and Reduce-Reduce Conflicts

We start our discussion by categorizing conflicts with respect to their origin.

1. *Mehrdeutigkeits-Konflikte* sind Konflikte, die ihre Ursache in einer Mehrdeutigkeit der zu Grunde liegenden Grammatik haben. Solche Konflikte weisen damit auf ein tatsächliches Problem der Grammatik hin. Wir hatten ein Beispiel für solche Konflikte gesehen, als wir in Abbildung 10.1 versucht hatten, die Syntax arithmetischer Ausdrücke ohne die syntaktischen Kategorien *product* und *factor* zu beschreiben.

   Wir hatten damals bereits gesehen, dass wir das Problem durch die Einführung von Operator-Präzedenzen lösen können. Falls dies nicht möglich ist, dann bleibt nur das Umschreiben der Grammatik.

2. *Look-Ahead-Konflikte* sind Reduce-Reduce-Konflikte, bei denen die Grammatik zwar einerseits eindeutig ist, für die aber andererseits ein Look-Ahead von einem Token nicht ausreichend ist um den Konflikt zu lösen.

3. *Mysteriöse Konflikte* entstehen erst beim Übergang von den LR-Zuständen zu den LALR-Zuständen durch das Zusammenfassen von Zuständen mit dem gleichen Kern. Diese Konflikte treten also genau dann auf, wenn das Konzept einer LALR-Grammatik nicht ausreichend ist um die Syntax der zu parsenden Sprache zu beschreiben.

Wir betrachten die letzten beiden Fälle nun im Detail und zeigen Wege auf, wie die Konflikte gelöst werden können.

10.3.1 Look-Ahead-Konflikte

Ein Look-Ahead-Konflikt liegt dann vor, wenn die Grammatik zwar eindeutig ist, aber ein Look-Ahead von einem Token nicht ausreicht um zu entscheiden, mit welcher Regel reduziert werden soll. Abbildung 10.5 zeigt die Grammatik `Look-Ahead.ipynb`[1], die zwar eindeutig ist, aber nicht die LR(1)-Eigenschaft hat und damit erst recht keine LALR(1) Grammatik ist.

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

[1] Diese Grammatik habe ich im Netz auf der Seite von Pete Jinks unter der Adresse
http://www.cs.man.ac.uk/~pjj/cs212/ho/node19.html
gefunden.

```
1  a : b 'U' 'V'
2    | c 'U' 'W'
3  b : 'X'
4  c : 'X'
```

Figure 10.5: Eine eindeutige Grammatik ohne die LR(1)-Eigenschaft.

$a \to b$ "U" "V"

verwendet werden und folglich ist das "X" als ein b zu interpretieren. Ist das zweite Token hinter dem "X" hingegen ein "W", so ist die zu verwendende Regel

$a \to c$ "U" "W"

und folglich ist das "X" als c zu lesen.

```
1  a : b 'V'
2    | c 'W'
3  b : 'X' 'U'
4  c : 'X' 'U'
```

Figure 10.6: Eine zu Abbildung 10.5 äquivalente LR(1)-Grammatik.

Das Problem bei dieser Grammatik ist, dass sie versucht, abhängig vom Kontext ein "X" wahlweise als ein b oder als ein c zu interpretieren. Es ist offensichtlich, wie das Problem gelöst werden kann: Wenn der Kontext "U", der sowohl auf b als auch auf c folgt, mit in die Regeln für b und c aufgenommen wird, dann verschwindet der Konflikt, denn dann hat der Zustand, in dem früher der Konflikt auftrat, die Form

$\{b \to$ "X" "U" $\bullet$: "V", $c \to$ "X" "U" $\bullet$: "W" $\}$.

Hier entscheidet sich nun anhand des nächsten Tokens, mit welcher Regel wir in diesem Zustand reduzieren müssen: Ist das nächste Token ein "V", so reduzieren wir mit der Regel

$b \to$ "X" "U",

ist das nächste Token hingegen der Buchstabe "W", so nehmen wir stattdessen die Regel

$c \to$ "X" "U" $\bullet$: "W".

Abbildung 10.6 zeigt die entsprechend modifizierte Grammatik, die Sie unter

Formal-Languages/tree/master/Python/Chapter-10/Look-Ahead-Solved.ipynb

im Netz finden.

10.3.2 Mysterious Reduce-Reduce Conflicts

A conflict is called a mysterious reduce-reduce conflict if the conflict results from the merger of states that happens when we go from an LR parsing table to an LALR parsing table. The grammar in Figure 10.7 on page 137 is the same as the grammar shown in Figure 9.14 on page 127 in the previous chapter. Then we had seen that this grammar is an LR grammar, but not an LALR grammar. Let us see what happens if we use PLY to generate the states for this grammar.

When we run PLY to produce the parsing table, we get the states shown in Figure 10.8 on page 137. This Figure only shows two states, `state 6` and `state 9`. I have taken the liberty to annotate the extended marked rules occurring in these states with their follow sets. Taken by itself, none of these two states has a conflict since the follow sets of the respective rules are disjoint. However, it is obvious that these two states have the same core and

```
s : 'v' a 'y'
  | 'w' b 'y'
  | 'v' b 'z'
  | 'w' a 'z'

a : X

b : X
```

Figure 10.7: A grammar that generates a mysterious reduce-reduce conflict.

should have been merged. The resulting state would have the form

$$\{a \to X\bullet : \{\texttt{'y'}, \texttt{'z'}\}, b \to X\bullet : \{\texttt{'y'}, \texttt{'z'}\}\}$$

and obviously has a reduce-reduce conflict if the next token is either 'y' or 'z'.

```
state 6

    (5) a -> X . : 'y'
    (6) b -> X . : 'z'

    y               reduce using rule 5 (a -> X .)
    z               reduce using rule 6 (b -> X .)

state 9

    (6) b -> X . : 'y'
    (5) a -> X . : 'z'

    y               reduce using rule 6 (b -> X .)
    z               reduce using rule 5 (a -> X .)
```

Figure 10.8: A grammar that generates a mysterious reduce-reduce conflict.

Interestingly, PLY does not merge these states and is therefore able to generate a parse table without conflicts. On the other hand, PLY claims to generate LALR tables. Therefore, I have written an email to David Beazley asking whether this behaviour is a feature or a bug. He has classified this example as an "*interesting curiosity*".

Exercise 29: Use PLY to implement a *Python* parser that is able to evaluate formulas from propositional logic. The language should support the operators "<->" (equivalence), "->" (implication), "|" (disjunction), "&" (conjunction), and "!" (negation). The operator "!" should bind stronger than the operator "&", which in turn binds stronger than the operator "|". The operators "|" and "&" are both left associative. The operator "->" associates to the right and binds weaker than the operator "|". The operator "<->" in not associative and is the weakest of all operators in terms of binding strength. Furthermore, the language should support parenthesis, variables, and the assignment operator ":=". ◇

Chapter 11

Assembler

A compiler translates programs written in a high level language like `C` or *Java* into some low level representation. This low level representation can be either machine code or some form of assembler code. For the programming language *Java*, the command `javac` compiles a program written in *Java* into *Java* byte code. This byte code is then executed using the *Java* virtual machine (JVM).

The compiler that we are going to develop in the next chapter generates a particular form of assembler code know as JVM assembler code. This assembler code can be translated directly into *Java* byte code, which is also the byte code generated by the program `javac`. The program for translating JVM assembler into bytecode is called *Jasmin*. You can download *Jasmin* at

http://sourceforge.net/projects/jasmin.

Jasmin takes an byte code produced by *Jasmin* can be executed using the command `java` just like any other "`.class`"-file. This chapter will discuss the syntax and semantics of *Jasmin* assembler code. *Jasmin* is discussed in more detail in the book *Java Virtual Machine* [MD97].

By the way, *Java* isn't the only programming language that is translated into *Java* Byte code. Here is a list of some of the better knwon programming languages that use the JVM architecture.

1. Kotlin is developed by JetBrains. It is is fully interoperable with *Java* and compiles to JVM bytecode. It is widely used in Android development as a replacement for *Java*.

   As of the 28^{th} of November 2024, *Kotlin* occupies the 20^{th} place in the Tiobe index.

2. Scala blends functional and object-oriented programming paradigms. Scala compiles to JVM bytecode and is often used for concurrent and distributed applications.

   Currently, *Scala* occupies the 30^{th} place in the Tiobe index.

3. Groovy is a dynamic scripting language that is easy to learn for *Java* programmers because its syntax is close to *Java*. It is often used in scripting, configuration, and in build systems like Gradle.

4. Clojure is a modern Lisp dialect designed for functional programming. It compiles to JVM bytecode and supports concurrency.

5. JRuby is a JVM-based version of the Ruby programming language. It is fully compatible with *Ruby* but additionally support the use *Java* libraries.

6. Jython is a JVM-based version of Python version 2.7.

7. Frege is a version of the pure functional language Haskell that compiles to JVM bytecode.

The main reason these languages compile to the JVM is to take advantage of *Java*'s rich library ecosystem. Another reason is the fact that compiling a language to the JVM makes this language available for all those hardware platforms that support *Java*.

11.1 Introduction into Jasmin Assembler

```
1  .class public Hello
2  .super java/lang/Object
3
4  .method public <init>()V
5      aload 0
6      invokenonvirtual java/lang/Object/<init>()V
7      return
8  .end method
9
10 .method public static main([Ljava/lang/String;)V
11     .limit locals 1
12     .limit stack  2
13     getstatic     java/lang/System/out Ljava/io/PrintStream;
14     ldc           "Hello World!"
15     invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
16     return
17 .end method
```

Figure 11.1: An assembler program to print "`Hello World!`".

To get used to the syntax of *Jasmin* assembler, we start with a small program that prints the string

"`Hello World!`"

on the standard output stream. Figure 11.1 on page 139 shows the program `Hello.jas`. We discuss this program line by line.

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. The lines 4 – 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. The lines 10 – 17 defines the method `main` that does the actual work.
   (a) Line 10 uses the directive "`.method`" to declare the name of the method and its signature. The string

   `main([Ljava/lang/String;)V`

   specifies the signature:
   i. The string "`main`" is the name of the method that is defined.
   ii. The character "`[`" specifies that the first argument is an array.
   iii. The character "`L`" specifies that this array consists of objects.
   iv. The string "`java/lang/String;`" specifies that these objects are objects of the class "`java.lang.String`".

v. Finally, the character "V" specifies that the return type of the method `main` is "`void`".

(b) Line 11 uses the directive "`.limit locals`" to specify the number of local variables used by the method `main`. In this case, there is just one local variable. This variable corresponds to the parameter of the method main. The assembler file shown in Figure 11.1 on page 139 corresponds to the *Java* code shown in Figure 11.2 below. The method `main` has one local variable, which is the parameter `args`. The information on the number of local variables is needed by the *Java* virtual machine in order to allocate memory for these variables on the stack.

```
1  public class Hello {
2      public static void main(String[] args) {
3          System.out.println("Hello World!");
4      }
5  }
```

Figure 11.2: Printing `Hello world` in *Java*.

(c) For the purpose of the following discussion, we basically assume that there exist two types of processors: Those that store the objects they work upon in registers and those that store these objects on a stack residing in main memory. The Java virtual machine is of the second type. Hence, *Jasmin* assembler programs do not refer to registers but rather refer to this stack[1]. Line 12 uses the directive

"`.limit stack`"

to specify the 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 the command `getstatic` in line 13 pushes the object

`java.lang.System.out`

onto the stack. This is an object of class "`java.io.PrintStream`". Then, the command `ldc` in line 14 pushes a reference to the string "`Hello World`" onto the stack. The other instructions do not push anything onto the stack.

(d) Line 15 calls the method `println`, which is a method of the class "`java.io.PrintStream`". It also specifies that `println` takes one argument of type `java.lang.String` and returns nothing.

In reality, `println` needs a second argument. This argument is the object `java.lang.System.out` that we had previously pushed on the stack using the method `getstatic`.

(e) Line 16 returns from the method `main`.

(f) In line 17 the directive "`.end`" marks the end of the code corresponding to the method `main`.

Before we proceed, let us assume that we are working on a Unix operating system and that there is an executable file called `jasmin` somewhere in our path that contains the following code:

```
#!/bin/bash
java -jar /usr/local/lib/jasmin.jar $@
```

Of course, for this to work the directory `/usr/local/lib/` has to contain the file "`jasmin.jar`". If we were working on a Windows operating system, we would have a file called `jasmin.bat` somewhere in our `PATH`. This file would contain the following line:

```
java -jar %USERPROFILE%/Dropbox/Software/jasmin-2.4/jasmin.jar %*
```

Of course, for this to work the directory `%USERPROFILE%/Dropbox/Software/jasmin-2.4` has to contain the file "`jasmin.jar`".

[1] In reality, all real processors make use of registers. However, it is possible to simulate a stack machine using a real processor and that is what is done in the Java virtual machine.

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. Every command takes its arguments from the top of the stack and replaces these arguments with the result of the operation performed by the command. For example, if we want to add two values, then we first have to push both values onto the stack. Next, performing the add operation will pop these values from the stack and then push their sum onto the stack.

11.2 Assembler Instructions

We proceed to discuss some of the JVM instructions. Since there are more than 160 JVM instructions, we can only discuss a subset of all instructions. We restrict ourselves to those instructions that deal with integers: For example, there is an instruction called `iadd` that adds two 32 bit integers. There are also instructions like `fadd` that adds two floating point numbers and `dadd` that adds two double precision floating point numbers but, since our time is limited, we won't discuss these instructions. Before we are able to discuss the different instructions we have to discuss how the main memory is organized in the JVM: In the JVM, the memory is split into four parts:

1. The program memory contains the program code as a sequence of bytes.
2. The operands of the different machine instructions are put onto the stack. Furthermore, the stack contains the arguments and the local variables of a procedure. However, in the context of the JVM the procedures are called *methods* instead of procedures.

   The register `SP` points to the top of the stack. If a method is called, the arguments of the method are placed on the stack. The register `LV` (local variables) points to the first argument of the current method. On top of the arguments, the local variables of the method are put on the stack. Both the arguments and the local variables can be accessed via the register `LV` by specifying their offset from the first argument. We will discuss the register `LV` in more detail when we discuss the invocation of methods.
3. The heap is used for dynamically allocated memory. Newly created objects are located in the heap.
4. The constant pool contains the definitions of constants and also the addresses of methods in program memory.

In the following, we will be mostly concerned with the stack. We proceed to discuss some of the assembler instructions.

Arithmetical and Logical Instructions

1. The instruction "`iadd`" adds those values that are on top of the stack and replaces these values by their sum. Figure 11.3 on page 142 show how this command works. The left part of the figure shows the stack as it is before the command `iadd` is executed, while the right part of the figure depicts the situation after the execution of this command. Note that after the command is executed, the stack pointer points to the position where formerly the first argument had been stored.

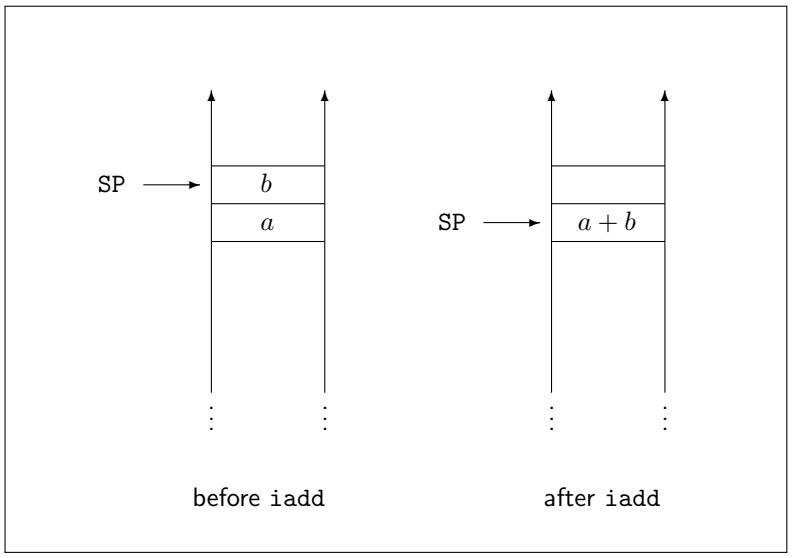

Figure 11.3: The effect of `iadd`.

2. The instruction "`isub`" subtracts the integer value on top of the stack from the value that is found on the position next to the top of the stack. Figure 11.4 on page 142 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.

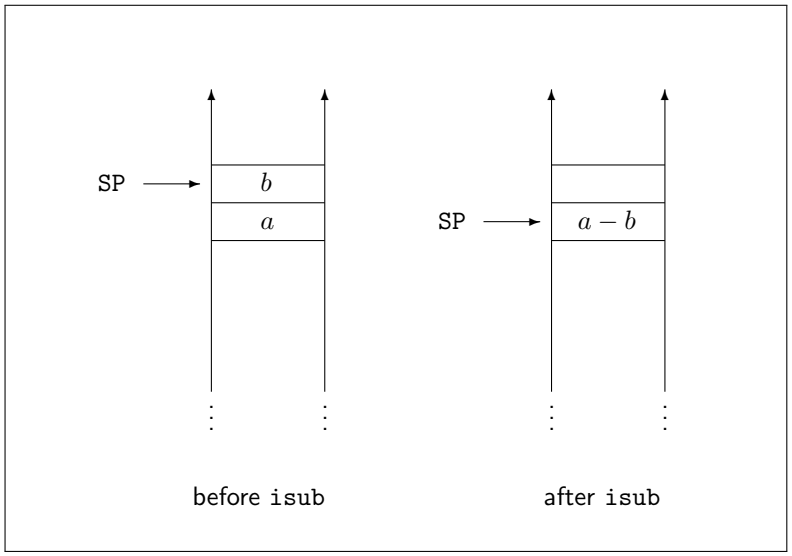

Figure 11.4: The effect of `isub`.

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.

   Note: Integer division in *Java* is different from integer division in *Python*. In *Python*, integer division is always

rounded down (i.e. rounded to negative infinity), while integer division in *Java* rounds towards 0. While there is no difference for natural numbers, we have $-5 / 2 = -2$ in *Java*, while we have $-5 // 2 = -3$ in *Python*.[2].

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.

   Again, it is important to note that the *Java* implementation of $a \,\%\, b$ differs from the implementation in *Java*. For example, in *Java* we have $-5 \,\%\, 3 = -2$, while we have $-5 \,\%\, 3 = 1$ in *Python*.

6. The instruction "`iand`" computes the bitwise "and" of the values on top of the stack.

7. The instruction "`ior`" computes the bitwise "or" of the integer values that are on top of the stack.

8. The instruction "`ixor`" computes the bitwise "*exclusive or*" of the integer values that are on top of the stack.

Shift Instructions

1. The instruction "`ishl`" shifts the value a to the left by b`[4:0]` bits. Here, a and b are assumed to be the two values on top of the stack: b is the value on top of the stack and a is the value below b. b`[4:0]` denotes the natural number that results from the 5 lowest bits of b. Figure 11.5 on page 143 pictures this command.

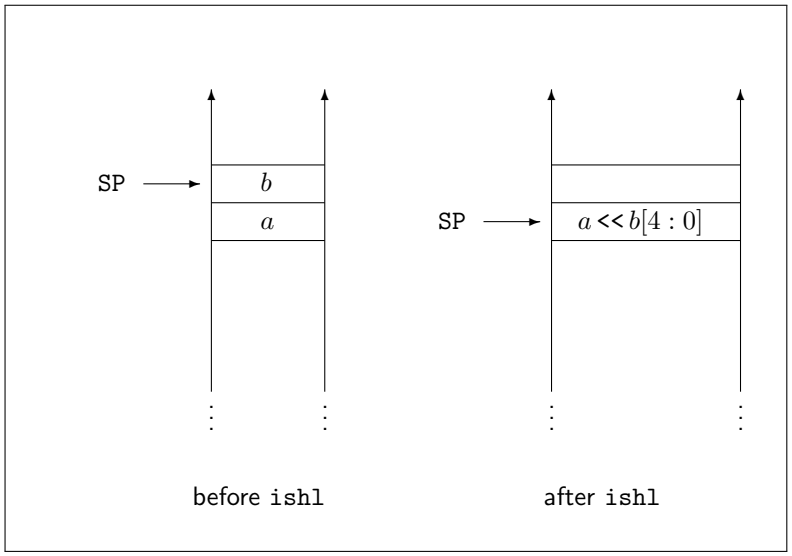

Figure 11.5: The effect of `ishl`.

2. The instruction "`ishr`" shifts the value a to the right by b`[4:0]` bits. Here, a and b are assumed to be the two values on top of the stack: b is the value on top of the stack and a is the value below b. b`[4:0]` denotes the natural number that results from the 5 lowest bits of b. Note that this instruction performs an *arithmetic shift*, i.e. the sign bit is preserved.

11.2.1 Instructions to Manipulate the Stack

1. The instruction "`dup`" duplicates the value that is on top of the stack. Figure 11.6 on page 144 pictures this command.

2. The instruction "`pop`" removes the value that is on top of the stack. Figure 11.7 on page 144 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 the old value.

[2] In mathematics, integer division, which is also known as Euclidean division is defined in the same was as it is implemented in *Python*. The different implementation of this operator in *Java* is mostly a reflection of the poor educational standards of the United States of America.

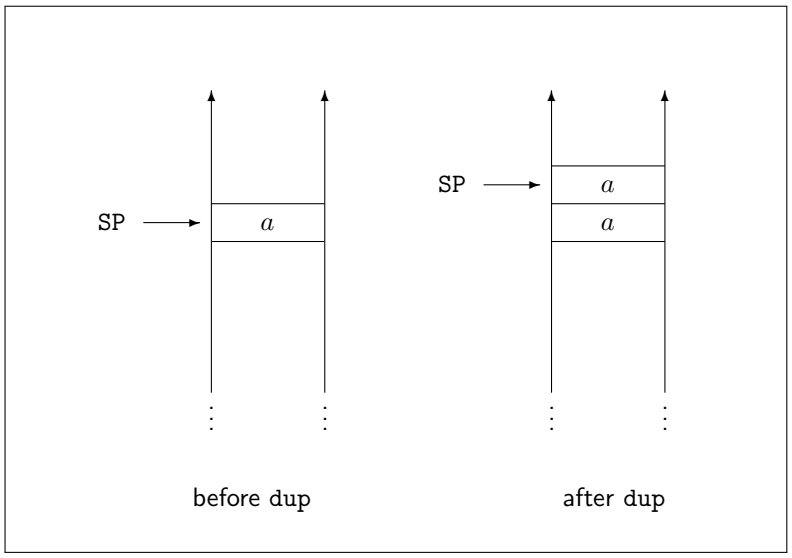

Figure 11.6: The effect of `dup`.

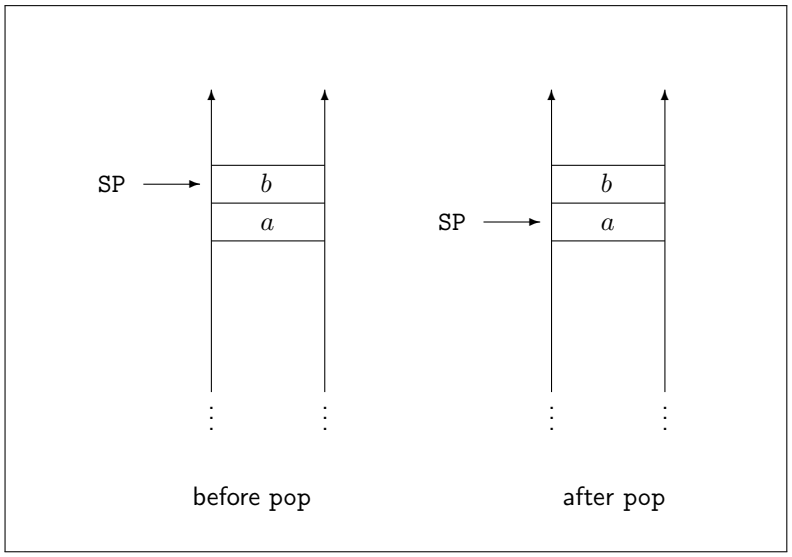

Figure 11.7: The effect of `pop`.

3. The instruction "`nop`" does nothing. The name is short for "`no operation`".

4. The instruction "`bipush` b" pushes the byte b that is given as argument onto the stack. The argument b is constricted to a byte in order to limit the number of bytes that are needed to represent this instruction in memory. Figure 11.8 on page 145 pictures this command.

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.

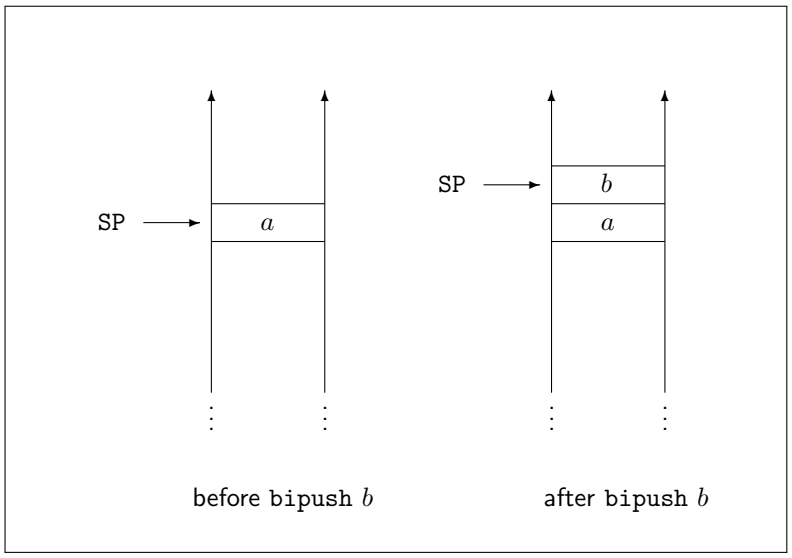

Figure 11.8: The effect of `bipush` b.

6. The instruction "`iload` v" reads the local variable with index v and pushes it on top of the stack. Figure 11.9 on page 145 pictures this command. Note that `LV` denotes the register pointing to the beginning of the local variables of a method.

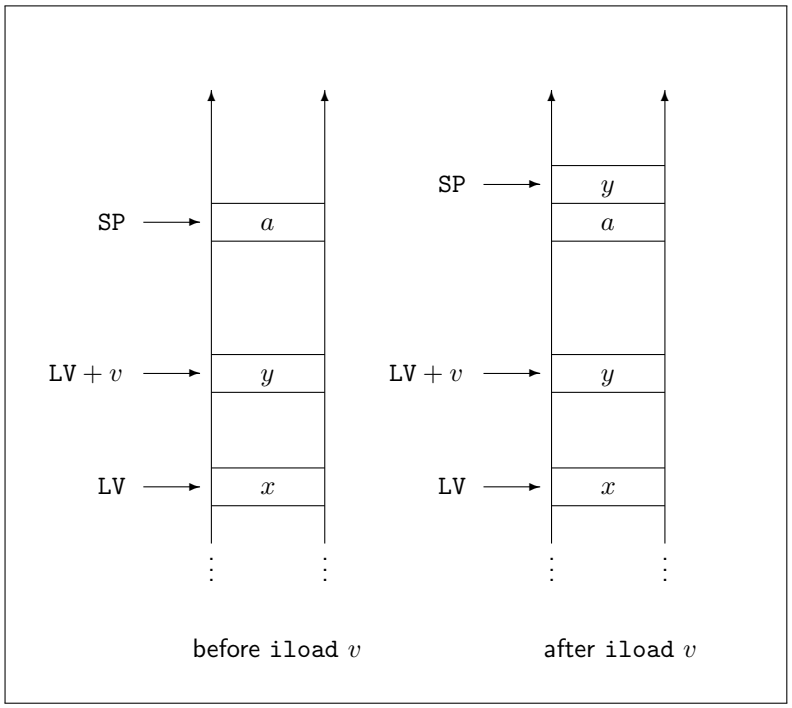

Figure 11.9: The effect of `iload` v.

7. The instruction "`istore` v" removes the value which is on top of the stack and stores this value at the location for the local variable with number v. Hence, v is interpreted as an index into the local variable table of the method. Figure 11.10 on page 146 pictures this command.

8. The instruction "`ldc` c" pushes the constant c onto the stack. This constant can be an integer, a single

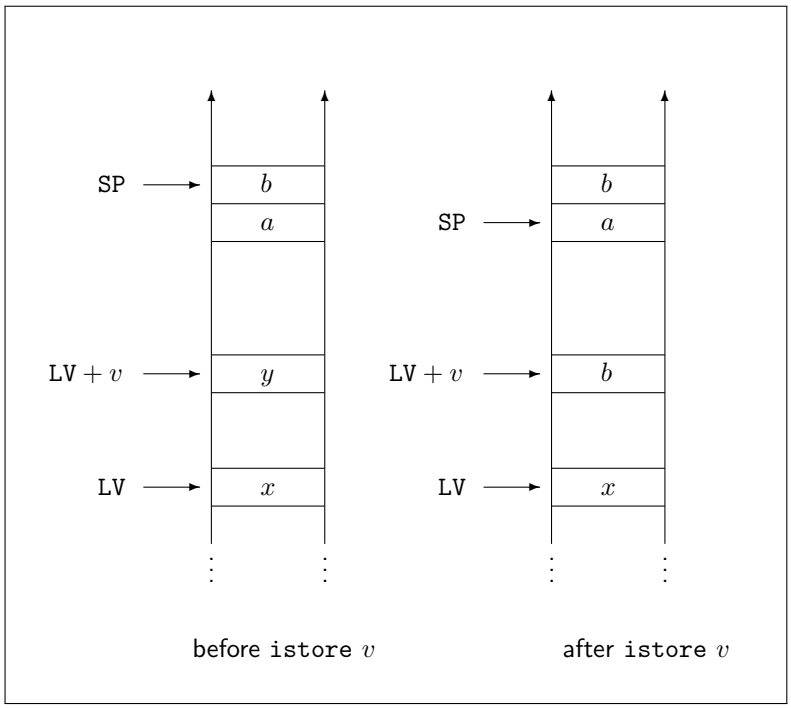

Figure 11.10: The effect of `istore` v.

precision floating point number, or a (pointer to) a string. If c is a string, this string is actually stored in the so called *constant pool* and in this case the command "`ldc` c" will only push a pointer to the string onto the stack.

Branching Commands

In this subsection we discuss those commands that change the control flow.

1. The instruction "`goto` l" jumps to the label l. Here the label l is a label name that has to be declared inside the method containing this `goto` command. A label with name *target* is declared using the syntax

   target:

   The next section presents an example assembler program that demonstrates this command.

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 n onto the stack. After the method returns, this integer is replaced with the return value $\texttt{sum}(n)$.

10. The instruction "`ireturn`" returns from the method that is currently invoked. This method also returns a value to the calling procedure. In order for `ireturn` to be able to return a value v, this value v has to be pushed onto the stack before the command `ireturn` is executed.

    In general, if a method taking n arguments $a_1, \cdots, a_n$ is to be called, then first the arguments $a_1, \cdots, a_n$ have to be pushed onto the stack. When the method m is done and has computed its result r, the arguments $a_1, \cdots, a_n$ will have been replaced with the single value r.

11. The instruction "`return`" returns from the method that is currently invoked. However, in contrast to the command `ireturn`, this command is used if the method that has been invoked does not return a result.

11.3 An Example Program

Figure 11.11 shows the `C` program `sum.c` that computes the sum

$$\sum_{i=1}^{6^2} i.$$

The function $\texttt{sum}(n)$ computes the sum $\sum_{i=1}^{n} i$ and the function `main` calls this function with the argument $6 \cdot 6$. Figure 11.12 on page 149 shows how this program can be translated into the assembler program `Sum.jas`. We discuss the implementation of this program next.

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`. This method takes an argument which is an array of strings. It does not return a value.

```
1  #import "stdio.h"
2
3  int sum(int n) {
4      int s;
5      s = 0;
6      while (n != 0) {
7          s = s + n;
8          n = n - 1;
9      };
10     return s;
11 }
12 int main() {
13     printf("%d\n", sum(6*6));
14     return 0;
15 }
```

Figure 11.11: A C function to compute $\sum_{i=1}^{36} i$.

5. Line 11 specifies that there is just one local variable.
6. Line 12 specifies that the stack will contain at most 3 temporary values.
7. Line 13 pushes the object `java.lang.out` onto the stack. We need this object later in order to invoke `println`.
8. Line 14 pushes the number 6 onto the stack.
9. Line 15 duplicates the value 6. Therefore, after line 15 is executed, the stack contains three elements: The object `java.lang.out`, the number 6, and again the number 6.
10. Line 16 multiplies the two values on top of the stack and replaces them with their product, which happens to be 36.
11. Line 17 calls the method `sum` defined below. After this call has finished, the number 36 on top of the stack is replaced with the value $\texttt{sum}(36)$.
12. Line 18 prints the value that is on top of the stack. This is possible, because the object `java.lang.out` is directly below the value $\texttt{sum}(36)$.
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.
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 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 in line 32 and the result is written to the local variable s in line 33. The combined effect of these instructions is therefore to perform the assignment

    ```
    s = s + n;
    ```

```
1  .class public Sum
2  .super java/lang/Object
3
4  .method public <init>()V
5      aload 0
6      invokenonvirtual java/lang/Object/<init>()V
7      return
8  .end method
9
10 .method public static main([Ljava/lang/String;)V
11     .limit locals 1
12     .limit stack  3
13     getstatic     java/lang/System/out Ljava/io/PrintStream;
14     ldc           6
15     dup
16     imul
17     invokestatic  Sum/sum(I)I
18     invokevirtual java/io/PrintStream/println(I)V
19     return
20 .end method
21
22 .method public static sum(I)I
23     .limit locals 2
24     .limit stack  2
25     ldc    0
26     istore 1                    ; s = 0;
27 loop:
28     iload  0                    ; n
29     ifeq   finish               ; if (n == 0) goto finish;
30     iload  1                    ; s
31     iload  0                    ; n
32     iadd
33     istore  1                   ; s = s + n;
34     iload   0                   ; n
35     ldc     1
36     isub
37     istore  0                   ; n = n - 1;
38     goto    loop
39 finish:
40     iload   1
41     ireturn                     ; return s;
42 .end method
```

Figure 11.12: An assembler program to compute the sum $\sum_{i=1}^{36} i$.

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 loop and test whether n is zero.

20. The declaration in line 42 terminates the definition of the method `sum`.

Exercise 30:

(a) Implement an assembler program that computes the function `factorial`, which takes a natural number n as input and returns $n!$. Test your program by printing $n!$ for $n = 1, \cdots, 10$.

(b) Implement the function `factorial` in a recursive manner.

11.4 Disassembler*

Sometimes it is useful to transform a file consisting of *Java* byte code back into something that looks like assembler code. After all, a *Java* class file is a binary file and can therefore only be viewed via commands like `od` that produce an *octal* or *hexadecimal dump* of the given file. The command `javap` is a *disassembler*, i.e. it takes a *Java* byte code file and transforms it in something that looks similar to *Jasmin* assembler. For example, Figure 11.13 shows the class *Java* class `Sum` to compute the sum

$$\sum_{i=1}^{36} i$$

written in *Java*. If this program is stored in a file called "`Sum.java`", we can compile it via the following command:

```
javac Sum.java
```

This will produce a class file with the name "`Sum.class`" containing the byte code.

```
1  public class Sum {
2      public static void main(String[] args) {
3          System.out.println(sum(6 * 6));
4      }
5
6      static int sum(int n) {
7          int s = 0;
8          for (int i = 0; i <= n; ++i) {
9              s += i;
10         }
11         return s;
12     }
13 }
```

Figure 11.13: A *Java* program computing $\sum_{i=0}^{6^2} i$.

Next, in order to *decompile* the "`.class`" file, we run the command

```
javap -c Sum.class
```

The output of this command is shown in Figure 11.14. We see that the syntax used by `javap` differs a bit from the syntax used by *Jasmin*. It is rather unfortunate that the company developing *Java* has decided not to make their assembler public. Still, we can see that the output produced by `javap` is quite similar to the input accepted by `jasmin`.

```
Compiled from "Sum.java"
public class Sum {
  public Sum();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: bipush        36
       5: invokestatic  #3                  // Method sum:(I)I
       8: invokevirtual #4                  // Method java/io/PrintStream.println:(I)V
      11: return

  static int sum(int);
    Code:
       0: iconst_0
       1: istore_1
       2: iconst_0
       3: istore_2
       4: iload_2
       5: iload_0
       6: if_icmpgt     19
       9: iload_1
      10: iload_2
      11: iadd
      12: istore_1
      13: iinc          2, 1
      16: goto          4
      19: iload_1
      20: ireturn
}
```

Figure 11.14: The output of "`javap -c Sum.class`".

Chapter 12

Development of a Simple Compiler

In this chapter, we construct a compiler that translates a fragment of the C language into *Java* assembler. The fragment of the C language translated by the compiler is referred to as *Integer*-C, because only the data type `int` is available. Although it would be feasible to support additional data types, the extra effort would no longer be proportionate to the educational benefit of the example.

A compiler essentially consists of the following components:

1. The scanner reads the file to be translated and breaks it down into a sequence of tokens. We will develop our scanner using the PLY tool.
2. The parser reads the sequence of tokens and produces an abstract syntax tree as a result. We will use PLY for the creation of the parser.
3. The type-checker checks the abstract syntax tree for type errors. Since the language *Integer*-C contains only a single data type, this phase is not needed for our compiler.
4. The code generator then translates the parse tree into a sequence of assembler commands that conform to the syntax of *Jasmin*.
5. This can be followed by an *optimization phase*. Due to time constraints we cannot cover this topic.
6. The resulting assembler program can be translated into Java bytecode using *Jasmin*.
7. The Java bytecode is then executed by the JVM, i.e. the *Java* virtual machine.

In compilers whose target code is a RISC assembler program, a code similar to JVM code is usually generated first. Such a code is referred to as *intermediate code*. It then remains the task of a so-called *backend* to generate an assembler program for a given processor architecture. The most challenging task here is to find a register allocation for the variables, such that as many variables as possible can be held in registers. Due to time constraints, we cannot address the topic of register allocation in this lecture.

12.1 The Programming Language Integer-C

We now introduce the language *Integer*-C, which our compiler is intended to translate. In this context, we also refer to it as the *source language* of our compiler. Figure 12.1 on page 153 shows the grammar of the source language in Extended Backus-Naur Form (EBNF). The grammar for *Integer*-C uses the following terminals:

1. ID is an abbreviation for Identifier and represents a sequence of digits, letters, and underscores that begins with a letter. An ID designates either a variable or the name of a function.
2. NUMBER stands for a sequence of digits interpreted as a decimal number.
3. In addition, there are a number of operators such as `"+"`, `"-"`, etc., as well as keywords like `"if"`, `"else"`, and `"while"`.

$$
\begin{array}{rcl}
\textit{program} & \rightarrow & \textit{function}+ \\
\textit{function} & \rightarrow & \text{“}\texttt{int}\text{”}\ \textsc{Id}\ \text{“(”}\ \textit{paramList}\ \text{“)”}\ \ \text{“\{”}\ \textit{decl}+\ \textit{stmnt}+\ \text{“\}”} \\
\textit{paramList} & \rightarrow & (\text{“}\texttt{int}\text{”}\ \textsc{Id}\ (\text{“,”}\ \ \text{“}\texttt{int}\text{”}\ \textsc{Id})*)? \\
\textit{decl} & \rightarrow & \text{“}\texttt{int}\text{”}\ \textsc{Id}\ \text{“;”} \\
\textit{stmnt} & \rightarrow & \text{“\{”}\ \textit{stmnt}*\ \text{“\}”} \\
 & | & \textsc{Id}\ \text{“=”}\ \textit{expr}\ \text{“;”} \\
 & | & \text{“}\texttt{if}\text{”}\ \ \text{“(”}\ \textit{boolExpr}\ \text{“)”}\ \textit{stmnt} \\
 & | & \text{“}\texttt{if}\text{”}\ \ \text{“(”}\ \textit{boolExpr}\ \text{“)”}\ \textit{stmnt}\ \text{“}\texttt{else}\text{”}\ \textit{stmnt} \\
 & | & \text{“}\texttt{while}\text{”}\ \ \text{“(”}\ \textit{boolExpr}\ \text{“)”}\ \textit{stmnt} \\
 & | & \text{“}\texttt{return}\text{”}\ \textit{expr}\ \text{“;”} \\
 & | & \textit{expr}\ \text{“;”} \\
\textit{boolExpr} & \rightarrow & \textit{expr}\ (\text{“==”} \mid \text{“}\texttt{!=}\text{”} \mid \text{“<=”} \mid \text{“>=”} \mid \text{“<”} \mid \text{“>”})\ \textit{expr} \\
 & | & \text{“}\texttt{!}\text{”}\ \textit{boolExpr} \\
 & | & \textit{boolExpr}\ (\text{“}\&\&\text{”} \mid \text{“}||\text{”})\ \textit{boolExpr} \\
\textit{expr} & \rightarrow & \textit{expr}\ (\text{“+”} \mid \text{“−”} \mid \text{“*”} \mid \text{“/”} \mid \text{“\%”})\ \textit{expr} \\
 & | & \text{“(”}\ \textit{expr}\ \text{“)”} \\
 & | & \textsc{Number} \\
 & | & \textsc{Id}\ (\text{“(”}(\textit{expr}\ (\text{“,”}\ \textit{expr})*)?\ \text{“)”})?
\end{array}
$$

Figure 12.1: An EBNF grammar for *Integer*-C

According to the grammar provided above, a program is a list of functions. A function consists of the declaration of its signature, followed by a list of declarations (*decl*) and commands (*stmt*) enclosed in curly braces. The grammar shown in Figure 12.1 is ambiguous:

1. The grammar has the *Dangling-Else Problem*. We will solve this problem via operator precedences.
2. For the operators used in arithmetic and Boolean expressions, precedences must be set to resolve ambiguities in the interpretation of these expressions.

Figure 12.2 shows a simple INTEGER-C program. The function *sum*(n) calculates the sum

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

and the function *main*() calls the function `sum` with the argument $6 \cdot 6$. The function `println` used in the program outputs its argument followed by a newline. We assume that this function is predefined.

12.2 Developing the Scanner and the Parser

We construct both the scanner and the parser using *Ply*. Figure 12.3 shows the scanner. The scanner recognizes the operators, keywords, identifiers, and numbers. As the scanner is mostly similar to those scanners that we have already seen before, there are only a few points worth mentioning.

1. If a token does not have to be manipulated by the scanner, then it can be defined by a simple equation of the form

```
1  int sum(int n) {
2      int s;
3      s = 0;
4      while (n != 0) {
5          s = s + n;
6          n = n - 1;
7      };
8      return s;
9  }
10 int main() {
11     int n;
12     n = 6 * 6;
13     println(sum(n));
14 }
```

Figure 12.2: A simple INTEGER-C program to compute the sum $\sum_{i=1}^{6 \cdot 6} i$.

```
t_name = regexp
```

where *name* is the name of the token and *regexp* is the regular expression defining the token. We have used this shortcut to define most of the tokens recognized by our scanner.

2. While we do not need to declare tokens for those operators that consist of just one token, we have to declare those tokens that consist of two or more characters. For example, the operator "==" is represented by the token `'EQ'` and is defined in line 9.

3. Our programming language supports single line comments that start with the string "//" and extend to the end of the line. These comments are implemented via the token `COMMENT` that is defined in line 16–18. Note that the function `t_COMMENT` does not return a value. Therefore, these comments are simply discarded. This is also the reason that we have to use a function to define this token.

4. The most interesting aspect is the implementation of keywords like "`while`" or "`return`". Note that we have defined separate tokens for all of the keywords but we have not defined any of these tokens. For example, we have not defined `t_return`. The reason is that, syntactically, all of the keywords are identifiers and are recognized by the regular expression

   ```
   r'[a-zA-Z][a-zA-Z0-9_]*'
   ```

   that is used for recognizing identifiers in line 28. However, the function `t_ID` that recognizes identifiers does not immediately return the token `'ID'` when it has scanned the name of an identifier. Rather, it first checks whether this name is a predefined keyword. The predefined keywords are the key of the dictionary `Keywords` that is defined in line 38-43. The values of the keywords in this dictionary are the token types. Therefore, the function `t_ID` sets the token type of the token that is returned to the value stored in the dictionary `Keywords`. In case a name is not defined in this dictionary, the token type is set to `'ID'`.

5. The scanner implements the function `t_newline` in line 6. This function is needed to update the attribute `lineno` of the scanner. This attribute stores the line number and is needed for error messages.

Now we are ready to present the specification of our parser. For reasons of space, we have split the grammar into six parts. We start with Figure 12.4 on page 156.

1. Line 3 declares the start symbol `program`.

2. Line 5–12 declare the operator precedences.

```
import ply.lex as lex

tokens = [ 'NUMBER', 'ID', 'EQ', 'NE', 'LE', 'GE', 'AND', 'OR',
           'INT', 'IF', 'ELSE', 'WHILE', 'RETURN'
         ]

t_NUMBER = r'0|[1-9][0-9]'

t_EQ  = r'=='
t_NE  = r'!='
t_LE  = r'<='
t_GE  = r'>='
t_AND = r'&&'
t_OR  = r'\|\|'

def t_COMMENT(t):
    r'//[^\n]*'
    pass

Keywords = { 'int'   : 'INT',
             'if'    : 'IF',
             'else'  : 'ELSE',
             'while' : 'WHILE',
             'return': 'RETURN'
           }

def t_ID(t):
    r'[a-zA-Z][a-zA-Z0-9_]*'
    t.type = Keywords.get(t.value, 'ID')
    return t

literals = ['+','-','*','/','%','(',')','{','}',';','=','<','>','!',',']

t_ignore  = ' \t\r'

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

def t_error(t):
    print(f"Illegal character '{t.value[0]}' in line {t.lineno}.")
    t.lexer.skip(1)

__file__ = 'main'

lexer = lex.lex()
```

Figure 12.3: The scanner for *Integer-C*.

(a) The operator "||" that represents logical or has the lowest precedence and associates to the left.

(b) The operator "&&" that represents logical and binds stronger than "||" but not as strong as the negation

```
1  import ply.yacc as yacc
2
3  start = 'program'
4
5  precedence = (
6      ('left', 'OR'),
7      ('left', 'AND'),
8      ('right', '!'),
9      ('nonassoc', 'EQ', 'NE', 'LE', 'GE', '<', '>'),
10     ('left', '+', '-'),
11     ('left', '*', '/', '%')
12 )
13
14 def p_program_one(p):
15     "program : function"
16     p[0] = ('program', p[1])
17
18 def p_program_more(p):
19     "program : function program"
20     p[0] = ('program', p[1]) + p[2][1:]
21
22 def p_function(p):
23     "function : INT ID '(' param_list ')' '{' decl_list stmnt_list '}'"
24     p[0] = ('fct', p[2], p[4], p[7], p[8])
```

Figure 12.4: Grammar for Integer-C, part 1.

operator "!".

(c) The negation operator "!" is right associative because we want to interpret an expression of the form !!a as !(!a).

(d) The comparison operators "==", "!=", "<=", ">=", "<", and ">" are non-associative, since our programming languages disallows the chaining of comparison operators, i.e. an expression of the form $x < y < z$ is syntactically invalid.

(e) The arithmetical operators "+" and "-" have a lower precedence than the arithmetical operators "*", "/", and "%".

3. A `program` is a non-empty list of `function` definitions. Therefore, it is either a single function definition (line 14–16) or it is a function definition followed by more function definitions (line 18–20).

   The purpose of the parser is to turn the given program into an abstract syntax tree. This abstract syntax tree is represented as a nested tuple. A program consisting of the functions $f_1, \cdots, f_n$ is represented as the nested tuple.

   $$(\text{"program"}, f_1, \cdots, f_n).$$

   If in line 19 a `program` consisting of a `function` and another `program` is parsed, the expression `p[2]` in line 20 refers to the abstract syntax tree representing the `program` that follows the `function`. The expression `p[2][1:]` discards the keyword "program" that is at the start of this nested tuple. Hence, `p[2][1:]` is just a tuple of the functions following the first `function`. Therefore, the assignment to `p[0]` creates a nested tuple that starts with the keyword "program" followed by all the functions defined in this program.

4. A `function` definition starts with the keyword "int" that is followed by the name of the function, an opening parenthesis, a list of the parameters of the function, a closing parenthesis, an opening brace, a list of declarations, a list of statements, and finally a closing brace.

The initial version of the programming language C, as detailed in the book "*The C Programming Language*" [KR78], required that all variable declarations precede any statements, prohibiting the mixing of declarations and statements. While this limitation was lifted in subsequent versions of the language, our implementation still enforces this restriction.

```
25  def p_param_list_empty(p):
26      "param_list :"
27      p[0] = ('.', )
28
29  def p_param_list_one(p):
30      "param_list : INT ID"
31      p[0] = ('.', p[2])
32
33  def p_param_list_more(p):
34      "param_list : INT ID ',' ne_param_list"
35      p[0] = ('.', p[2]) + p[4][1:]
36
37  def p_ne_param_list_one(p):
38      "ne_param_list : INT ID"
39      p[0] = ('.', p[2])
40
41  def p_ne_param_list_more(p):
42      "ne_param_list : INT ID ',' ne_param_list"
43      p[0] = ('.', p[2]) + p[4][1:]
44
45  def p_decl_list_one(p):
46      "decl_list :"
47      p[0] = ('.',)
48
49  def p_decl_list_more(p):
50      "decl_list : INT ID ';' decl_list"
51      p[0] = ('.', p[2]) + p[4][1:]
52
53  def p_stmnt_list_one(p):
54      "stmnt_list :"
55      p[0] = ('.',)
56
57  def p_stmnt_list_more(p):
58      "stmnt_list : stmnt stmnt_list"
59      p[0] = ('.', p[1]) + p[2][1:]
```

Figure 12.5: Grammar for Integer-C, part 2.

Figure 12.5 shows the definition of parameter lists, declaration lists, and statement lists.

1. The definition of a list of parameters is quite involved, because parameter lists may be empty and, furthermore, different parameters have to be separated by ",". Therefore, we have to introduce the additional syntactical variable `ne_param_list`, which represents a non-empty parameter list. The grammar rules for `param_list` are as follows:

```
param_list
     : /* epsilon */
     | 'int' ID
     | 'int' ID ',' ne_param_list
     ;
ne_param_list
     : 'int' ID
     | 'int' ID ',' ne_param_list
     ;
```

We use the "." as a key to represent lists of any kind as nested tuples.

2. The grammar rules for a list of declarations are as follows:

```
decl_list
    : /* epsilon */
    | 'int' ID ';' decl_list
    ;
```

Observe that with this definition a list of declarations may be empty. The grammar rules for a list of declarations are simpler than the grammar rules for a parameter list because every variable declaration is ended with a semicolon, while parameters are separated by commas and the last parameter must not be followed by a comma.

3. The grammar rules for a list of statements are as follows:

```
stmnt_list
    : /* epsilon */
    | stmnt stmnt_list
    ;
```

Figure 12.6 on page 159 shows how the variable `stmnt` is defined. The grammar rules for statements are as follows:

```
stmnt : '{' stmnt_list '}'
      | ID '=' expr ';'
      | 'if' '(' bool_expr ')' stmnt
      | 'if' '(' bool_expr ')' stmnt 'else' stmnt
      | 'while' '(' bool_expr ')' stmnt
      | 'return' expr ';'
      | expr ';'
      ;
```

Figure 12.7 on page 160 shows how Boolean expressions are defined. The grammar rules are as follows:

```
bool_expr : bool_expr '||' bool_expr
          | bool_expr '&&' bool_expr
          | '!' bool_expr
          | '(' bool_expr ')'
          | expr '==' expr
          | expr '!=' expr
          | expr '<=' expr
          | expr '>=' expr
          | expr '<'  expr
          | expr '>'  expr
          ;
```

```
60  def p_stmnt_block(p):
61      "stmnt : '{' stmnt_list '}'"
62      p[0] = p[2]
63
64  def p_stmnt_assign(p):
65      "stmnt : ID '=' expr ';'"
66      p[0] = ('=', p[1], p[3])
67
68  def p_stmnt_if(p):
69      "stmnt : IF '(' bool_expr ')' stmnt"
70      p[0] = ('if', p[3], p[5])
71
72  def p_stmnt_if_else(p):
73      "stmnt : IF '(' bool_expr ')' stmnt ELSE stmnt"
74      p[0] = ('if-else', p[3], p[5], p[7])
75
76  def p_stmnt_while(p):
77      "stmnt : WHILE '(' bool_expr ')' stmnt"
78      p[0] = ('while', p[3], p[5])
79
80  def p_stmnt_return(p):
81      "stmnt : RETURN expr ';'"
82      p[0] = ('return', p[2])
83
84  def p_stmnt_expr(p):
85      "stmnt : expr ';'"
86      p[0] = p[1]
```

Figure 12.6: Grammar for Integer-C, part 3.

Figure 12.8 on page 161 shows how arithmetic expressions are defined. The grammar rules are as follows:

```
expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | expr '%' expr
     | '(' expr ')'
     | NUMBER
     | ID
     | ID '(' expr_list ')'
```

Finally, Figure 12.9 on page 162 shows how `expr_list` is defined.

```
expr_list : /* epsilon */
          | expr
          | expr ',' ne_expr_list
          ;

ne_expr_list : expr
             | expr ',' ne_expr_list
             ;
```

```
87  def p_bool_expr_or(p):
88      "bool_expr : bool_expr OR bool_expr"
89      p[0] = ('||', p[1], p[3])
90
91  def p_bool_expr_and(p):
92      "bool_expr : bool_expr AND bool_expr"
93      p[0] = ('&&', p[1], p[3])
94
95  def p_bool_expr_neg(p):
96      "bool_expr : '!' bool_expr"
97      p[0] = ('!', p[2])
98
99  def p_bool_expr_paren(p):
100     "bool_expr : '(' bool_expr ')'"
101     p[0] = p[2]
102
103 def p_bool_expr_eq(p):
104     "bool_expr : expr EQ expr"
105     p[0] = ('==', p[1], p[3])
106
107 def p_bool_expr_ne(p):
108     "bool_expr : expr NE expr"
109     p[0] = ('!=', p[1], p[3])
110
111 def p_bool_expr_le(p):
112     "bool_expr : expr LE expr"
113     p[0] = ('<=', p[1], p[3])
114
115 def p_bool_expr_ge(p):
116     "bool_expr : expr GE expr"
117     p[0] = ('>=', p[1], p[3])
118
119 def p_bool_expr_lt(p):
120     "bool_expr : expr '<' expr"
121     p[0] = ('<', p[1], p[3])
122
123 def p_bool_expr_gt(p):
124     "bool_expr : expr '>' expr"
125     p[0] = ('>', p[1], p[3])
```

Figure 12.7: Grammar for Integer-C, part 4.

Furthermore, this function shows the definition of the function `p_error`, which is called if PLY detects a syntax error. PLY will detect a syntax error in the case that the state of the generated shift/reduce parser has no action for the next input token. The argument `p` to the function `p_error` is the first token for which the action of the shift/reduce parser is undefined. In this case `p.value` is the part of the input string corresponding to this token.

Exercise 31: Extend the parser that is available at

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

so that it supports the ternary C-operator for conditional expressions. For example, in C, the following assignment using the ternary operator can be used to assign to the variable `max` the maximum of the values of the variables `x`

```
126 def p_expr_plus(p):
127     "expr : expr '+' expr"
128     p[0] = ('+', p[1], p[3])
129
130 def p_expr_minus(p):
131     "expr : expr '-' expr"
132     p[0] = ('-', p[1], p[3])
133
134 def p_expr_times(p):
135     "expr : expr '*' expr"
136     p[0] = ('*', p[1], p[3])
137
138 def p_expr_divide(p):
139     "expr : expr '/' expr"
140     p[0] = ('/', p[1], p[3])
141
142 def p_expr_modulo(p):
143     "expr : expr '%' expr"
144     p[0] = ('%', p[1], p[3])
145
146 def p_expr_group(p):
147     "expr : '(' expr ')'"
148     p[0] = p[2]
149
150 def p_expr_number(p):
151     "expr : NUMBER"
152     p[0] = ('Number', p[1])
153
154 def p_expr_id(p):
155     "expr : ID"
156     p[0] = p[1]
157
158 def p_expr_fct_call(p):
159     "expr : ID '(' expr_list ')'"
160     p[0] = ('call', p[1]) + p[3][1:]
```

Figure 12.8: Grammar for Integer-C, part 5.

and `y`:

```
max = x > y ?  x :  y;
```

Extend the parser so it can process the assignment shown above. ◇

```
161 def p_expr_list_empty(p):
162     "expr_list :"
163     p[0] = ('.',)
164 
165 def p_expr_list_one(p):
166     "expr_list : expr"
167     p[0] = ('.', p[1])
168 
169 def p_expr_list_more(p):
170     "expr_list : expr ',' ne_expr_list"
171     p[0] = ('.', p[1]) + p[3][1:]
172 
173 def p_ne_expr_list_one(p):
174     "ne_expr_list : expr"
175     p[0] = ('.', p[1])
176 
177 def p_ne_expr_list_more(p):
178     "ne_expr_list : expr ',' ne_expr_list"
179     p[0] = ('.', p[1]) + p[3][1:]
180 
181 def p_error(p):
182     if p:
183         print(f'Syntax error at token "{p.value}" in line {p.lineno}.')
184     else:
185         print('Syntax error at end of input.')
186 
187 parser = yacc.yacc(write_tables=False, debug=True)
```

Figure 12.9: Grammar for Integer-C, part 6.

12.3 Code Generation

Next, we discuss the generation of code. We structure our representation by discussing the code generation for arithmetic expressions, Boolean expression, statements, and function definitions separately.

12.3.1 Translation of Arithmetic Expressions

Given an arithmetic expression e, the translation of e is supposed to generate some code that, when executed, places the result of evaluating the expression e onto the stack. To this end we define a function `compile_expr` that has the following signature:

$$\texttt{compile\_expr} : \mathit{Expr} \times \mathit{SymbolTable} \times \mathit{ClassName} \to \big\langle \mathit{List}[\mathit{AsmCmd}], \mathbb{N} \big\rangle.$$

A call of this function has the form

$$\texttt{compile\_expr}(\texttt{expr}, \texttt{st}, \texttt{name}).$$

The interpretation of the arguments is as follows:

(a) `expr` is the arithmetic expression that is to be translated into assembler code.

(b) `st` is the symbol table: Concretely, this is a dictionary that maps the variable names to their position in the local variable frame. For example, if "x" is a variable that occupies the third slot in the local variable frame, then we have

```
st['x'] = 2,
```

because the first variable in the local variable frame has index 0. We will discuss later how the positions of the variables in the local variable frame is assigned.

(c) `name` is the name of the class that is to be used by our compiler.

As we generate *Java* assembler and in *Java* every function has to be a part of a class, all functions that we create have to be static functions that are defined inside the *Java* `.class` file that is generated. The parameter `name` specifies the name of this class.

The argument `name` is only needed when function calls are translated.

The function `compile_expr` returns a pair.

(a) The first component of this pair is a list of *Java* assembler commands that adhere to the syntax recognized by *Jasmin*. In general, when the expression that is translated is complex, the execution of these assembler commands might need considerable room on the stack. However, it has to be guaranteed that when the execution of these commands finishes, the stack is back to its original height plus one because the net effect of executing these commands must be to put the value of the expression on the stack.

(b) The second component of the return value of `compile_expr` is a natural number. This natural number tells us how much the stack might grow when `expr` is evaluated. This information is needed because the *Java* virtual machine needs this information in advance: In *Java*, every function has to declare how much space it might use on the stack. This declaration is done using the pseudo assembler command

```
.limit stack sz
```

where *sz* is the maximum size of the stack. Controlling the maximum height of the stack is a security feature of *Java* that prevents exploits that utilize stack overflows.

In the following, we discuss the evaluation of the different arithmetic expressions one by one.

Translation of a Variable

If the expression that is to be compiled is a variable v, we have to load this variable onto the stack using the command `iload`. This command has one parameter which is the index of the variable on the local variable frame. This index

is stored in the symbol table `st`. Since we just need one entry on the stack to store the variable, we have

$$\texttt{compile\_expr}(v, \texttt{st}, \texttt{name}) = \big\langle \texttt{[iload st[}v\texttt{]]},\ 1 \big\rangle \quad \text{if } v \text{ is a variable.}$$

The code for translating a variable is shown in Figure 12.10 on page 164.

```
1  def compile_expr(expr, st, class_name):
2      if isinstance(expr, str):
3          return [f'iload {st[expr]}'], 1
4      ...
```

Figure 12.10: Translation of a variable.

Translation of a Constant

We can load a constant c onto the stack using the assembler command

$$\texttt{ldc}\ c.$$

As we only need the room to store c on the stack, we have

$$\texttt{compile\_expr}(c, \texttt{st}, \texttt{name}) = \big\langle \texttt{[ldc}\ c\ \texttt{]},\ 1 \big\rangle \quad \text{if } c \text{ is a constant.}$$

The code for translating a constant is shown in Figure 12.11 on page 164.

```
5  def compile_expr(expr, st, class_name):
6      ...
7      elif expr[0] == 'Number':
8          _, n = expr
9          return [f'ldc {n}'], 1
10     ...
```

Figure 12.11: Translation of a variable.

Translation of an Arithmetic Operator

In order to translate an expression of the form

lhs "+" *rhs*

into assembler code, we first have to recursively translate the expressions *lhs* and *rhs* into assembler code. Later, when this code is executed the values of the expressions *lhs* und *rhs* are placed on the stack. We add these values using the command `iadd`. If the evaluation of *lhs* needs s_1 words on the stack and the evaluation of *rhs* needs s_2 words on the stack, then the evaluation of *lhs* + *rhs* needs

$$\max(s_1, 1 + s_2)$$

words on the stack, because when *rhs* is evaluated, the value of *lhs* occupies already one position on the stack and hence the evaluation of *rhs* can only use the memory cells that are above the position where *lhs* is stored. Therefore, the compilation of *lhs* + *rhs* can be specified as follows:

$$\begin{array}{ll} & \texttt{compile\_expr}(\textit{lhs}, \texttt{st}, \texttt{name}) = \langle L_1, s_1 \rangle \\ \wedge & \texttt{compile\_expr}(\textit{rhs}, \texttt{st}, \texttt{name}) = \langle L_2, s_2 \rangle \\ \rightarrow & \texttt{compile\_expr}(\textit{lhs} + \textit{rhs}, \texttt{st}, \texttt{name}) = \big\langle L_1 + L_2 + \texttt{[iadd]}, \max(s_1, 1 + s_2) \big\rangle. \end{array}$$

Figure 12.12 on page 165 shows how the translation of arithmetic operators is implemented. The translation of subtraction, multiplication, and division is similar.

```
11  def compile_expr(expr, st, class_name):
12      ...
13      elif expr[0] in  ['+', '-', '*', '/', '%']:
14          op, lhs, rhs = expr
15          L1, sz1 = compile_expr(lhs, st, class_name)
16          L2, sz2 = compile_expr(rhs, st, class_name)
17          OpToCmd = { '+': 'iadd', '-': 'isub', '*': 'imul', '/': 'idiv', '%': 'irem' }
18          Cmd     = indent(OpToCmd[op])
19          return L1 + L2 + [Cmd], max(sz1, 1 + sz2)
20      ...
```

Figure 12.12: Translation of the addition of expressions.

Translation of a Function Call

In order to translate a function call of the form $f(a_1, \cdots, a_n)$ we first have to translate the arguments a_1, $\cdots$, a_n. Then there are two cases:

1. f is a user defined function. In this case, the byte code treats f as a static function of the class `class_name`. This static function can be called using the assembler command `invokestatic`. In order to calculate the stack size that needs to be reserved for the evaluation of $f(a_1, \cdots, a_n)$, let us assume that we have

   $$\texttt{compile\_expr}(a_i, \texttt{st}, \texttt{class\_name}) = \langle L_i, s_i \rangle,$$

   i.e. evaluation of the i^{th} argument is done by the list of assembler commands L_i and needs a stack size of s_i. As we are evaluating the arguments a_i in the order from a_1 to a_n, the evaluation of a_i needs a stack size of $i - 1 + s_i$, since the results from the evaluation of the arguments a_1, $\cdots$, a_{i-1} are already placed on the stack. Therefore, if we define

   $$s := \max(s_1, 1 + s_2, \cdots, i - 1 + s_i, \cdots, n - 1 + s_n),$$

   then s is the total amount of stack size needed to evaluate the arguments. Furthermore, let us define the signature `fs` of the function f as the string

   $$\texttt{fs} := \texttt{class\_name/f(}\underbrace{\texttt{I} \cdots \texttt{I}}_{n}\texttt{)I}.$$

   Then we can define the value of $\texttt{compile\_expr}\big(f(a_1, \cdots, a_n), \texttt{st}, \texttt{class\_name}\big)$ as follows:

   $$\texttt{compile\_expr}\big(f(a_1, \cdots, a_n), \texttt{st}, \texttt{class\_name}\big) := \big\langle L_1 + \cdots + L_n + \texttt{[invokestatic fs]}, \max(s, 1) \big\rangle.$$

2. If f is the function `println`, then *Jasmin* treats the function f as a method of the predefined *Java* object `java.lang.System.out`. This method can be invoked using the assembler command `invokevirtual`. As before, let us assume that we have

   $$\texttt{compile\_expr}(a_i, \texttt{st}, \texttt{class\_name}) = \langle L_i, s_i \rangle.$$

   This time, we have to start by putting the object `java.lang.System.out` onto the stack before we can evaluate any of the arguments. Therefore, in order to calculate the stack size we now define

$$s := \max(1 + s_1, 2 + s_2, \cdots, i - 1 + s_i, \cdots, n + s_n).$$

In this case we have to define the signature `fs` of f as the string

$$\texttt{fs} := \texttt{java/io/PrintStream/println(}\underbrace{\texttt{I}\cdots\texttt{I}}_{n}\texttt{)V}.$$

Furthermore, we define the command `gs` for putting the object `java.lang.System.out` onto the stack as

$$\texttt{gs} := \texttt{getstatic java/lang/System/out Ljava/io/PrintStream;}.$$

Then we can define the value of $\texttt{compile\_expr}\big(f(a_1, \cdots, a_n), \texttt{st}, \texttt{class\_name}\big)$ as follows:

$$\texttt{compile\_expr}\big(f(a_1, \cdots, a_n), \texttt{st}, \texttt{class\_name}\big) := \big\langle \texttt{[gs]} + L_1 + \cdots + L_n + \texttt{[invokestatic fs]}, \max(s, 1)\big\rangle.$$

Figure 12.12 on page 165 shows how the translation of function calls is implemented.

```
21  def compile_expr(expr, st, class_name):
22      ...
23      elif expr[0] == 'call' and expr[1] == 'println':
24          _, _, *args = expr
25          CmdLst    = ['getstatic java/lang/System/out Ljava/io/PrintStream;')
26          stck_size = 0
27          cnt       = 0
28          for arg in args:
29              L, sz_arg = compile_expr(arg, st, class_name)
30              stck_size = max(stck_size, cnt + 1 + sz_arg)
31              CmdLst   += L
32              cnt      += 1
33          CmdLst += [f'invokevirtual java/io/PrintStream/println({"I"*cnt})V']
34          return CmdLst, stck_size
35      elif expr[0] == 'call' and expr[1] != 'println':
36          _, f, *args = expr
37          CmdLst    = []
38          stck_size = 0
39          cnt       = 0
40          for arg in args:
41              L, sz_arg = compile_expr(arg, st, class_name)
42              stck_size = max(stck_size, cnt + sz_arg)
43              CmdLst   += L
44              cnt      += 1
45          CmdLst += [f'invokestatic {class_name}/{f}({"I"*cnt})I']
46          return CmdLst, max(stck_size, 1)
47      ...
```

Figure 12.13: Translation of function calls.

12.3.2 Translation of Boolean Expressions

Boolean expressions are build from equations and inequations using the logical operators "!" (logical not), "&&" (logical and), and "||" (logical or). To compile Boolean expressions we define a function `compile_bool` that has the following signature:

$$\texttt{compile\_bool} : \mathit{BoolExpr} \times \mathit{SymbolTable} \times \mathit{ClassName} \to \big\langle \mathit{List}[\mathit{AsmCmd}], \mathbb{N} \big\rangle.$$

The interpretation of the arguments is similar to the interpretation of the arguments of the function `compile_expr`. When we execute the list of assembler commands that result from the compilation of a Boolean expression, we expect that the execution of these commands either puts the number 1 (representing `True`) or the number 0 (representing `False`) onto the stack. We start our discussion of the function `compile_bool` with the translation of equations.

Translation of Equations

In the *Java* virtual machine, the logical values `True` and `False` are represented by the integers 1 and 0, respectively. Therefore, the code produced from translating an equation of the form

lhs == *rhs*

should either place the number 1 or 0 onto the stack: If the value of *lhs* is equal to the value of *rhs*, the number 1 has to be put on the stack, otherwise the value 0 has to be put onto the stack. Let us assume that we have

$$\texttt{compile\_expr}(\mathit{lhs}, \texttt{st}, \texttt{class\_name}) = \langle L_1, s_1 \rangle \quad \text{and} \quad \texttt{compile\_expr}(\mathit{rhs}, \texttt{st}, \texttt{class\_name}) = \langle L_2, s_2 \rangle.$$

Then the stack size needed for evaluating the equation *lhs* == *rhs* is given by the expression

$$\max(s_1, 1 + s_2)$$

and the function `compile_bool` can be specified as follows:

$$\begin{array}{lcl}
\mathit{compile_bool}(\mathit{lhs} == \mathit{rhs}, \texttt{st}, \texttt{class_name}) & = & \big\langle \quad L_1 \\
& & + \quad L_2 \\
& & + \quad [\,\texttt{if_icmpeq}\ \mathit{true}\,] \\
& & + \quad [\,\texttt{bipush}\ 0\,] \\
& & + \quad [\,\texttt{goto}\ \mathit{next}\,] \\
& & + \quad [\,\mathit{true}\texttt{:}\,] \\
& & + \quad [\,\texttt{bipush}\ 1\,] \\
& & + \quad [\,\mathit{next}\texttt{:}\,], \qquad\qquad s \big\rangle
\end{array}$$

Here, *true* and *next* have to be new labels that do not occur elsewhere in the code for the function that is compiled. Let me explain this equation in detail:

1. L_1 and L_2 are the lists of assembler commands that evaluate *lhs* and *rhs*, respectively.
2. Executing L_1 and L_2 leaves two values on the stack. These values are then compared using the assembler command `if_icmpeq`. If these value are the same, execution proceeds at the label *true*.
3. Otherwise execution commences with the next instruction and hence the value 0 is pushed onto the stack.
4. Next, the control flow jumps to the label *next*, which is at the end of the generated list of assembler commands. Hence, in this case the execution of equation *lhs* == *rhs* is finished.
5. If the values of *lhs* and *rhs* are the same, the number 1 is pushed onto the stack.

The translation of a negated equation of the form

lhs != *rhs*

is similar to the translation of an equation: We only have to replace the command `if_icmpeq` with the command `if_icmpne`. Similarly, if the inequation has the form

lhs <= *rhs*

we have to replace the command `if_icmpeq` with the command `if_icmple`. If the inequation has the form

lhs >= *rhs*

we have to replace the command `if_icmpeq` with the command `if_icmpge`. If the inequation has the form

lhs < *rhs*

we have to replace the command `if_icmpeq` with the command `if_icmplt`. If the inequation has the form

lhs > *rhs*

we have to replace the command `if_icmpeq` with the command `if_icmpgt`. Figure 12.14 on page 168 shows how the translation of equations and inequations is implemented.

```
1   def compile_bool(expr, st, class_name):
2       if expr[0] in ['==', '!=', '<=', '>=', '<', '>']:
3           OpToCmd = { '==': 'if_icmpeq',
4                       '!=': 'if_icmpne',
5                       '<=': 'if_icmple',
6                       '>=': 'if_icmpge',
7                       '<' : 'if_icmplt',
8                       '>' : 'if_icmpgt'
9                     }
10          op, lhs, rhs = expr
11          L1, sz1    = compile_expr(lhs, st, class_name)
12          L2, sz2    = compile_expr(rhs, st, class_name)
13          true_label = new_label()
14          next_label = new_label()
15          CmdLst     = L1 + L2
16          cmd        = OpToCmd[op]
17          CmdLst    += [indent(cmd + ' ' + true_label)]
18          CmdLst    += [indent('bipush 0')]
19          CmdLst    += [indent('goto ' + next_label)]
20          CmdLst    += [' ' * 4 + true_label + ':']
21          CmdLst    += [indent('bipush 1')]
22          CmdLst    += [' ' * 4 + next_label + ':']
23          return CmdLst, max(sz1, 1 + sz2)
24          ...
```

Figure 12.14: Translation of equations.

Translation of Binary Boolean Operators

In order to translate an expression of the form

lhs && *rhs*

into assembler code, we first have to recursively translate the expressions *lhs* and *rhs* into assembler code. Later, when this code is executed the values of the expressions *lhs* und *rhs* are placed on the stack. We combine these values using the command `iand`. If the evaluation of *lhs* needs s_1 words on the stack and the evaluation of *rhs* needs s_2 words on the stack, then the evaluation of *lhs* + *rhs* needs

$$\max(s_1, 1 + s_2)$$

words on the stack, because when *rhs* is evaluated, the value of *lhs* occupies already one position on the stack and hence the evaluation of *rhs* can only use the memory cells that are above the position where *lhs* is stored. Therefore, the compilation of *lhs* && *rhs* can be specified as follows:

$$\begin{array}{ll} & \texttt{compile\_bool}(\textit{lhs}, \texttt{st}, \texttt{name}) = \langle L_1, s_1 \rangle \\ \wedge & \texttt{compile\_bool}(\textit{rhs}, \texttt{st}, \texttt{name}) = \langle L_2, s_2 \rangle \\ \rightarrow & \texttt{compile\_bool}(\textit{lhs} + \textit{rhs}, \texttt{st}, \texttt{name}) = \big\langle L_1 + L_2 + \texttt{[iand]}, \max(s_1, 1 + s_2) \big\rangle. \end{array}$$

The translation of an expression of the form

lhs || *rhs*

is similar: Instead of using the command `iand` we have to use the command `ior` instead. Figure 12.15 on page 169 shows how the translation of binary logical operators is implemented.

```
25  def compile_expr(expr, st, class_name):
26      ...
27      elif expr[0] in ['&&', '||']:
28          op, lhs, rhs = expr
29          OpToCmd      = { '&&': iand, '||': 'ior' }
30          L1, sz1      = compile_bool(lhs, st, class_name)
31          L2, sz2      = compile_bool(rhs, st, class_name)
32          cmd          = OpToCmd[op]
33          CmdLst       = L1 + L2 + [indent(cmd)]
34          return CmdLst, max(sz1, 1 + sz2)
35      ...
```

Figure 12.15: Translation of binary logical operators.

It should be noted that our translation of the binary logical operators is different from what happens in the language `C`. In `C`, the evaluation of logical operators is lazy: if an expression of the form

lhs && *rhs*

is evaluated, the evaluation is stopped as soon the evaluation of *lhs* returns 0 because then there is no point to evaluate *rhs*, since if the value of *lhs* is 0, the result of the expression *lhs* && *rhs* is 0, independent from the value of the expression *rhs*. Similarly, if an expression of the form

lhs || *rhs*

is evaluated and the evaluation of *lhs* is 1, then the result of the evaluation of the expression *lhs* || *rhs* is 1, regardless of the value of *rhs*.

Translation of Negations

The translation of an expression of the form `!`*expr* is not as straightforward as the translation of conjunctions and disjunctions. The reason is that *Jasmin* has no assembler command of the form `inot` that negates a logical value. But there is another way: Since we represent the truth values as 0 and 1, we can specify negation arithmetically as follows:

$$!x = 1 - x.$$

Therefore, if we have

$$\texttt{compile\_bool}(\textit{arg}, \texttt{st}, \texttt{class\_name}) = \langle L, s \rangle$$

we can define:

$$\texttt{compile\_bool}(\texttt{!}\textit{arg}, \texttt{st}, \texttt{class\_name}) = \big\langle \texttt{[bipush 1]} + L + \texttt{[isub]}, 1 + s \big\rangle.$$

Figure 12.16 on page 170 shows how the translation of the negation operator is implemented.

```
36  def compile_expr(expr, st, class_name):
37      ...
38      elif expr[0] == '!':
39          _, arg = expr
40          L, sz  = compile_expr(arg, st, class_name)
41          CmdLst = ['bipush 1'] + L + ['isub']
42          return CmdLst, max(sz1, sz + 1)
43      ...
```

Figure 12.16: Translation of the negation operator.

12.3.3 Translation of Statements

Next, we show how statements are compiled. First of all, we agree that the execution of a statement must not change the size of the stack: The size of stack before the execution of a statement must be the same as the size of the stack after after the statement has been executed. Of course, during the execution of the statement the stack may well grow and shrink. But once the execution of the statement has finished, the stack has to be cleaned from all intermediate values that have been put on the stack during the execution of the statement.

To compile statements we define a function `compile_stmnt` that has the following signature:

$$\texttt{compile\_stmnt} : \mathit{Stmnt} \times \mathit{SymbolTable} \times \mathit{ClassName} \to \big\langle \mathit{List}[\mathit{AsmCmd}], \mathbb{N} \big\rangle.$$

The interpretation of the arguments is similar to the interpretation of the arguments of the function `compile_expr`. The only difference is that the first argument now is an abstract syntax tree that represents a statement.

Translation of Assignments

To begin with, we show how an assignment of the form

$$x = \mathit{expr}$$

is translated. The idea is to evaluate *expr*. As a consequence of this evaluation the value of *expr* remains on the stack, from where it can be stored in the variable x using the assembler command `istore`. Therefore, if we have

$$\texttt{compile\_expr}(\mathit{expr}, \texttt{st}, \texttt{class\_name}) = \langle L, s \rangle$$

we can define:

$$\texttt{compile\_stmnt}(x = \mathit{expr}, \texttt{st}, \texttt{class\_name}) = \big\langle L + \texttt{[istore st[x]]}, s \big\rangle.$$

Figure 12.17 on page 170 shows how the translation of an assignment is implemented.

```
1   def compile_stmnt(stmnt, st, class_name):
2       if stmnt[0] == '=':
3           _, var, expr = stmnt
4           CmdLst, sz = compile_expr(expr, st, class_name)
5           CmdLst    += [f'istore {st[var]}']
6           return CmdLst, sz
7       ...
```

Figure 12.17: Translation of an assignment statement.

The Translation of Branching Commands

Next, we show how to translate a branching command of the form

`if` (*bool_expr*) *stmnt*.

Obviously, we first have to translate the Boolean expression *bool_expr*. If we have

$$\texttt{compile\_expr}(\textit{bool\_expr}, \texttt{st}, \texttt{class\_name}) = \langle L_1, s_1 \rangle$$

and, furthermore,

$$\texttt{compile\_stmnt}(\textit{stmnt}, \texttt{st}, \texttt{class\_name}) = \langle L_2, s_2 \rangle,$$

then we can define:

$$\begin{array}{lcll} \textit{compile\_stmnt}\big(\texttt{if}\ (\textit{expr})\ \textit{stmnt}, \texttt{st}, \texttt{class\_name}\big) & = & \big\langle & L_1 \\ & & + & [\ \texttt{ifeq}\ \textit{else}\ ] \\ & & + & L_2 \\ & & + & [\ \textit{else}\texttt{:}\ ], \qquad \max(s_1, s_2) \big\rangle \end{array}$$

This works because executing the list of assembler commands L_1 leaves either a 0 or a 1 on the stack. If *expr* is false, the top of the stack stores the number 0. In this case, the branching command `ifeq` branches to the label *else* and the assembler commands in the list L_2 are not executed. If *expr* is true, the top of the stack stores the number 1. Therefore, the branching command `ifeq` does not branch and the assembler commands in the list L_2 are executed, i.e. *stmnt* is executed.

The translation of a branching command of the form

`if` (*bool_expr*) $\textit{stmnt}_1$ `else` $\textit{stmnt}_2$

is similar. Let us assume that we have the following:

1. $\texttt{compile\_expr}(\textit{bool\_expr}, \texttt{st}, \texttt{class\_name}) = \langle L_1, s_1 \rangle$
2. $\texttt{compile\_stmnt}(\textit{stmnt}_1, \texttt{st}, \texttt{class\_name}) = \langle L_2, s_2 \rangle$, and
3. $\texttt{compile\_stmnt}(\textit{stmnt}_2, \texttt{st}, \texttt{class\_name}) = \langle L_3, s_3 \rangle$.

Then we define:

$$\begin{array}{lcll} \textit{compile\_stmnt}\big(\texttt{if}\ (\textit{bool\_expr})\ \textit{stmnt}_1\ \texttt{else}\ \textit{stmnt}_2, \texttt{st}, \texttt{class\_name}\big) & = & \big\langle & L_1 \\ & & + & [\ \texttt{ifeq}\ \textit{else}\ ] \\ & & + & L_2 \\ & & + & [\texttt{goto}\ \textit{next}] \\ & & + & [\ \textit{else}\texttt{:}\ ] \\ & & + & L_3 \\ & & + & [\ \textit{next}\texttt{:}\ ], \qquad \max(s_1, s_2, s_3) \big\rangle \end{array}$$

Figure 12.18 on page 172 shows how the translation of branching statements is implemented.

Translation of a Loop

If we want to translate a `while` loop of the form

`while` (*cond*) *stmnt*

we recursively compute

$$\texttt{compile\_expr}(\textit{cond}, \texttt{st}, \texttt{class\_name}) = \langle L_1, s_1 \rangle$$

and

$$\texttt{compile\_stmnt}(\textit{stmnt}, \texttt{st}, \texttt{class\_name}) = \langle L_2, s_2 \rangle,$$

```
def compile_stmnt(stmnt, st, class_name):
    ...
    elif stmnt[0] == 'if':
        _, expr, sub_stmnt = stmnt
        L1, sz1    = compile_bool(expr, st, class_name)
        L2, sz2    = compile_stmnt(sub_stmnt, st, class_name)
        else_label = new_label()
        CmdLst = L1 + [f'ifeq {else_label}'] + L2 + [else_label + ':']
        return CmdLst, max(sz1, sz2)
    elif stmnt[0] == 'if-else':
        _, expr, then_stmnt, else_stmnt = stmnt
        L1, sz1    = compile_bool(expr, st, class_name)
        L2, sz2    = compile_stmnt(then_stmnt, st, class_name)
        L3, sz3    = compile_stmnt(else_stmnt, st, class_name)
        else_label = new_label()
        next_label = new_label()
        if_stmnt   = f'ifeq {else_label}'
        else_stmnt = else_label + ':'
        next_stmnt = next_label + ':'
        goto_stmnt = f'goto {next_label}'
        CmdLst = L1 + [if_stmnt] + L2 + [goto_stmnt, else_stmnt] + L3 + [next_stmnt]
        return CmdLst, max(sz1, sz2, sz3)
    ...
```

Figure 12.18: Translation of branching statements.

Then we can define

$$\begin{aligned} \textit{compile\_stmnt}(\texttt{while (}\textit{cond}\texttt{) }\textit{stmnt}, \texttt{st}, \texttt{class\_name}) \;=\; \big\langle\; & [\textit{loop}:] \\ & + \; L_1 \\ & + \; [\,\texttt{ifeq }\textit{next}\,] \\ & + \; L_2 \\ & + \; [\texttt{goto }\textit{loop}] \\ & + \; [\,\textit{next}\texttt{:}\,], \qquad \max(s_1, s_2)\big\rangle \end{aligned}$$

Figure 12.19 on page 173 shows how the translation of a loop is implemented.

Translation of a Return Statement

In order to translate a return statement of the form

`return` *expr*`;`

we first have to translate the expression *expr*. Assume that

$\texttt{compile\_expr}(\textit{expr}, \texttt{st}, \texttt{class\_name}) = \langle L, s \rangle$.

Then we can define

$\texttt{compile\_stmnt}(\texttt{return }\textit{expr}\texttt{;}, \texttt{st}, \texttt{class\_name}) = \big\langle L + [\texttt{ireturn}], s \big\rangle$.

Figure 12.20 on page 173 shows how a return statement is translated.

```
def compile_stmnt(stmnt, st, class_name):
    ...
    elif stmnt[0] == 'while':
        _, expr, body_stmnt = stmnt
        L1, sz1    = compile_bool(expr, st, class_name)
        L2, sz2    = compile_stmnt(body_stmnt, st, class_name)
        loop_label = new_label()
        next_label = new_label()
        if_stmnt   = f'ifeq {next_label}'
        loop_stmnt = loop_label + ':'
        next_stmnt = next_label + ':'
        goto_stmnt = f'goto {loop_label}'
        CmdLst = [loop_stmnt] + L1 + [if_stmnt] + L2 + [goto_stmnt, next_stmnt]
        return CmdLst, max(sz1, sz2)
    ...
```

Figure 12.19: Translation of a loop.

```
def compile_stmnt(stmnt, st, class_name):
    ...
    elif stmnt[0] == 'return':
        _, expr = stmnt
        CmdLst, sz = compile_expr(expr, st, class_name)
        CmdLst    += ['ireturn']
        return CmdLst, sz
    ...
```

Figure 12.20: Translation of a return statement.

Translating a Block Statement

The translation of block statement of the form

$$\{\mathit{stmnt}_1\,;\ \cdots\ \mathit{stmnt}_n\,;\}$$

proceeds by translating the statements stmnt_i separately. Assume that

$$\texttt{compile\_stmnt}(\mathit{stmnt}_i, \texttt{st}, \texttt{class\_name}) = \langle L_i, s_i \rangle \quad \text{for } i = 1, \cdots, n.$$

Then we define

$$\mathit{compile\_stmnt}\big(\{\mathit{stmnt}_1\,;\ \cdots\ \mathit{stmnt}_n\,;\}, \texttt{st}, \texttt{class\_name}\big) := \big\langle L_1 + \cdots + L_n, \max(s_1, \cdots, s_n) \big\rangle.$$

Figure 12.21 on page 174 shows how the translation of a block statement is implemented.

Translation of Expression Statements

When we translate an expression statement of the form

$$\mathit{expr}\ ;$$

we assume that the evaluation of the expression does not leave a value on the stack. At present, the only expression

```
def compile_stmnt(stmnt, st, class_name):
    ...
    elif stmnt[0] == '.':
        _, *stmnt_lst = stmnt
        CmdLst = []
        size   = 0
        for s in stmnt_lst:
            L, sz = compile_stmnt(s, st, class_name)
            CmdLst += L
            size   = max(size, sz)
        return CmdLst, size
```

Figure 12.21: Translation of an expression statement.

whose evaluation does not create a value is an invocation of the function `println`. Then in order to evaluate the expression statement, we just have to evaluate the expression. Therefore, we have

$$\texttt{compile\_stmnt}(\textit{expr};, \texttt{st}, \texttt{class\_name}) = \texttt{compile\_expr}(\textit{expr}, \texttt{st}, \texttt{class\_name}).$$

Figure 12.22 on page 174 shows how the translation of an expression statement is implemented.

```
def compile_stmnt(stmnt, st, class_name):
    ...
   else: # it must be an expression statement
        CmdLst, sz = compile_expr(stmnt, st, class_name)
        return CmdLst, sz
```

Figure 12.22: Translation of an expression statement.

12.3.4 Translation of a Function Definition

Figure 12.23 on page 175 shows how a function definition is translated. In order to understand how this works we have to understand what type of code the function `compile` has to generate. Depending on whether the function that is translated is the function `main` or not there are two cases.

1. If the function that is to be compiled has the name "`main`", the code that has to be generated has the following form:

```
        .method public static main([Ljava/lang/String;)V
        .limit locals l
        .limit stack s
            s_1
            ⋮
            s_n
            return
        .end method
```

```
def compile_fct(fct_def, class_name):
    global label_counter
    label_counter = 0
    _, name, parameters, variables, stmnts = fct_def
    _, *parameters = parameters
    _, *variables  = variables
    _, *stmnts     = stmnts
    m   = len(parameters)
    n   = len(variables)
    st  = {}
    cnt = 0
    for var in parameters:
        st[var] = cnt
        cnt    += 1
    for var in variables:
        st[var] = cnt
        cnt    += 1
    CmdLst = []
    size   = 0
    for stmnt in stmnts:
        L, sz = compile_stmnt(stmnt, st, class_name)
        CmdLst += L
        size = max(size, sz)
    limit_locals = f'.limit locals {m+n}'
    limit_stack  = f'.limit stack  {size}'
    if name != 'main':
        method = f'.method public static {name}({"I"*m})I'
        CmdLst = [method, limit_locals, limit_stack] + CmdLst + ['.end method']
        return CmdLst, sz
    else:
        method = '.method public static main([Ljava/lang/String;)V'
        CmdLst = [method, limit_locals, limit_stack] + CmdLst + \
                 ['return', '.end method']
        return CmdLst, sz
```

Figure 12.23: Translation of a function definition.

Here l is the number of all variables used by the function `main`, while s is the maximum height of the stack. Finally, $s_1, \cdots, s_n$ are the assembler commands that result from translating the block inside the function.

2. Otherwise, the following code is to be generated:

```
1        .method public static f(I···I)I
2        .limit locals l
3        .limit stack s
4           s_1
5           ⋮
6           s_n
7        .end method
```

   Here f is the name of the function. The signature of f is a string of the form

$$f(\underbrace{\texttt{I}\cdots\texttt{I}}_{m})\texttt{I}$$

   where m is the number of the parameters. Furthermore, l is the number of all local variables and s is the maximum height of the stack. Finally, s_1, $\cdots$, s_n are the assembler commands that result from translating the block inside the function.

We proceed to discuss the implementation shown in Figure 12.23 line by line.

1. We initialize the global variable `label_counter` to 0. Every time a new label is generated, this counter will be incremented by the function `new_label` that creates labels of the form `l1`, `l2`, etc.
2. The nested tuple that is produced by the parser from a function definition has the form

   (`'fct'`, *name*, *parameters*, *variables*, *stmnts*).

   (a) *name* is the name of the function.
   (b) *parameters* is the abstract syntax tree representing the list of parameters.
   (c) *variables* is the abstract syntax tree representing the list of all local variables declared in the function.
   (d) *stmnts* is the the abstract syntax tree representing the list of all statements in the body of the function.
3. Initially, the parameters, variables, and statements are stored as nested tuples. In line 5-7 these nested tuples are transformed into proper lists.
4. m is the number of parameters.
5. n is the number of local variables.
6. The `for` loops in line 12–17 initialize the symbol table `st`. Given a parameter or local variable x, the expression `st[`x`]` returns the position in the local variable frame where the parameter or variable is stored.
7. The `for` loop in line 20–23 translates the statements in the body of the function one by one. Furthermore, it computes the size of the stack that is needed.
8. Line 24 declares the size of the local variable frame. The pseudo-command `.limit locals` reserves memory for the parameters and the local variables.
9. Line 25 declares the size of the stack needed by the function.
10. Depending on whether the function that is compiled is the function `main` or not, the correct code is returned.

```
1  import os
2
3  def compile_program(file_name):
4      directory = os.path.dirname(file_name)
5      base      = os.path.basename(file_name)
6      base      = base[:-2]
7      with open(file_name, 'r') as handle:
8          program = handle.read()
9      ast = yacc.parse(program)
10     _, *fct_lst = ast
11     CmdLst      = []
12     for fct in fct_lst:
13         L, _ = compile_fct(fct, base)
14         CmdLst += L + ['\n']
15     with open(directory + '/' + base + '.jas', 'w') as handle:
16         handle.write('.class public ' + base + '\n');
17         handle.write('.super java/lang/Object\n\n');
18         handle.write('.method public <init>()V\n');
19         handle.write('    aload 0\n');
20         handle.write('    invokenonvirtual java/lang/Object/<init>()V\n');
21         handle.write('    return\n');
22         handle.write('.end method\n\n');
23         for cmd in CmdLst:
24             handle.write(cmd + '\n')
```

Figure 12.24: Translation of a program.

12.3.5 Compiling a Program

We conclude this chapter by presenting the function `compile_program` that is shown in Figure 12.24 on page 177. This function takes the name of a file as its argument. This file is expected to contain a program written in the subset of the programming language `C` that has been presented in this chapter. If the name of this file is, for example, `Test.c`, then the function `compile_program` generates a file with the name `Test.jas`, that contains an assembler program that adheres to the rules of *Jasmin*. We proceed to discuss the function `compile_program` line by line.

1. In line 1 we import the module `os`. This module contains the submodule `path` which contains the functions `dirname` and `basename`.

   (a) The function `dirname` takes a string specifying a *file path*, e.g. something like

   `'~/Dropbox/Formal-Languages/Ply/Examples/Test.c'`

   and returns the name of the directory. In the example given above it would return the string

   `'~/Dropbox/Formal-Languages/Ply/Examples'`.

   (b) The function `basename` takes a *file path* and returns the name of the filename of the directory. In the example given previously it would return the string `'Test.c'`.

   The module `os` is aware of the differences in naming directories that exist in different operating systems.

2. After computing the `directory` of the file and its `base` in line 4 and 5, we remove the file extension `'.c'` in line 6.

3. Line 7 and 8 read the file and store its content into the string `program`.

4. In line 9 our parser converts the program into an abstract syntax tree that is represented as a nested tuple.
5. `fct_lst` is a list of all function definitions occurring in the given program.
6. `CmdLst` is a list of all assembler commands that will be generated.
7. The `for` loop in line 12–14 translates all function definitions into assembler code and concatenates the assembler statements into the list `CmdLst`.
8. In line 15 the output file is opened for writing.
9. Line 16–22 write the initialization code to the output file.
10. The `for` loop in line 23–24 writes the generated assembler commands to the output file.

Figure 12.25 on page 178 shows an integer `C` program for computing the sum $\sum_{i=1}^{6^2} i$. When we translate this program using the compiler developed in this chapter, we get the program that is shown in Figure 12.26 on page 179.

```
1   int sum(int n) {
2       int s;
3       s = 0;
4       while (n != 0) {
5           s = s + n;
6           n = n - 1;
7       }
8       return s;
9   }
10
11  int main() {
12      int n;
13      n = 6 * 6;
14      println(sum(n));
15  }
```

Figure 12.25: A `C` program for computing the sum $\sum_{i=1}^{6^2} i$.

```
1  .class public MySum
2  .super java/lang/Object
3
4  .method public <init>()V
5      aload 0
6      invokenonvirtual java/lang/Object/<init>()V
7      return
8  .end method
9
10 .method public static sum(I)I
11 .limit locals 2
12 .limit stack  2
13         ldc 0
14         istore 1
15     l3: iload 0
16         ldc 0
17         if_icmpne l1
18         bipush 0
19         goto l2
20     l1: bipush 1
21     l2: ifeq l4
22         iload 1
23         iload 0
24         iadd
25         istore 1
26         iload 0
27         ldc 1
28         isub
29         istore 0
30         goto l3
31     l4: iload 1
32         ireturn
33 .end method
34
35 .method public static main([Ljava/lang/String;)V
36 .limit locals 1
37 .limit stack  2
38         ldc 6
39         ldc 6
40         imul
41         istore 0
42         getstatic java/lang/System/out Ljava/io/PrintStream;
43         iload 0
44         invokestatic MySum/sum(I)I
45         invokevirtual java/io/PrintStream/println(I)V
46         return
47 .end method
```

Figure 12.26: Assembler program generated by our compiler.

Exercise 32: Extend the compiler that is presented in this chapter and is available at

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

so that a new kind of `for`-loops is supported. Figure 12.27 on page 180 shows an example of the use of this kind of `for` loop. ◇

```
int sum(int n) {
    int s;
    int i;
    s = 0;
    for (i = 1 .. n) {
        s = s + i;
    }
    return s;
}
```

Figure 12.27: A function for computing the sum $\sum_{i=1}^{n} i$.

Bibliography

[ASUL06] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

[Bac59] John W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich ACM-GAMM conference. In *IFIP Congress*, pages 125–131, 1959.

[CS70] John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, 1971.

[Ear70] Jay C. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[Ers58] A. P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1(8):3–6, 1958.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[Kas65] Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, Air Force Cambridge Research Lab, Bedford, MA, 1965.

[Kle56] Stephen C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata studies*, pages 3–40. Princeton University Press, 1956.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, December 1965.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1978.

[MD97] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly, 1997.

[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.

[RS59] Michael Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[Sip12] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, third edition, 2012.

[You67] Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.

Index

$L(F)$, 27, 32
Σ, 26
$\doteq$, 14
$\emptyset$, 12
λ-generating, 110
$\rightsquigarrow$, 32
PLY, 16
det(F), 34
ε, 12
ε-closure, 32
ε-transition, 31
Σ^*, 4
λ, 4
closure$(\mathcal{M})$, 108
ASCII-Alphabet, 4
CYK-Algorithmus, 92
DFA, 31
EBNF-Grammar, 73
FSM, 26
PLY, 17
UNICODE-Alphabet, 4
Integer-C, 151

accepted language, 27
accepting state, 25
accepting states, set of, 26
alphabet, 4
ambiguous grammar, 65
augmented grammar, 109, 111

closure of a set of marked rules, 108
Cocke-Younger-Kasami-Algorithmus, 92
complement of a language, 48, 50
complete, finite state machine, 27
concatenation, 6
configuration (of an NFA), 31
context-free grammar, 60
context-free language, 7

dead state, 27
death of an FSM, 26
derivation-step, 61
deterministic finite automaton, 31

e.m.R., 119
Earley-Objekt, 92

empty string, λ, 4
equivalence of regular expressions, 14
extended marked rule, 119

finite state machine, 26
formal language, 4, 7
functional token definitions, 17

grammar rule, 60

identification of states, 44
immediate token definition, 17
input alphabet, 26

Kleene closure, 11

left-recursive, 66
leftmost derivation, 99
length of a string, 6

marked rule, 107

Nfa, 31
non-deterministic FSM, 31
non-terminal, 59
non-terminals, 60

palindrome, 63
parse-tree, 64
parser configuration, 100
ply, 17
power of a language, 10
power of a string, 11
prime number, 7
product of languages, 10
Pumping Lemma for regular languages, 52
pumping lemma for regular languages, 52

reachable, 44
reduce-reduce conflict, 114, 121
regular expression, 10, 12
regular language, 7, 48
reversal of a string, 51
rightmost derivation, 99

scanner, 16
scanner states, 20

separable, 44
separates, 44
set difference, 50
shift-reduce conflict, 114, 121
shift-reduce parser, 100
SLR grammar, 114
start state, 25, 26
start symbol, 61
string, 4
symbol, 4
symbol table, 162
syntactic variable, 59
syntactic category, 59
syntactic variables, 60

terminal, 59
terminals, 60
token, 16
tokens, 60
transition function, 26

universal language, 7