Formal Languages and Automata

David Suendermann

http://suendermann.com

Baden-Wuerttemberg Cooperative State University

Stuttgart, Germany

- **The most up-to-date version of this document as well as auxiliary material can be found online at**

  `http://suendermann.com`

- **If you are running Windows, please install the complete UNIX emulation package Cygwin, so everybody has the same tool set available:**

  `http://cygwin.com`

- **A comprehensive (though German) script by my colleague Karl Stroetmann covers many of the topics discussed in this lecture:**

  `http://wwwlehre.dhbw-stuttgart.de/~stroetma/Formale-Sprachen/formale-sprachen.pdf`

Outline

1. **introduction**
2. **regular expressions**
   - **compact description of sets of strings**
   - **fundamental component of script languages (Perl, Python, grep, sed, awk, etc.) and of most modern programming languages (.NET, SQL Server 2008, Java, etc.)**
3. **the scanner generator JFlex**
4. **finite-state machines**
   ...are able to detect regular expressions
5. **formal grammars**

Outline (cont.)

6. **context-free languages**

   most programming languages are context-free

7. **Antlr**

   ...a parser generator

Outline

1. **introduction**
2. **regular expressions**
   - **compact description of sets of strings**
   - **fundamental component of script languages (Perl, Python, grep, sed, awk, etc.) and of most modern programming languages (.NET, SQL Server 2008, Java, etc.)**
3. **the scanner generator JFlex**
4. **finite-state machines**
   ...are able to detect regular expressions
5. **formal grammars**

Example applications of formal languages and automata

- **HTML and web browsers**
- **speech recognition and understanding grammars**
- **dialog systems and AI (Siri, Watson)**
- **regular expression matching**
- **compilers and interpreters of programming languages**

- **An alphabet Σ is a finite, non-empty set of characters (symbols):**

$$\Sigma = \{c_1, \cdots, c_n\}. \tag{1}$$

- **examples:**
  1. **The alphabet $\Sigma_{\mathrm{bin}} = \{0, 1\}$ can express integers in the binary system.**
  2. **The English language is based on the alphabet $\Sigma_{\mathrm{en}} = \{\mathtt{a}, \cdots, \mathtt{z}, \mathtt{A}, \cdots, \mathtt{Z}\}$.**
  3. **The alphabet $\Sigma_{\mathrm{ASCII}} = \{0, \cdots, 127\}$ represents the set of ASCII characters [American Standard Code for Information Interchange] coding letters, digits, and special and control characters.**

Alphabets: ASCII code chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

- A **word** of the alphabet Σ is a sequence (list) of symbols of Σ:

$$w = c_1 \cdots c_n \quad \text{with} \quad c_1, \ldots, c_n \in \Sigma. \tag{2}$$

- The **empty word** is written as

$$w = \varepsilon. \tag{3}$$

- The set of all words of an alphabet Σ is represented by Σ^*.
- In programming languages, words are also referred to as **strings**.
- examples:
  1. Using the aforementioned set Σ_{bin}, we can define the words

$$w_1 = 01100 \quad \text{and} \quad w_2 = 11001 \quad \text{with} \quad w_1, w_2 \in \Sigma^*_{\mathrm{bin}}. \tag{4}$$

  2. Using the aforementioned set Σ_{en}, we can define the word

$$w = \texttt{example} \quad \text{with} \quad w \in \Sigma^*_{\mathrm{en}}. \tag{5}$$

- **We refer to the length of a word w as $|w|$, e.g.:**

$$w = \texttt{example} \quad \textbf{with} \quad w \in \Sigma_{\mathrm{en}}^* \longrightarrow |w| = 7. \tag{6}$$

- **We access individual symbols within words using the terminology**

$$w[i] \quad \textbf{with} \quad i \in \{1, 2, \cdots, |w|\}. \tag{7}$$

- **We define the concatenation of the words $w_1, w_2, \ldots, w_n$ as**

$$w = w_1 w_2 \cdots w_n. \tag{8}$$

- **concatenation example:**

$$w_1 = 01 \quad \textbf{and} \quad w_2 = 10 \longrightarrow$$
$$w_1 w_2 = 0110 \quad \textbf{and} \quad w_2 w_1 = 1001. \tag{9}$$

(Concatenation) power of a word

- **The nth power of a word w concatenates the same word n times:**

$$w^n = w^{n-1}w \quad \text{with} \quad w^0 = \varepsilon \quad \text{and} \quad n \in \mathbb{I}, n \neq 0. \tag{10}$$

- **In the following, we will be frequently using the set of integers**

$$\mathbb{I} = \{0, 1, \cdots\}. \tag{11}$$

- **Given the alphabet Σ, we refer to the subset $L \subseteq \Sigma^*$ as formal language.**

- **examples:**

  1. **We define**

  $$L_{\mathbb{I}} = \{1w | w \in \Sigma^*_{\mathrm{bin}}\} \cup \{0\}. \qquad (12)$$

  Then, $L_{\mathbb{I}}$ is the set of all those words that represent integers using the binary system (all words starting with 1 and the word 0. Hence, we have

  $$100 \in L_{\mathbb{I}} \quad \textbf{but} \quad 010 \notin L_{\mathbb{I}}. \qquad (13)$$

2. **We define the function**

$$d : L_{\mathbb{I}} \longrightarrow \mathbb{I} \tag{14}$$

as the function returning the decimal-system representation of a word in the language $L_{\mathbb{I}}$. This gives us

(a) $d(0) = 0$,

(b) $d(1) = 1$,

(c) $d(w0) = 2d(w)$ **for** $|w| > 0$,

(d) $d(w1) = 2d(w) + 1$ **for** $|w| > 0$.

Formal languages: examples (cont.)

3. **We define the language $L_{\mathbb{P}}$ as the language representing prime numbers in the binary system:**

$$L_{\mathbb{P}} = \{w \in L_{\mathbb{I}} | d(w) \in \mathbb{P}\}. \tag{15}$$

One way to formally express the set of all prime numbers is

$$\mathbb{P} = \{p \in \mathbb{I} | \{t \in \mathbb{I} | \exists k \in \mathbb{I} : kt = p\} = \{1, p\}\}. \tag{16}$$

4. **We define the language $L_C \subset \Sigma^*_{\mathrm{ASCII}}$ as the set of all** `C` **functions with a declaration of the form**

$$\texttt{char*}\ f(\texttt{char*}\ x); \quad (17)$$

that is, L_C contains the ASCII code of all those `C` **functions processing and returning a string.**

5. **Using the alphabet $\Sigma_{\mathrm{ASCII}+} = \Sigma_{\mathrm{ASCII}} \cup \{\dagger\}$, we define the universal language**

$$L_u = \{f\dagger x\dagger y\} \quad \text{with} \tag{18}$$

   (a) $f \in L_C$,
   (b) $x, y \in \Sigma^*_{\mathrm{ASCII}}$,
   (c) applying f to x terminates and returns y.

- **These examples show that formal languages have a wide scope.**
- **Testing whether a word belongs to $L_{\mathbb{I}}$ is straightforward whereas the same test for $L_{\mathbb{P}}$ or L_C is more complicated.**
- **Later, we will see that there is no algorithm to do this test for L_u.**

- **Given an alphabet Σ and the formal languages $L_1, L_2 \subseteq \Sigma^*$, we define the product**

$$L_1 \cdot L_2 = \{w_1 w_2 | w_1 \in L_1, w_2 \in L_2\}. \tag{19}$$

- **example:**

Using the alphabet Σ_{en}, we define the languages

$$L_1 = \{\mathtt{ab}, \mathtt{bc}\} \quad \textbf{and} \quad L_2 = \{\mathtt{ac}, \mathtt{cb}\}. \tag{20}$$

The product is

$$L_1 \cdot L_2 = \{\mathtt{abac}, \mathtt{abcb}, \mathtt{bcac}, \mathtt{bccb}\}. \tag{21}$$

(Concatenation) power of a language

- **Given an alphabet Σ, the formal language $L \subseteq \Sigma^*$, and the integer $n \in \mathbb{I}$, we define the nth power of L (recursively) as**

$$L^n = L^{n-1} \cdot L \quad \text{with} \quad L^0 = \{\varepsilon\}. \tag{22}$$

- **Using the alphabet Σ_{en}, we define the language**

$$L = \{\mathtt{ab}, \mathtt{ba}\}. \tag{23}$$

This gives us

$$L^0 = \{\varepsilon\},$$

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

$$L^2 = \{\mathtt{ab}, \mathtt{ba}\} \cdot \{\mathtt{ab}, \mathtt{ba}\} = \{\mathtt{abab}, \mathtt{abba}, \mathtt{baab}, \mathtt{baba}\}. \tag{24}$$

- **Given an alphabet Σ and a formal language $L \subseteq \Sigma^*$, we define the Kleene star as**

$$L^* = \bigcup_{n \in \mathbb{I}} L^n. \tag{25}$$

- **example:**

Using the alphabet Σ_{en}, we define the language

$$L = \{\mathtt{a}\}. \tag{26}$$

This gives us

$$L^* = \{\mathtt{a}^n | n \in \mathbb{I}\}. \tag{27}$$

- **Given the alphabet Σ_{bin} and the language**

$$L = \{1\}. \tag{28}$$

a) Formally describe the language

$$L' = L^* \backslash \{\varepsilon\}. \tag{29}$$

b) Formally describe the set

$$D = \{d(w) | w \in L'\}. \tag{30}$$

c) Formally describe the language

$$L'_{-} = \{w | w - 1 \in L'\}. \tag{31}$$

d) Formally describe the language

$$L'_{+} = \{w | w + 1 \in L'\}. \tag{32}$$

Hint: Here, the operators $+$ and $-$ perform addition and substraction of binary numbers.

Outline

1. **introduction**
2. **regular expressions**
   - **compact description of sets of strings**
   - **fundamental component of script languages (Perl, Python, grep, sed, awk, etc.) and of most modern programming languages (.NET, SQL Server 2008, Java, etc.)**
3. **the scanner generator JFlex**
4. **finite-state machines**

   ...are able to detect regular expressions
5. **formal grammars**

- **live demonstration:**
  - **Vi**
  - **extract e-mail or IP addresses from large numbers of files**

- **Using the alphabet Σ, we refer to the set of all regular expressions as R.**
- **We introduce a function**

$$L : R \rightarrow 2^{\Sigma^*} \tag{33}$$

assigning a formal language $L(r) \subseteq \Sigma^*$ to each regular expression r.

- **Here, 2^S denotes the power set of a set S.**
- **E.g.,**

$$2^{\Sigma_{\rm bin}} = 2^{\{0,1\}} = \{\emptyset, \{0\}, \{1\}, \{0,1\}\}, \tag{34}$$

and

$$\begin{aligned} 2^{\Sigma^*_{\rm bin}} &= 2^{\{\varepsilon,0,1,00,01,\ldots\}} \\ &= \{\emptyset, \{\varepsilon\}, \{0\}, \{1\}, \{00\}, \{01\}, \ldots \\ &\quad \ldots \{\varepsilon, 0\}, \{\varepsilon, 1\}, \{\varepsilon, 00\}, \{\varepsilon, 01\}, \ldots \\ &\quad \ldots \{010, 1110, 10101\}, \ldots\}. \end{aligned} \tag{35}$$

- **The set of regular expressions (R) is defined as follows:**

1. **The regular expression $\emptyset$ is associated with the empty language:**

$$L(\emptyset) = \{\} \quad \text{with} \quad \emptyset \in R. \tag{36}$$

2. **The regular expression ε is associated with the language containing only the empty word:**

$$L(\varepsilon) = \{\varepsilon\} \quad \text{with} \quad \varepsilon \in R. \tag{37}$$

3. **Each symbol in the alphabet Σ is also a regular expression:**

$$c \in \Sigma \longrightarrow c \in R;$$
$$L(c) = \{c\}. \tag{38}$$

4. **We define the infix operator "$+$" generating new regular expressions by merging the languages of the regular expressions r_1 and r_2:**

$$r_1 \in R, r_2 \in R \longrightarrow r_1 + r_2 \in R;$$
$$L(r_1 + r_2) = L(r_1) \cup L(r_2). \tag{39}$$

The set of regular expressions (cont.)

5. **We define the infix operator "·" generating new regular expressions using the product of the languages representing the regular expressions r_1 and r_2:**

$$r_1 \in R, r_2 \in R \longrightarrow r_1 \cdot r_2 \in R;$$
$$L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2). \quad (40)$$

6. **We define the Kleene star of the language representing a regular expression r:**

$$r \in R \longrightarrow r^* \in R;$$
$$L(r^*) = L^*(r). \quad (41)$$

7. **Brackets can be used to group regular expressions without changing them:**

$$r \in R \longrightarrow (r) \in R;$$
$$L((r)) = L(r). \quad (42)$$

Operator precedences

- **To save brackets, we introduce the following operator precedences:**

  I. **“(”, “)” (strongest)**

  II. **“$*$”**

  III. **“$\cdot$”**

  IV. **“$+$” (weakest)**

- **example:**

$$a + b \cdot c^* = a + (b \cdot (c^*)). \tag{43}$$

- **For the sake of further simplicity, the product operator “$\cdot$” can also be omitted, e.g.:**

$$a + b \cdot c^* = a + bc^*. \tag{44}$$

- **For all the following examples, we are using the alphabet**

$$\Sigma_{\mathbf{abc}} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}. \tag{45}$$

1. **The regular expression**

$$r_1 = (\mathtt{a} + \mathtt{b} + \mathtt{c})(\mathtt{a} + \mathtt{b} + \mathtt{c}) \tag{46}$$

describes all the words of exactly two symbols:

$$L(r_1) = \{w \in \Sigma_{\mathbf{abc}}^* \big| |w| = 2\}. \tag{47}$$

2. **The regular expression**

$$r_2 = (\mathtt{a} + \mathtt{b} + \mathtt{c})(\mathtt{a} + \mathtt{b} + \mathtt{c})^* \tag{48}$$

describes all the words of one or more symbols:

$$L(r_1) = \{w \in \Sigma_{\mathbf{abc}}^* \big| |w| \geq 1\}. \tag{49}$$

3. **The regular expression**

$$r_3 = (\mathtt{b} + \mathtt{c})^*\mathtt{a}(\mathtt{b} + \mathtt{c})^* \quad (50)$$

describes all the words containing exactly one $\mathtt{a}$**:**

$$L(r_3) = \{w \in \Sigma^*_{\mathbf{abc}} \big| |\{i \in \mathbb{I} | w[i] = \mathtt{a}\}| = 1\} \quad (51)$$

where $|S|$ refers to the number of elements in a set S.

Regular expressions: exercise

a) **Using the alphabet $\Sigma_{\mathrm{abc}} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, give a regular expression r_a for all the words $w \in \Sigma_{\mathrm{abc}}^*$ containing exactly one a or exactly one b.**

b) **Which language is expressed by r_a?**

c) **Using the alphabet $\Sigma_{\mathrm{abc}} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, give a regular expression r_b for all the words containing at least one a and one b.**

d) **Using the alphabet $\Sigma_{\mathrm{bin}} = \{0, 1\}$, give a regular expression for all the words whose third last symbol is 1.**

e) **Using the alphabet Σ_{bin}, give a regular expression for all the words not containing the string 110.**

f) **Which language is expressed by the regular expression**

$$r_f = (1 + \varepsilon)(00^*1)^*0^*? \tag{52}$$

1. $r_1 + r_2 \doteq r_2 + r_1$ (commutative law)
   The symbol $\doteq$ means that the formal languages represented by these regular expressions are identical, i.e.:
   $$L(r_1 + r_2) = L(r_2 + r_1). \quad (53)$$
   This equivalence can be proven using the commutativity of merged sets:
   $$L(r_1 + r_2) = L(r_1) \cup L(r_2) = L(r_2) \cup L(r_1) = L(r_2 + r_1). \quad (54)$$
2. $(r_1 + r_2) + r_3 \doteq r_1 + (r_2 + r_3)$ (associative law)
3. $(r_1 r_2) r_3 \doteq r_1 (r_2 r_3)$ (associative law)
4. $\emptyset r \doteq \emptyset$
5. $\varepsilon r \doteq r$
6. $\emptyset + r \doteq r$
7. $(r_1 + r_2) r_3 \doteq r_1 r_3 + r_2 r_3$ (distributive law)
8. $r_1 (r_2 + r_3) \doteq r_1 r_2 + r_1 r_3$ (distributive law)

- **We want to prove that**

$$\emptyset r \doteq \emptyset. \tag{55}$$

- **According to Equation 53, to prove Equation 55, we have to show that**

$$L(\emptyset r) = L(\emptyset). \tag{56}$$

One way to do so is:

$$\begin{aligned} L(\emptyset r) &\overset{\text{Eq.40}}{=} L(\emptyset) \cdot L(r) \\ &\overset{\text{Eq.36}}{=} \emptyset \cdot L(r) \\ &\overset{\text{Eq.19}}{=} \{w_1 w_2 | w_1 \in \emptyset, w_2 \in L(r)\} \\ &= \emptyset \\ &\overset{\text{Eq.36}}{=} L(\emptyset) \;\square \end{aligned} \tag{57}$$

Algebraic operations on regular expressions (cont.)

9. $r + r \doteq r$

10. $(r^*)^* \doteq r^*$

11. $\emptyset^* \doteq \varepsilon$

12. $\varepsilon^* \doteq \varepsilon$

13. $r^* \doteq \varepsilon + r^* r$

14. $r^* \doteq (\varepsilon + r)^*$

15. $\left\{ r \doteq rs + t \quad \text{with} \quad \varepsilon \notin L(s) \right\} \longrightarrow r \doteq ts^*$ **(proof by Arto Salomaa)**

Algebraic operations on regular expressions: example for Rule 15

- **Using only the 15 algebraic operations, we want to prove that**

$$\varrho^*\varrho \doteq \varrho\varrho^* \quad \text{with} \quad \varrho \in R \quad \text{and} \quad \varepsilon \notin L(\varrho). \tag{58}$$

- **Setting**

$$r = \varrho^*\varrho, \tag{59}$$

$$s = \varrho, \tag{60}$$

$$t = \varrho, \tag{61}$$

we have

$$\begin{aligned} rs + t &= \varrho^*\varrho\varrho + \varrho \\ &\overset{5,7}{\doteq} (\varrho^*\varrho + \varepsilon)\varrho \\ &\overset{1,13}{\doteq} \varrho^*\varrho \\ &= r. \end{aligned} \tag{62}$$

- **This fulfills the conditions of Rule 15, leading to the conclusion**

$$\varrho^*\varrho = r \doteq ts^* = \varrho\varrho^* \quad \text{with} \quad \varepsilon \notin L(\varrho) \ \square \tag{63}$$

Algebraic operations on regular expressions: exercise

a) Simplify the following regular expression:

$$\boldsymbol{r} = 0(\varepsilon + 0 + 1)^* + (\varepsilon + 1)(1 + 0)^* + \varepsilon. \tag{64}$$

b) Prove the equivalence using only algebraic operations

$$\boldsymbol{r}^* \doteq \varepsilon + \boldsymbol{r}^*. \tag{65}$$

c) Prove the equivalence using only algebraic operations

$$10(10)^* \doteq 1(01)^*0. \tag{66}$$

d) Prove the equivalence

$$(1 + \varepsilon)(0(1 + \varepsilon))^*1^* \doteq (0 + 10)^*1^*. \tag{67}$$

Outline

1. **introduction**
2. **regular expressions**
   - **compact description of sets of strings**
   - **fundamental component of script languages (Perl, Python, grep, sed, awk, etc.) and of most modern programming languages (.NET, SQL Server 2008, Java, etc.)**
3. **the scanner generator JFlex**
4. **finite-state machines**
   ...are able to detect regular expressions
5. **formal grammars**

- **A scanner is a tool to split an input text into individual tokens.**
- **E.g., the scanner used for the `C` compiler distinguishes the following tokens:**
  1. **key words (`if`, `while`)**
  2. **operators (+, +=, <)**
  3. **constants:**
     a) **numbers (`123`, `1.23e-2`)**
     b) **strings in single quotes (`'abc'`)**
     c) **strings in double quotes (`"abc"`)**
  4. **variable, function, type names**
  5. **comments**
  6. **white space (blanks, tabs, newline, carriage return)**

Scanners: example

- **looking at the `C` expression**

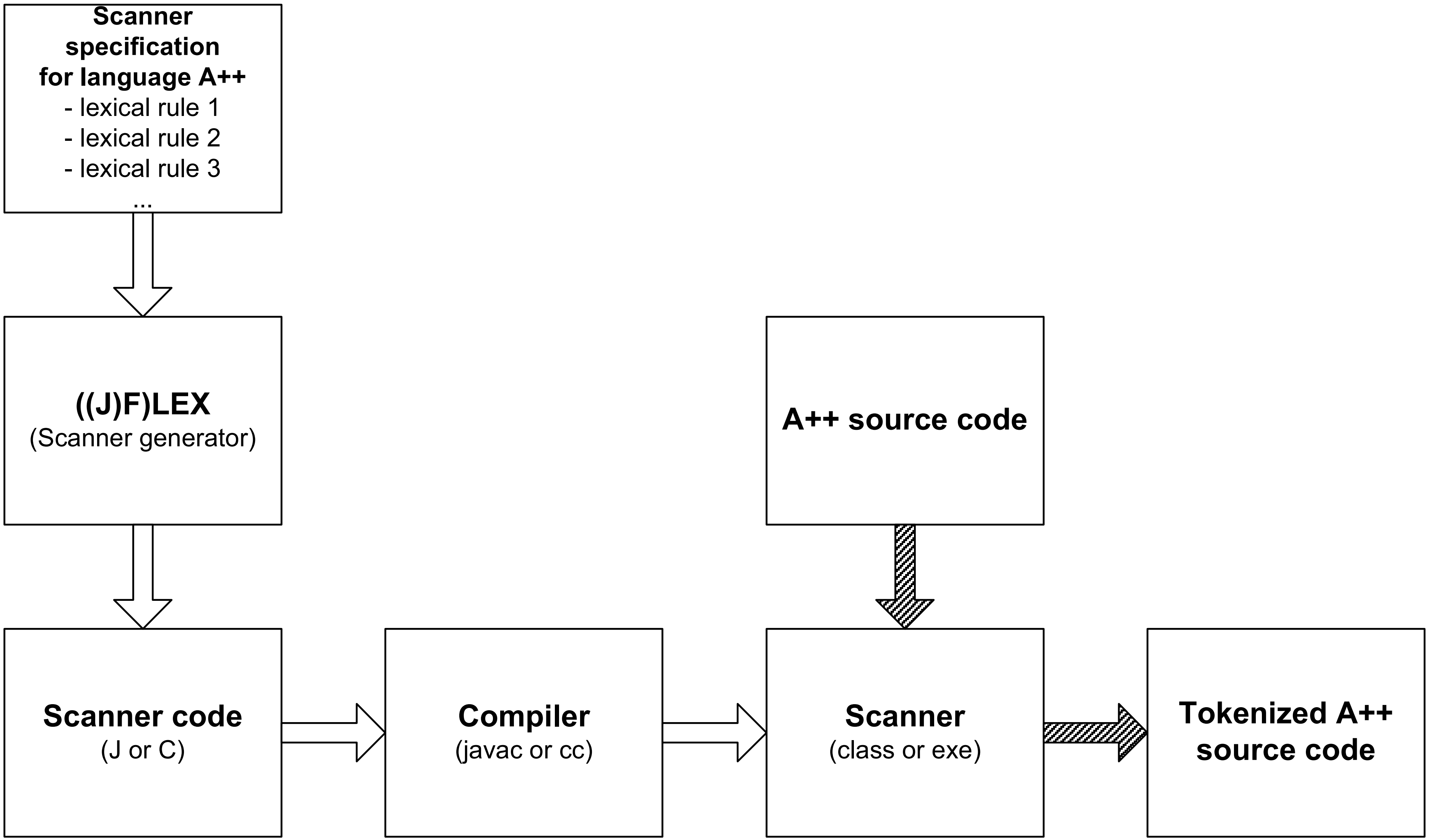
  `sum=3+2;`

- **This expression would be tokenized as**

token	token type
`sum`	**identifier**
=	**assignment operator**
3	**number**
+	**addition operator**
2	**number**
;	**end of statement**

- **JFlex is a scanner generator.**
- **Given a specification of token types, it automatically generates a scanner.**
- **Tokens types are specified by regular expressions.**
- **JFlex is a free, open-source software.**
- **JFlex is written in Java, i.e. it is platform-independent.**
- **The scanner JFlex produces is also a Java program.**

A scanner generator for the (hypothetical) language A++



- **Download and install the JDK, e.g. from**

  `http://jdk6.java.net/`

- **Download and install/unpack JFlex from**

  `http://jflex.de`

- **If you are running Windows/Cygwin, make sure your environment variables reflect the new installations.**

- **This can be done by editing the file** `profile` **which (depending on your specific folder structure) can be found, for example, in**

  `c:\cygwin\etc`

- **In particular,** `profile` **should contain entries similar to the following:**
  - **to add the location of the JDK:**

    ```
    export PATH=$PATH:/cygdrive/c/Program\
    Files/Java/jdk1.6.0_26/bin
    ```

  - **to add the location of JFlex:**

    ```
    export CLASSPATH=$CLASSPATH';c:\jflex\lib\JFlex.jar'
    ```

- **To test the proper installation, download and unpack the file package** `fla_*.zip` **from**

  `http://suendermann.com`

  and run the following command from a new Cygwin shell:

  ```
  java JFlex.Main example.flex
  javac Count.java
  java Count input.txt
  ```

- A JFlex specification consists of three parts:
  1. the user code contains
     - * package declarations
     - * import commands
  2. options and declarations
  3. lexical rules
     - * Regular expressions describe strings the scanner is supposed to recognize.
     - * It is also defined how the scanner processes these strings.
- These parts are separated by the string %% at the beginning of a line.

JFlex specifications: `example.flex`

```
%%

%class Count
%standalone
%unicode

%{
    int mCount = 0;
%}

%eof{
    System.out.println("Total: " + mCount);
%eof}

%%

[1-9][0-9]*  { mCount += new Integer(yytext()); }
.|\n         { /* skip */                        }
```

JFlex specifications: example (cont.)

- **generating the Java code of the scanner:**

```
$ java JFlex.Main example.flex
Reading "example.flex"
Constructing NFA : 12 states in NFA
Converting NFA to DFA :
...
5 states before minimization, 3 states in minimized DFA
Writing code to "Count.java"
```

- **...and compiling it:**

```
javac Count.java
```

JFlex specifications: example (cont.)

- **Our example scanner adds up all integers found in an input file.**
- **An example imput (**`input.txt`**) reads**

```
John has 3 apples and 5 oranges.
George bought 5 bananas.
How many fruits do they have altogether?
```

- **applying the scanner to this input**

```
java Count input.txt
```

- **... produces the output**

```
Total: 13
```

- **Let us discuss our example in more detail:**
- **Line 3**
  specifies the scanner class's name (`Count`).
- **Line 4**
  The option `%standalone` means that the generated program is not component of a parser but an individual app (stand-alone scanner). This is why the class `Count` comes with the method `main()`.
- **Lines 7 to 9**
  Using the key words `%{` and `%}`, we define the variable `mCount`. Here, we can also define additional methods.
- **Lines 11 to 14**
  Using the key words `%eof{` and `%eof}`, we define a command to be executed when reaching the end of file.

JFlex specifications: example (cont.)

- **Lines 17 and 18**
  contain the scanner rules. A rule has the form

  regex{*action*}

  – *regex* **is a regular expression**
  – *action* **is Java code to be executed when** *regex* **was found.**

- **Line 17**
  `[1-9][0-9]*` **matches an integer whose string can be accessed by the function** `yytext()`**.**

- **Line 18**
  `.|\n` **matches any character except newline (**`.`**) or (|) newline (**`\n`**). This line is necessary since standalone scanners print all characters not matched by a rule to stdout.**

- **The minimal syntax of regular expressions as discussed before was introduced to be able to show their equivalence to finite state machines (as done later on).**
- **Practical implementations of regular expressions (e.g. in JFlex) use a richer and more powerful syntax.**
- **Regular expressions in JFlex are based on the ASCII alphabet.**
- **We distinguish between the set of operator symbols**

$$O = \{\texttt{.}, \texttt{*}, \texttt{+}, \texttt{?}, \texttt{!}, \texttt{-}, \texttt{\~{}}, \texttt{|}, \texttt{(}, \texttt{)}, \texttt{[}, \texttt{]}, \texttt{\{}, \texttt{\}}, \texttt{<}, \texttt{>}, \texttt{/}, \texttt{\textbackslash}, \texttt{\^{}}, \texttt{\$}, \texttt{"}\} \quad (68)$$

and the set of regular expressions

1. $c \in \Sigma_{\mathrm{ASCII}} \backslash O \longrightarrow c \in R$
2. "." $\in R$

   any character but newline (`\n`)

3. $x \in \{\texttt{a}, \texttt{b}, \texttt{f}, \texttt{n}, \texttt{r}, \texttt{t}, \texttt{v}\} \longrightarrow \backslash x \in R$
defines the following control characters

- `\a` **(alert)**
- `\b` **(backspace)**
- `\f` **(form feed)**
- `\n` **(newline)**
- `\r` **(carriage return)**
- `\t` **(tabulator)**
- `\v` **(vertical tabulator)**

4. $a, b, c \in \{\texttt{0}, \cdots, \texttt{7}\} \longrightarrow \backslash abc \in R$ **octal representation of a character's ASCII code (e.g.** `\040` **represents the empty space " ")**

ASCII code chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Regular expressions in JFlex (cont.)

5. $c \in O \longrightarrow \backslash c \in R$
   escaping operator symbols
6. $r_1, r_2 \in R \longrightarrow r_1 r_2 \in R$
   concatenation
7. $r_1, r_2 \in R \longrightarrow r_1 \,|\, r_2 \in R$
   infix operation using "|" rather than "+"
8. $r \in R \longrightarrow r* \in R$
   Kleene star
9. $r \in R \longrightarrow r+ \in R$
   variation of the Kleene star:
   $$r+ = rr* \tag{69}$$
10. $r \in R \longrightarrow r? \in R$
    optional presence:
    $$r? = r \,|\, \varepsilon \tag{70}$$

11. $r \in R, n \in \mathbb{I} \longrightarrow r\{n\} \in R$

12. $r \in R;\ m, n \in \mathbb{I};\ m \leq n \longrightarrow r\{m, n\} \in R$
concatenation of between m and n times r

13. $r \in R \longrightarrow \hat{}r \in R$
r has to be at the beginning of line

14. $r \in R \longrightarrow r\$ \in R$
r has to be at the end of line

15. $r_1, r_2 \in R \longrightarrow r_1/r_2 \in R$
The same as r_1r_2, however, the method `yytext()` returns only the contents of r_1. The trailing context r_2 can be processed by the next rule. For an example, see `exampleTrailingContext.flex`.

16. $r \in R \longrightarrow (r) \in R$
Grouping regular expressions with brackets.

17. **Ranges**
    - `[aeiou]` $\doteq$ `a|e|i|o|u`
    - `[a-z]` $\doteq$ `a|b|c|` $\cdots$ `|z`
    - `[a-zA-Z0-9]`: **alphanumeric characters**
    - `[^0-9]`: **all ASCII characters w/o digits**
18. `[ ]` $\in R$

    empty space
19. `[^]` $\in R$

    any character
20. $w \in \{\Sigma_{\mathrm{ASCII}} \backslash \{\backslash, "\}\}^* \longrightarrow "w" \in R$

    verbatim text
21. $r \in R \longrightarrow\ !r \in R$

    negation

22. $r \in R \longrightarrow \tilde{}r \in R$

    The upto operator matches the shortest string ending with r.

23. **predefined character classes**

    `[:jletter:]` **matches characters c for which calling the Java method** `Character.isJavaIdentifierStart(`c`)` **returns** `true`

    `[:jletterdigit:]` $\longleftrightarrow$ `isJavaIdentifierPart()`

    `[:letter:]` $\longleftrightarrow$ `isLetter()`

    `[:digit:]` $\longleftrightarrow$ `isDigit()`

    `[:uppercase:]` $\longleftrightarrow$ `isUppercase()`

    `[:lowercase:]` $\longleftrightarrow$ `isLowercase()`

Regular expressions in JFlex: precedences

I. “`(`”, “`)`” (strongest)

II. “`*`”, “`+`”, “`?`”

III. “`!`”

IV. concatenation

V. “`|`” (weakest)

example:

`!a*b|c+de` $\doteq$ `(((!(a*))b)|(((c+)d)e))`

1. `[a-zA-Z][a-zA-Z0-9_]*`
   typical variable names in programming languages

2. `0|[1-9][0-9]*`
   integer

3. `\/\/.*`
   `C++` **comment (one-liner)**

4. `"/*" !([^]* "*/" [^]*) "*/"`
   `C` **comment**

5. `"/*" ~ "*/"`
   `C` **comment (using the upto operator)**

6. `!(!`r_1`|!`r_2`)`
   intersection of two regular expressions using de Morgan's law

$$r_1 \wedge r_2 \leftrightarrow \neg(\neg r_1 \vee \neg r_2) \tag{71}$$

   example: $r_1 =$ `[ab]{3}`, $r_2 =$ `a*`

1. **write a JFlex program removing `C` and `C++` comments from an input source**
2. **write a JFlex program extracting the plain text from an HTLM source**
3. **write a JFlex program computing average exam scores per student from a score sheet (`exam.txt`):**

```
Exam: Formal Languages and Automata

Exercise:                  1. 2. 3. 4. 5. 6.
Ronald Reagan:             9 12 10  6  6  0
Arnold Schwarzenegger:     4  4  2  0  -  -
James Dean:                9 12 12  9  9  6
```

using the formula

$$\texttt{avgScore} = \mathbf{5} - \mathbf{4} \cdot \frac{\texttt{sumPoints}}{\texttt{maxPoints}} \quad \textbf{with} \quad \texttt{maxPoints} = \mathbf{60.} \tag{72}$$

Outline

1. **introduction**
2. **regular expressions**
   - **compact description of sets of strings**
   - **fundamental component of script languages (Perl, Python, grep, sed, awk, etc.) and of most modern programming languages (.NET, SQL Server 2008, Java, etc.)**
3. **the scanner generator JFlex**
4. **finite-state machines**

   ...are able to detect regular expressions
5. **formal grammars**

Finite-state machines: introduction

- **We will introduce finite state machines (FSMs) and show how a regular expression can be converted into an FSM and the other way around.**

- **We will see that FSMs can be deterministic or non-deterministic which can be transformed into each other.**

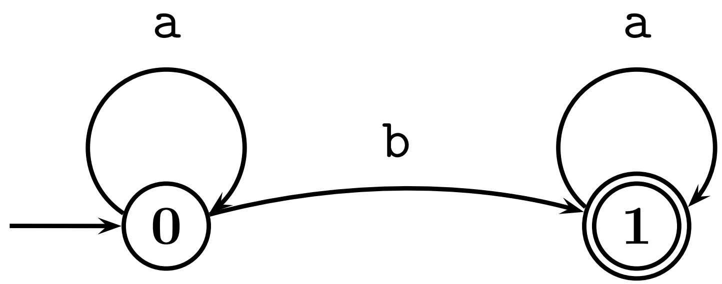
- **The purpose of the FSMs discussed in the following is**
  - **to read a string and**
  - **to decide whether the string is element of the language represented by the FSM.**
- **The output of these FSMs is binary:** `true` **or** `false`**.**
- **As its name implies, FSMs have a finite (i.e., fixed) number of states.**

FSMs: working principle

1. **In the beginning, the FSM is in an initial state.**
2. **For every input $c \in \Sigma$, the FSM changes to a new state depending on c and the current state.**
3. **After reading the entire input string, the FSM is in a certain state. If this state belongs to the set of so-called final (or accept) states the string is element of the accepted language.**

FSMs: example

- **a simple FSM recognizing the regular expression** `a*ba*`



- **This FSM has two states, 0 and 1.**
- **0 is the initial state (with an arrow "pointing at it from anywhere" (Sipser, 2006))**
- **1 is a final state (represented as a double circle)**

- **An FSM is a quintuple**

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle \tag{73}$$

with the following components

1. **Q is the finite set of states.**
2. **Σ is the input alphabet.**
3. **$\delta : Q \times \Sigma \rightarrow Q \cup \{\Omega\}$ is the state-transition function. If $\delta(q, c) = \Omega$, the FSM announces an error, i.e. rejects the input.**
4. **$q_0 \in Q$ is the initial state.**
5. **$F \subseteq Q$ is the set of final states.**

- **Using the above mentioned example, the FSM is expressed as**

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle \tag{74}$$

with

1. $Q = \{0, 1\}$
2. $\Sigma = \{\mathtt{a}, \mathtt{b}\}$
3. $\delta(0, \mathtt{a}) = 0; \delta(0, \mathtt{b}) = 1; \delta(1, \mathtt{a}) = 1; \delta(1, \mathtt{b}) = \Omega$
4. $q_0 = 0$
5. $F = \{1\}$

Language accepted by an FSM

- **In order to formally define the language accepted by an FSM, we generalize the state transition function δ to a function**

$$\delta' : Q \times \Sigma^* \rightarrow Q \cup \{\Omega\} \tag{75}$$

  whose second argument is a string.

- **We define**
  - $\delta'(q, \varepsilon) = q$
  - $\delta'(q, w) = \begin{cases} \delta'(\delta(q, c), v) & \text{if} \quad \delta(q, c) \neq \Omega \\ \Omega & \text{otherwise} \end{cases}$

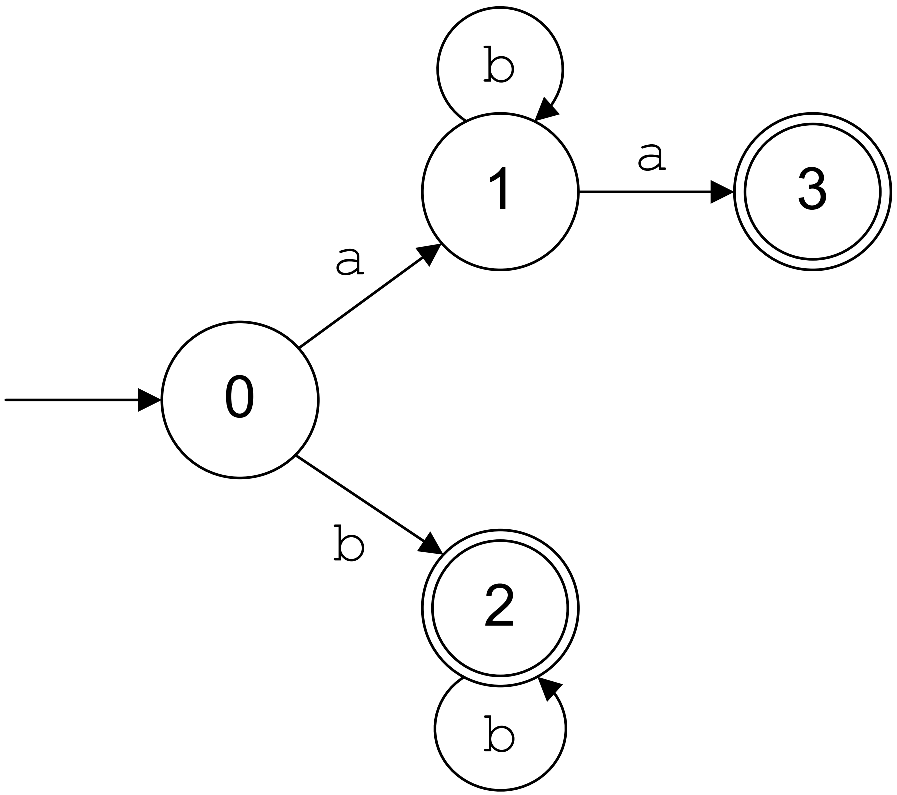
  with $\quad w = cv; c \in \Sigma; v \in \Sigma^* \quad$ **for** $\quad |w| > 0$

- **E.g., we can show that $\delta'(0, \texttt{aba}) = 1$ for the above example.**

- **the language accepted by an FSM $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ (aka regular language) is defined as**

$$L(A) = \{w \in \Sigma^* | \delta'(q_0, w) \in F\}. \tag{76}$$

1. **We are given this graphical representation of an FSM A:**



a) Give a regular expression describing $L(A)$.

b) Give a formal definition of A.

2. Give

- **a regular expression,**
- **a graphical representation, and**
- **a formal definition**

of a deterministic FSM A whose language $L(A) \subset \{\mathtt{a}, \mathtt{b}\}^*$ contains all those words featuring the substring `ab`

a) at the beginning,

b) at arbitrary position,

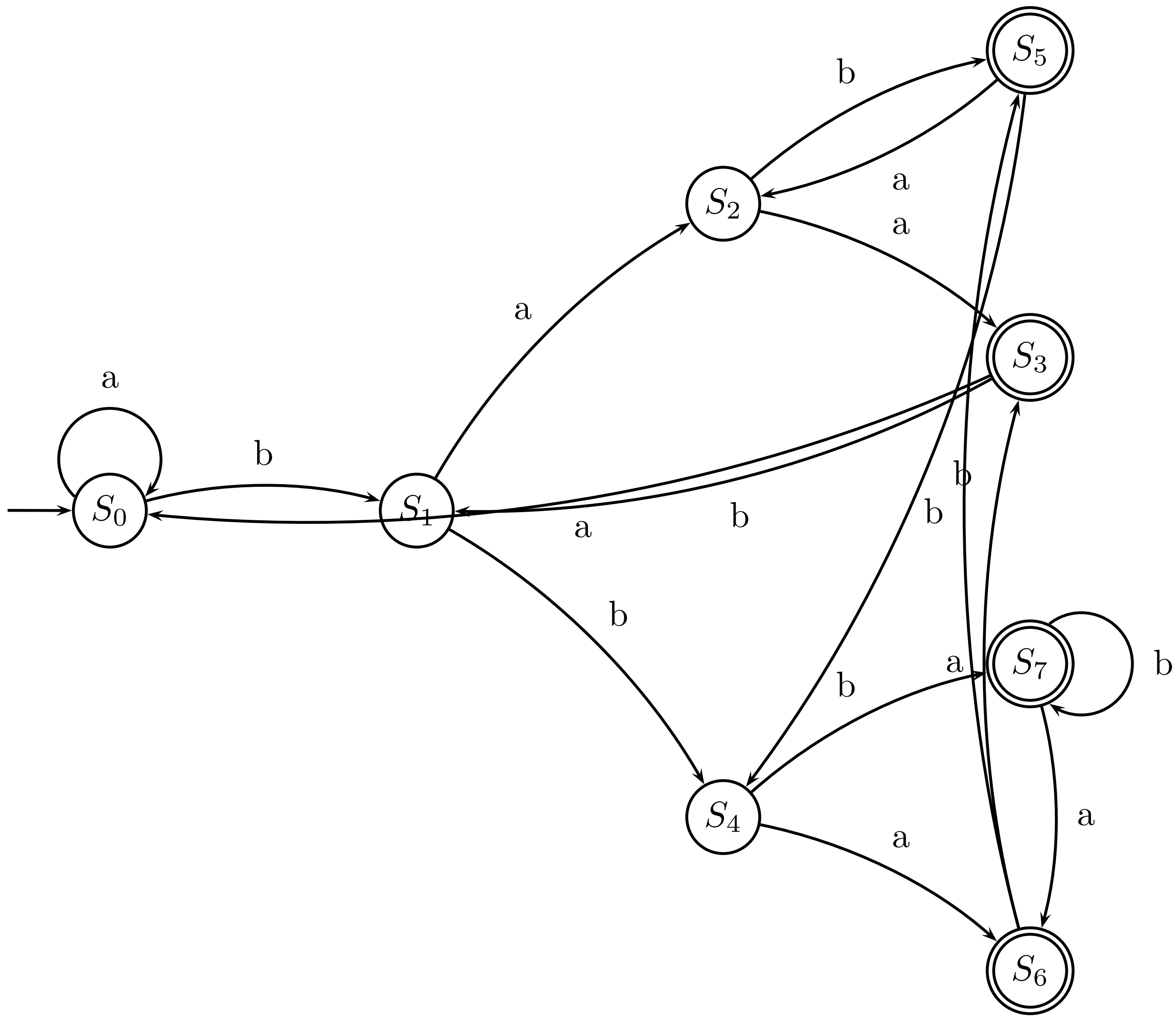
c) at the end.

FSM: determinism

- **So far, we have discussed deterministic FSMs, i.e. every state has exactly one transition for every possible input.**
- **We also refer to deterministic FSMs as deterministic finite automata (DFAs).**
- **Often, DFAs can be rather complex as in the following example accepting a language specified by the regular expression**

$$(\mathtt{a}+\mathtt{b})^*\mathtt{b}(\mathtt{a}+\mathtt{b})(\mathtt{a}+\mathtt{b}) \tag{77}$$

FSM: example of a DFA



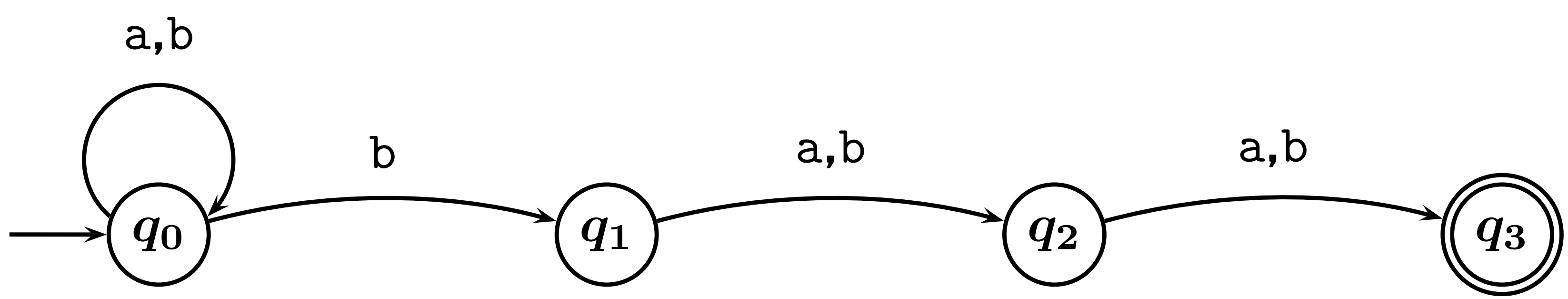
FSM: determinism (cont.)

- **We can simplify such an FSM when we permit that an input can lead to**
  - **one transition,**
  - **multiple transitions, or**
  - **no transition.**
- **That is, an FSM selects its next state from a set of states where the set depends on the current state and the input.**
- **We call this a non-deterministic FSM or non-deterministic finite automaton (NFA).**
- **For the same example with the regular expression**

$$(\mathtt{a}+\mathtt{b})^*\mathtt{b}(\mathtt{a}+\mathtt{b})(\mathtt{a}+\mathtt{b}) \tag{78}$$

...

- **...we get the following NFA:**



- **This FSM is non-deterministic since, in state q_0 with the input `b`, the FSM has to "guess" the next state.**

- **An example string `abab` can be read in three ways:**

  1. $q_0 \overset{a}{\mapsto} q_0 \overset{b}{\mapsto} q_0 \overset{a}{\mapsto} q_0 \overset{b}{\mapsto} q_0$ **(failure)**
  2. $q_0 \overset{a}{\mapsto} q_0 \overset{b}{\mapsto} q_0 \overset{a}{\mapsto} q_0 \overset{b}{\mapsto} q_1$ **(failure)**
  3. $q_0 \overset{a}{\mapsto} q_0 \overset{b}{\mapsto} q_1 \overset{a}{\mapsto} q_2 \overset{b}{\mapsto} q_3$ **(success)**

- **Even though NFAs seem to be based on guesswork, in the following, we will see that they are as powerful as DFAs.**

- **For the formal description of an NFA, we introduce the spontaneous transition, i.e., state changes without reading an input symbol:**

$$q_1 \stackrel{\varepsilon}{\mapsto} q_2. \tag{79}$$

- **An NFA is a quintuple**

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle \tag{80}$$

with the following components

1. **Q is the finite set of states.**
2. **Σ is the input alphabet.**
3. **δ is a relation on $Q \times \{\Sigma \cup \{\varepsilon\}\} \times Q$. I.e.,**

$$\delta \subseteq Q \times \{\Sigma \cup \{\varepsilon\}\} \times Q \tag{81}$$

4. **$q_0 \in Q$ is the initial state.**
5. **$F \subseteq Q$ is the set of final states.**

- **The above mentioned NFA example is expressed as**

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle \tag{82}$$

with

1. $Q = \{q_0, q_1, q_2, q_3\}$
2. $\Sigma = \{\mathtt{a}, \mathtt{b}\}$
3. $\delta = \{\langle q_0, \mathtt{a}, q_0\rangle, \langle q_0, \mathtt{b}, q_0\rangle, \langle q_0, \mathtt{b}, q_1\rangle, \langle q_1, \mathtt{a}, q_2\rangle, \langle q_1, \mathtt{b}, q_2\rangle, \langle q_2, \mathtt{a}, q_3\rangle, \langle q_2, \mathtt{b}, q_3\rangle\}$
4. $q_0 = q_0$
5. $F = \{q_3\}$

NFA: exercise

- **Given an FSM A whose language $L(A) \subset \{\mathtt{a}, \mathtt{b}\}^*$ contains all those words featuring the substring** `aba`**, what is**
  - **a regular expression representing $L(A)$,**
  - **a graphical representation of A,**
  - **a formal definition of A?**

- **Now, we want to show that an NFA A can be transformed to a DFA $\det(A)$ sharing the same language, i.e.**

$$L(A) = L(\det(A)) \tag{83}$$

- **The idea is that $\det(A)$ computes the set of all the states A can assume.**
- **A set M of states of A is a final state of $\det(A)$ if M contains a final state of A.**
- **To show this, we define three auxiliary functions.**
- **First, the ε closure**

$$ec : Q \rightarrow 2^Q \tag{84}$$

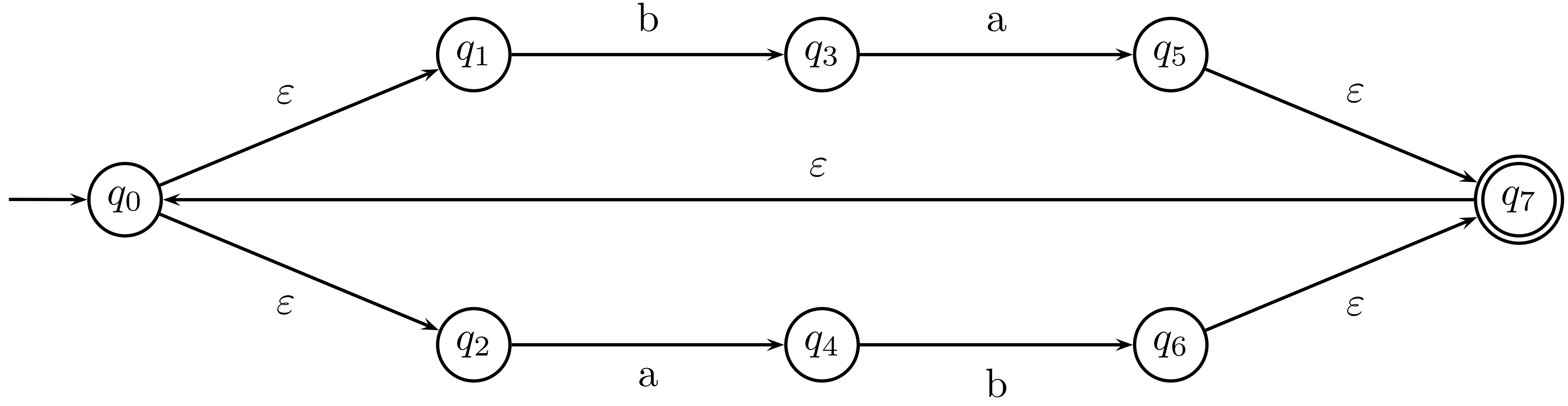
returns the set of all those states, the NFA can change to by means of an ε transition coming from state q.

- **Formal definition of ec:**

$$q \in ec(q); \tag{85}$$

$$p \in ec(q) \wedge \langle p, \varepsilon, r \rangle \in \delta \quad \rightarrow \quad r \in \ ec(q). \tag{86}$$

- **an example NFA with ε transitions:**



- **calculating the ε closure for all states:**
  - $ec(q_0) = \{q_0, q_1, q_2\}$,
  - $ec(q_1) = \{q_1\}$,
  - $ec(q_2) = \{q_2\}$,
  - $ec(q_3) = \{q_3\}$,
  - $ec(q_4) = \{q_4\}$,
  - $ec(q_5) = \{q_5, q_7, q_0, q_1, q_2\}$,
  - $ec(q_6) = \{q_6, q_7, q_0, q_1, q_2\}$.
  - $ec(q_7) = \{q_7, q_0, q_1, q_2\}$.

- **Second, we transform the relation δ into a function**

$$\delta^* : Q \times \Sigma \rightarrow 2^Q. \tag{87}$$

- **Here, $\delta^*(q, c)$ returns the set of all those states, the NFA can change to coming from state q reading the symbol c followed by any number of ε transitions.**

- **Formally, we have**

$$\delta^*(q_1, c) = \bigcup_{q_2 \in Q:\ \langle q_1, c, q_2 \rangle \in \delta} ec(q_2). \tag{88}$$

- **examples (based on the above NFA):**

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

- **Third, we transform the function δ^* into a function**

$$\Delta^* : 2^Q \times \Sigma \rightarrow 2^Q. \tag{89}$$

- **Here, $\Delta^*(M, c)$ returns the set of all those states, the NFA can change to coming from a set of states M reading the symbol c followed by any number of ε transitions.**

- **Formally, we have**

$$\Delta^*(M, c) = \bigcup_{q \in M} \delta^*(q, c). \tag{90}$$

- **examples (based on the above NFA):**

1. $\Delta^*(\{q_0, q_1, q_2\}, \mathtt{a}) = \{q_4\}$,
2. $\Delta^*(\{q_3\}, \mathtt{a}) = \{q_5, q_7, q_0, q_1, q_2\}$,
3. $\Delta^*(\{q_3\}, \mathtt{b}) = \{\}$,

- **We are now ready to transform an NFA A into a DFA:**

$$\det(A) = \langle 2^Q, \Sigma, \Delta^*, ec(q_0), \hat{F} \rangle \qquad (91)$$

with

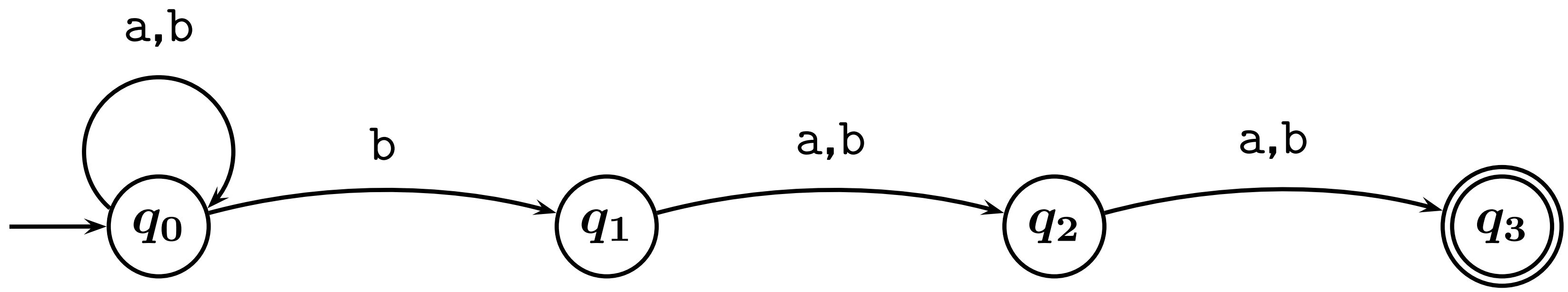
$$\hat{F} = \{M \in 2^Q | M \cap F \neq \{\}\}. \qquad (92)$$

- **That is, the set of final states $\hat{F}$ is the set of all subsets of Q containing a final state.**

Equivalence of DFA and NFA: example

- **returning to the example FSM expressing the regular expression**

$$(\mathtt{a}+\mathtt{b})^*\mathtt{b}(\mathtt{a}+\mathtt{b})(\mathtt{a}+\mathtt{b}) \tag{93}$$



- **The initial state:**

$$S_0 = ec(q_0) = \{q_0\}. \tag{94}$$

- **The state transition function: Starting with the initial state...**
  - $\Delta^*(\{q_0\}, \mathtt{a}) = \{q_0\} = S_0$.

- **exploring the set of states...**
  - $S_1 = \Delta^*(\{q_0\}, \mathtt{b}) = \{q_0, q_1\}$.
  - $S_2 = \Delta^*(\{q_0, q_1\}, \mathtt{a}) = \{q_0, q_2\}$.
  - $S_4 = \Delta^*(\{q_0, q_1\}, \mathtt{b}) = \{q_0, q_1, q_2\}$
  - $S_3 = \Delta^*(\{q_0, q_2\}, \mathtt{a}) = \{q_0, q_3\}$.
  - $S_5 = \Delta^*(\{q_0, q_2\}, \mathtt{b}) = \{q_0, q_1, q_3\}$.
  - $S_6 = \Delta^*(\{q_0, q_1, q_2\}, \mathtt{a}) = \{q_0, q_2, q_3\}$.
  - $S_7 = \Delta^*(\{q_0, q_1, q_2\}, \mathtt{b}) = \{q_0, q_1, q_2, q_3\}$.

- **transitions with repetitive states...**
  - $\boldsymbol{\Delta^*(\{q_0, q_3\}, \texttt{a}) = \{q_0\} = S_0}$.
  - $\boldsymbol{\Delta^*(\{q_0, q_3\}, \texttt{b}) = \{q_0, q_1\} = S_1}$.
  - $\boldsymbol{\Delta^*(\{q_0, q_1, q_3\}, \texttt{a}) = \{q_0, q_2\} = S_2}$.
  - $\boldsymbol{\Delta^*(\{q_0, q_1, q_3\}, \texttt{b}) = \{q_0, q_1, q_2\} = S_4}$.
  - $\boldsymbol{\Delta^*(\{q_0, q_2, q_3\}, \texttt{a}) = \{q_0, q_3\} = S_3}$.
  - $\boldsymbol{\Delta^*(\{q_0, q_2, q_3\}, \texttt{b}) = \{q_0, q_1, q_3\} = S_5}$.
  - $\boldsymbol{\Delta^*(\{q_0, q_1, q_2, q_3\}, \texttt{a}) = \{q_0, q_2, q_3\} = S_6}$.
  - $\boldsymbol{\Delta^*(\{q_0, q_1, q_2, q_3\}, \texttt{b}) = \{q_0, q_1, q_2, q_3\} = S_7}$.

- **Now, we can define the DFA**

$$\det(A) = \langle \hat{Q}, \Sigma, \Delta^*, S_0, \hat{F} \rangle \tag{95}$$

with

- **the set of states**

$$\hat{Q} = \{S_0, \cdots, S_7\}, \tag{96}$$

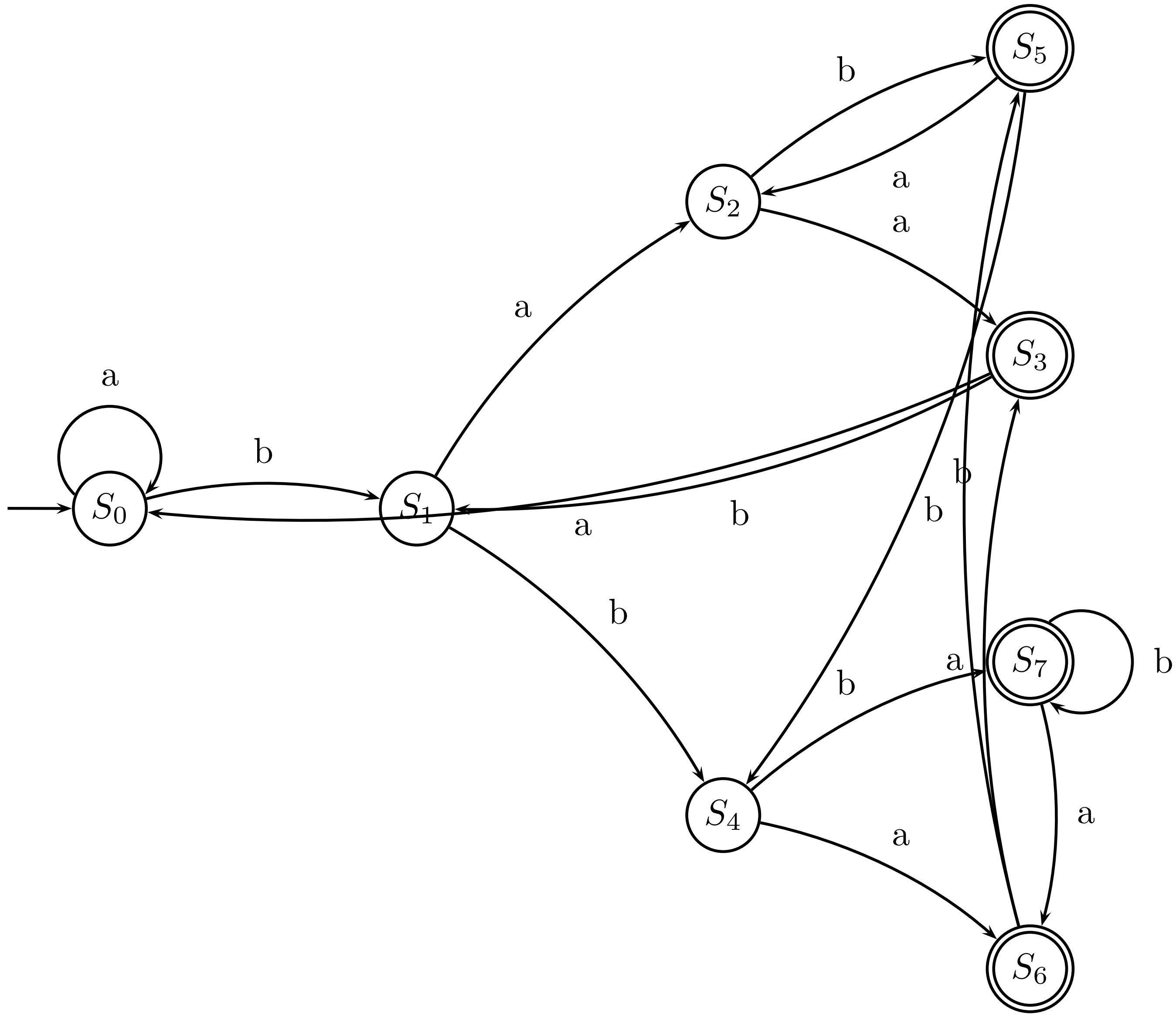
- **the state transition function Δ^* as summarized as follows:**

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

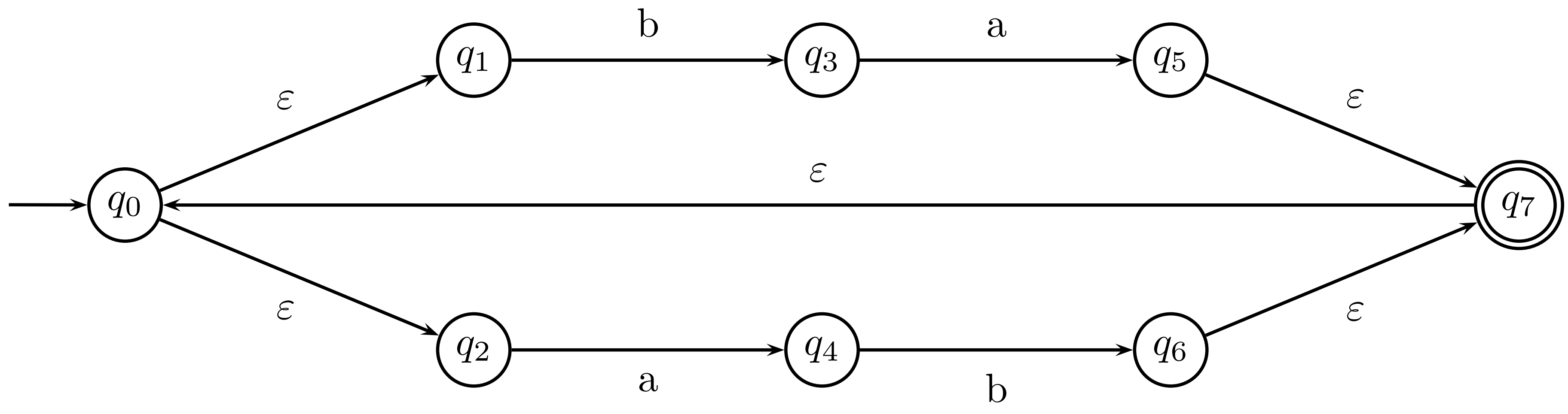
- **and the set of final states (each DFA state containing the NFA final state q_3)**

$$\hat{F} = \{S_3, S_5, S_6, S_7\}. \tag{97}$$

Equivalence of DFA and NFA: example (cont.)



- **We are given the following NFA A:**



a) Determine $\mathrm{det}(A)$.

b) Draw $\mathrm{det}(A)$'s graph.

c) Give a regular expression representing the same language as A.

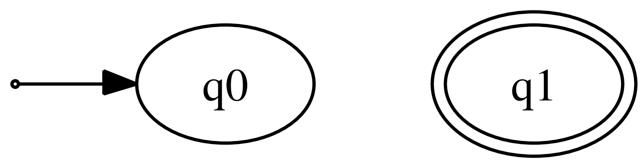
- **Given a regular expression r, we want to derive an NFA $A(r)$ accepting the same language:**

$$L(A(r)) = L(r). \tag{98}$$

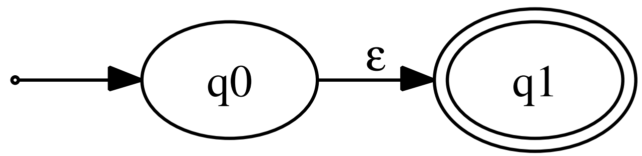
- **Deriving the transformation rules, we will be using two properties of $A(r)$:**
  - **There are no transitions to the initial state.**
  - **There are no transitions from the final state.**

- **Assuming Σ is the alphabet which r is based on, we define**

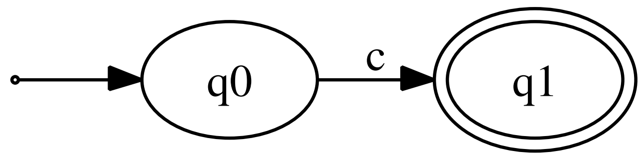
  1. $A(\emptyset) = \langle\{q_0, q_1\}, \Sigma, \{\}, q_0, \{q_1\}\rangle$



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

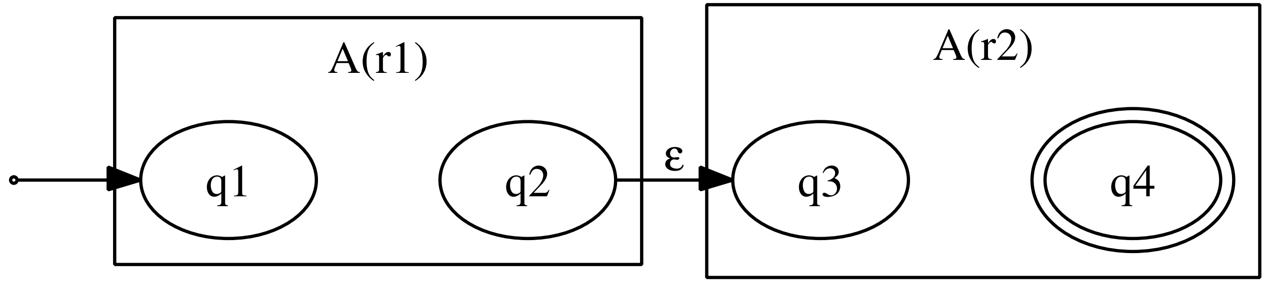


3. $A(c) = \langle \{q_0, q_1\}, \Sigma, \{\langle q_0, c, q_1\rangle\}, q_0, \{q_1\}\rangle$

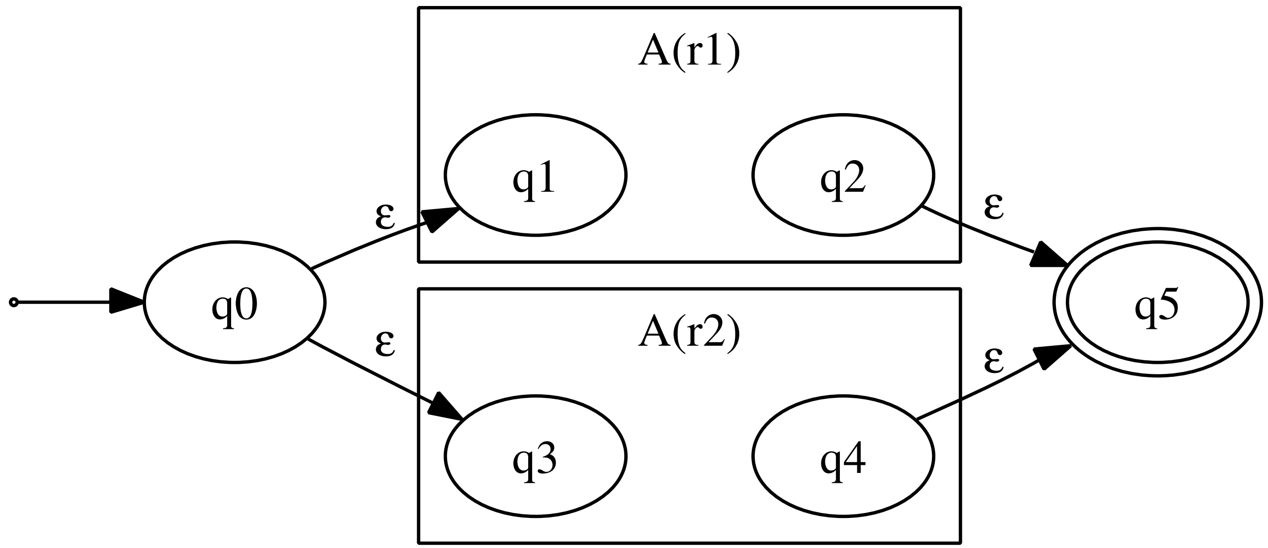


4. $A(r_1 r_2) = \langle Q_1 \cup Q_2, \Sigma, \{\langle q_2, \varepsilon, q_3\rangle\} \cup \delta_1 \cup \delta_2, q_1, \{q_4\}\rangle$ **with**
   $A(r_1) = \langle Q_1, \Sigma, \delta_1, q_1, \{q_2\}\rangle$,
   $A(r_2) = \langle Q_2, \Sigma, \delta_2, q_3, \{q_4\}\rangle$.

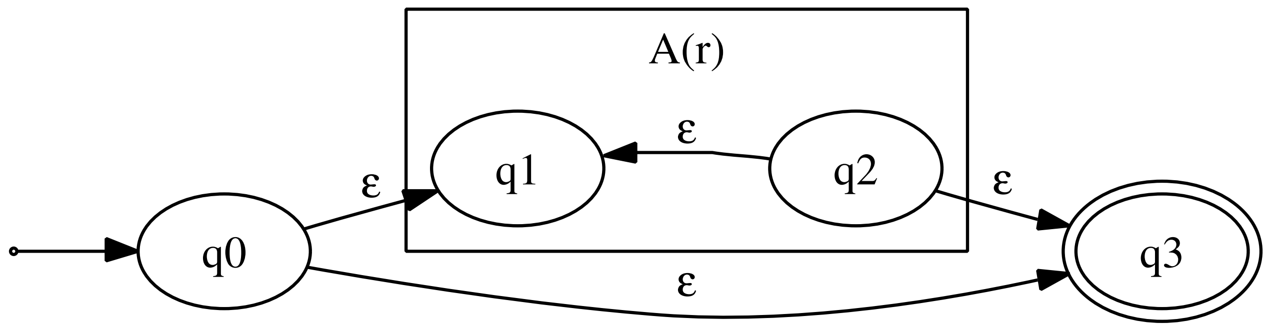
4. $A(r_1 r_2)$ **(cont.)**



5. $A(r_1 + r_2) = \langle \{q_0, q_5\} \cup Q_1 \cup Q_2, \Sigma,$
$\{\langle q_0, \varepsilon, q_1\rangle, \langle q_0, \varepsilon, q_3\rangle, \langle q_2, \varepsilon, q_5\rangle, \langle q_4, \varepsilon, q_5\rangle\} \cup \delta_1 \cup \delta_2, q_0, \{q_5\}\rangle$



6. $A(r^*) = \langle \{q_0, q_3\} \cup Q, \Sigma,$
$\{\langle q_0, \varepsilon, q_1\rangle, \langle q_2, \varepsilon, q_1\rangle, \langle q_0, \varepsilon, q_3\rangle, \langle q_2, \varepsilon, q_3\rangle\} \cup \delta, q_0, \{q_3\}\rangle$ with
$A(r) = \langle Q, \Sigma, \delta, q_1, \{q_2\}\rangle$.



Note: In Transformation Rules 4 and 5 (and often also 6), states connected by ε transitions can be merged.

- **Determine an NDA accepting the same language as the regular expression**

$$(\mathtt{a} + \mathtt{b})\mathtt{a}^*\mathtt{b} \tag{99}$$

Transformation of DFAs into regular expressions

- **We have learned how to convert**
  - **regular expressions $\longrightarrow$ NFAs,**
  - **NFAs $\longrightarrow$ DFAs.**
- **To complete the circle, we now investigate how to convert**
  - **DFAs $\longrightarrow$ regular expressions.**
- **That is, given an DFA A, we want to derive a regular expression $r(A)$ accepting the same language:**

$$L(r(A)) = L(A). \tag{100}$$

Transformation of DFAs into regular expressions: formal derivation

- **The DFA to be converted is of the form**

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle \quad \text{with} \quad Q = \{q_1, \cdots, q_n\}. \tag{101}$$

- **Now, we introduce the auxiliary regular expression**

$$r^{(k)}(p_1, p_2) \quad \text{with} \quad k \in \{0, \cdots, n+1\}; p_1, p_2 \in Q \tag{102}$$

being the regular expression representing all those strings that make A change from p_1 to p_2 without visiting any state q_i with $i \geq k$.

- **According to the above definition of $r^{(k)}(p_1, p_2)$, for $k = 0$, we are not allowed to visit any state changing from p_1 to p_2.**
- **Hence, the only way to change from p_1 to p_2 is to read a single symbol as expressed by the state transition function δ:**

$$r^{(0)}(p_1, p_2) = \begin{cases} c_1 + \cdots + c_l + \varepsilon & \text{for } p_1 = p_2 \\ c_1 + \cdots + c_l + \emptyset & \text{otherwise} \end{cases} \tag{103}$$

$$\text{with} \quad c_1, \ldots, c_l \in \{c \in \Sigma | \delta(p_1, c) = p_2\}$$

- **For $k > 0$, we have**

$$\begin{aligned} r^{(k)}(p_1, p_2) &= r^{(k-1)}(p_1, p_2) + & (104)\\ & r^{(k-1)}(p_1, q_{k-1}) \cdot & (105)\\ & \left(r^{(k-1)}(q_{k-1}, q_{k-1})\right)^* \cdot & (106)\\ & r^{(k-1)}(q_{k-1}, p_2) & (107) \end{aligned}$$

- **This formula recursively expresses $r^{(k)}$ by reference to $r^{(k-1)}$ whose only difference is the permission of q_{k-1} as intermediate state.**
- **Equation 104 expresses the transition from p_1 to p_2 without visiting q_{k-1} (and q_k, q_{k+1}, etc.)**
- **Alternatively, A may change**
  - **first from p_1 to q_{k-1} (without visiting q_k, q_{k+1}, etc.) (Equation 105),**
  - **then arbitrarily often from q_{k-1} to q_{k-1} (without...) (Equation 106),**
  - **and finally from q_{k-1} to p_2 (without...) (Equation 107).**

Transformation of DFAs into regular expressions: formal derivation (cont.)

- **Naturally, the regular expression imposing no restriction on which intermediate states can be visited is**

$$r(p_1, p_2) = r^{(n+1)}(p_1, p_2). \quad (108)$$

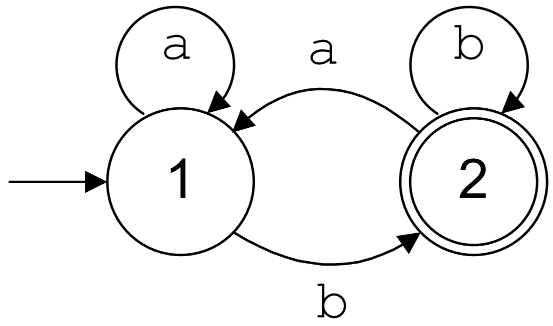
- **Considering**
  - **the initial state q_0 and**
  - **the set of final states $F = \{t_1, \cdots, t_m\}$,**

  we can define the regular expression describing exactly those strings for which A changes from its initial to one of its final states:

$$r(A) = r(q_0, t_1) + \cdots + r(q_0, t_m). \quad (109)$$

Transformation of DFAs into regular expressions: exercise

- **Determine a regular expression accepting the same language as this DFA:**



- Given the DFA

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle, \qquad (110)$$

we want to derive a DFA

$$A^- = \langle Q^-, \Sigma, \delta^-, q_0, F^- \rangle, \qquad (111)$$

accepting the same language, i.e.,

$$L(A) = L(A^-) \qquad (112)$$

for which the number of states (elements of Q^-) is minimal.

- The idea is to identify the set V comprising all the pairs of distiguishable states.

- That is, being in the states p or q, respectively, there is a symbol c which makes the DFA change to the states s and t, respectively, which, in turn, are distinguishable.

- Formally, we have

$$\delta(p, c) = s, \delta(q, c) = t, \langle s, t \rangle \in V. \qquad (113)$$

Minimization of DFAs: the algorithm

1. **We initialize V with all those pairs for which one member is a final state and the other is not:**

$$V = \{\langle p,q\rangle \in Q \times Q | (p \in F \wedge q \notin F) \vee (p \notin F \wedge q \in F)\}. \quad (114)$$

2. **While we can find a pair of states $\langle p,q\rangle$ and a symbol c such that the states $\delta(p,c)$ and $\delta(q,c)$ are distinguishable, we keep adding this pair and its reverse to V:**

while$(\exists\langle p,q\rangle \in Q \times Q : \exists c \in \Sigma : \langle\delta(p,c),\delta(q,c)\rangle \in V \wedge \langle p,q\rangle \notin V)$ (115)

$\{$

$$V = V \cup \{\langle p,q\rangle,\langle q,p\rangle\}$$

$\}$

a) If we have a pair of states $\langle p, q \rangle$ and attempting to read the symbol c results in a reject (Ω) for one of the states and does not for the other, p and q are distinguishable:

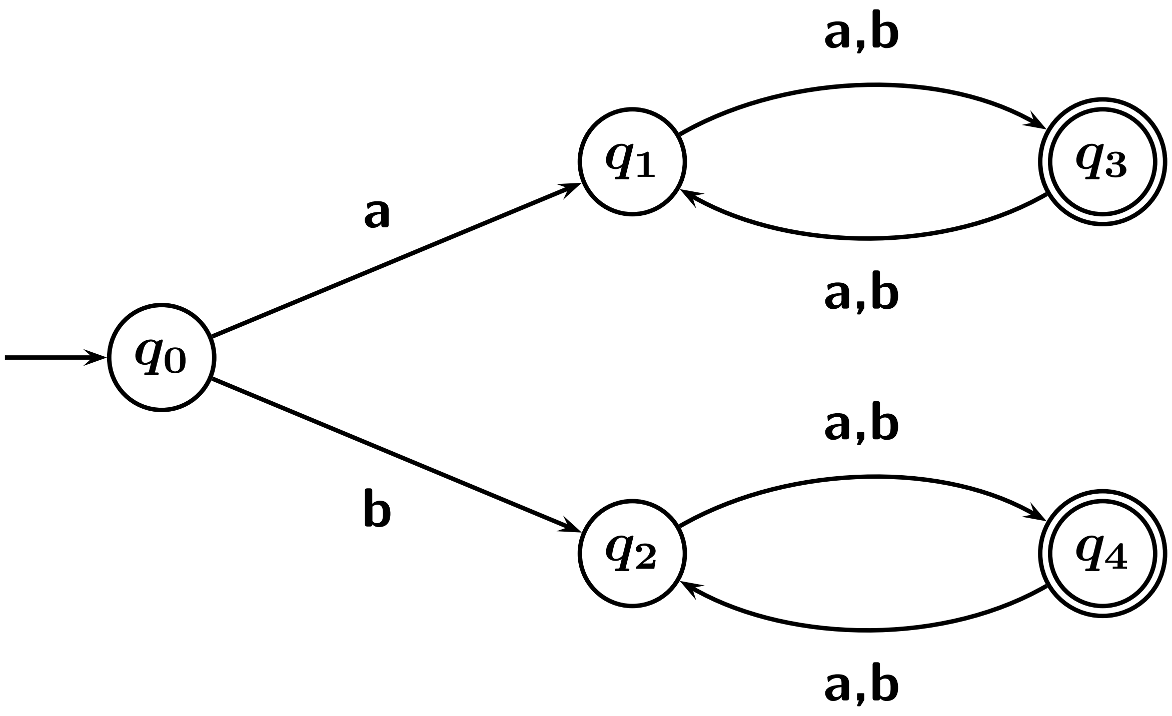
$$\delta(p, c) = \Omega \;\wedge\; \delta(q, c) \neq \Omega \;\vee\; \delta(p, c) \neq \Omega \;\wedge\; \delta(q, c) = \Omega \qquad (116)$$

can be added to the condition in Eq. 115.

b) If we have a pair of states $\langle p, q \rangle$ and reading all possible symbols $c \in \Sigma$ results the same successor states p and q are indistinguishable:

$$\langle p, q \rangle \in Q \times Q : \forall c \in \Sigma : \delta(p, c) = \delta(q, c) \rightarrow \langle p, q \rangle, \langle q, p \rangle \notin V. \qquad (117)$$

- **We want to minimize this DFA with 5 states:**



- **This is the formal definition of the DFA:**

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle \tag{118}$$

with

1. $Q = \{q_0, q_1, q_2, q_3, q_4\}$
2. $\Sigma = \{\texttt{a}, \texttt{b}\}$
3. $\delta = \ldots$ **(skipped to save space, see graph)**
4. $q_0 = q_0$
5. $F = \{q_3, q_4\}$

- **For the sake of practicality, we represent the set V by means of a two-dimensional table with the elements of Q as columns and rows and V's elements as cells featuring the symbol $\times$.**

- **Analogously, we represent state pairs that are definitely not members of V using the symbol $\circ$.**

1. **By determining all combinations of states in $F = \{q_3, q_4\}$ and $Q \backslash F = \{q_0, q_1, q_2\}$, we get the following initial state of V:**

	q_0	q_1	q_2	q_3	q_4
q_0				×	×
q_1				×	×
q_2				×	×
q_3	×	×	×		
q_4	×	×	×		

2. **Furthermore, the cases $\langle q_i, q_i \rangle | i \in \{0, \cdots, 4\}$ are naturally indistinguishable since they are identical:**

	q_0	q_1	q_2	q_3	q_4
q_0	○			×	×
q_1		○		×	×
q_2			○	×	×
q_3	×	×	×	○	
q_4	×	×	×		○

3. **Now, we iterate over all the remaining state-pairs and symbols. In doing so, we can skip the cases $\langle q_i, q_j\rangle | i, j \in \{0, \cdots, 4\}; j < i$ due to the symmetry of the distinguishability of states.**

   - $\delta(q_0, \mathtt{a}) = q_1; \delta(q_1, \mathtt{a}) = q_3; \langle q_1, q_3\rangle \in V \longrightarrow \langle q_0, q_1\rangle, \langle q_1, q_0\rangle \in V$
   - $\delta(q_0, \mathtt{a}) = q_1; \delta(q_2, \mathtt{a}) = q_4; \langle q_1, q_4\rangle \in V \longrightarrow \langle q_0, q_2\rangle, \langle q_2, q_0\rangle \in V$
   - $\delta(q_1, \mathtt{a}) = q_3; \delta(q_2, \mathtt{a}) = q_4; \langle q_3, q_4\rangle \notin V$ **(as of yet)**
     $\delta(q_1, \mathtt{b}) = q_3; \delta(q_2, \mathtt{b}) = q_4; \langle q_3, q_4\rangle \notin V$ **(as of yet)**
   - $\delta(q_3, \mathtt{a}) = q_1; \delta(q_4, \mathtt{a}) = q_2; \langle q_1, q_2\rangle \notin V$ **(as of yet)**
     $\delta(q_3, \mathtt{b}) = q_1; \delta(q_4, \mathtt{b}) = q_2; \langle q_1, q_2\rangle \notin V$ **(as of yet)**

- **Since no other distinguishable state pairs could be found, we fill empty cells with $\circ$:**

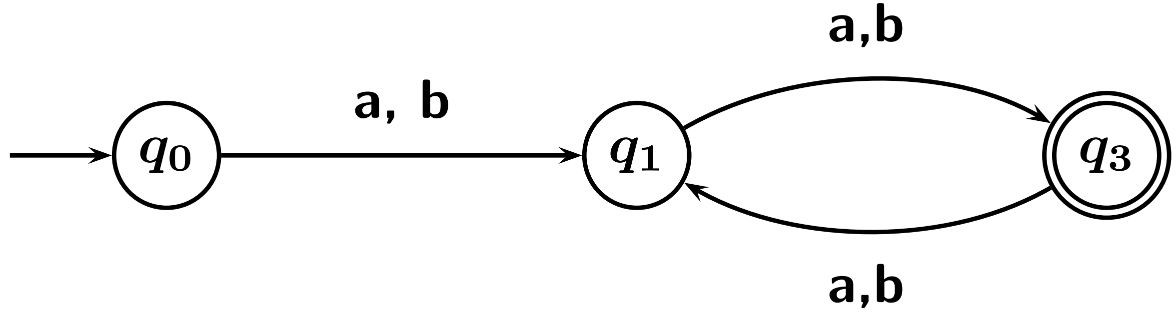
	q_0	q_1	q_2	q_3	q_4
q_0	$\circ$	$\times$	$\times$	$\times$	$\times$
q_1	$\times$	$\circ$	$\circ$	$\times$	$\times$
q_2	$\times$	$\circ$	$\circ$	$\times$	$\times$
q_3	$\times$	$\times$	$\times$	$\circ$	$\circ$
q_4	$\times$	$\times$	$\times$	$\circ$	$\circ$

- **From the table, we can derive the following (non-diagonal, non-symmetrical) indistinguishable state pairs:**

  a) $\langle q_1, q_2 \rangle$,

  b) $\langle q_3, q_4 \rangle$.

- **This is the minimized DFA after merging indistinguishable states:**



- **Derive a minimal DFA accepting the language**

$$L(\mathtt{a}(\mathtt{ba})^*). \tag{119}$$

- **Hint: Solve the exercise in three steps:**
  1. **Derive an NFA accepting L.**
  2. **Transform the NFA into a DFA.**
  3. **Minimize the DFA.**

- **Earlier in this lecture, we have seen that there can be multiple regular expressions describing the same language.**
- **We have also learned that using algebraic transformation rules to prove equivalence of regular expressions can be very difficult or even impossible.**
- **In the following, we will learn a straight-forward algorithm proving equivalence of regular expressions based on FSMs.**
- **According to the textbook of Hopcroft and Ullman** *Introduction to Automata Theory, Languages, and Computation* **(1979), the algorithm involves four steps.**

1. Given the regular expressions r_1 and r_2, derive NFAs A_1 and A_2 accepting their respective languages:

$$L(r_1) = L(A_1) \quad \text{and} \quad L(r_2) = L(A_2). \tag{120}$$

2. Transform the NFAs A_1 and A_2 into the DFAs D_1 and D_2.

3. Minimize the DFAs D_1 and D_2 yielding the DFAs M_1 and M_2.

4. If $r_1 \doteq r_2$, then M_1 and M_2 must be identical except for possible differences in state names.

Note: If you can show equivalence in any intermediate stage of the algorithm, this is enough to prove $r_1 \doteq r_2$ (e.g. if $A_1 = A_2$).

- **Reusing two exercises from an earlier section, prove the following equivalences:**

a) $\mathtt{10(10)^*} \doteq \mathtt{1(01)^*0}$,

b) $(\mathtt{1} + \varepsilon)(\mathtt{0}(\mathtt{1} + \varepsilon))^*\mathtt{1}^* \doteq (\mathtt{0} + \mathtt{10})^*\mathtt{1}^*$.

1. **introduction**
2. **regular expressions**
   - **compact description of sets of strings**
   - **fundamental component of script languages (Perl, Python, grep, sed, awk, etc.) and of most modern programming languages (.NET, SQL Server 2008, Java, etc.)**
3. **the scanner generator JFlex**
4. **finite-state machines**
   ...are able to detect regular expressions
5. **formal grammars**

Formal languages vs. formal grammars

- **In the introduction, we have learned that a formal language is a set of words composed of symbols of a given alphabet.**
- **We have learned about several ways to describe (words accepted by) a language:**
  - **regular expressions,**
  - **DFAs,**
  - **NFAs.**

  Yet another way to do so are
  - **formal grammars.**

- **According to Noam Chomsky (*1928), a grammar is a quatruple**

$$G = \langle V_N, V_T, P, S \rangle \tag{121}$$

with

1. **the set of non-terminal symbols V_N,**
2. **the set of terminal symbols V_T,**
3. **the set of production rules P of the form**

$$\alpha \longrightarrow \beta \tag{122}$$

$$\text{with} \quad \alpha \in V^* V_N V^*, \beta \in V^*, V = V_N \cup V_T$$

4. **the distinguished start symbol $S \in V_N$.**

- **For the sake of simplicity, we will be using the short form**

$$\alpha \longrightarrow \beta_1 | \cdots | \beta_n \quad \text{replacing} \quad \begin{array}{l} \alpha \longrightarrow \beta_1 \\ \vdots \\ \alpha \longrightarrow \beta_n \end{array} \tag{123}$$

- **We want to define a grammar**

$$G = \langle V_N, V_T, P, S \rangle \tag{124}$$

to describe identifiers of the `C` **programming language.**

- **that is, alpha-numeric words which must not start with a digit and may also contain an underscore (_)**

- **We have**

1. $V_N = \{I, R, L, D\}$ **(identifier, rest, letter, digit),**
2. $V_T = \{\mathtt{a}, \cdots, \mathtt{z}, \mathtt{A}, \cdots, \mathtt{Z}, \mathtt{0}, \cdots, \mathtt{9}, \_\}$,
3. $$\begin{aligned} P: \quad I &\rightarrow LR|\_R|L|\_ \\ R &\rightarrow LR|DR|\_R|L|D|\_ \\ L &\rightarrow \mathtt{a}|\cdots|\mathtt{z}|\mathtt{A}|\cdots|\mathtt{Z} \\ D &\rightarrow \mathtt{0}|\cdots|\mathtt{9} \end{aligned}$$
4. $S = I$.

- **We can define the operation of grammars by means of derivations.**
- **Given the grammar**

$$G = \langle V_N, V_T, P, S \rangle, \tag{125}$$

we define the relation

$$x \Rightarrow_G y \text{ iff } \exists u, v, p, q \in V^* : (x = upv) \wedge (p \rightarrow q \in P) \wedge (y = uqv) \tag{126}$$

pronounced as "G derives in one step".

- **We also define the relation**

$$x \Rightarrow_G^* y \text{ iff } \exists w_0, \ldots, w_n \tag{127}$$

$$\text{with } w_0 = x, w_n = y, w_{i-1} \Rightarrow_G w_i \text{ for } i \in \{1, \cdots, n\}$$

pronounced as "G derives in zero or more steps".

- **We are given the grammar**

$$G = \langle V_N, V_T, P, S \rangle \tag{128}$$

with

1. $V_N = \{S\}$,
2. $V_T = \{0\}$,
3. $$P: \quad S \rightarrow 0S \quad 1$$ $$S \rightarrow 0 \quad 2$$
4. $S = S$.

- **Derivations of G have the general form**

$$S \Rightarrow_1 0S \Rightarrow_1 00S \Rightarrow_1 \cdots \Rightarrow_1 0^{n-1}S \Rightarrow_2 0^n. \tag{129}$$

- **Apparently, the language accepted by G is**

$$L(G) = \{0^n | n \in \mathbb{I}; n > 0\}. \tag{130}$$

- **We are given the grammar**

$$G = \langle V_N, V_T, P, S \rangle \tag{131}$$

with

1. $V_N = \{S\}$,
2. $V_T = \{0, 1\}$,
3. $$P: \quad S \to 0S1 \quad 1$$ $$S \to 01 \quad 2$$
4. $S = S$.

- **Derivations of G have the general form**

$$S \Rightarrow_1 0S1 \Rightarrow_1 00S11 \Rightarrow_1 \cdots \Rightarrow_1 0^{n-1}S1^{n-1} \Rightarrow_2 0^n1^n. \tag{132}$$

- **Apparently, the language accepted by G is**

$$L(G) = \{0^n1^n | n \in \mathbb{I}; n > 0\}. \tag{133}$$

- **We are given the grammar**

$$G = \langle V_N, V_T, P, S \rangle \tag{134}$$

with

1. $V_N = \{S, B, C\}$,
2. $V_T = \{0, 1, 2\}$,
3. $$P: \quad \begin{aligned} S &\rightarrow 0SBC & \mathbf{1} \\ S &\rightarrow 0BC & \mathbf{2} \\ CB &\rightarrow BC & \mathbf{3} \\ 0B &\rightarrow 01 & \mathbf{4} \\ 1B &\rightarrow 11 & \mathbf{5} \\ 1C &\rightarrow 12 & \mathbf{6} \\ 2C &\rightarrow 22 & \mathbf{7} \end{aligned}$$
4. $S = S$.

- **Derivations of G have the general form**

$$S \Rightarrow_1 0SBC \Rightarrow_1 00SBCBC \Rightarrow_1 \cdots \Rightarrow_1 0^{n-1}S(BC)^{n-1} \Rightarrow_2 0^n(BC)^n$$
$$\Rightarrow_3^* 0^nB^nC^n \Rightarrow_{4,5}^* 0^n1^nC^n \Rightarrow_{6,7}^* 0^n1^n2^n \quad (135)$$

- **The language accepted by G is**

$$L(G) = \{0^n1^n2^n | n \in \mathbb{I}; n > 0\}. \quad (136)$$

- **These three derivation examples represent different classes of grammars or languages characterized by different properties.**

- **A widely used classification scheme of formal grammars and languages is the Chomsky hierarchy.**

- **Given the grammar**

$$G = \langle V_N, V_T, P, S \rangle, \tag{137}$$

we define the following grammar/language classes

  - **Type 0 or** *unrestricted*

    if there are no restrictions.

  - **Type 1 or** *context-sensitive*

    if all productions are of the form

$$\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2 \text{ with } A \in V_N; \alpha_1, \alpha_2 \in V^*, \beta \in VV^* \tag{138}$$

Exception:

$$\langle S \rightarrow \varepsilon \rangle \in P \quad \longrightarrow \quad \alpha_1, \alpha_2 \in (V \backslash \{S\})^*, \beta \in (V \backslash \{S\})(V \backslash \{S\})^* \tag{139}$$

- **Type 2 or** ***context-free***

  if all productions are of the form

  $$A \rightarrow \beta \text{ with } A \in V_N; \beta \in VV^* \tag{140}$$

  Exception:

  $$\langle S \rightarrow \varepsilon \rangle \in P \quad \longrightarrow \quad \beta \in (V \backslash \{S\})(V \backslash \{S\})^* \tag{141}$$

- **Type 3 or** ***regular***

  if all productions are of the form

  $$A \rightarrow aB \text{ or} \tag{142}$$

  $$A \rightarrow a \text{ with } A, B \in V_N; a \in V_T$$

  Exception:

  $$\langle S \rightarrow \varepsilon \rangle \in P \quad \longrightarrow \quad B \in V_N \backslash \{S\} \tag{143}$$

- **For each grammar/language type, there is also a corresponding type of automaton:**

grammar	language	automaton
Type 0	**unrestricted**	**Turing machine**
Type 1	**context-sensitive**	**linear-bounded non-deterministic Turing machine**
Type 2	**context-free**	**non-deterministic pushdown automaton**
Type 3	**regular**	**finite state machine**

Formal grammars vs. formal languages vs. automata: main focus of this class

- **For each grammar/language type, there is also a corresponding type of automaton:**

grammar	language	automaton
Type 0	unrestricted	Turing machine
Type 1	context-sensitive	linear-bounded non-deterministic Turing machine
Type 2	context-free	non-deterministic pushdown automaton
Type 3	regular	finite state machine

- **Returning to our example on identifiers of the `C` programming language:**

$$\begin{aligned} P: \quad I &\longrightarrow LR|\_R|L|\_ \\ R &\longrightarrow LR|DR|\_R|L|D|\_ \\ L &\longrightarrow \mathtt{a}|\cdots|\mathtt{z}|\mathtt{A}|\cdots|\mathtt{Z} \\ D &\longrightarrow \mathtt{0}|\cdots|\mathtt{9} \end{aligned}$$

- **This grammar is context-free but not regular.**
- **An equivalent regular grammar could have the following productions:**

$$\begin{aligned} P: \quad I &\longrightarrow \mathtt{A}|\cdots|\mathtt{Z}|\mathtt{a}|\cdots|\mathtt{z}|\_| \\ &\qquad \mathtt{A}R|\cdots|\mathtt{Z}R|\mathtt{a}R|\cdots|\mathtt{z}R|\_R \\ R &\longrightarrow \mathtt{A}|\cdots|\mathtt{Z}|\mathtt{a}|\cdots|\mathtt{z}|\_|\mathtt{0}|\cdots|\mathtt{9}| \\ &\qquad \mathtt{A}R|\cdots|\mathtt{Z}R|\mathtt{a}R|\cdots|\mathtt{z}R|\_R|\mathtt{0}R|\cdots|\mathtt{9}R \end{aligned}$$

- **Returning to the three derivation examples:**

  I.

  - **The grammar with** $P = \{\langle S \rightarrow 0S\rangle, \langle S \rightarrow 0\rangle\}$ **is regular.**
  - **So is the accepting language** $L = \{0^n | n \in \mathbb{I}; n > 0\}$.

  II.

  - **The grammar with** $P = \{\langle S \rightarrow 0S1\rangle, \langle S \rightarrow 01\rangle\}$ **is context-free.**
  - **So is the accepting language** $L = \{0^n 1^n | n \in \mathbb{I}; n > 0\}$.

III.

- **The last grammar is unrestricted.**
- **The only production preventing the grammar from being context-sensitive is $CB \rightarrow BC$.**
- **We can, however, replace this production by the three context-sensitive productions**

$$\begin{aligned} CB &\rightarrow CX \\ CX &\rightarrow BX \\ BX &\rightarrow BC \end{aligned} \tag{144}$$

  without changing the grammar's behavior.
- **The resulting grammar is context-sensitive.**
- **So is the accepting language $L = \{0^n 1^n 2^n | n \in \mathbb{I}; n > 0\}$.**

I. We are given the grammar

$$G = \langle V_N, V_T, P, S \rangle \tag{145}$$

with

1. $V_N = \{S, A, B\}$,
2. $V_T = \{0\}$,
3. $P:$

$$\begin{array}{rcll} S & \longrightarrow & \varepsilon & \mathbf{1} \\ S & \longrightarrow & ABA & \mathbf{2} \\ AB & \longrightarrow & 00 & \mathbf{3} \\ 0A & \longrightarrow & 000A & \mathbf{4} \\ A & \longrightarrow & 0 & \mathbf{5} \end{array}$$

4. $S = S$.

a) What is G's highest type?

b) Show how G derives the word 00000.

c) Formally describe the language $L(G)$.

d) Define a regular grammar G' equivalent to G.

II. An octal constant is a finite sequence of digits starting with 0 followed by at least one digit randing from 0 to 7. Define a regular grammar encoding exactly the set of possible octal constants.

III. We are given the grammar

$$G = \langle V_N, V_T, P, S \rangle \qquad (146)$$

with

1. $V_N = \{S, N, E\}$,
2. $V_T = \{0, 1, \mathtt{t}\}$,
3. $$\begin{array}{rrcll} P: & S & \longrightarrow & 0NS & \mathbf{1} \\ & S & \longrightarrow & 1ES & \mathbf{2} \\ & S & \longrightarrow & \mathtt{t} & \mathbf{3} \\ & N\mathtt{t} & \longrightarrow & \mathtt{t}0 & \mathbf{4} \\ & E\mathtt{t} & \longrightarrow & \mathtt{t}1 & \mathbf{5} \end{array}$$

$$\boldsymbol{N}0 \longrightarrow 0\boldsymbol{N} \quad \mathbf{6}$$

$$\boldsymbol{N}1 \longrightarrow 1\boldsymbol{N} \quad \mathbf{7}$$

$$\boldsymbol{E}0 \longrightarrow 0\boldsymbol{E} \quad \mathbf{8}$$

$$\boldsymbol{E}1 \longrightarrow 1\boldsymbol{E} \quad \mathbf{9}$$

4. $S = S$**.**

a) What is G's highest type?

b) Formally describe the language $L(G)$.

Outline (cont.)

6. **context-free languages**

   most programming languages are context-free

7. **Antlr**

   ...a parser generator

Limitations of regular languages: the pumping lemma

- Given a language L, the pumping lemma is a way to disprove the regularity of L.
- Informally, is says that sufficiently long words in L may be pumped to produce a new word within L.
- Here, pumping refers to the repetition of the middle section of the word.
- Formally, we have:
  - L is a regular language.
  - Then, there exists an integer $n \in \mathbb{I}$ such that all words $s \in L$ with a length greater than or equal to n can be split into three parts u, v, and w satisfying the following conditions:
    1. $s = uvw$,
    2. $v \neq \varepsilon$,
    3. $|uv| \leq n$,
    4. $\forall h \in \mathbb{I}(uv^h w \in L)$.

- **The pumping lemma can be written in a single formula as follows:**

$$\textbf{reg}(L) \quad \rightarrow \quad \exists n \in \mathbb{I}\ \forall s \in L(|s| \geq n \rightarrow \exists u, v, w \in \Sigma^*(s = uvw \\ \wedge v \neq \varepsilon \wedge |uv| \leq n \wedge \forall h \in \mathbb{I}(uv^h w \in L))) \tag{147}$$

- **In order to disprove regularity of languages, this formula can be transformed into**

$$\forall n \in \mathbb{I}\ \exists s \in L(|s| \geq n \wedge \forall u, v, w \in \Sigma^* \exists h \in \mathbb{I}(\neg(s = uvw \\ \wedge v \neq \varepsilon \wedge |uv| \leq n \wedge uv^h w \in L))) \rightarrow \neg\textbf{reg}(L) \tag{148}$$

- **Given the alphabet $\Sigma = \{(,)\}$,**
- **we define a language L consisting of k opening brackets followed by k closing brackets:**

$$L = \{(^k)^k | k \in \mathbb{I}\}. \tag{149}$$

- **According to Eq. 148, for all possible integers n, we need to find an $s \in L$ whose length is greater than or equal to n, e.g.**

$$s = (^n)^n. \tag{150}$$

- **Now, we just have to show that there is no way to satisfy Conditions 1 to 4 with this s.**

- **Considering that $s = uvw$ (1), $|uv| \leq n$ (3), and $v \neq \varepsilon$ (2), we know that**

$$u = (^l, \quad v = (^m, \quad w = (^p)^n \tag{151}$$

with

$$l + m + p = n; m \geq 1 \tag{152}$$

i.e.

$$l + p \leq n - 1. \tag{153}$$

- **Now, if we are able to show that Condition 4 cannot be fulfilled, we are done.**

- **That is, we need to show that**

$$\neg\forall h \in \mathbb{I}(uv^h w \in L) \quad \textbf{or} \quad \exists h \in \mathbb{I}(uv^h w \notin L). \tag{154}$$

- **For $h = 0$, we would obtain the word**

$$uw = (^{l+p})^n \tag{155}$$

- **According to Eq. 153, $l + p \neq n$, hence $uw \notin L$ which completes the proof that**

$$\neg\textbf{reg}(L). \tag{156}$$

The pumping lemma: example (cont.)

- **In conclusion, we see that the language**

$$L = \{(^k)^k | k \in \mathbb{I}\}. \qquad (157)$$

is not regular.

- **That is, regular languages are not capable of counting brackets.**
- **Hence, for most common programming languages, regular languages/grammars/expressions are not powerful enough.**
- **In the following, we will learn more about context-free languages which are able to cope with most common programming languages.**

The pumping lemma: exercise I

- **We are given the language L comprising all the words of the form $\mathtt{a}^n$ where n is a square number:**

$$L = \{\mathtt{a}^{n^2} | n \in \mathbb{I}\}. \tag{158}$$

- **Prove that L is not a regular language.**

- **We are given the language**

$$L = \{\mathtt{a}^k\mathtt{b}^l | k, l \in \mathbb{I}\}. \tag{159}$$

- **Apply the pumping theorem of regular languages.**
- **Define a grammar G of the highest possible type accepting L .**

Outline (cont.)

6. **context-free languages**

   most programming languages are context-free

7. **Antlr**

   ...a parser generator

- Context-free grammars have a non-terminal symbol on the right.
- This type of grammar is sufficiently powerful to describe most scenarios in computer programs.
- A parser is able to verify the validity of the program and erive an abstract syntax tree for later execution of the code.
- The automaton underlying a parser is the pushdown automaton (PDA) employing a stack.
- Arbitrary context-free grammars are represented by non-deterministic PDAs whereas computationally efficient parsers are usually limited to deterministic PDAs.
- In contrast to FSMs, non-deterministic PDAs are more powerful than deterministic ones and cannot be algorithmically transformed into the latter.

- **A syntax tree represents the syntactic structure of a string according to a formal grammar.**
- **Starting from the start symbol (root), every word of the language can be represented by a tree whose leaves are terminals and the inner nodes are non-terminals representing grammar rules.**
- **Consider the grammar**

$$G = \langle V_N, V_T, P, S \rangle \tag{160}$$

with

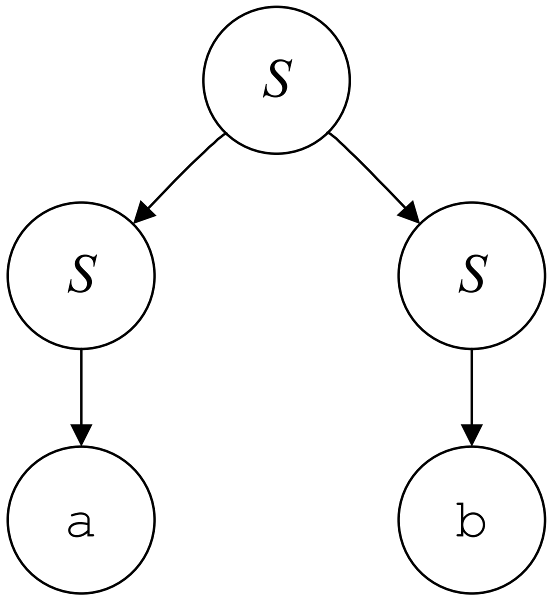
1. $V_N = \{S\}$,
2. $V_T = \{\mathtt{a}, \mathtt{b}\}$,
3. $$\begin{aligned} P: \quad & S \rightarrow SS && 1 \\ & S \rightarrow \mathtt{a} && 2 \\ & S \rightarrow \mathtt{b} && 3 \end{aligned}$$
4. $S = S$.

Syntax trees (cont.)

- **The word** `ab` **can be derived by** G **as**

$$S \Rightarrow_1 SS \Rightarrow_2 \mathtt{a}S \Rightarrow_3 \mathtt{ab} \qquad (161)$$

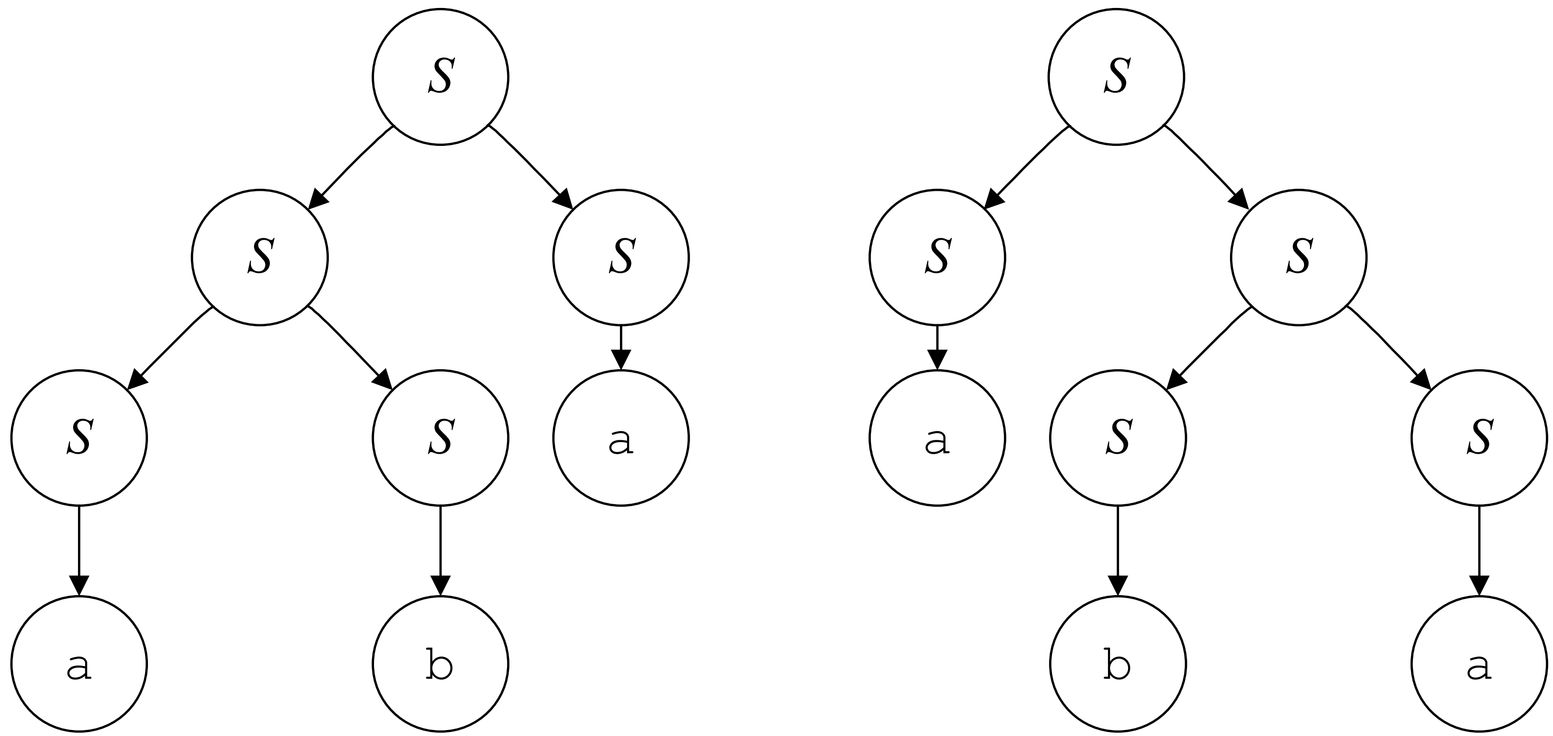
- **This derivation can be represented by a syntax tree:**



- **The word** `aba` **can be derived by G as**

$$S \Rightarrow_1 SS \Rightarrow_1 SSS \Rightarrow_2 \mathtt{a}SS \Rightarrow_3 \mathtt{ab}S \Rightarrow_2 \mathtt{aba} \qquad (162)$$

- **This derivation can be represented by two different syntax trees, that is, the derivation is ambiguous:**



Leftmost derivation

- **By always replacing the leftmost non-terminal, some cases of ambiguity can be overcome.**
- **The derivation of Eq. 162 is non-ambiguously represented by the left of the above trees.**
- **Unfortunately, there might be multiple leftmost derivations for a given word.**
- **E.g., the word** aba **can be derived by Eq. 162 as well as by**

$$S \Rightarrow_1 SS \Rightarrow_2 \mathtt{a}S \Rightarrow_1 \mathtt{a}SS \Rightarrow_3 \mathtt{ab}S \Rightarrow_2 \mathtt{aba} \tag{163}$$

- **This derivation can be represented by the right of the above trees.**

- **Consider the grammar**

$$G = \langle V_N, V_T, P, S \rangle \tag{164}$$

with

1. $V_N = \{S\}$,
2. $V_T = \{*, +, (,), \mathtt{a}, \mathtt{b}, \mathtt{c}\}$,
3. $$\begin{aligned} P: \quad S &\rightarrow S{*}S & 1 \\ S &\rightarrow S{+}S & 2 \\ S &\rightarrow (S) & 3 \\ S &\rightarrow \mathtt{a} & 4 \\ S &\rightarrow \mathtt{b} & 5 \\ S &\rightarrow \mathtt{c} & 6 \end{aligned}$$
4. $S = S$.

Syntax trees: exercise (cont.)

a) **Draw a leftmost-derived syntax tree for the word** `a+(b+a)*c`**.**

b) **Show that this grammar does not account for the precedence difference between * and + by drawing two different leftmost-derived syntax trees for the word** `a+b*c`**.**

c) **Write a context-free grammar G' with $L(G') = L(G)$ accounting for the precedence difference between * and +.**

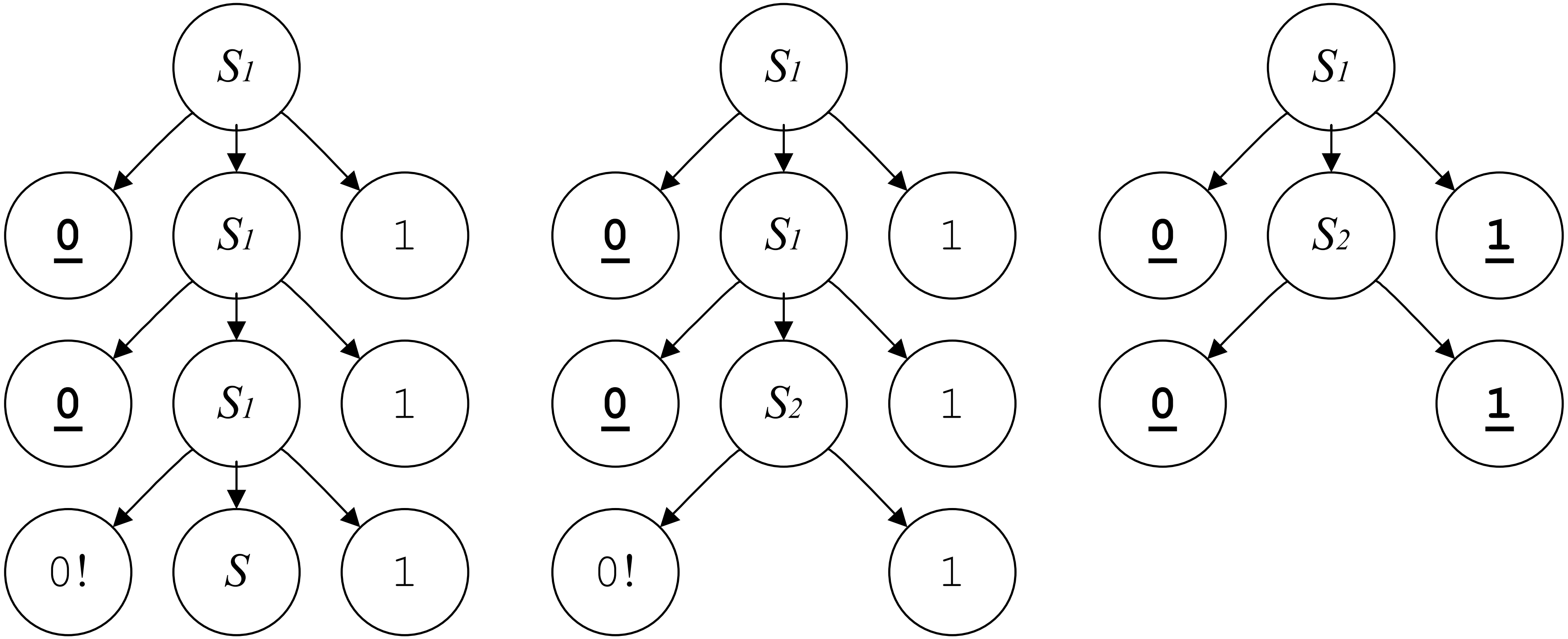
Top-down parsing

- There are multiple approaches to producing syntax trees from grammars.
- A popular technique is top-down parsing.
- An LL parser is based on the top-down approach parsing the input from left to right producing a leftmost derivation.
- Drawbacks:
  - possible exponential time complexity for ambiguous grammars
  - no termination for left-recursive grammars

A parsing algorithm

1. **Pick the leftmost non-processed symbol of the input word s. If there is none and all leaves of the syntax tree are matched the word is in the language, otherwise not.**
2. **Compare s with the left-most non-matched leaf of the syntax tree. If there is none roll back to the last possible alternative derivation and continue with Step 3.**
3. **If the leaf is a non-terminal symbol, extend the leaf by means of the first not yet tried derivation until a terminal symbol t shows up at the leftmost position.**
4. **If $s = t$, continue with Step 1, otherwise roll back to the last possible alternative derivation and continue with Step 3.**

- **We consider the example in Eq. 131 and show how the parsing algorithm deals with the words $w_1 = 0011$ and $w_2 = 0010$.**



- **Taking a look at Eq. 160, we see that the algorithm cannot succeed since it will result in an infinite loop replacing Rule 1 into itself over and over.**
- **This problem is referred to as left recursion.**
- **A grammar is left-recursive if a non-terminal symbol can derive a sentence with itself as the leftmost symbol.**
- **Examples:**
  - **immediate left recursion**

$$A \rightarrow A\alpha \tag{165}$$

  - **indirect left recursion**

$$\begin{aligned} A &\rightarrow B\alpha \\ B &\rightarrow A\beta \end{aligned} \tag{166}$$

A parsing algorithm: exercise

- **Rewrite the grammar in Eq. 160 to eliminate left recursion and show that the word** `aba` **can be parsed by the parsing algorithm.**

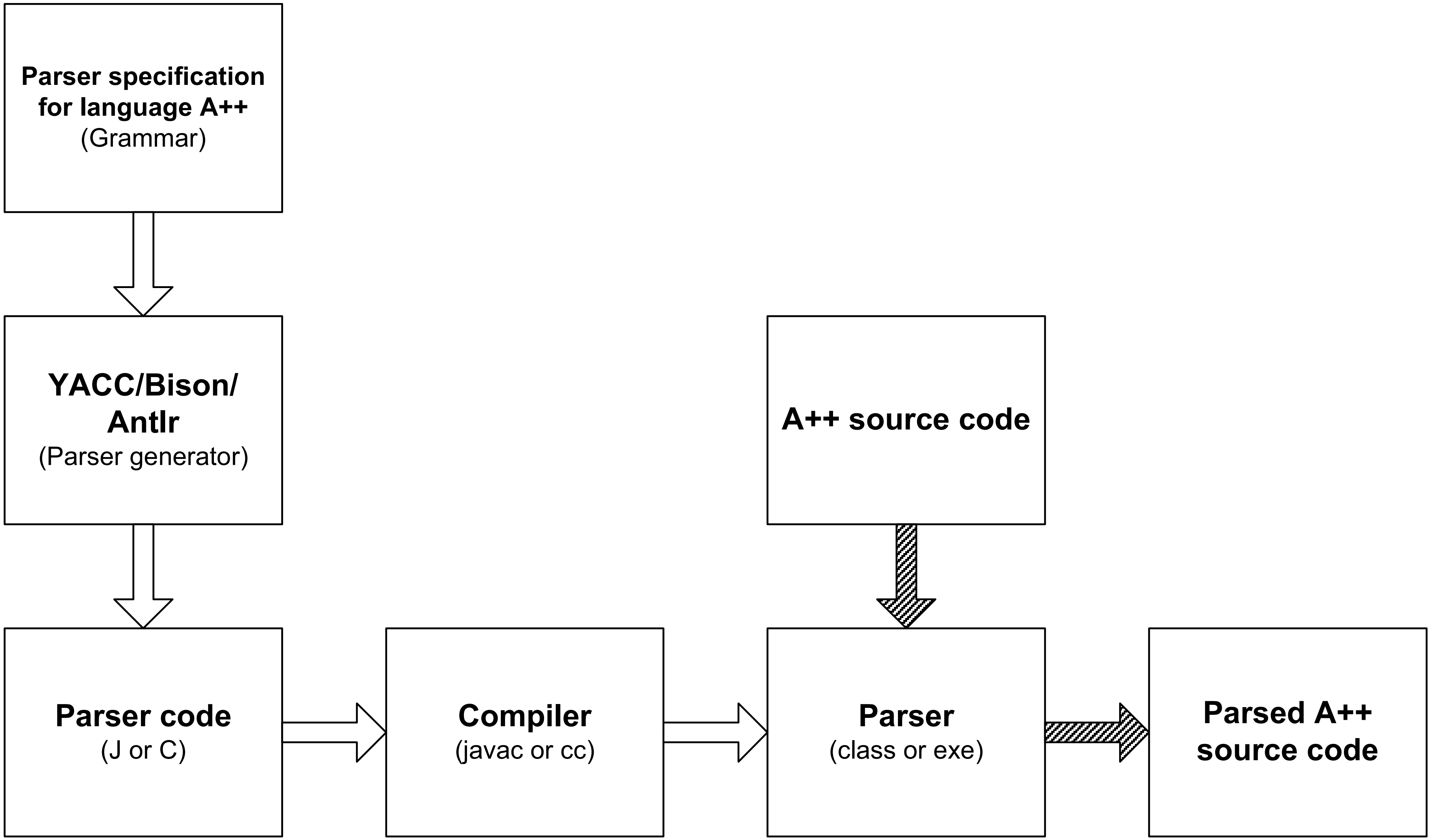
Outline (cont.)

6. **context-free languages**
   most programming languages are context-free
7. **Antlr**
   ...a parser generator

Antlr

- **Antlr (Another tool for language recognition) is a parser generator.**
- **Given a grammar specification, it performs the syntactical analysis (aka parsing) of a source text.**
- **Antlr is a free, open-source software.**
- **Antlr is written in Java, i.e. it is platform-independent.**
- **The parser Antlr produces is also a Java program.**

A parser generator for the (hypothetical) language A++



- **We assume you have installed the JDK, as formerly required by JFlex.**
- **Download the ANTLR Java complete binary jar from**

  ```
  http://antlr.org/download.html
  ```

- **Add the location of Antlr to your class path (see details in the JFlex-related instructions), e.g.:**

  ```
  export
  CLASSPATH=$CLASSPATH';c:\antlr\antlr-3.4-complete.jar'
  ```

- **To test the proper installation, use example files from the package** `fla_*.zip` **by running the following command from a new Cygwin shell:**

  ```
  java org.antlr.Tool expr.g
  javac ParseExpr.java
  echo '2 * 3 + (5 - 4) / 2' | java ParseExpr
  ```

- **Grammar specifications in Antlr are based on the so-called Extended Backus-Naur Form (EBNF) which is more compact than what we have used to describe formal grammars so far.**
- **These are additional constructs used by the EBNF derivative of Antlr (most which we already know from the operator set of JFlex):**

  a) the operator $*$ matching 0 or more repetitions of an expression,

  b) the operator $+$ matching 1 or more repetitions of an expression,

  c) the operator ? matching an optional expression,

  d) the operator $|$ separating alternatives,

  e) the operator .. to define ranges,

  f) parentheses to structure expressions.

EBNF: example

- **This is a grammar describing arithmetic expressions:**

$$\begin{aligned} S &\longrightarrow E \\ E &\longrightarrow P((\texttt{'+'}|\texttt{'-'})P)* \\ P &\longrightarrow F((\texttt{'*'}|\texttt{'/'})F)* \\ F &\longrightarrow \texttt{'('}E\texttt{')'}|N \\ N &\longrightarrow (\texttt{'1'}..\texttt{'9'})(\texttt{'0'}..\texttt{'9'})* \end{aligned} \tag{167}$$

- **Words decribed by this grammar include**

```
1
1+2
1+2-3
1+2*3
(1+2*3)/456
```

The following code represents the above grammar in Antlr format:

```
grammar expr;

start:expr;
expr:product(('+'|'-')product)*;
product:factor(('*'|'/')factor)*;
factor:'('expr')'|NUMBER;
NUMBER:('1'..'9')('0'..'9')*;
WS:(' '|'\t'|'\n'|'\r'){skip();};
```

- **In Line 1, we specify the name of our grammar (`expr`) using the keyword `grammar`.**
- **The grammar file name needs to be composed of the grammar name concatenated with the suffix `.g`; that is, our grammar needs to be saved as `expr.g`.**
- **The variable on the left of the first grammar rule, i.e. `start`, is used as start symbol.**
- **Terminals are specified using single quotes (e.g. `'+'`).**
- **By convention, non-terminal symbols are represented by variables starting with a lower-case letter (such as `expr`) unless they match terminals only, in which case they have to start with an upper-case letter (e.g. `NUMBER`).**
- **The non-terminal symbol `WS` defines all those terminals supposed to be treated as white space.**
- **The semantic action associated with the symbol `WS` in our case is `skip()` which means that white spaces are ignored.**

- **In order to generate the parser, we run Antlr using the command**

  ```
  java org.antlr.Tool expr.g
  ```

  producing the following files:

  - `exprParser.java` **containing the Parser code,**
  - `exprLexer.java` **containing the Scanner code, and**
  - `expr.tokens` **containing a mapping table between symbols used in the grammar and IDs used in the parser code.**

- **In order to run the Parser from the command line, we need to write a driver program invoking both classes** `exprParser` **and** `exprLexer`**.**

The following code implements Scanner and Parser generated from `expr.g`:

```
import org.antlr.runtime.*;

public class ParseExpr
{
  public static void main(String[] args) throws Exception
  {
    ANTLRInputStream input = new ANTLRInputStream(System.in);
    exprLexer lexer = new exprLexer(input);
    CommonTokenStream ts = new CommonTokenStream(lexer);
    exprParser parser = new exprParser(ts);
    parser.expr();
  }
}
```

- **Next, we need to compile the driver program by**

```
javac ParseExpr.java
```

- **Finally, we are able to execute the parser applying it to an input word, for example:**

```
echo '2 * 3 + (5 - 4) / 2' | java ParseExpr
```

- **This input is a valid expression for the grammar we specified.**
- **As we did not define any semantic actions in the grammar, the parser does not return anything but terminates silently.**
- **Now, let us try to parse a word not matched by the grammar, e.g.**

```
echo '2 * + 3 + (5 - 4) / 2' | java ParseExpr
```

- **This time, we receive the error message**

```
line 1:4 no viable alternative at input '+'
```

telling us that at Line 1, Character 5 (characters are enumerated starting with 0), the parser did not know how to handle the input symbol `'+'`.

Write parsers in Antlr for the following languages:

1. **Well-formulated formulas of propositional logic.**
2. **A simple HTML document (supporting the tags** `<html>`**,** `<head>`**,** `<title>`**,** `<body>`**,** `<p>`**,** `<br>`**).**
3. **Simplified English with the following non-terminals (tags):**
   - `S`**: sentence,**
   - `NP`**: noun phrase,**
   - `VP`**: verb phrase,**
   - `PP`**: prepositional phrase,**
   - `N`**: noun,**
   - `V`**: verb,**
   - `P`**: verb,**
   - `A`**: article.**

   Define a number of matching terminals to test the parser.

Extending the parser to evaluate expressions

- **The above example parser was able to verify whether the syntax of an input expression is correct.**
- **In order to produce a runnable program, the parser needs to be extended by executable code interpreting the parsable rules of the input expression.**
- **This can be done by injecting Java code directly into the grammar definition.**
- **The following code exemplifies how our grammar** `expr.g` **can be modified to calculate the result of an input expression.**

Extending the parser to evaluate expressions (cont.)

```
grammar exprEval;

start:expr{System.out.println($expr.result);} EOF;
expr returns [int result]
  :x=product{$result=$x.result;}
    (
      '+'y=product{$result+=$y.result;}
      |'-'y=product{$result-=$y.result;}
    )*;
product returns [int result]
  :x=factor{$result=$x.result;}
    (
      '*'y=factor{$result*=$y.result;}
      |'/'y=factor{$result/=$y.result;}
    )*;
factor returns [int result]
  :'('x=expr')'{$result=$x.result;}
  |NUMBER{$result=new Integer($NUMBER.text);};
NUMBER:('1'..'9')('0'..'9')*;
WS:(' '|'\t'|'\n'|'\r'){skip();};
```

Extending the parser to evaluate expressions (cont.)

- **After modifying the driver program (which now needs to call the method `parser.start()` directly) we can execute for instance**

```
echo '((4*2)+4)/3' | java ParseExprEval
```

which returns the expected result `4`.

- **These are the additional features we are using:**

1. **Java code can be injected at any place inside the grammar rules by using curly brackets.**
2. **Objects associated with non-terminal symbols can be called inside the Java code using their name preceded by `$` (e.g. `$expr`).**
3. **The built-in symbol `EOF` forces the parser to process the entire input string which prevents incomplete parse results to be returned.**
4. **Return parameters of a rule can be defined by extending the rule header by the keyword `returns` followed by a type and a variable name in square brackets (e.g. `expr returns [int result]`). This parameter can be used inside the rule escaping it by `$` (e.g. `$result`).**

Evaluation with variables

- **The above example is already a compiler in that it does not only parse input expressions but also evaluates them.**
- **To extend our language's functionality, we want to do two more enhancements:**
  a) **Allow for multiple statements to be evaluated.**
  b) **Allow for variables to be used.**
- **In order to do this, we can use the following additional features:**

5. **Code encapsulated by the keyword** `header{}` **is inserted right at the top of the parser code.**
6. **Code encapsulated by the keyword** `members{}` **is inserted right at the top of the parser class.**
7. **A useful Java class to store variables and their values is** `TreeMap`**.**

Evaluation with variables (cont.)

```
grammar exprComp;
@header
{
  import java.util.TreeMap;
}
@members
{
  TreeMap<String, Integer> varTable = new TreeMap<String, Integer>();
}
start:statement+ EOF;
statement:expr{System.out.println($expr.result);}(';')*
  |VAR '=' expr{varTable.put($VAR.text, $expr.result);}(';')*;
expr returns [int result]
  :x=product{$result=$x.result;}
    (
      '+'y=product{$result+=$y.result;}
      |'-'y=product{$result-=$y.result;}
    )*;
product returns [int result]
  :x=factor{$result=$x.result;}
    (
      '*'y=factor{$result*=$y.result;}
      |'/'y=factor{$result/=$y.result;}
    )*;
factor returns [int result]
  :'('x=expr')'{$result=$x.result;}
  |NUMBER{$result=new Integer($NUMBER.text);}
  |VAR{$result = varTable.get($VAR.text);};
NUMBER:('1'..'9')('0'..'9')*;
WS:(' '|'\t'|'\n'|'\r'){skip();};
VAR:('a'..'z'|'A'..'Z')+;
```

- **In the above example, we were lucky since expressions could be evaluated right at the time of parsing.**
- **In more complex scenarios (e.g., user-defined functions), it is necessary to parse the entire input first generating an abstract syntax tree (AST).**
- **Only after the AST has been generated, the actual evaluation is carried out.**
- **In Antlr, this can be achieved by putting the evaluation logic into external Java classes referenced from within the grammar.**

- **In the following example, we develop a parser which is to differentiate a given formula with respect to x.**
- **In order to simplify things, we want to start with formulas which are sums of constants and xs (e.g. $4 + x + c + x$).**
- **This is how the grammar** `diff.g` **can look like:**

```
grammar diff;
expr returns [Expr result]:
  f=addend{$result=$f.result;}
  ('+' g=addend{$result=new Sum($result,$g.result);})* EOF;
addend returns [Expr result]:
  NUM{$result=new Number($NUM.text);}|
  VAR{$result=new Variable($VAR.text);};
NUM:('0'..'9');
VAR:('a'..'z'|'A'..'Z');
WS : (' '|'\t'|'\n'|'\r') { skip(); };
```

Abstract syntax trees: example (cont.)

- **This grammar refers to four classes (**`Expr`**,** `Sum`**,** `Number`**,** `Variable`**) all of which have to be coded in respective external Java source files.**
- **Since the** `result` **of the rules** `expr` **and** `addend` **is an instantiation of either of the classes** `Sum`**,** `Number`**, or** `Variable` **but the result itself needs to be of type** `Expr`**, the latter needs to be an abstract class whereas the former are extensions:**

Abstract syntax trees: example (cont.)

```
public abstract class Expr {
    public abstract Expr diff(String x);
}

...

public class Number extends Expr {
    private Integer mValue;

    public Number(Integer value) {
        mValue = value;
    }
    public Number(String value) {
        mValue = new Integer(value);
    }
    public Expr diff(String x) {
        return new Number(0);
    }
    public String toString() {
        return mValue.toString();
    }
}
```

- **And this is a driver class acommodating the AST and executing the differentiation with respect to x:**

```
import org.antlr.runtime.*;

public class ParseDiff
{
  public static void main(String[] args) throws Exception
  {
    ANTLRInputStream input = new ANTLRInputStream(System.in);
    diffLexer lexer = new diffLexer(input);
    CommonTokenStream ts = new CommonTokenStream(lexer);
    diffParser parser = new diffParser(ts);
    Expr expr=parser.expr();
    Expr diff=expr.diff("x");
    System.out.println(diff);
  }
}
```

Solutions to selected exercises

- **Given the alphabet Σ_{bin} and the language**

$$L = \{1\}. \tag{168}$$

a) Formally describe the language

$$L' = L^* \backslash \{\varepsilon\}. \tag{169}$$

- **According to Equation 25, we have**

$$\begin{aligned} L' &= \{L^0 \cup L^1 \cup L^2 \cup \cdots\} \backslash \{\varepsilon\} \\ &= \{\varepsilon, 1, 11, \ldots\} \backslash \{\varepsilon\} \\ &= \{1, 11, \ldots\} \\ &= \{11^n | n \in \mathbb{I}\}. \end{aligned} \tag{170}$$

b) Formally describe the set

$$D = \{d(w)|w \in L'\}. \tag{171}$$

– Using Equations 170 and 14, we have

$$\begin{aligned} D &= \{d(\mathtt{1}), d(\mathtt{11}), d(\mathtt{111}), \ldots\} \\ &= \{1, 3, 7, \ldots\} \\ &= \{2 \cdot 2^n - 1|n \in \mathbb{I}\} \end{aligned} \tag{172}$$

c) Formally describe the language

$$L'_{-} = \{w | w - 1 \in L'\}. \qquad (173)$$

– **The condition**

$$w - 1 \in \{1, 11, 111, \ldots\} \qquad (174)$$

is equivalent to

$$w \in \{1 + 1, 11 + 1, 111 + 1, \ldots\}. \qquad (175)$$

– **Hence, we have**

$$\begin{aligned} L'_{-} &= \{1 + 1, 11 + 1, 111 + 1, \ldots\} \qquad (176) \\ &= \{10, 100, 1000, \ldots\} \\ &= \{100^n | n \in \mathbb{I}\}. \end{aligned}$$

d) Formally describe the language

$$L'_+ = \{w | w + 1 \in L'\}. \tag{177}$$

– **The condition**

$$w + 1 \in \{1, 11, 111, \ldots\} \tag{178}$$

is equivalent to

$$w \in \{1 - 1, 11 - 1, 111 - 1, \ldots\}. \tag{179}$$

– **Hence, we have**

$$\begin{aligned} L'_+ &= \{1 - 1, 11 - 1, 111 - 1, \ldots\} \quad (180) \\ &= \{0, 10, 110, \ldots\} \\ &= \{1^n 0 | n \in \mathbb{I}\}. \end{aligned}$$

a) Using the alphabet $\Sigma_{\mathbf{abc}} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, give a regular expression r_a for all the words $w \in \Sigma_{\mathbf{abc}}^*$ containing exactly one $\mathtt{a}$ or exactly one $\mathtt{b}$.

- **Similarly to Equation 50, we have**

$$r_a = (\mathtt{b} + \mathtt{c})^*\mathtt{a}(\mathtt{b} + \mathtt{c})^* + (\mathtt{a} + \mathtt{c})^*\mathtt{b}(\mathtt{a} + \mathtt{c})^* \qquad (181)$$

b) Which language is expressed by r_a?

- **Similarly to Equation 51, we have**

$$L(r_a) = \{w \in \Sigma^*_{\mathbf{abc}} \big| |\{i \in \mathbb{I} | w[i] = \mathtt{a}\}| = 1 \vee |\{i \in \mathbb{I} | w[i] = \mathtt{b}\}| = 1\} \quad (182)$$

- **Alternatively, one can write**

$$\begin{aligned} L(r_a) &= \{w \in \Sigma^*_{\mathbf{abc}} \big| |\{i \in \mathbb{I} | w[i] = \mathtt{a}\}| = 1\} \cup \quad (183) \\ &\quad \{w \in \Sigma^*_{\mathbf{abc}} \big| |\{i \in \mathbb{I} | w[i] = \mathtt{b}\}| = 1\} \end{aligned}$$

c) Using the alphabet $\Sigma_{\mathrm{abc}} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, give a regular expression r_b for all the words containing at least one $\mathtt{a}$ and one $\mathtt{b}$.

$$r_a = (\mathtt{a}+\mathtt{b}+\mathtt{c})^*\mathtt{a}(\mathtt{a}+\mathtt{b}+\mathtt{c})^*\mathtt{b}(\mathtt{a}+\mathtt{b}+\mathtt{c})^* + (\mathtt{a}+\mathtt{b}+\mathtt{c})^*\mathtt{b}(\mathtt{a}+\mathtt{b}+\mathtt{c})^*\mathtt{a}(\mathtt{a}+\mathtt{b}+\mathtt{c})^* \quad (184)$$

d) Using the alphabet $\Sigma_{\mathrm{bin}} = \{0, 1\}$, give a regular expression for all the words whose third last symbol is 1.

$$r_d = (0 + 1)^* 1 (0 + 1)(0 + 1) \tag{185}$$

e) Using the alphabet Σ_{bin}, give a regular expression for all the words not containing the string 110.

- **Not containing the string 110 means that 1 must be followed by 0 except for at the end of the word which can be preceded by an arbitrary number of 1.**
- **A possible solution is**

$$r_e = 0^*(100^*)^*1^*. \tag{186}$$

- **To check the validity of a regular expression cadidate, it is useful to control that prototypical words are covered by the candidate, e.g.**

$$\varepsilon, 0, 1, 0^*, 1^*, 0^*1^*, 0^*10^* \in L(r_e) \tag{187}$$

and that others are not (i.e., those featuring 110), e.g.

$$110, 0^*111^*0 \notin L(r_e). \tag{188}$$

f) Which language is expressed by the regular expression

$$r_f = (1 + \varepsilon)(00^*1)^*0^*? \qquad (189)$$

- **To understand what a regular expression is doing, it is useful to point out prototypical words covered by the regular expression, e.g.**

$$\varepsilon, 0, 1, 0^*, 10^*, 0^*10^* \in L(r_f) \qquad (190)$$

and that others that are not, e.g.

$$11, 111^* \notin L(r_f). \qquad (191)$$

- **Apparently, $L(r_f)$ contains all those words not containing two (or more) 1 in sequence.**

- **Hence, we formally describe $L(r_f)$ as the set of all the words with zero occurences of the string 11:**

$$L(r_f) = \{w \in \Sigma^*_{\mathrm{bin}} \big| |\{i \in \mathbb{I} | w[i]w[i+1] = 11\}| = 0\}. \qquad (192)$$

a) Simplify the following regular expression:

$$r = 0(\varepsilon + 0 + 1)^* + (\varepsilon + 1)(1 + 0)^* + \varepsilon. \tag{193}$$

$$\begin{aligned} r &= 0(\varepsilon + 0 + 1)^* + (\varepsilon + 1)(1 + 0)^* + \varepsilon \\ &\overset{14,1}{\doteq} 0(0 + 1)^* + (\varepsilon + 1)(0 + 1)^* + \varepsilon \\ &\overset{7,5}{\doteq} 0(0 + 1)^* + (0 + 1)^* + 1(0 + 1)^* + \varepsilon \\ &\overset{1,7}{\doteq} \varepsilon + (0 + 1)(0 + 1)^* + (0 + 1)^* \\ &\overset{\text{Eq.58,13}}{\doteq} (0 + 1)^* + (0 + 1)^* \\ &\overset{9}{\doteq} (0 + 1)^*. \end{aligned} \tag{194}$$

b) Prove the equivalence using only algebraic operations

$$\boldsymbol{r}^* \doteq \varepsilon + \boldsymbol{r}^*. \tag{195}$$

$$\begin{aligned} \varepsilon + \boldsymbol{r}^* &\overset{\mathbf{13}}{\doteq} \varepsilon + \varepsilon + \boldsymbol{r}^*\boldsymbol{r} \\ &\overset{\mathbf{9}}{\doteq} \varepsilon + \boldsymbol{r}^*\boldsymbol{r} \\ &\overset{\mathbf{13}}{\doteq} \boldsymbol{r}^* \ \square \end{aligned} \tag{196}$$

c) Prove the equivalence using only algebraic operations

$$\mathtt{10(10)^* \doteq 1(01)^*0}. \tag{197}$$

- **We set**

$$r = \mathtt{1(01)^*0}, \tag{198}$$

$$s = \mathtt{10}, \tag{199}$$

$$t = \mathtt{10}. \tag{200}$$

- **This yields**

$$\begin{aligned} rs + t &= 1(01)^*010 + 10 && (201)\\ &\overset{8}{\doteq} 1((01)^*010 + 0) \\ &\overset{5,7}{\doteq} 1((01)^*01 + \varepsilon)0 \\ &\overset{13}{\doteq} 1(01)^*0 \\ &= r. \end{aligned}$$

(202)

- **With the observation that $\varepsilon \notin L(r)$, this fulfills the conditions of Rule 15, leading to the conclusion**

$$1(01)^*0 = r \doteq ts^* = 10(10)^* \quad \square \qquad (203)$$

1. **write a JFlex program removing `C` and `C++` comments from an input source**

```
java JFlex.Main removeCppComment.flex
javac removeCppComment.java
java removeCppComment example.cpp
```

2. **write a JFlex program extracting the plain text from an HTLM source**

```
java JFlex.Main html2text.flex
javac html2text.java
java html2text teaching.html
```

3. **write a JFlex program computing average exam scores per student from a score sheet (`exam.txt`)**

```
java JFlex.Main examScore.flex
javac examScore.java
java examScore exam.txt
```

FSM: solution to exercise

1.

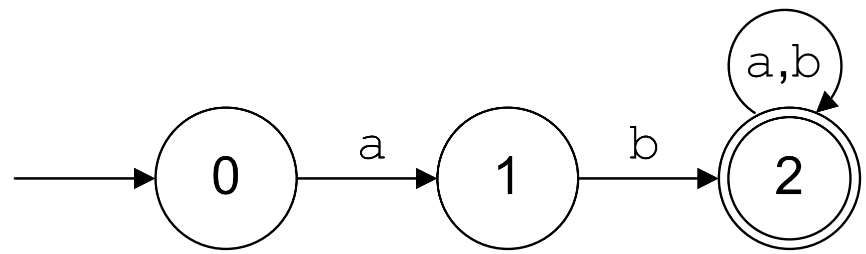
a) $r = \mathtt{ab}^*\mathtt{a} + \mathtt{bb}^*$

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

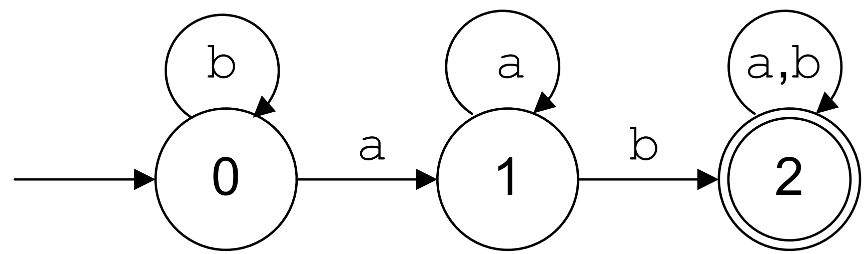
with

1. $Q = \{0, 1, 2, 3\}$
2. $\Sigma = \{\mathtt{a}, \mathtt{b}\}$
3. $\delta(0, \mathtt{a}) = 1; \delta(0, \mathtt{b}) = 2; \delta(1, \mathtt{a}) = 3; \delta(1, \mathtt{b}) = 1;$
   $\delta(2, \mathtt{a}) = \Omega; \delta(2, \mathtt{b}) = 2; \delta(3, \mathtt{a}) = \Omega; \delta(3, \mathtt{b}) = \Omega$
4. $q_0 = 0$
5. $F = \{2, 3\}$

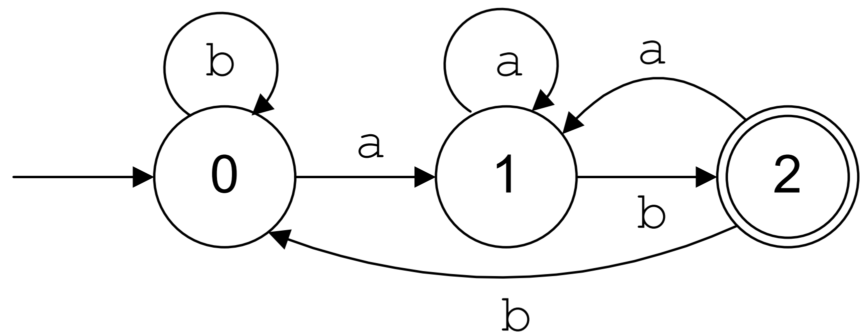
2. a) $\boldsymbol{r} = \mathtt{ab(a+b)^*}$



b) $\boldsymbol{r} = \mathtt{(a+b)^*ab(a+b)^*}$



c) $\boldsymbol{r} = \mathtt{(a+b)^*ab}$



Notes for the compiler lab project

Notes for the compiler lab project

- **important dates:**

proposal due	**May 2**
code due	**May 14**
presentations	**May 16**

- **Please submit your proposals to all of the following e-mail addresses:**

  `david@suendermann.com`

  `david@speechcycle.com`

  `suendermann@dhbw-stuttgart.de`

- **Up to two students can work together in a team.**
- **Presentations are to be in English and have a duration of 15 minutes.**