
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://www1ehre.dhbw-stuttgart.de/~stroetma/Formale-Sprachen/formale-sprachen.pdf`

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

6. context-free languages

most programming languages are context-free

7. Antlr

...a parser generator

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**

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

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

- **examples:**

1. The alphabet $\Sigma_{\text{bin}} = \{0, 1\}$ can express integers in the binary system.
2. The English language is based on the alphabet $\Sigma_{\text{en}} = \{a, \dots, z, A, \dots, Z\}$.
3. The alphabet $\Sigma_{\text{ASCII}} = \{0, \dots, 127\}$ 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	NULL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	S0	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 :

$$w = c_1 \cdots c_n \quad \text{with} \quad c_1, \dots, c_n \in \Sigma. \quad (2)$$

- The **empty word** is written as

$$w = \varepsilon. \quad (3)$$

- 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

$$w_1 = 01100 \quad \text{and} \quad w_2 = 11001 \quad \text{with} \quad w_1, w_2 \in \Sigma_{\text{bin}}^*. \quad (4)$$

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

$$w = \text{example} \quad \text{with} \quad w \in \Sigma_{\text{en}}^*. \quad (5)$$

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

$$w = \text{example} \quad \text{with} \quad w \in \Sigma_{\text{en}}^* \longrightarrow |w| = 7. \quad (6)$$

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

$$w[i] \quad \text{with} \quad i \in \{1, 2, \dots, |w|\}. \quad (7)$$

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

$$w = w_1 w_2 \dots w_n. \quad (8)$$

- **concatenation example:**

$$w_1 = 01 \quad \text{and} \quad w_2 = 10 \longrightarrow$$

$$w_1 w_2 = 0110 \quad \text{and} \quad w_2 w_1 = 1001. \quad (9)$$

(Concatenation) power of a word

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

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

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

$$\mathbb{I} = \{0, 1, \dots\}. \quad (11)$$

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

1. We define

$$L_{\mathbb{I}} = \{1w|w \in \Sigma_{\text{bin}}^*\} \cup \{0\}. \quad (12)$$

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

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

2. We define the function

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

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 $|w| > 0$.

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

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

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

$$\mathbb{P} = \{p \in \mathbb{I} \mid \{t \in \mathbb{I} \mid \exists k \in \mathbb{I} : kt = p\} = \{1, p\}\}. \quad (16)$$

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

`char* f(char* x);` (17)

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

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

$$L_u = \{f^\dagger x^\dagger y\} \quad \text{with} \quad (18)$$

- (a) $f \in L_C$,
- (b) $x, y \in \Sigma_{\text{ASCII}}^*$,
- (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**

$$L_1 \cdot L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}. \quad (19)$$

- **example:**

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

$$L_1 = \{ab, bc\} \quad \text{and} \quad L_2 = \{ac, cb\}. \quad (20)$$

The product is

$$L_1 \cdot L_2 = \{abac, abcb, bcac, bccb\}. \quad (21)$$

(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 n th power of L (recursively) as

$$L^n = L^{n-1} \cdot L \quad \text{with} \quad L^0 = \{\epsilon\}. \quad (22)$$

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

$$L = \{ab, ba\}. \quad (23)$$

This gives us

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

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

$$L^2 = \{ab, ba\} \cdot \{ab, ba\} = \{abab, abba, baab, baba\}. \quad (24)$$

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

$$L^* = \bigcup_{n \in \mathbb{I}} L^n. \quad (25)$$

- **example:**

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

$$L = \{a\}. \quad (26)$$

This gives us

$$L^* = \{a^n \mid n \in \mathbb{I}\}. \quad (27)$$

- Given the alphabet Σ_{bin} and the language

$$L = \{1\}. \quad (28)$$

- a) Formally describe the language

$$L' = L^* \setminus \{\varepsilon\}. \quad (29)$$

- b) Formally describe the set

$$D = \{d(w) \mid w \in L'\}. \quad (30)$$

- c) Formally describe the language

$$L'_- = \{w \mid w - 1 \in L'\}. \quad (31)$$

- d) Formally describe the language

$$L'_+ = \{w \mid w + 1 \in L'\}. \quad (32)$$

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

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

$$L : R \rightarrow 2^{\Sigma^*} \quad (33)$$

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.,

$$2^{\Sigma_{\text{bin}}} = 2^{\{\emptyset, \{0\}, \{1\}, \{0, 1\}\}}, \quad (34)$$

and

$$2^{\Sigma_{\text{bin}}^*} = 2^{\{\epsilon, 0, 1, 00, 01, \dots\}} \quad (35)$$

$$\begin{aligned} &= \{\emptyset, \{\epsilon\}, \{0\}, \{1\}, \{00\}, \{01\}, \dots \\ &\quad \dots \{\epsilon, 0\}, \{\epsilon, 1\}, \{\epsilon, 00\}, \{\epsilon, 01\}, \dots \\ &\quad \dots \{010, 1110, 10101\}, \dots\}. \end{aligned}$$

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

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

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

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

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

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

$$\begin{aligned} c \in \Sigma &\longrightarrow c \in R; \\ L(c) &= \{c\}. \end{aligned} \quad (38)$$

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

$$\begin{aligned} r_1 \in R, r_2 \in R &\longrightarrow r_1 + r_2 \in R; \\ L(r_1 + r_2) &= L(r_1) \cup L(r_2). \end{aligned} \quad (39)$$

5. We define the infix operator “ \cdot ” generating new regular expressions using the **product** of the languages representing the regular expressions r_1 and r_2 :

$$\begin{aligned} r_1 \in R, r_2 \in R &\longrightarrow r_1 \cdot r_2 \in R; \\ L(r_1 \cdot r_2) &= L(r_1) \cdot L(r_2). \end{aligned} \tag{40}$$

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

$$\begin{aligned} r \in R &\longrightarrow r^* \in R; \\ L(r^*) &= L^*(r). \end{aligned} \tag{41}$$

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

$$\begin{aligned} r \in R &\longrightarrow (r) \in R; \\ L((r)) &= L(r). \end{aligned} \tag{42}$$

- To save brackets, we introduce the following **operator precedences**:
 - I. “(”, “)” (strongest)
 - II. “*”
 - III. “.”
 - IV. “+” (weakest)

- **example:**

$$a + b \cdot c^* = a + (b \cdot (c^*)). \quad (43)$$

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

$$a + b \cdot c^* = a + bc^*. \quad (44)$$

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

$$\Sigma_{abc} = \{a, b, c\}. \quad (45)$$

1. The regular expression

$$r_1 = (a + b + c)(a + b + c) \quad (46)$$

describes all the words of exactly two symbols:

$$L(r_1) = \{w \in \Sigma_{abc}^* \mid |w| = 2\}. \quad (47)$$

2. The regular expression

$$r_2 = (a + b + c)(a + b + c)^* \quad (48)$$

describes all the words of one or more symbols:

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

3. The regular expression

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

describes all the words containing exactly one a:

$$L(r_3) = \{w \in \Sigma_{abc}^* \mid |\{i \in \mathbb{I} \mid w[i] = a\}| = 1\} \quad (51)$$

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

Regular expressions: exercise

- a) Using the alphabet $\Sigma_{abc} = \{a, b, c\}$, give a regular expression r_a for all the words $w \in \Sigma_{abc}^*$ containing exactly one a or exactly one b.
- b) Which language is expressed by r_a ?
- c) Using the alphabet $\Sigma_{abc} = \{a, b, c\}$, give a regular expression r_b for all the words containing at least one a and one b.
- d) Using the alphabet $\Sigma_{bin} = \{0, 1\}$, 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

$$r_f = (1 + \varepsilon)(00^*1)^*0^*? \quad (52)$$

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. $\epsilon r \doteq r$

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:

$$L(\emptyset^r) \stackrel{\text{Eq.40}}{=} L(\emptyset) \cdot L(r) \tag{57}$$

$$\stackrel{\text{Eq.36}}{=} \emptyset \cdot L(r)$$

$$\stackrel{\text{Eq.19}}{=} \{w_1 w_2 \mid w_1 \in \emptyset, w_2 \in L(r)\}$$

$$= \emptyset$$

$$\stackrel{\text{Eq.36}}{=} L(\emptyset) \quad \square$$

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

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

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

- Setting

$$r = Q^*Q, \quad (59)$$

$$s = Q, \quad (60)$$

$$t = Q, \quad (61)$$

we have

$$rs + t = Q^*QQ + Q \quad (62)$$

$\stackrel{5,7}{\doteq}$

$$(Q^*Q + \varepsilon)Q$$

$\stackrel{1,13}{\doteq}$

$$Q^*Q$$

$$= r.$$

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

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

a) Simplify the following regular expression:

$$r = 0(\varepsilon + 0 + 1)^* + (\varepsilon + 1)(1 + 0)^* + \varepsilon. \quad (64)$$

b) Prove the equivalence using only algebraic operations

$$r^* \doteq \varepsilon + r^*. \quad (65)$$

c) Prove the equivalence using only algebraic operations

$$10(10)^* \doteq 1(01)^*0. \quad (66)$$

d) Prove the equivalence

$$(1 + \varepsilon)(0(1 + \varepsilon))^*1^* \doteq (0 + 10)^*1^*. \quad (67)$$

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**

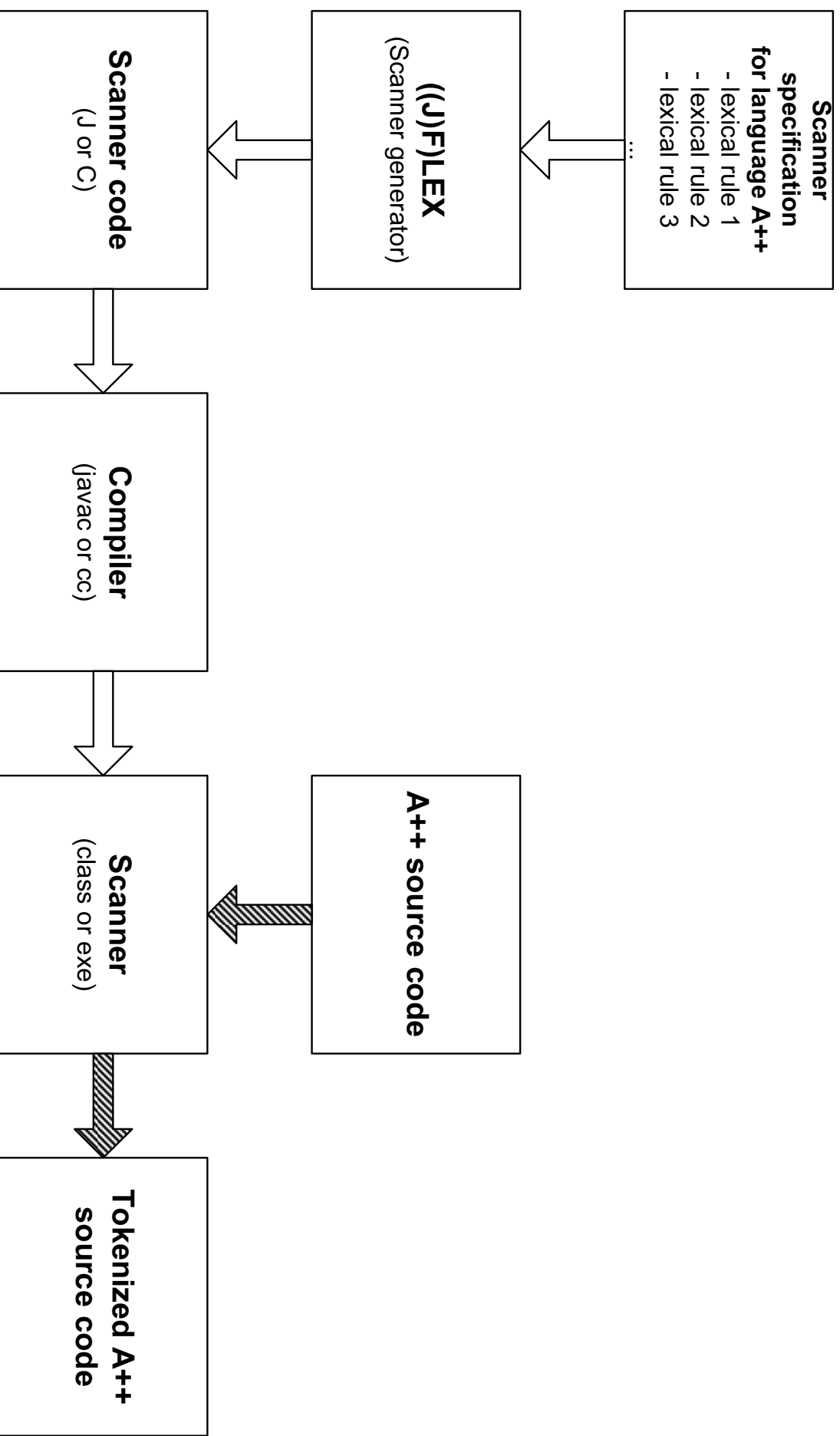
`sum=3+2;`

- **This expression would be tokenized as**

token	token type
<code>sum</code>	identifier
<code>=</code>	assignment operator
<code>3</code>	number
<code>+</code>	addition operator
<code>2</code>	number
<code>;</code>	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`

JFlex: installation (cont.)

- 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

```
1  %%
2
3  %class Count
4  %standalone
5  %unicode
6
7  %{
8      int mCount = 0;
9  %}
10
11 %eof{
12     System.out.println("Total: " + mCount);
13 %eof}
14
15 %%
16
17 [1-9] [0-9] * { mCount += new Integer(yttext()); }
18 .|\n      { /* skip */ }
```

- **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
```

- **Our example scanner adds up all integers found in an input file.**
- **An example input (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.

- **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
$$O = \{., *, +, ?, !, -, \sim, |, (,), [,], \{, \}, <, >, /, \backslash, \hat{, } \$, \}$$
(68) and the set of regular expressions
 1. $c \in \Sigma_{\text{ASCII}} \setminus O \longrightarrow c \in R$
 2. $“.” \in R$
any character but newline ($\backslash n$)

3. $x \in \{a, b, f, n, r, t, v\} \longrightarrow \backslash x \in R$

defines the following control characters

`\a` (alert)

`\b` (backspace)

`\f` (form feed)

`\n` (newline)

`\r` (carriage return)

`\t` (tabulator)

`\v` (vertical tabulator)

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

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:
$$r^+ = r r^* \tag{69}$$
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. $r \in R \longrightarrow \^r \in R$
 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|...|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. $w \in \{\Sigma_{\text{ASCII}} \setminus \{\backslash, \text{"}\}\}^* \longrightarrow \text{"}w\text{"} \in R$
verbatim text

21. $r \in R \longrightarrow !r \in R$
negation

22. $r \in R \longrightarrow \sim r \in R$

The **upto** operator matches the **shortest** string ending with r .

23. predefined character classes

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

`[::j]letterdigit::]` \longleftrightarrow `isJavaIdentifierPart()`

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

`[::d]igit::]` \longleftrightarrow `isDigit()`

`[::u]ppercase::]` \longleftrightarrow `isUppercase()`

`[::l]owercase::]` \longleftrightarrow `isLowercase()`

- I. “(”, “)” (strongest)
- II. “*”, “+”, “?”
- III. “|”
- IV. concatenation
- V. “|” (weakest)

example:

$$i a^* b | c + d e \doteq ((((! (a^*)) b) | (((c +) d) e)))$$

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. `!(~!r1|~!r2)`

intersection of two regular expressions using de Morgan's law

$$r_1 \wedge r_2 \leftrightarrow \neg(\neg r_1 \vee \neg r_2) \quad (71)$$

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 HTML 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

$$\text{avgScore} = 5 - 4 \cdot \frac{\text{sumPoints}}{\text{maxPoints}} \quad \text{with} \quad \text{maxPoints} = 60. \quad (72)$$

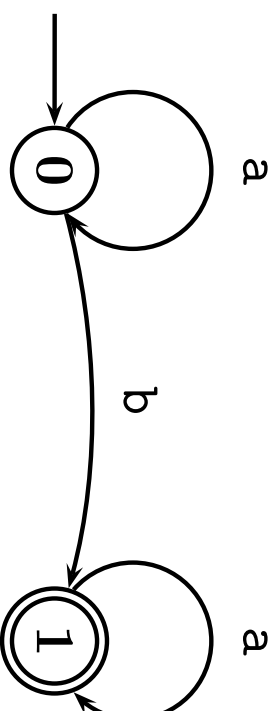
1. introduction
2. regular expressions
 - compact description of sets of strings
 - fundamental component of script languages (Perl, Python, grep, sed, awk, etc.) and of most modern programming languages (.NET, SQL Server 2008, Java, etc.)
3. the scanner generator JFlex
4. **finite-state machines**
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

- 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.**

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.

- 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

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle \quad (74)$$

with

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

- 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

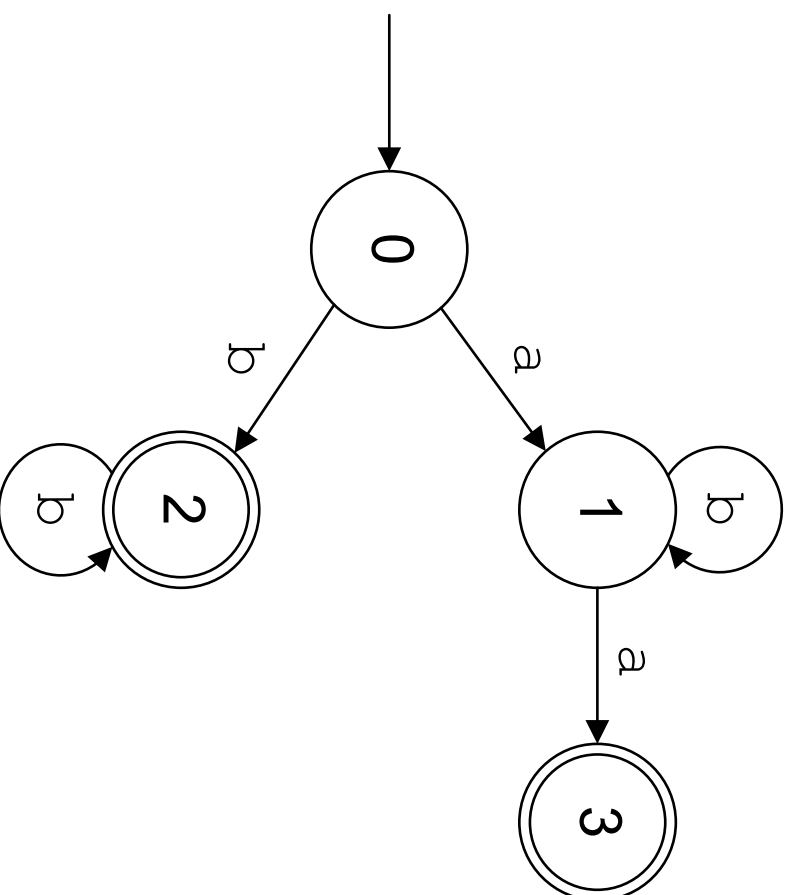
$$\begin{aligned} - \delta'(q, \varepsilon) &= q \\ - \delta'(q, w) &= \begin{cases} \delta'(\delta(q, c), v) & \text{if } \delta(q, c) \neq \Omega \\ \Omega & \text{otherwise} \end{cases} \end{aligned}$$

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

- E.g., we can show that $\delta'(0, \text{aba}) = 1$ 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

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

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

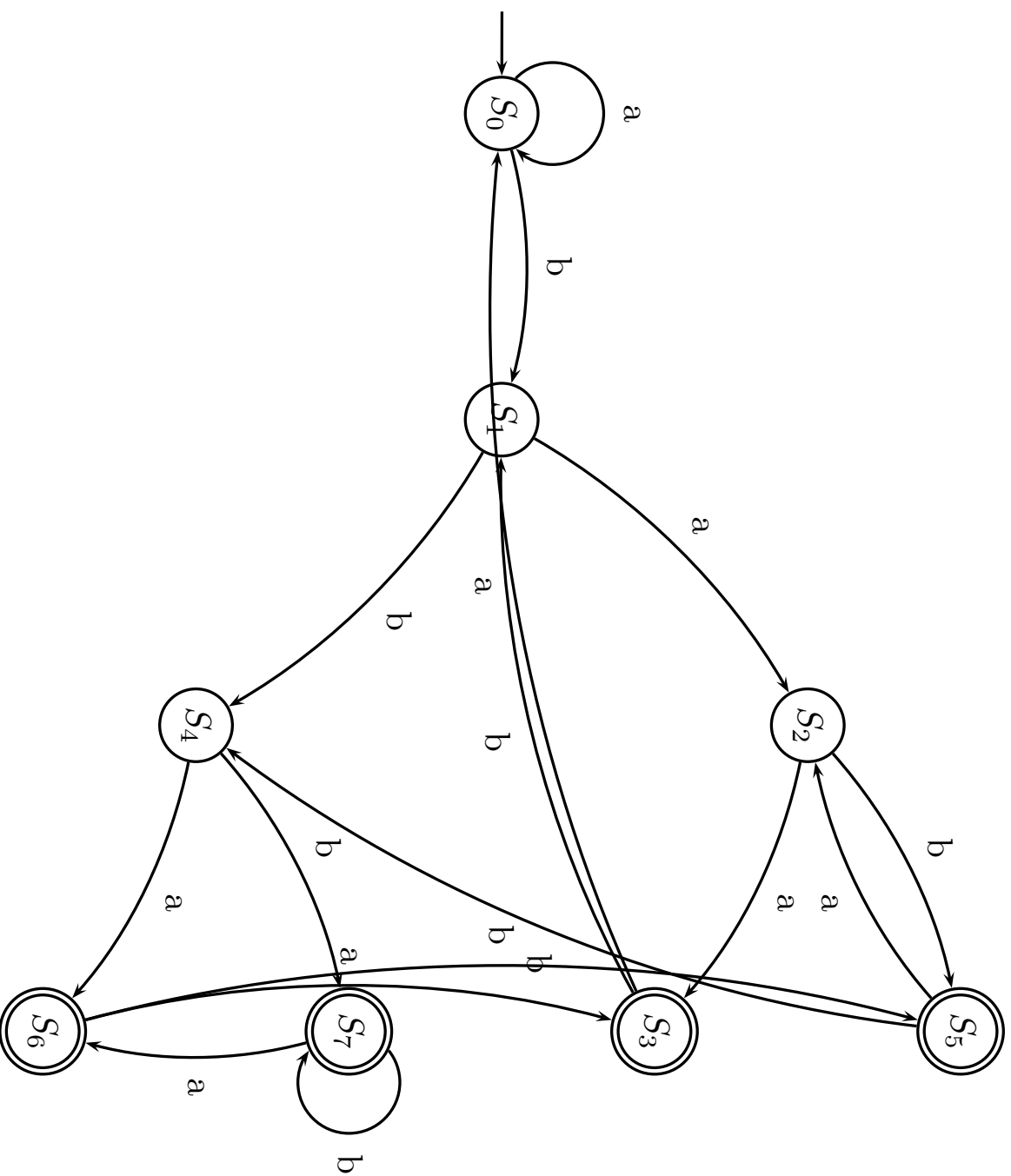
- a regular expression,
- a graphical representation, and
- a formal definition

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

- a) at the beginning,
- b) at arbitrary position,
- c) at the end.

- 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
$$(a + b)^*b(a + b)(a + b) \quad (77)$$

FSM: example of a DFA

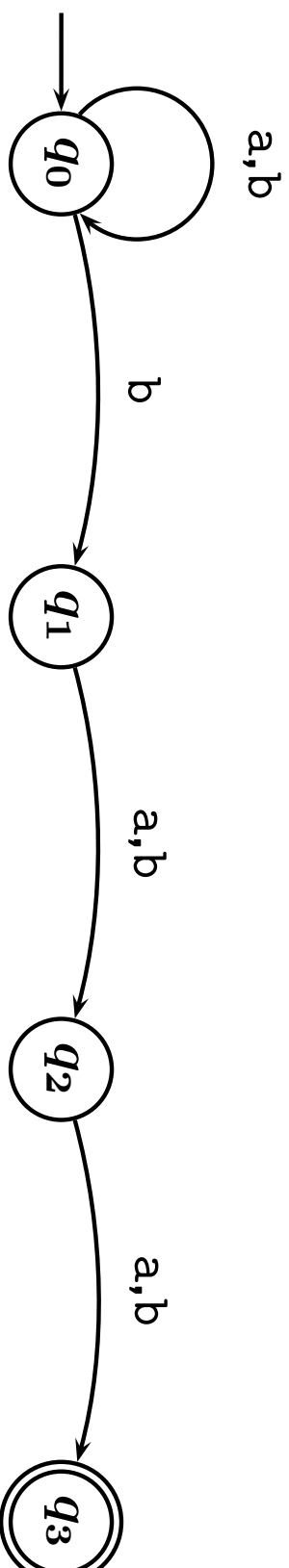


- We can simplify such an FSM when we permit that an input can lead to
 - one transition,
 - multiple transitions, or
 - no transition.
- That is, an FSM selects its next state from a set of states where the set depends on the current state and the input.
- We call this a **non-deterministic** FSM or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

(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. $q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0$ (failure)
 2. $q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1$ (failure)
 3. $q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3$ (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:

$$q_1 \xrightarrow{\epsilon} q_2. \tag{79}$$

- 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 $Q \times \{\Sigma \cup \{\epsilon\}\} \times Q$. I.e.,
$$\delta \subseteq Q \times \{\Sigma \cup \{\epsilon\}\} \times Q \tag{81}$$
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

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle \quad (82)$$

with

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

- **Given an FSM A whose language $L(A) \subset \{a, b\}^*$ 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 $\text{det}(A)$ sharing the same language, i.e.

$$L(A) = L(\text{det}(A)) \quad (83)$$

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

- First, the ε closure

$$ec : Q \rightarrow 2^Q \quad (84)$$

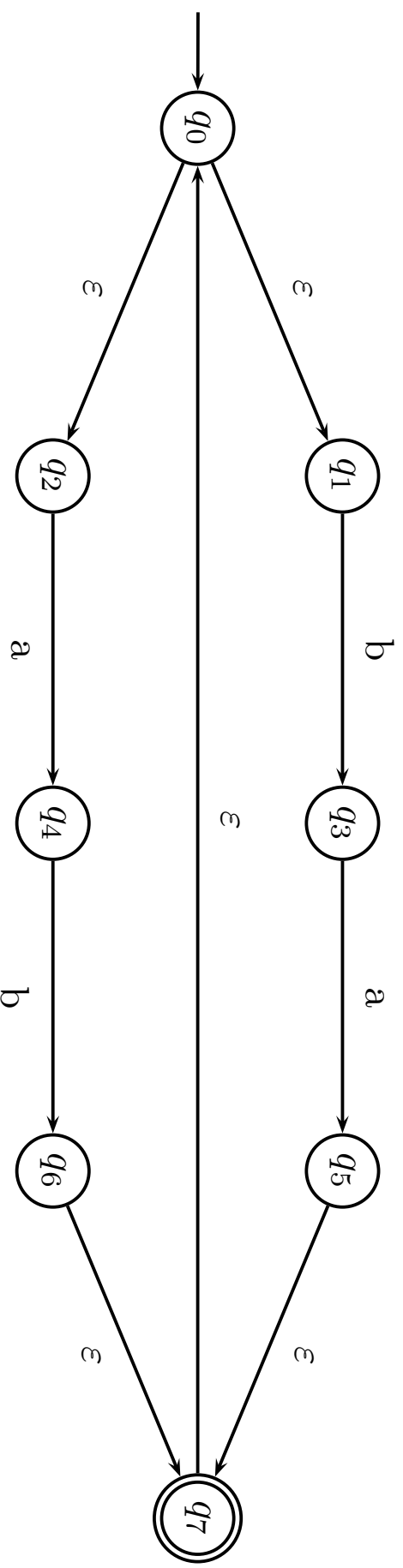
returns the set of all those states, the NFA can change to by means of an ε transition coming from state q .

- Formal definition of ec :

$$q \in ec(q); \quad (85)$$

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

- 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

$$\delta^* : Q \times \Sigma \rightarrow 2^Q. \quad (87)$$

- 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

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

- examples (based on the above NFA):

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

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

$$\Delta^* : 2^Q \times \Sigma \rightarrow 2^Q. \quad (89)$$

- 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

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

- examples (based on the above NFA):

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

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

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

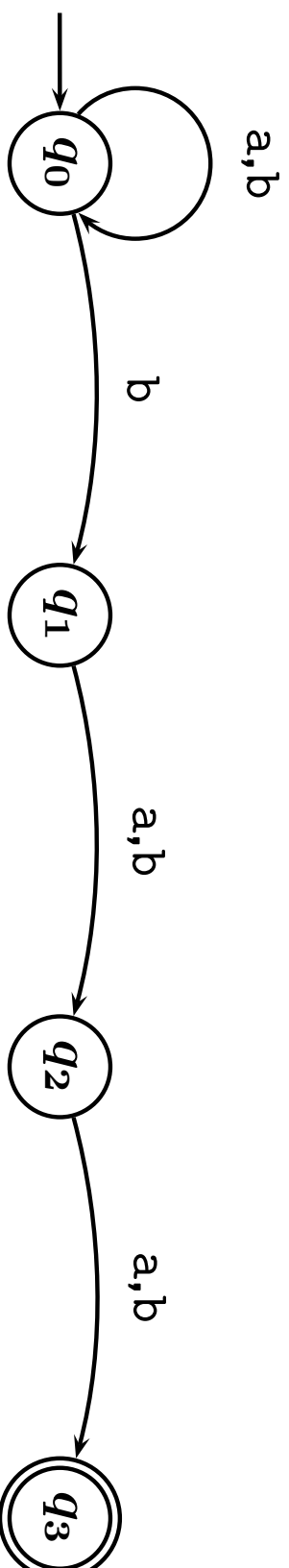
with

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

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

- returning to the example FSM expressing the regular expression

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



- The initial state:

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

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

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

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

Equivalence of DFA and NFA: example (cont.)

- Now, we can define the DFA

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

with

- the set of states

$$\hat{Q} = \{S_0, \dots, S_7\}, \quad (96)$$

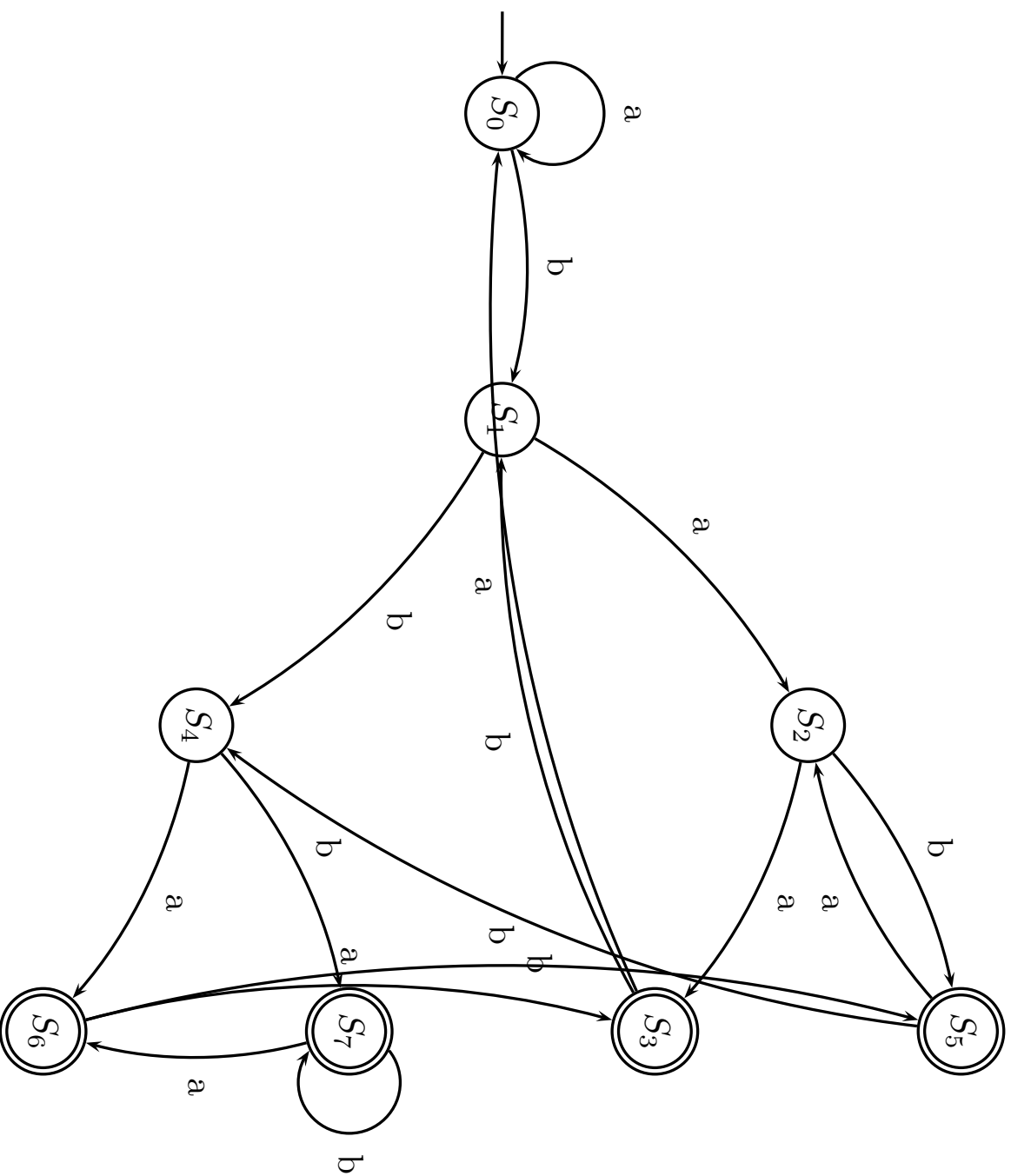
- the state transition function Δ^* as summarized as follows:

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

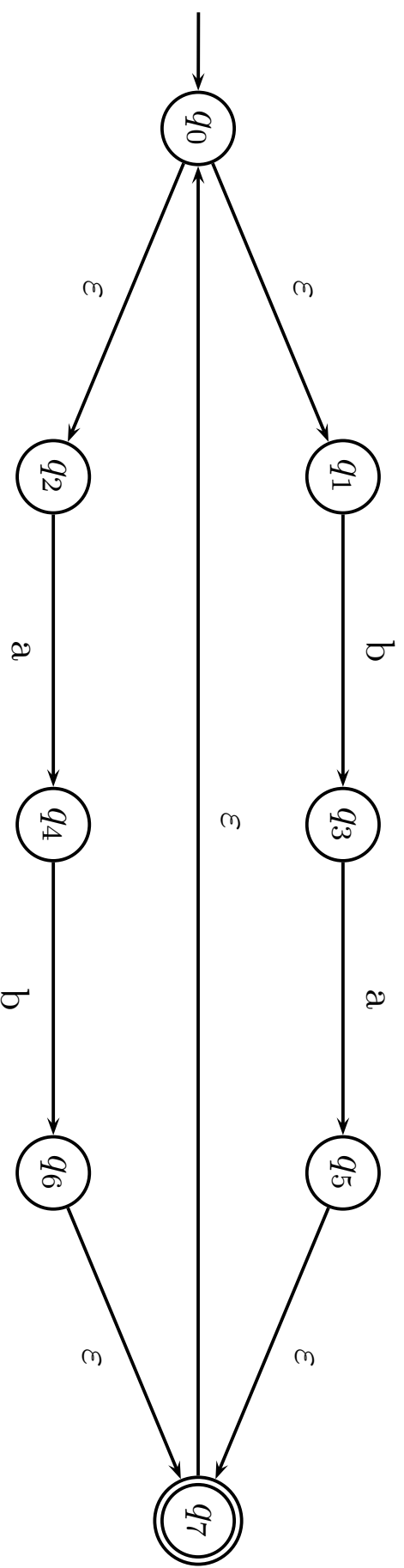
- and the set of final states (each DFA state containing the NFA final state q_3)

$$\hat{F} = \{S_3, S_5, S_6, S_7\}. \quad (97)$$

Equivalence of DFA and NFA: example (cont.)



- We are given the following NFA A :

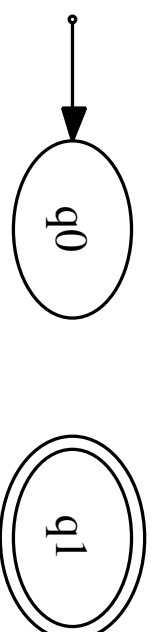


- Determine $\text{det}(A)$.**
- Draw $\text{det}(A)$'s graph.**
- Give a regular expression representing the same language as A .**

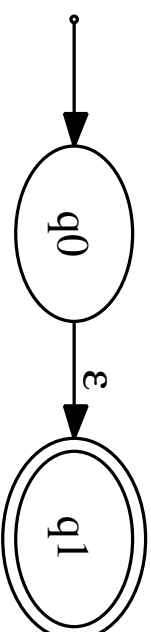
- Given a regular expression r , we want to derive an NFA $A(r)$ accepting the same language:

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

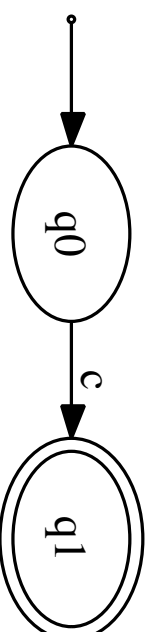
- Deriving the transformation rules, we will be using two properties of $A(r)$:
 - There are no transitions to the initial state.
 - There are no transitions from the final state.
- Assuming Σ is the alphabet which r is based on, we define
 1. $A(\emptyset) = \langle \{q_0, q_1\}, \Sigma, \{q_0, q_1\} \rangle$



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

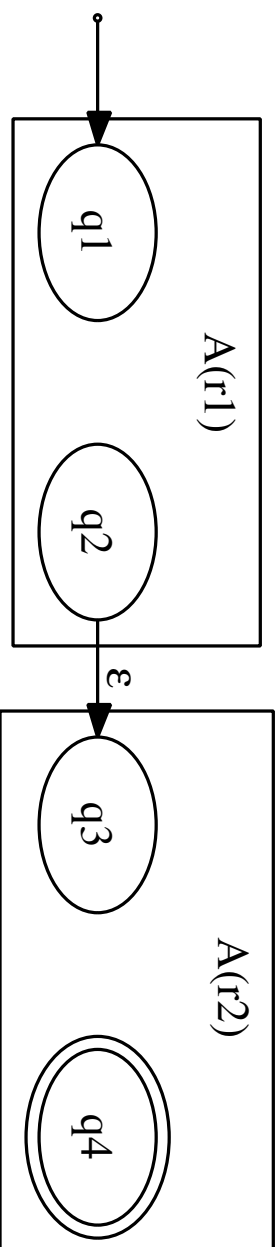


3. $A(c) = \langle \{q_0, q_1\}, \Sigma, \{\langle q_0, c, q_1 \rangle\}, q_0, \{q_1\} \rangle$

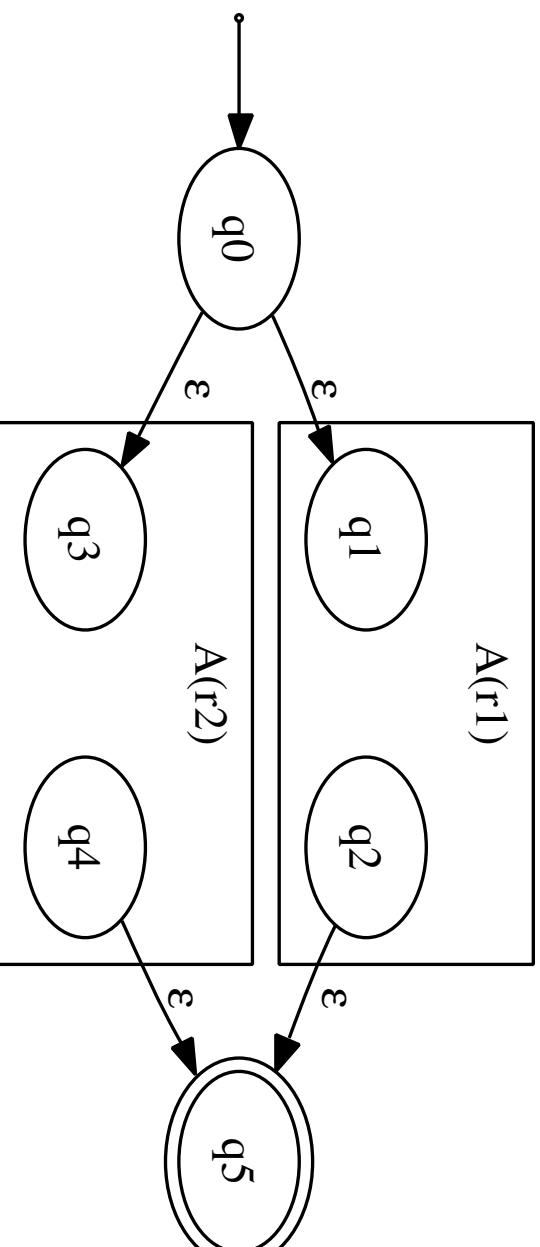


4. $A(r_1 r_2) = \langle Q_1 \cup Q_2, \Sigma, \{\langle q_2, \varepsilon, q_3 \rangle\} \cup \delta_1 \cup \delta_2, q_0, \{q_4\} \rangle$ with
 $A(r_1) = \langle Q_1, \Sigma, \delta_1, q_1, \{q_2\} \rangle,$
 $A(r_2) = \langle Q_2, \Sigma, \delta_2, q_3, \{q_4\} \rangle.$

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



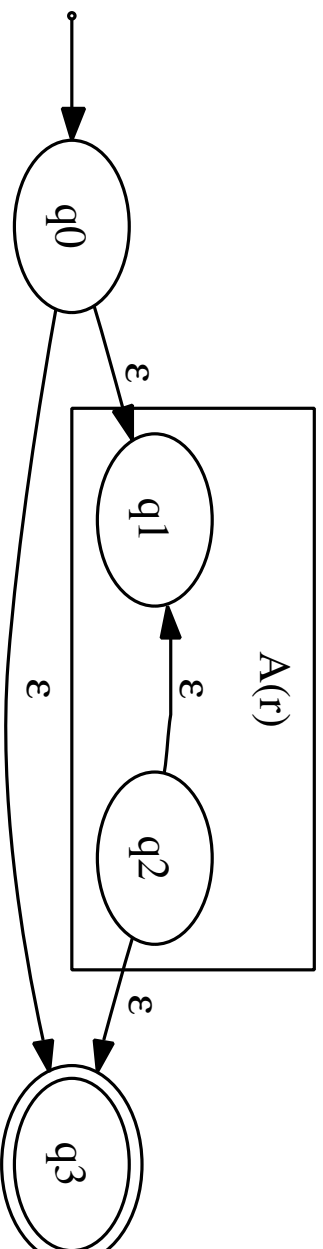
5. $A(r_1 + r_2) = \langle \{q_0, q_5\} \cup Q_1 \cup Q_2, \Sigma, \{ \langle q_0, \epsilon, q_1 \rangle, \langle q_0, \epsilon, q_3 \rangle, \langle q_2, \epsilon, q_5 \rangle, \langle q_4, \epsilon, q_5 \rangle \} \cup \delta_1 \cup \delta_2, q_0, \{q_5\} \rangle$



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**

$$(a + b)a^*b$$

(99)

- 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:

$$L(r(A)) = L(A). \quad (100)$$

- The DFA to be converted is of the form

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

- Now, we introduce the auxiliary regular expression

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

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 :

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

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

Transformation of DFAs into regular expressions: formal derivation ($k > 0$)

- For $k > 0$, we have

$$r^{(k)}(p_1, p_2) = r^{(k-1)}(p_1, p_2) + \quad (104)$$

$$r^{(k-1)}(p_1, q_{k-1}) \cdot \quad (105)$$

$$\left(r^{(k-1)}(q_{k-1}, q_{k-1}) \right)^* \cdot \quad (106)$$

$$r^{(k-1)}(q_{k-1}, p_2) \quad (107)$$

- 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).

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

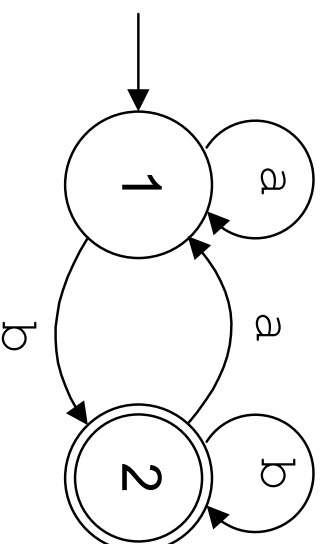
$$r(p_1, p_2) = r^{(n+1)}(p_1, p_2). \quad (108)$$

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

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

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

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



- Given the DFA

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

we want to derive a DFA

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

accepting the same language, i.e.,

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

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

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

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

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

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 :

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

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

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**:

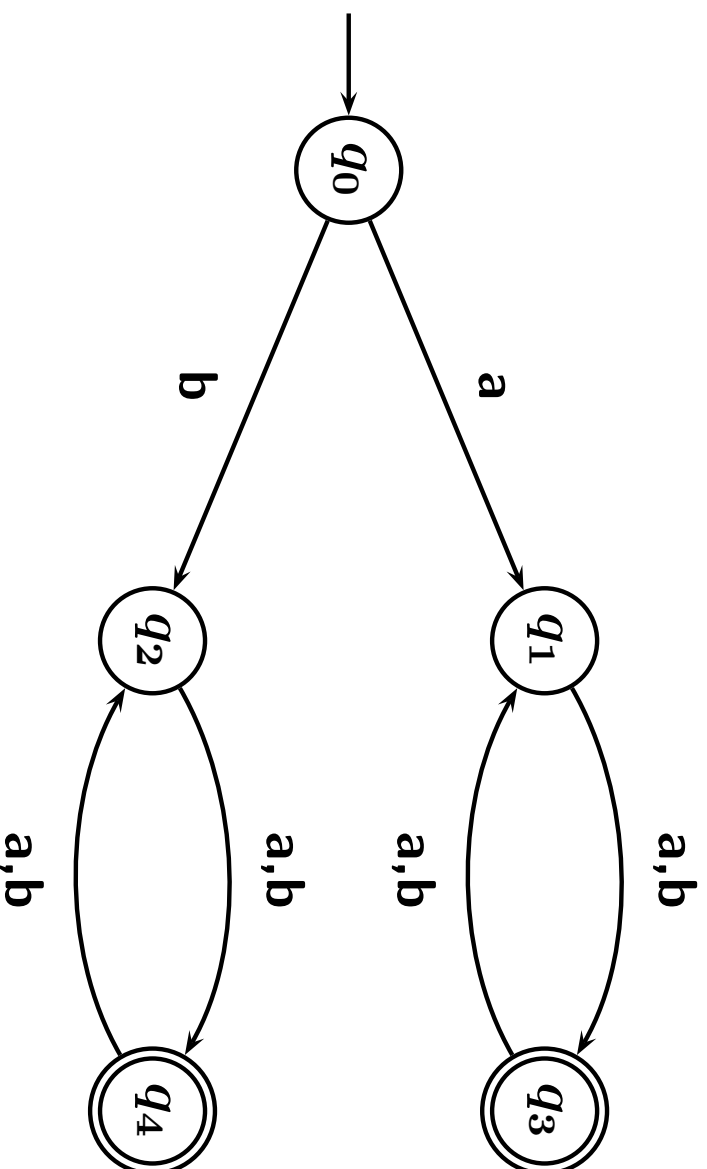
$$\delta(p, c) = \Omega \wedge \delta(q, c) \neq \Omega \vee \delta(p, c) \neq \Omega \wedge \delta(q, c) = \Omega \quad (116)$$

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**:

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

- We want to minimize this DFA with 5 states:



- This is the formal definition of the DFA:

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle \quad (118)$$

with

1. $Q = \{q_0, q_1, q_2, q_3, q_4\}$
 2. $\Sigma = \{a, b\}$
 3. $\delta = \dots$ (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 $Q \setminus F = \{q_0, q_1, q_2\}$, 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 $\langle q_i, q_i \rangle | i \in \{0, \dots, 4\}$ 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 $\langle q_i, q_j \rangle | i, j \in \{0, \dots, 4\}; j < i$ due to the **symmetry** of the distinguishability of states.

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

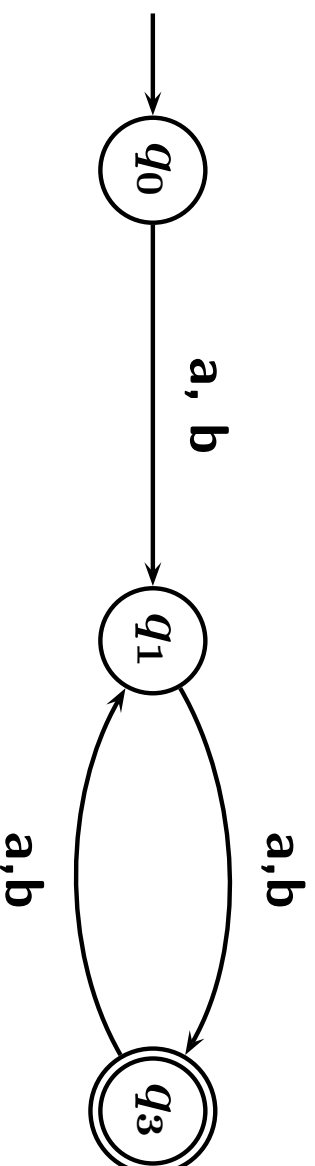
Minimization of DFAs: example (cont.)

- 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**

$$L(a(ba)^*).$$

(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:

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

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) $10(10)^* \doteq 1(01)^*0$,

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

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**

- 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 quadruple

$$G = \langle V_N, V_T, P, S \rangle \quad (121)$$

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

$$\alpha \rightarrow \beta \quad (122)$$

with $\alpha \in V^*V_NV^*$, $\beta \in V^*$, $V = V_N \cup V_T$

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

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

$$\alpha \rightarrow \beta_1 | \dots | \beta_n \quad \text{replacing} \quad \alpha \rightarrow \beta_1 \quad (123)$$

⋮

$$\alpha \rightarrow \beta_n$$

- We want to define a grammar

$$G = \langle V_N, V_T, P, S \rangle \quad (124)$$

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. $V_T = \{a, \dots, z, A, \dots, Z, 0, \dots, 9, _ \}$,
3. $P : \begin{array}{l} I \rightarrow LR_R|L_ \\ R \rightarrow LR|DR|_R|L|D|_ \\ L \rightarrow a|\dots|z|A|\dots|Z \\ D \rightarrow 0|\dots|9 \end{array}$
4. $S = I$.

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

$$G = \langle V_N, V_T, P, S \rangle, \quad (125)$$

we define the relation

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

pronounced as “ **G derives in one step**”.

- We also define the relation

$$x \Rightarrow_G^* y \text{ iff } \exists w_0, \dots, w_n \quad (127)$$

with $w_0 = x, w_n = y, w_{i-1} \Rightarrow_G w_i$ for $i \in \{1, \dots, n\}$
pronounced as “ **G derives in zero or more steps**”.

- We are given the grammar

$$G = \langle V_N, V_T, P, S \rangle \quad (128)$$

with

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

- Derivations of G have the general form

$$S \Rightarrow_1 0S \Rightarrow_1 00S \Rightarrow_1 \dots \Rightarrow_1 0^{n-1}S \Rightarrow_2 0^n. \quad (129)$$

- Apparently, the language accepted by G is

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

- 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. $P : \begin{array}{l} S \rightarrow 0S1 \quad 1 \\ S \rightarrow 01 \quad 2 \end{array}$
4. $S = S$.

- Derivations of G have the general form

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

- Apparently, the language accepted by G is

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

- We are given the grammar

$$G = \langle V_N, V_T, P, S \rangle \quad (134)$$

with

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

$S \rightarrow 0SBC$	1
$S \rightarrow 0BC$	2
$CB \rightarrow BC$	3
$0B \rightarrow 01$	4
$1B \rightarrow 11$	5
$1C \rightarrow 12$	6
$2C \rightarrow 22$	7
4. $S = S$.

- Derivations of G have the general form

$$\begin{aligned} S &\Rightarrow_1 0SBC \Rightarrow_1 00SBCBC \Rightarrow_1 \dots \Rightarrow_1 0^{n-1}S(BC)^{n-1} \Rightarrow_2 0^n(BC)^n \\ &\Rightarrow_3^* 0^n B^n C^n \Rightarrow_{4,5}^* 0^n 1^n C^n \Rightarrow_{6,7}^* 0^n 1^n 2^n \end{aligned} \quad (135)$$

- The language accepted by G is

$$L(G) = \{0^n 1^n 2^n \mid n \in \mathbb{I}; n > 0\}. \quad (136)$$

- 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

$$G = \langle V_N, V_T, P, S \rangle, \quad (137)$$

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

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

Exception:

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

The Chomsky hierarchy (cont.)

- **Type 2 or context-free**

if all productions are of the form

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

Exception:

$$\langle S \rightarrow \epsilon \rangle \in P \longrightarrow \beta \in (V \setminus \{S\})(V \setminus \{S\})^* \quad (141)$$

- **Type 3 or regular**

if all productions are of the form

$$A \rightarrow aB \text{ or} \quad (142)$$

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

Exception:

$$\langle S \rightarrow \epsilon \rangle \in P \longrightarrow B \in V_N \setminus \{S\} \quad (143)$$

- 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

- 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:

$$P : I \rightarrow LR_R|L|_$$

$$R \rightarrow LR|DR|_R|L|D|_$$

$$L \rightarrow a|\dots|z|A|\dots|Z$$

$$D \rightarrow 0|\dots|9$$

- This grammar is context-free but not regular.

- An equivalent regular grammar could have the following productions:

$$P : I \rightarrow A|\dots|Z|a|\dots|z|_|$$

$$AR|\dots|ZR|aR|\dots|zR|_R$$

$$R \rightarrow A|\dots|Z|a|\dots|z|_|0|\dots|9|$$

$$AR|\dots|ZR|aR|\dots|zR|_R|0R|\dots|9R$$

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

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

$$CB \rightarrow CX$$

(144)

$$CX \rightarrow BX$$

$$BX \rightarrow BC$$

without changing the grammar's behavior.

- The resulting grammar is context-sensitive.
- So is the accepting language $L = \{0^n 1^n 2^n \mid n \in \mathbb{I}; n > 0\}$.

I. We are given the grammar

$$G = \langle V_N, V_T, P, S \rangle \quad (145)$$

with

1. $V_N = \{S, A, B\}$,

2. $V_T = \{0\}$,

3. $P:$ $S \rightarrow \epsilon$ 1

$S \rightarrow ABA$ 2

$AB \rightarrow 00$ 3

$0A \rightarrow 000A$ 4

$A \rightarrow 0$ 5

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 ranging from 0 to 7. Define a regular grammar encoding exactly the set of possible octal constants.

III. We are given the grammar

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

with

1. $V_N = \{S, N, E\}$,
2. $V_T = \{0, 1, t\}$,
3. $P : \begin{array}{ll} S \rightarrow 0NS & 1 \\ S \rightarrow 1ES & 2 \\ S \rightarrow t & 3 \\ Nt \rightarrow t0 & 4 \\ Et \rightarrow t1 & 5 \end{array}$

The Chomsky hierarchy: exercises (cont.)

$$N0 \rightarrow 0N \quad 6$$

$$N1 \rightarrow 1N \quad 7$$

$$E0 \rightarrow 0E \quad 8$$

$$E1 \rightarrow 1E \quad 9$$

4. $S = S$.

a) What is G 's highest type?

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

6. context-free languages

most programming languages are context-free

7. Antlr

...a parser generator

- Given a language L , the **pumping lemma** is a way to **disprove** the regularity of L .
- Informally, it 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.
 - 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. $v \neq \epsilon$,
 3. $|uv| \leq n$,
 4. $\forall h \in \mathbb{I} : uv^h w \in L$.

The pumping lemma: example

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

$$L = \{ ({}^k) {}^k \mid k \in \mathbb{I} \}. \quad (147)$$

- Using Condition 1 of the pumping lemma, we have

$$s = uvw = ({}^k) {}^k, \quad k \in \mathbb{I}. \quad (148)$$

- Apparently, the middle term v must have the form

$$v = ({}^l) {}^m \quad (149)$$

with

$$- l, m \in \mathbb{I},$$

$$- l + m > 0 \quad (\text{due to Condition 2}),$$

$$- l \leq k,$$

$$- m \leq k.$$

The pumping lemma: example (cont.)

- **Case $l = 0$**
 - We have
$$u = ({}^k) m', v = () m, w = () k - m - m'. \quad (150)$$
 - **Applying Condition 4 leads to**
$$s' = ({}^k) m' ({}^h m) k - m - m' \quad \text{for } h \in \mathbb{I}. \quad (151)$$
 - The word s' can only be member of the language L as given in Equation 147 if the number of opening and closing brackets are identical:
$$k \stackrel{!}{=} m' + h m + k - m - m'. \quad (152)$$
 - This holds only for $h = 1$ and not arbitrary $h \in \mathbb{I}$, so, $l = 0$ is not a valid case.

The pumping lemma: example (cont.)

- **Case $m = 0$**
 - **We have**
$$u = (x^{k-l-l'}, v = (x^l, w = (x^{l'})^k. \tag{153}$$
 - **Applying Condition 4 leads to**
$$s' = (x^{k-l-l'} (x^{hl} (x^{l'})^k) \text{ for } h \in \mathbb{I}. \tag{154}$$
 - **The word s' can only be member of the language L as given in Equation 147 if the number of opening and closing brackets are identical:**
$$k - l - l' + hl + l' \stackrel{!}{=} k. \tag{155}$$
 - **This holds only for $h = 1$ and not arbitrary $h \in \mathbb{I}$, so, $l = 0$ is not a valid case.**

The pumping lemma: example (cont.)

- Case $l > 0, m > 0$

– We have

$$u = ({}^{k-l}, v = ({}^l)_m, w = ({}^{k-m}) \quad (156)$$

– Applying Condition 4 leads to

$$s' = ({}^{k-l} (({}^l)_m)_h)^{k-m} \quad \text{for } h \in \mathbb{I}. \quad (157)$$

– The word s' can only be member of the language L as given in Equation 147 if the number of opening and closing brackets are identical:

$$k - l + lh \stackrel{!}{=} mh + k - m. \quad (158)$$

– This holds for arbitrary $h \in \mathbb{I}$ only $l = m$.

– However, inserting the example case $h = 2$ into Equation 157

$$s'_{h=2} = ({}^k)_l ({}^m)_k. \quad (159)$$

shows that s' will only be accepted for $l = m = 0$, i.e., $v = \varepsilon$ contradicting Condition 2.

The pumping lemma: example (cont.)

- In conclusion, we see that the language

$$L = \{ ({}^k) {}^k \mid k \in \mathbb{I} \}.$$

(160)

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 a^n where n is a square number:

$$L = \{a^{2^n} \mid n \in \mathbb{I}\}. \quad (161)$$

- Prove that L is not a regular language.

The pumping lemma: exercise II

- We are given the language

$$L = \{a^k b^l \mid k, l \in \mathbb{I}\}.$$

(162)

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

6. context-free languages

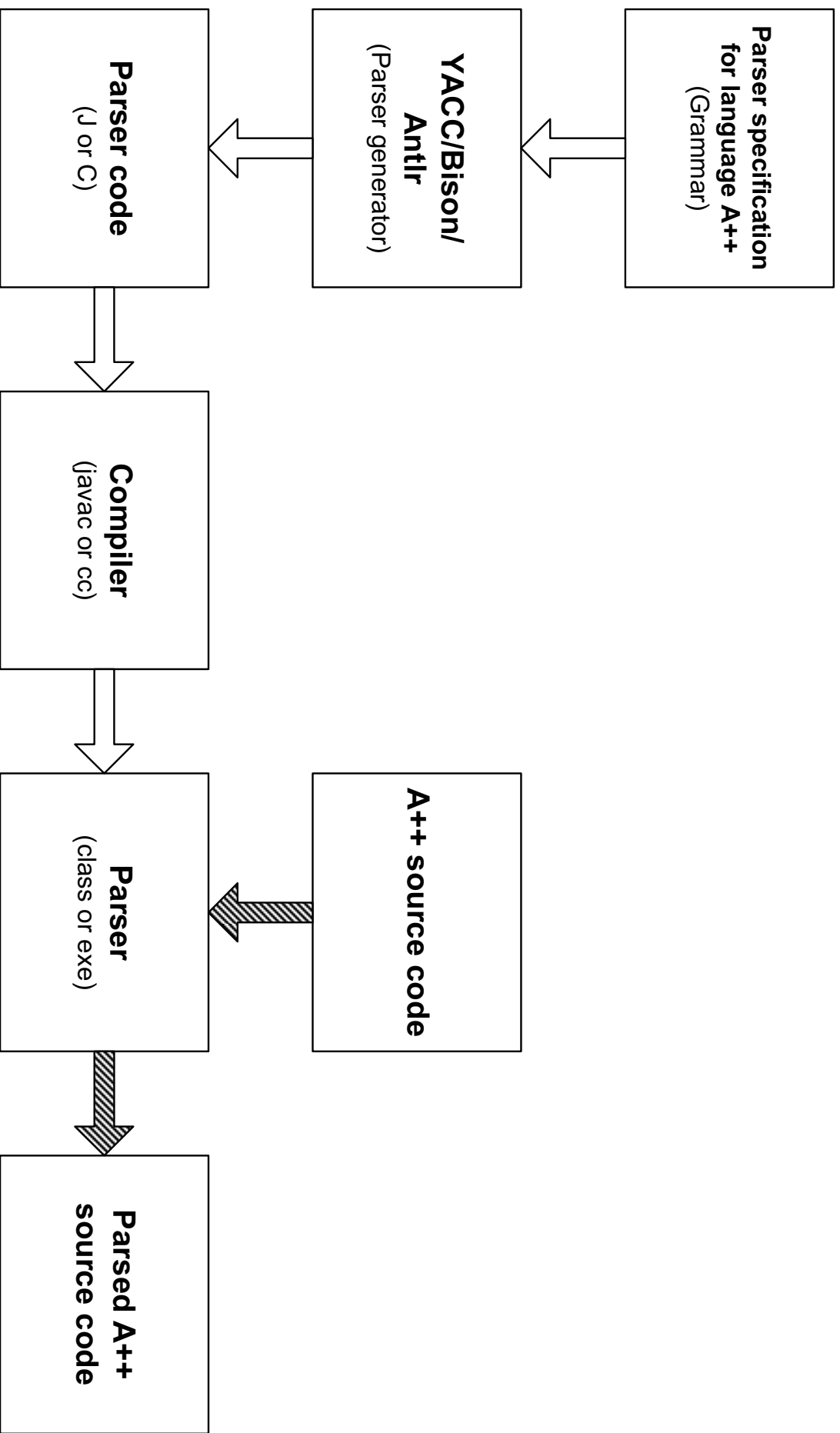
most programming languages are context-free

7. *Antlr*

...a parser generator

- **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](http://antlr.org/download.html) 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.

- This is a grammar describing arithmetic expressions:

$$S \rightarrow E$$

$$E \rightarrow P(('+' | '-') P) *$$

$$P \rightarrow F(('*' | '/') F) *$$

$$F \rightarrow '(' E ') | N$$

$$N \rightarrow ('1' .. '9') ('0' .. '9') * \quad (163)$$

- Words described 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:

```
1  grammar expr;
2
3  start: expr;
4  expr: product('(' '+' | '-'') product)*;
5  product: factor('(' '*' | '/'') factor)*;
6  factor: '(' expr')' | NUMBER;
7  NUMBER: ('1'..'9') ('0'..'9')*;
8  WS: (' ' | '\t' | '\n' | '\r') {skip();};
```

Grammars in Antlr (cont.)

- 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:

```
1  import org.antlr.runtime.*;
2
3  public class ParseExpr
4  {
5      public static void main(String[] args) throws Exception
6      {
7          ANTLRInputStream input = new ANTLRInputStream(System.in);
8          exprLexer lexer = new exprLexer(input);
9          CommonTokenStream ts = new CommonTokenStream(lexer);
10         exprParser parser = new exprParser(ts);
11         parser.expr();
12     }
13 }
```

- 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>`, `
`).
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.

- 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()};
```

- **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`).**

- 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 x s (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(); };
```

- **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 accommodating 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

$$L = \{1\}. \tag{164}$$

- a) Formally describe the language

$$L' = L^* \setminus \{\epsilon\}. \tag{165}$$

- According to Equation 25, we have

$$\begin{aligned} L' &= \{L^0 \cup L^1 \cup L^2 \cup \dots\} \setminus \{\epsilon\} \\ &= \{\epsilon, 1, 11, \dots\} \setminus \{\epsilon\} \\ &= \{1, 11, \dots\} \\ &= \{11^n \mid n \in \mathbb{I}\}. \end{aligned} \tag{166}$$

b) Formally describe the set

$$D = \{d(w) \mid w \in L'\}. \quad (167)$$

– Using Equations 166 and 14, we have

$$\begin{aligned} D &= \{d(1), d(11), d(111), \dots\} \\ &= \{1, 3, 7, \dots\} \\ &= \{2 \cdot 2^n - 1 \mid n \in \mathbb{I}\} \end{aligned} \quad (168)$$

c) Formally describe the language

$$L'_- = \{w \mid w - 1 \in L'\}. \quad (169)$$

– The condition

$$w - 1 \in \{1, 11, 111, \dots\} \quad (170)$$

is equivalent to

$$w \in \{1 + 1, 11 + 1, 111 + 1, \dots\}. \quad (171)$$

– Hence, we have

$$\begin{aligned} L'_- &= \{1 + 1, 11 + 1, 111 + 1, \dots\} & (172) \\ &= \{10, 100, 1000, \dots\} \\ &= \{100^n \mid n \in \mathbb{I}\}. \end{aligned}$$

d) Formally describe the language

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

– The condition

$$w + 1 \in \{1, 11, 111, \dots\} \quad (174)$$

is equivalent to

$$w \in \{1 - 1, 11 - 1, 111 - 1, \dots\}. \quad (175)$$

– Hence, we have

$$\begin{aligned} L'_+ &= \{1 - 1, 11 - 1, 111 - 1, \dots\} \\ &= \{0, 10, 110, \dots\} \\ &= \{1^n 0 | n \in \mathbb{I}\}. \end{aligned} \quad (176)$$

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

- Similarly to Equation 50, we have

$$r_a = (b + c)^*a(b + c)^* + (a + c)^*b(a + c)^* \quad (177)$$

b) Which language is expressed by r_a ?

- Similarly to Equation 51, we have

$$L(r_a) = \{w \in \Sigma_{abc}^* \mid |\{i \in \mathbb{I} \mid w[i] = a\}| = 1 \vee |\{i \in \mathbb{I} \mid w[i] = b\}| = 1\} \quad (178)$$

- Alternatively, one can write

$$\begin{aligned} L(r_a) &= \{w \in \Sigma_{abc}^* \mid |\{i \in \mathbb{I} \mid w[i] = a\}| = 1\} \cup \\ &\quad \{w \in \Sigma_{abc}^* \mid |\{i \in \mathbb{I} \mid w[i] = b\}| = 1\} \end{aligned} \quad (179)$$

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

$$r_a = (a + b + c)^*a(a + b + c)^*b(a + b + c)^* + (a + b + c)^*b(a + b + c)^*a(a + b + c)^* \quad (180)$$

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

$$r_d = (0 + 1)^* 1 (0 + 1)(0 + 1) \quad (181)$$

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

$$r_e = 0^*(100^*)^*1^*. \quad (182)$$

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

$$\epsilon, 0, 1, 0^*, 1^*, 0^*1^*, 0^*10^* \in L(r_e) \quad (183)$$

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

$$110, 0^*111^*0 \notin L(r_e). \quad (184)$$

f) Which language is expressed by the regular expression

$$r_f = (1 + \varepsilon)(00^*1)^*0^*? \quad (185)$$

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

$$\varepsilon, 0, 1, 0^*, 10^*, 0^*10^* \in L(r_f) \quad (186)$$

and that others that are not, e.g.

$$11, 111^* \notin L(r_f). \quad (187)$$

- 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 occurrences of the string 11:

$$L(r_f) = \{w \in \Sigma_{\text{bin}}^* \mid |\{i \in \mathbb{I} \mid w[i]w[i+1] = 11\}| = 0\}. \quad (188)$$

a) Simplify the following regular expression:

$$r = 0(\varepsilon + 0 + 1)^* + (\varepsilon + 1)(1 + 0)^* + \varepsilon. \quad (189)$$

$$r \stackrel{\cdot}{=} 0(\varepsilon + 0 + 1)^* + (\varepsilon + 1)(1 + 0)^* + \varepsilon \quad (190)$$

$$\stackrel{14,1}{\dot{=}} 0(0 + 1)^* + (\varepsilon + 1)(0 + 1)^* + \varepsilon$$

$$\stackrel{7,5}{\dot{=}} 0(0 + 1)^* + (0 + 1)^* + 1(0 + 1)^* + \varepsilon$$

$$\stackrel{1,7}{\dot{=}} \varepsilon + (0 + 1)(0 + 1)^* + (0 + 1)^*$$

Eq.58,13

$$\stackrel{\cdot}{=} (0 + 1)^* + (0 + 1)^*$$

$$\stackrel{9}{\dot{=}} (0 + 1)^*.$$

b) Prove the equivalence using only algebraic operations

$$r^* \doteq \varepsilon + r^* . \tag{191}$$

$$\varepsilon + r^* \stackrel{13}{\doteq} \varepsilon + \varepsilon + r^* r \tag{192}$$

$$\stackrel{9}{\doteq} \varepsilon + r^* r$$

13

$$\stackrel{13}{\doteq} r^* \square$$

c) Prove the equivalence using only algebraic operations

$$10(10)^* \doteq 1(01)^*0. \quad (193)$$

• We set

$$r = 1(01)^*0, \quad (194)$$

$$s = 10, \quad (195)$$

$$t = 10. \quad (196)$$

- This yields

$$rs + t = 1(01)^*010 + 10 \quad (197)$$

$$\stackrel{8}{=} 1((01)^*010 + 0)$$

$$\stackrel{5,7}{=} 1((01)^*01 + \varepsilon)0$$

$$\stackrel{13}{=} 1(01)^*0$$

$$= r.$$

(198)

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

$$1(01)^*0 = r \stackrel{15}{=} ts^* = 10(10)^* \quad \square \quad (199)$$

- 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 HTML 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
```

1.

a) $r = ab^*a + bb^*$

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

with

1. $Q = \{0, 1, 2, 3\}$

2. $\Sigma = \{a, b\}$

3. $\delta(0, a) = 1; \delta(0, b) = 2; \delta(1, a) = 3; \delta(1, b) = 1;$

$\delta(2, a) = \Omega; \delta(2, b) = 2; \delta(3, a) = \Omega; \delta(3, b) = \Omega$

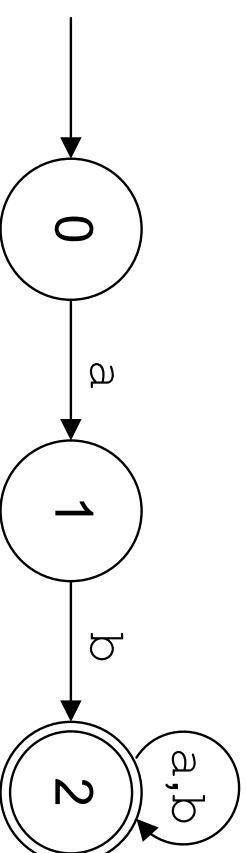
4. $q_0 = 0$

5. $F = \{2, 3\}$

FSM: solution to exercise (cont.)

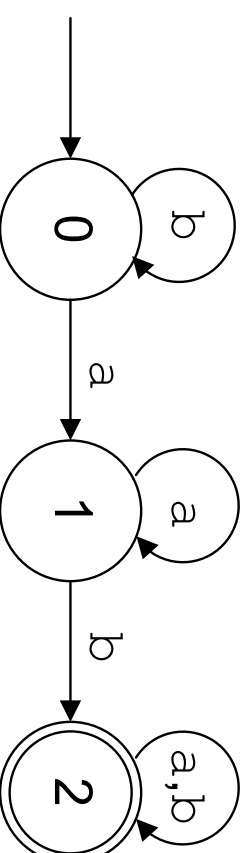
2. a)

$$r = ab(a + b)^*$$



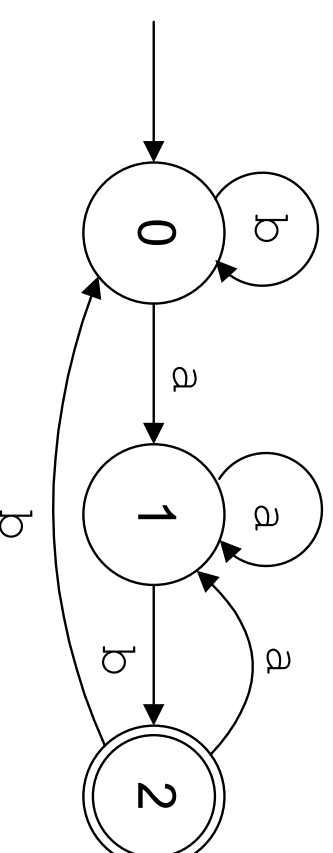
b)

$$r = (a + b)^*ab(a + b)^*$$



c)

$$r = (a + b)^*ab$$



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.**