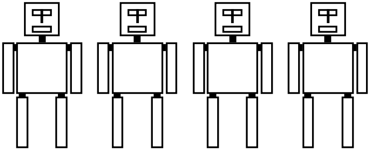
Formal Languages and Automata

Stephan Schulz & Jan Hladik

stephan.schulz@dhbw-stuttgart.de
jan.hladik@dhbw-stuttgart.de

with contributions from David Suendermann



DHBW Stuttgart
Duale Hochschule
Baden-Württemberg

Table of Contents

Outline

Introduction

- Stephan Schulz
  - Dipl.-Inform., U. Kaiserslautern, 1995
  - Dr. rer. nat., TU München, 2000
  - Visiting professor, U. Miami, 2002
  - Visiting professor, U. West Indies, 2005
  - Lecturer (Hildesheim, Offenburg, ...) since 2009
  - Industry experience: Building Air Traffic Control systems
    - System engineer, 2005
    - Project manager, 2007
    - Product Manager, 2013
  - Professor, DHBW Stuttgart, 2014

Research: Logic & Automated Reasoning

Introduction

- Jan Hladik
  - Dipl.-Inform.: RWTH Aachen, 2001
  - Dr. rer. nat.: TU Dresden, 2007
  - Industry experience: SAP Research
    - Work in publicly funded research projects
    - Collaboration with SAP product groups
    - Supervision of Bachelor, Master, and PhD students
  - Professor: DHBW Stuttgart, 2014

Research: Semantic Web, Semantic Technologies, Automated Reasoning

Literature

- Scripts
  - The most up-to-date version of this document as well as auxiliary material will be made available online at

    `http://wwwlehre.dhbw-stuttgart.de/~sschulz/fla2015.html` and

    `http://wwwlehre.dhbw-stuttgart.de/~hladik/FLA`
- Books
  - John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman: Introduction to Automata Theory, Languages, and Computation
  - Michael Sipser: Introduction to the Theory of Computation
  - Dirk W. Hoffmann: Theoretische Informatik
  - Ulrich Hedtstück: Einführung in die theoretische Informatik

Computing Environment

- For practical exercises, you will need a complete Linux/UNIX environment. If you do not run one natively, there are several options:
  - You can install VirtualBox (`https://www.virtualbox.org`) and then install e.g. Ubuntu (`http://www.ubuntu.com/`) on a virtual machine
  - For Windows, you can install the complete UNIX emulation package Cygwin from `http://cygwin.com`
  - For MacOS, you can install `fink` (`http://fink.sourceforge.net/`) or MacPorts (`https://www.macports.org/`) and the necessary tools
- You will need at least `flex`, `bison`, `gcc`, `grep`, `sed`, `AWK`, `make`, and a good text editor

Outline

Formal language concepts

Alphabet: finite set Σ of symbols (characters)

- $\{a, b, c\}$

Word: finite sequence w of characters (string)

- $ab \neq ba$

Language: (possibly infinite) set L of words

- $\{ab, ba\} = \{ba, ab\}$

Formal: L defined precisely

- opposed to natural languages, where there are borderline cases

Some formal languages

Example

- names in a phone directory
- phone numbers in a phone directory
- legal C identifiers
- legal C programs
- legal HTML 4.01 Transitional documents
- empty set
- ASCII strings
- Unicode strings

More?

Language classes

This course: four classes of different complexity and expressivity

1. regular languages: limited power, but easy to handle
   - "strings that start with a letter, followed by up to 7 letters or digits"
   - legal C identifiers
   - phone numbers
2. context-free languages: more expressive, but still feasible
   - "every `<token>` is matched by `</token>`"
   - nested dependencies
   - (most aspects of) legal C programs
   - many natural languages (English, German)

```
Jan says that we
  let
  the children
    help
    Hans
      paint
      the house
```

```
Jan sagt, dass wir
  die Kinder
    dem Hans
      das Haus
      anstreichen
    helfen
  ließen
```

Language classes (cont')

3. context-sensitive languages: even more expressive, difficult to handle computationally
   - "every variable has to be declared before it is used" (arbitrary sequence, arbitrary amounts of code in between)
   - cross-serial dependencies
   - (remaining aspects of) legal C programs
   - most remaining natural languages (Swiss German)

```
Jan säit das mer                    Jan says that we
  d'chind                             the children
    em Hans                             Hans
      es huus                             the house
  lönd                                let
    helfe                               help
      aastriche                           paint
```

4. recursively enumerable languages: most general (Chomsky) class; undecidable
   - all (valid) mathematical theorems
   - programs terminating on a particular input

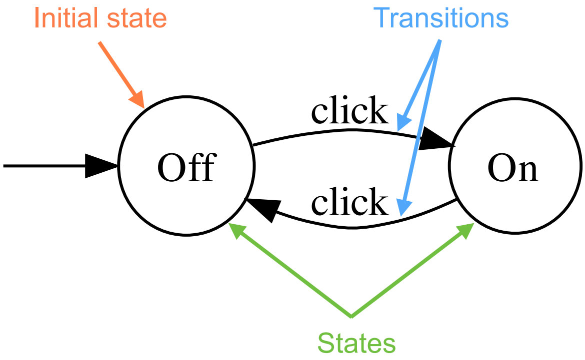
Automata

- abstract formal machine model, characterised by states, letters, transitions, and external memory
- accept words

For every language class discussed in this course, a machine model exists such that for every language L there is an automaton $\mathcal{A}(L)$ that accepts exactly the words in L.

regular	$\leadsto$	finite automaton
context-free	$\leadsto$	pushdown automaton
context-sensitive	$\leadsto$	linearly bounded Turing machine
recursively enumerable	$\leadsto$	(unbounded) Turing machine

Example: Finite Automaton



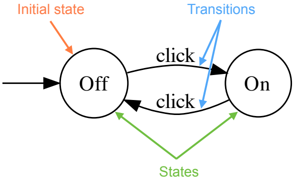
Example: Finite Automaton

Formally:

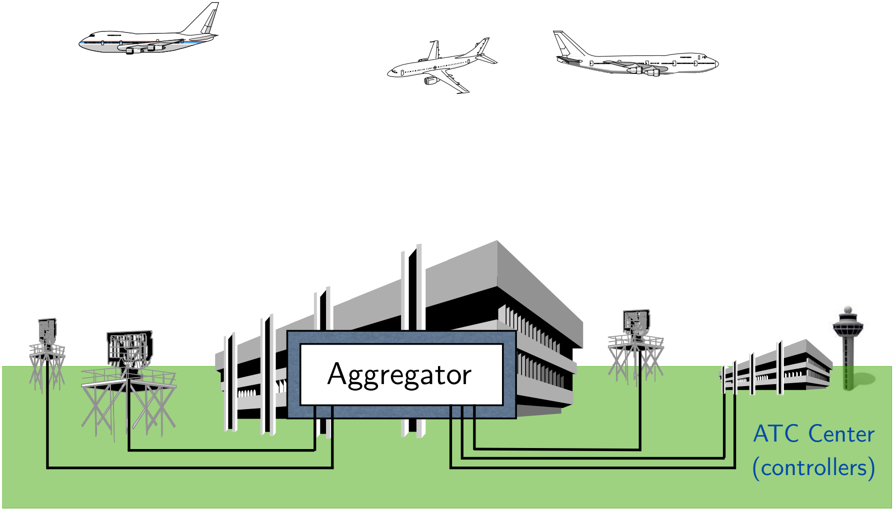
- $Q = \{\text{Off}, \text{On}\}$ is the set of states
- $\Sigma = \{click\}$ is the alphabet
- The transition function δ is given by

δ	click
Off	On
On	Off

- The initial state is Off
- There are no accepting states



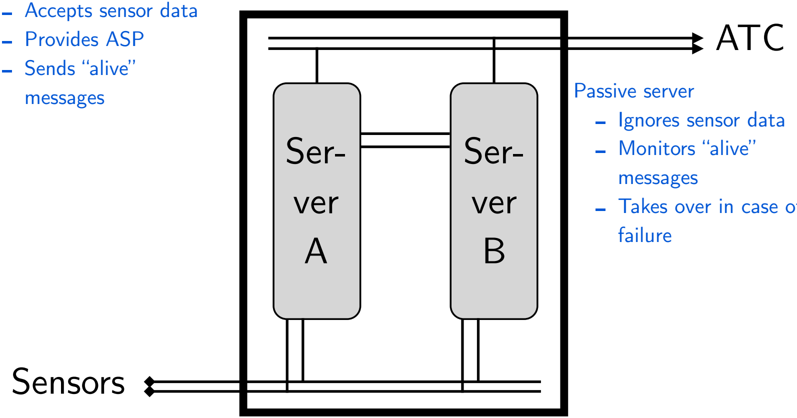
ATC scenario



ATC redundancy

Aktive server:

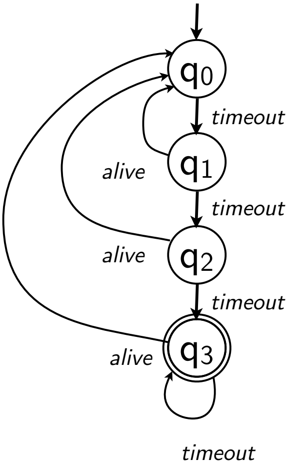
- Accepts sensor data
- Provides ASP
- Sends “alive” messages



Passive server

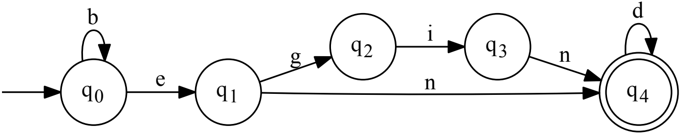
- Ignores sensor data
- Monitors “alive” messages
- Takes over in case of failure

DFA to the rescue



- Two events (“letters”)
  - timeout: 0.1 seconds have passed
  - alive: message from active server
- States q_0, q_1, q_2: Server is passive
  - No processing of input
  - No sending of alive messages
- State q_3: Server becomes active
  - Process input, provide output to ATC
  - Send alive messages every 0.1 seconds

Exercise: Automaton

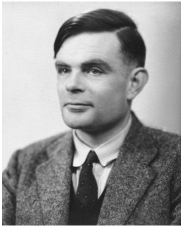


Does this automaton accept the words *begin*, *end*, *bind*, *bend*?

Turing Machine

“Universal computer”

- Very simple model of a computer
  - Infinite tape, one read/write head
  - Tape can store letters from a alphabet
  - FSM controls read/write and movement operations
- Very powerful model of a computer
  - Can compute anything any real computer can compute
  - Can compute anything an “ideal” real computer can compute
  - Can compute everything a human can compute (?)



Formal grammars

Formalism to generate (rather than accept) words over alphabet

terminal symbols: may appear in the produced word (alphabet)

non-terminal symbols: may not appear in the produced word (temporary symbols)

production rules: $l \to r$ means: l can be replaced by r anywhere in the word

Example

Grammar for arithmetic expressions over $\{0, 1\}$

$$\begin{array}{rcl}
\Sigma & = & \{0, 1, +, \cdot, (,)\} \\
N & = & \{E\} \\
P & = & \{E \to 0, E \to 1, \\
 & & \quad E \to (E) \\
 & & \quad E \to E + E \\
 & & \quad E \to E \cdot E\}
\end{array}$$

Exercise: Grammars

Using

- the non-terminal symbol S
- the terminal symbols b, d, e, g, i, n
- the production rules
  $S \rightarrow begin, beg \rightarrow e, in \rightarrow ind, in \rightarrow n, eg \rightarrow egg, ggg \rightarrow b$

can you generate the words $bend$ and end starting from the symbol S?

- If yes, how many steps do you need?
- If no, why not?

Questions about formal languages

- For a given language L, how can we find
  - a corresponding automaton $\mathcal{A}_L$?
  - a corresponding grammar G_L?
- What is the simplest automaton for L?
  - "simplest" meaning: weakest possible language class
  - "simplest" meaning: least number of elements
- How can we use formal descriptions of languages for compilers?
  - detecting legal words/reserved words
  - testing if the structure is legal
  - understanding the meaning by analysing the structure

More questions about formal languages

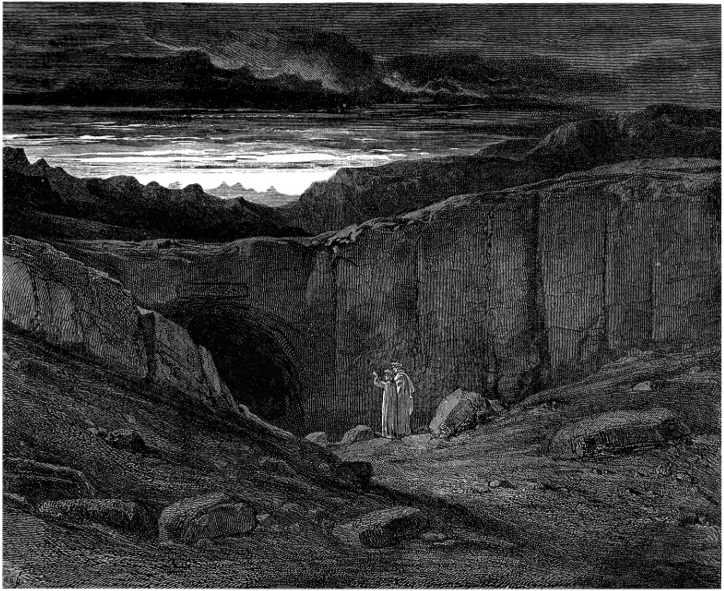
Closure properties: if L_1 and L_2 are in a class, does this also hold for

- the union of L_1 and L_2,
- the intersection of L_1 and L_2,
- the concatenation of L_1 and L_2,
- the complement of L_1?

Decision problems: for a word w and languages L_1 and L_2 (given by grammars or automata),

- does $w \in L_1$ hold?
- is L_1 finite?
- is L_1 empty?
- does $L_1 = L_2$ hold?

Abandon all hope…



Example applications for 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

Outline

Alphabets

Definition (Alphabet)

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

$$\Sigma = \{c_1, \ldots, c_n\}$$

Example

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

Alphabets: ASCII code chart

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

Words

Definition (Word)

- A word over the alphabet Σ is a finite sequence (list) of characters of Σ:

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

- The empty word with no characters is written as ε.
- The set of all words over an alphabet Σ is represented by Σ^*.

In programming languages, words are often referred to as strings.

Words

Example

1. Using Σ_{bin}, we can define the words $w_1, w_2 \in \Sigma_{\text{bin}}^*$:

$$w_1 = 01100 \quad \text{and} \quad w_2 = 11001$$

2. Using Σ_{en}, we can define the word $w \in \Sigma_{\text{en}}^*$:

$$w = \texttt{example}$$

Properties of words

Definition (Length, character access)

- The length $|w|$ of a word w is the number of characters in w.
- The number of occurrences of a character c in w is denoted as $|w|_c$.
- The individual characters within words are accessed using the terminology $w[i]$ with $i \in \{1, 2, \ldots, |w|\}$.

Example

- $|\texttt{example}| = 7$ and $|\varepsilon| = 0$
- $|\texttt{example}|_{\texttt{e}} = 2$ and $|\texttt{example}|_{\texttt{k}} = 0$
- $\texttt{example}[4] = \texttt{m}$

Appending words

Definition (Concatenation of words)

For words w_1 and w_2, the concatenation $w_1 \cdot w_2$ is defined as w_1 followed by w_2.

$w_1 \cdot w_2$ is often simply written as $w_1 w_2$.

Example

Let $w_1 = \texttt{01}$ and $w_2 = \texttt{10}$.
Then the following holds:

$$w_1 w_2 = \texttt{0110} \quad \text{and} \quad w_2 w_1 = \texttt{1001}$$

Iterated concatenation

In the following, we denote the set of natural numbers $\{0, 1, \ldots\}$ by $\mathbb{N}$.

Definition (Power of a word)

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

$$\begin{array}{lcll} w^0 & = & \varepsilon & \\ w^n & = & w^{n-1} \cdot w & \text{if } w > 0 \end{array}$$

Example

Let $w = ab$. Then:

$$\begin{array}{lcl} w^0 & = & \varepsilon \\ w^1 & = & ab \\ w^3 & = & ababab \end{array}$$

Exercise: Operations on words

Given the alphabet $\Sigma = \{a, b, c\}$ and the words

- $u = abc$
- $v = aa$
- $w = cb$

what is denoted by the following expressions?

1. $u^2 \cdot w$
2. $v \cdot \varepsilon \cdot w \cdot u^0$
3. $|u^3|_a$
4. $v \cdot a^2 \cdot (v[4])$
5. $(v \cdot a^2 \cdot v)[4]$
6. $|w^0|$
7. $|w^0 \cdot w|$

Formal languages

Definition (Formal language)

For an alphabet Σ, a formal language over Σ is a subset $L \subseteq \Sigma^*$.

Example

Let $L_{\mathbb{N}} = \{\mathtt{1}w \mid w \in \Sigma_{\text{bin}}^*\} \cup \{\mathtt{0}\}$.
Then $L_{\mathbb{N}}$ is the set of all words that represent integers using the binary system (all words starting with 1 and the word 0:

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

Numeric value of a binary word

Definition (Numeric value)

We define the function

$$n : L_{\mathbb{N}} \to \mathbb{N}$$

as the function returning the numeric value of a word in the language $L_{\mathbb{N}}$. This means

(a) $n(\mathtt{0}) = 0$,

(b) $n(\mathtt{1}) = 1$,

(c) $n(w\mathtt{0}) = 2 \cdot n(w)$ for $|w| > 0$,

(d) $n(w\mathtt{1}) = 2 \cdot n(w) + 1$ for $|w| > 0$.

Prime numbers as a language

Definition (Prime numbers)

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

$$L_{\mathbb{P}} = \{w \in L_{\mathbb{N}} \mid n(w) \in \mathbb{P}\}.$$

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

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

C functions as a language

Definition

We define the language $L_C \subset \Sigma^*_{\text{ASCII}}$ as the set of all C function definitions with a declaration of the form:

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

(where f and x are legal C identifiers).
Then L_C contains the ASCII code of all those definitions of C functions processing and returning a string.

C function evaluations as a language

Definition

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

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

(a) $f \in L_C$,

(b) $x, y \in \Sigma^*_{\mathrm{ASCII}}$,

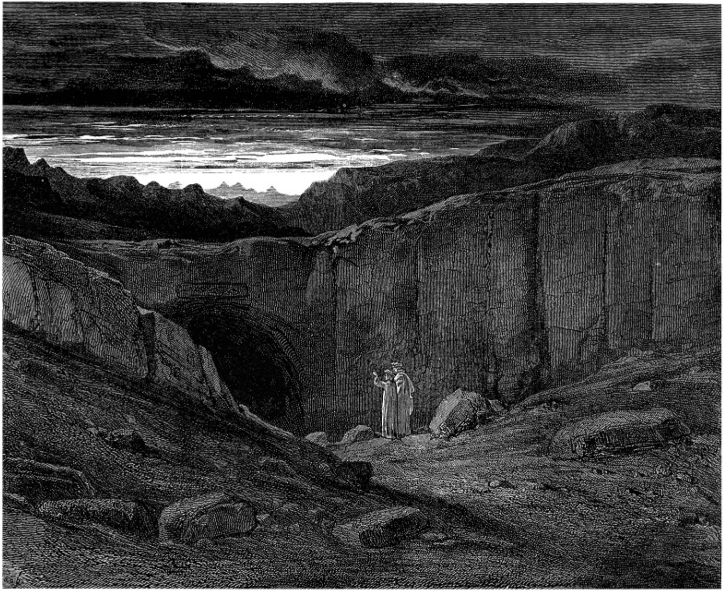
(c) applying f to x terminates and returns y.

Formal languages have a wide scope:

- Testing whether a word belongs to $L_{\mathbb{N}}$ is straightforward.
- The same test for $L_{\mathbb{P}}$ or L_C is more complex.
- Later, we will see that there is no algorithm to do this test for L_u.

End lecture 1

Abandon all hope…



Product

Definition (Product of formal languages)

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

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

Example

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

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

The product is

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

Power

Definition (Power of a language)

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

$$\begin{aligned} L^0 &= \{\varepsilon\} \\ L^n &= L^{n-1} \cdot L \end{aligned}$$

Example

Using the alphabet Σ_{en}, we define the language $L = \{\texttt{ab}, \texttt{ba}\}$. Thus:

$$\begin{aligned} L^0 &= \{\varepsilon\} \\ L^1 &= \{\varepsilon\} \cdot \{\texttt{ab}, \texttt{ba}\} &&= \{\texttt{ab}, \texttt{ba}\} \\ L^2 &= \{\texttt{ab}, \texttt{ba}\} \cdot \{\texttt{ab}, \texttt{ba}\} &&= \{\texttt{abab}, \texttt{abba}, \texttt{baab}, \texttt{baba}\} \end{aligned}$$

The Kleene Star operator

Definition (Kleene Star)

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

$$L^* = \bigcup_{n \in \mathbb{N}} L^n.$$

Example

Using the alphabet Σ_{en}, we define the language $L = \{\mathtt{a}\}$. Thus:

$$L^* = \{\mathtt{a}^n | n \in \mathbb{N}\}.$$

Exercise: formal languages

Given the alphabet Σ_{bin} and the language $L = \{\mathtt{1}\}$, formally describe the following:

a) the language $M = L^* \backslash \{\varepsilon\}$
b) the set $N = \{n(w) \mid w \in M\}$
c) the language $M^- = \{w \mid n(w) - \mathtt{1} \in N\}$
d) the language $M^+ = \{w \mid n(w) + \mathtt{1} \in N\}$

Outline

Outline

Regular expressions

Compact and convenient way to represent a set of strings

- Characterize tokens for compilers
- Describe search terms for a data base
- Filter through genomic data
- Extract URLs from web pages
- Extract email addresses from web pages

The set of all regular expressions (over an alphabet) is a formal language

Each single regular expression describes a formal language

Reminder: Power sets

Definition (Power set of a set)

- Assume a set S. Then the power set of S, written as 2^S, is the set of all subsets of S.
- In particular, if Σ is an alphabet, 2^{Σ^*} is the set of all subsets of Σ^* and hence the set of all possible formal languages over Σ.

Example

$$
\begin{aligned}
2^{\Sigma_{\text{bin}}} &= 2^{\{\mathtt{0},\mathtt{1}\}} = \{\emptyset, \{\mathtt{0}\}, \{\mathtt{1}\}, \{\mathtt{0},\mathtt{1}\}\},\\
2^{\Sigma_{\text{bin}}^*} &= 2^{\{\varepsilon,\mathtt{0},\mathtt{1},\mathtt{00},\mathtt{01},\ldots\}}\\
&= \{\emptyset, \{\varepsilon\}, \{\mathtt{0}\}, \{\mathtt{1}\}, \{\mathtt{00}\}, \{\mathtt{01}\}, \ldots\\
&\quad \ldots \{\varepsilon,\mathtt{0}\}, \{\varepsilon,\mathtt{1}\}, \{\varepsilon,\mathtt{00}\}, \{\varepsilon,\mathtt{01}\}, \ldots\\
&\quad \ldots \{\mathtt{010},\mathtt{1110},\mathtt{10101}\}, \ldots\}.
\end{aligned}
$$

Regular expressions and formal languages

A regular expression over Σ...

- ...is a word over the extended alphabet $\Sigma \cup \{\emptyset, \varepsilon, +, \cdot, ^*, (,)\}$
- ...describes a formal language over Σ

Terminology

The following terms are defined on the next slides:

- R_Σ is the set of all regular expressions over the alphabet Σ.
- The function $L : R_\Sigma \to 2^{\Sigma^*}$ assigns a formal language $L(r) \subseteq \Sigma^*$ to each regular expression r.

Regular expressions and their languages (1)

Definition (Regular expressions)

The set of regular expressions R_Σ over the alphabet Σ is defined as follows:

1. The regular expression $\emptyset$ denotes the empty language.
   $\emptyset \in R_\Sigma$ and $L(\emptyset) = \{\}$
2. The regular expression ε denotes the language containing only the empty word.
   $\varepsilon \in R_\Sigma$ and $L(\varepsilon) = \{\varepsilon\}$
3. Each symbol in the alphabet Σ is a regular expression.
   $c \in \Sigma \Rightarrow c \in R_\Sigma$ and $L(c) = \{c\}$

Regular expressions and their languages (2)

Definition (Regular expressions (cont'))

4. The operator $+$ denotes the union of the languages of r_1 and r_2.
   $r_1 \in R_\Sigma, r_2 \in R_\Sigma \Rightarrow r_1 + r_2 \in R_\Sigma$ and $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
5. The operator $\cdot$ denotes the product of the languages of r_1 and r_2.
   $r_1 \in R_\Sigma, r_2 \in R_\Sigma \Rightarrow r_1 \cdot r_2 \in R_\Sigma$ and $L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2)$
6. The Kleene star of a regular expression r denotes the Kleene star of the language of r.
   $r \in R_\Sigma \Rightarrow r^* \in R_\Sigma$ and $L(r^*) = (L(r))^*$
7. Brackets can be used to group regular expressions without changing their language.
   $r \in R_\Sigma \Rightarrow (r) \in R_\Sigma$ and $L((r)) = L(r)$

Equivalence of regular expressions

Definition (Equivalence and precedence)

- Two regular expressions r_1 and r_2 are equivalent if they denote the same language: $r_1 \doteq r_2$ if and only if $L(r_1) = L(r_2)$
- The operators have the following precedence:
  $(\ldots) \quad > \quad * \quad > \quad \cdot \quad > \quad +$
- The product operator $\cdot$ can be omitted.

Example

$$
\begin{aligned}
a + b \cdot c^* &\doteq a + (b \cdot (c^*)) \\
ac + bc^* &\doteq a \cdot c + b \cdot c^*
\end{aligned}
$$

Note: Some authors (and tools) use $|$ as the union operator.

Examples for regular expressions

Example

Let $\Sigma_{\text{abc}} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$.

- The regular expression $r_1 = (\mathtt{a} + \mathtt{b} + \mathtt{c})(\mathtt{a} + \mathtt{b} + \mathtt{c})$ describes all the words of exactly two symbols:

$$L(r_1) = \{w \in \Sigma^*_{\text{abc}} \big| |w| = 2\}$$

- The regular expression $r_2 = (\mathtt{a} + \mathtt{b} + \mathtt{c})(\mathtt{a} + \mathtt{b} + \mathtt{c})^*$ describes all the words of one or more symbols:

$$L(r_1) = \{w \in \Sigma^*_{\text{abc}} \big| |w| \geq 1\}$$

Exercise: regular expressions

1. Using the alphabet $\Sigma_{\text{abc}} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, give a regular expression r_1 for all the words $w \in \Sigma_{\text{abc}}^*$ containing exactly one $\mathtt{a}$ or exactly one $\mathtt{b}$.
2. Formally describe $L(r_1)$ as a set.
3. Using the alphabet $\Sigma_{\text{abc}} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, give a regular expression r_2 for all the words containing at least one $\mathtt{a}$ and one $\mathtt{b}$.
4. Using the alphabet $\Sigma_{\text{bin}} = \{\mathtt{0}, \mathtt{1}\}$, give a regular expression for all the words whose third last symbol is $\mathtt{1}$.
5. Using the alphabet Σ_{bin}, give a regular expression for all the words not containing the string $\mathtt{110}$.
6. Which language is described by the regular expression

$$r_6 = (\mathtt{1} + \varepsilon)(\mathtt{0}\mathtt{0}^*\mathtt{1})^*\mathtt{0}^*?$$

End lecture 2

Algebraic operations on regular expressions

Theorem

1. $r_1 + r_2 \doteq r_2 + r_1$ *(commutative law)*
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)*

Proof of some rules

Proof of Rule 1 ($r_1 + r_2 \doteq r_2 + r_1$).

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

$\square$

Proof of Rule 4 ($\emptyset r \doteq \emptyset$).

$$\begin{aligned}
L(\emptyset r) &\overset{\text{Def. concat}}{=} L(\emptyset) \cdot L(r) \\
&\overset{\text{Def. empty regexp}}{=} \emptyset \cdot L(r) \\
&\overset{\text{Def. product}}{=} \{w_1 w_2 | w_1 \in \emptyset, w_2 \in L(r)\} \\
&= \emptyset \\
&\overset{\text{Def. empty regexp}}{=} L(\emptyset)
\end{aligned}$$

$\square$

Algebraic operations on regular expressions (cont.)

Theorem

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. $\varepsilon \notin L(s)$ *and* $r \doteq rs + t \longrightarrow r \doteq ts^*$ *(proof by Arto Salomaa)*
16. $r^*r \doteq rr^*$ *(see Lemma: Kleene Star below)*
17. $\varepsilon \notin L(s)$ *and* $r \doteq sr + t \longrightarrow r \doteq s^*t$ *(Arden's Lemma)*

Lemma: Kleene Star (1)

Lemma (Kleene Star)

$$r^*r \doteq rr^*$$

Proof of Case 1: $\varepsilon \notin L(r)$.

$$\begin{aligned}
r^*r &\doteq (\varepsilon + r^*r)r && \text{(by 13. } (r')^* \doteq \varepsilon + (r')^*r'\text{)} \\
&\doteq (r^*r + \varepsilon)r && \text{(by 1. } r_1 + r_2 \doteq r_2 + r_1\text{)} \\
&\doteq r^*rr + r && \text{(by 7. } (r_1 + r_2)r_3 \doteq r_1r_3 + r_2r_3\text{)} \\
&\doteq rr^* && \text{(by 15. } r' \doteq r's + t \text{ with } r' = r^*r, s = r, t = r\text{)}
\end{aligned}$$

$\square$

Lemma: Kleene Star (2)

Proof of Case 2: $\varepsilon \in L(r)$.

We show $L(r^*r) = L(r^*) = L(rr^*)$

a) Proof of $L(r^*r) \subseteq L(r^*)$

$L(r^*r) = L(r^*) \cdot L(r)$
$= (L(r))^* \cdot L(r)$
$= (\bigcup_{i \geq 0} L(r)^i) \cdot L(r)$
$= \bigcup_{i \geq 0} (L(r)^i \cdot L(r))$
$= \bigcup_{i \geq 1} L(r)^i$
$\subseteq L(r^*)$

b) Proof of $L(r^*r) \supseteq L(r^*)$

$L(r^*r) = \{uv \mid u \in L(r^*), v \in L(r)\}$
$\supseteq \{uv \mid u \in L(r^*), v = \varepsilon\}$
$= \{u \mid u \in L(r^*)\}$
$= L(r^*)$

- a. and b. imply $L(r^*r) = L(r^*)$
- $L(rr^*) = L(r^*)$: strictly analoguous

□

A note on Aarto/Arden

- Aarto: $\varepsilon \notin L(s)$ and $r \doteq rs + t \longrightarrow r \doteq ts^*$
- Why do we need $\varepsilon \notin L(s)$?
  - This guarantees that only words of the form ts^* are in $L(r)$
  - Example: $r \doteq rs + t$ mit $s = b^*$, $t = a$.
    - If we could apply Aarto, the result would be $r \doteq a(b^*)^* \doteq ab^*$
    - But $L = \{ab^*\} \cup \{b^*\}$ also fulfills the equation, i.e. there is no single unique solution in this case
  - Intuitively: $\varepsilon \in L(s)$ is a second escape from the recursion that bypasses t
- The case for Arden's lemma ($\varepsilon \notin L(s)$ and $r \doteq sr + t \longrightarrow r \doteq s^*t$) is analoguous

Exercise: Algebra on regular expressions

1. Prove the equivalence using only algebraic operations

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

2. Simplify the following regular expression:

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

3. Prove the equivalence using only algebraic operations

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

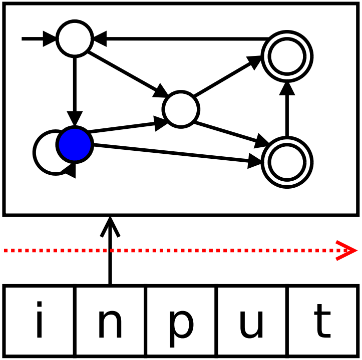
End lecture 3

Outline

Finite Automata: Motivation

- Simple model of computation
- Can recognize regular languages
- Equivalent to regular expressions
  - We can automatically generate a FA from a RE
  - We can automatically generate an RE from an FA
- Two variants:
  - Deterministic (DFA, now)
  - Non-deterministic (NFA, later)
- Easy to implement in actual programs

Deterministic Finite Automata: Idea



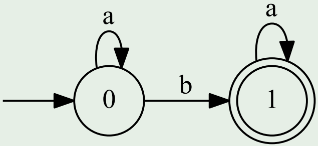
Deterministic finite automaton (DFA)

- is in one of finitely many states
- starts in the initial state
- processes input from left to right
  - changes state depending on character read
  - determined by transition function
  - no rewinding!
  - no writing!
- accepts input if
  - after reading the entire input
  - a final state is reached

DFA $\mathcal{A}$ for a^*ba^*

Example (Automaton $\mathcal{A}$)

$\mathcal{A}$ is a simple DFA recognizing the regular language $\mathtt{a}^*\mathtt{ba}^*$.



- $\mathcal{A}$ has two states, 0 and 1.
- It operates on the alphabet $\{a, b\}$.
- The transition function is indicated by the arrows.
- 0 is the initial state (with an arrow "pointing at it from anywhere").
- 1 is an accepting state (represented as a double circle).

DFA: formal definition

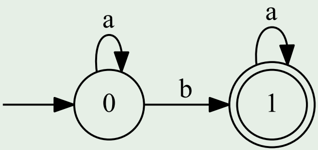
Definition (Deterministic Finite Automaton)

A deterministic finite automaton (DFA) is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ with the following components

- Q is a finite set of states.
- Σ is the (finite) input alphabet.
- $\delta : Q \times \Sigma \to Q \cup \{\Omega\}$ is the transition function.
  If $\delta(q, c) = \Omega$, the DFA announces an error, i.e. rejects the input.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of final (or accepting) states.

Formal definition of $\mathcal{A}$

Example



$\mathcal{A}$ is expressed as $(Q, \Sigma, \delta, q_0, F)$ with

- $Q = \{0, 1\}$
- $\Sigma = \{\mathtt{a}, \mathtt{b}\}$
- $\delta(0, \mathtt{a}) = 0; \delta(0, \mathtt{b}) = 1, \delta(1, \mathtt{a}) = 1; \delta(1, \mathtt{b}) = \Omega$
- $q_0 = 0$
- $F = \{1\}$

Language accepted by an DFA

Definition (Language accepted by an automaton)

The state transition function δ is generalised to a function δ' whose second argument is a word as follows:

- $\delta' : Q \times \Sigma^* \to Q \cup \{\Omega\}$
- $\delta'(q, \varepsilon) = q$
- $\delta'(q, wc) = \begin{cases} \delta(\delta'(q, w), c) & \text{if} \quad \delta(q, c) \neq \Omega \\ \Omega & \text{otherwise} \end{cases}$

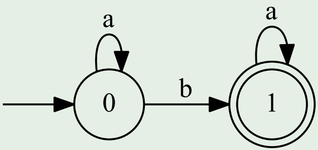
  with $c \in \Sigma; w \in \Sigma^*$

The language accepted by a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ is defined as

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

Language accepted by $\mathcal{A}$

Example



- $\delta'(0, aa) = \delta(\delta'(0, a), a) = \delta(\delta(\delta'(0, \varepsilon), a, a) = 0$
- $\delta'(1, aaa) = 1$
- $\delta'(0, bb) = \delta'(1, b) = \Omega$
- $L(\mathcal{A}) = \{w \in \{a, b\}^* \mid w = a^n b a^m$ and $n, m \in \mathbb{N}\}$

Run of a DFA

Definition (Run)

A run of an automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ on a word $w = c_1 \cdot c_2 \cdots c_n$ is a sequence

$$r = ((q_0, c_1, q_1), (q_1, c_2, q_2), \ldots, (q_{n-1}, c_n, q_n))$$

where

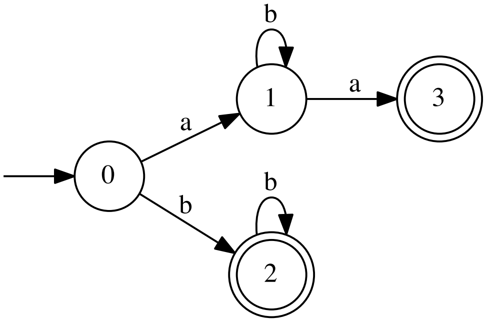
- $q_i \in Q$ holds for $1 \leq i \leq n$ and
- $\delta(q_i, c_{i+1}) = q_{i+i}$ holds for $0 \leq i \leq n-1$.

A run is accepting if $q_n \in F$ holds.

The language accepted by $\mathcal{A}$ can alternatively be defined as the set of all words for which there exists an accepting run of $\mathcal{A}$.

Exercise: DFA

1. Given this graphical representation of a DFA $\mathcal{A}$:



a) Give a regular expression describing $L(\mathcal{A})$.
b) Give a formal definition of $\mathcal{A}$.

Exercise: DFA

2 Give

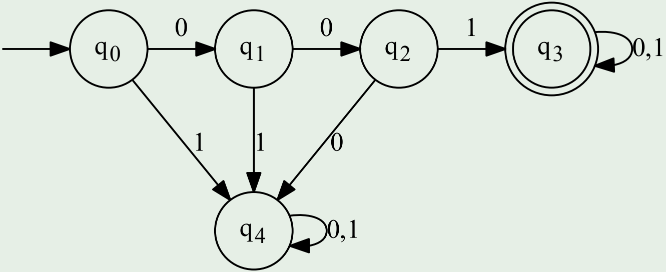
- a regular expression,
- a graphical representation, and
- a formal definition

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

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

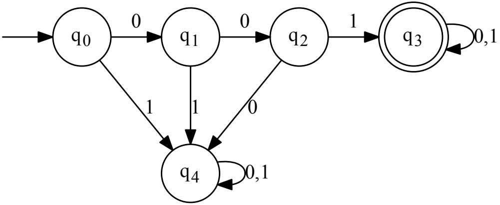
Another example

Example



Which language is recognized by the DFA?

Tabular representation of a DFA



$\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- Initial state: q_0
- $F = \{q_3\}$

		δ	0	1
$\rightarrow$	q_0	q_0	q_1	q_4
	q_1	q_1	q_2	q_4
	q_2	q_2	q_4	q_3
$*$	q_3	q_3	q_3	q_3
	q_4	q_4	q_4	q_4

DFA: Tabular representation in practice

```
Delta | 0   1
---------------
-> q0  | q1 q4
   q1  | q2 q4
   q2  | q4 q3
*  q3  | q3 q3
   q4  | q4 q4
```

```
> easim.py fsa001.txt 10101
Processing:  10101
q0 :: 1 -> q4
q4 :: 0 -> q4
q4 :: 1 -> q4
q4 :: 0 -> q4
q4 :: 1 -> q4
Rejected

> easim.py fsa001.txt 101
Processing:  101
q0 :: 1 -> q4
q4 :: 0 -> q4
q4 :: 1 -> q4
Rejected
```

DFAs in tabular form: exercise

- Give the following DFA ...
  - as a formal 5-tuple
  - as a diagram

```
parity    | 0    1
---------------------
->  even | even odd
*   odd  | odd  even
```

- Characterize the language accepted by the DFA

DFA exercise

- Assume
  - $\Sigma = \{a, b, c\}$
  - $L_1 = \{ubw | u \in \Sigma^*, w \in \Sigma\}$
  - $L_2 = \{ubw | u \in \Sigma, w \in \Sigma^*\}$
- Group 1 (your family name starts with A-M):
  Find a DFA $\mathcal{A}$ with $L(\mathcal{A}) = L_1$
- Group 2 (your family name does not start with A-M):
  Find a DFA $\mathcal{A}$ with $L(\mathcal{A}) = L_2$

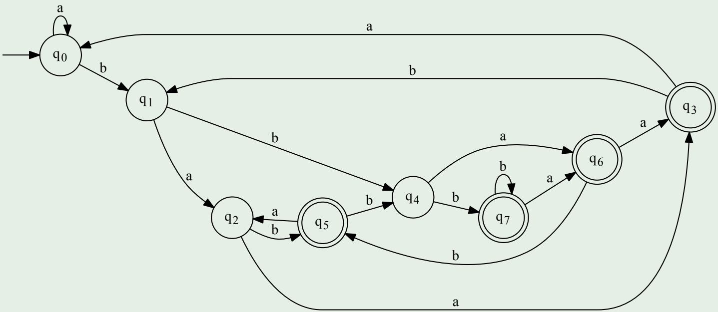
Outline

Drawbacks of deterministic automata

Deterministic automata:

- Transition function δ
  - exactly one transition from every configuration (possibly Ω)
- can be complex even for simple languages

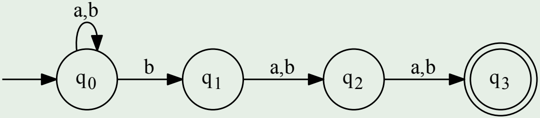
Example (DFA $\mathcal{A}$ for $(\mathtt{a}+\mathtt{b})^*\mathtt{b}(\mathtt{a}+\mathtt{b})(\mathtt{a}+\mathtt{b})$)



Non-Determinism

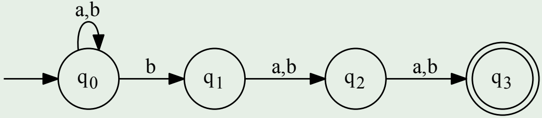
- FA can be simplified if one input can lead to
  - one transition,
  - multiple transitions, or
  - no transition.
- Intuitively, such an FA selects its next state from a set of states depending on the current state and the input
  - and always chooses the “right” one
- This is called a non-deterministic finite automaton (NFA)

Example (NFA $\mathcal{B}$ for $(\mathtt{a}+\mathtt{b})^*\mathtt{b}(\mathtt{a}+\mathtt{b})(\mathtt{a}+\mathtt{b})$)



Non-Deterministic automata

Example (Transitions in $\mathcal{B}$)

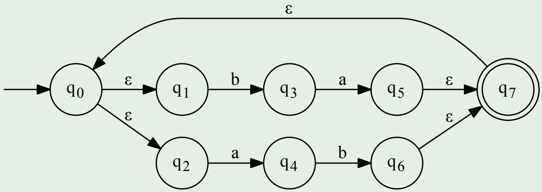


- In state q_0 with input `b`, the FA has to “guess” the next state.
- The string `abab` can be read in three ways:
  1. $q_0 \overset{a}{\mapsto} q_0 \overset{b}{\mapsto} q_0 \overset{a}{\mapsto} q_0 \overset{b}{\mapsto} q_0$ (failure)
  2. $q_0 \overset{a}{\mapsto} q_0 \overset{b}{\mapsto} q_0 \overset{a}{\mapsto} q_0 \overset{b}{\mapsto} q_1$ (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)
- An NFA accepts an input w if there exists an accepting run on w!

NFA: non-deterministic transitions and ε-transitions

- Non-deterministic transitions allow an NFA to go to more than one successor state
  - Instead of a function δ, an NFA has a transition relation Δ
- An NFA can additionally change its current state without reading an input symbol: $q_1 \overset{\varepsilon}{\mapsto} q_2$.
  - This is called a spontaneous transition or ε-transition
  - Thus, Δ is a relation on $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$

Example (NFA with ε-transitions)



NFA: Formal definition

Definition (NFA)

An NFA is a quintuple $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ with the following components:

1. Q is the finite set of states.
2. Σ is the input alphabet.
3. Δ is a relation on $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$.
4. $q_0 \in Q$ is the initial state.
5. $F \subseteq Q$ is the set of final states.

Run of a nondeteterministic automaton

Definition (Run of an NFA)

A run of an NFA $\mathcal{A}$ on a word w is a sequence of transitions

$$((q_0, c_1, q_1), (q_1, c_2, q_2), \ldots, (q_{n-1}, c_n, q_n))$$

such that the following conditions are satisfied:

- q_0 is the initial state, $q_i \in Q$, $c_i \in \Sigma \cup \{\varepsilon\}$,
- $(q_i, c_{i+1}, q_{i+1}) \in \Delta$ holds for $0 \leq i \leq n-1$,
- $c_1 \cdot c_2 \cdot \ldots \cdot c_n = w$.

It is accepting if q_n is a final state.

The slightly more complex definition is necessary to handle ε-transitions.

Language recognized by an NFA

Definition (Language recognized by an NFA)

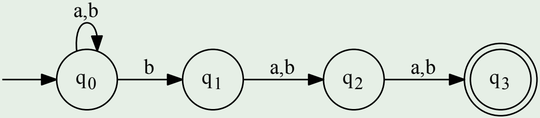
Assume an NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$. The language accepted by $\mathcal{A}$ is

$$L(\mathcal{A}) = \{w \mid \text{ there is an accepting run of } \mathcal{A} \text{ on } w\}$$

- Note that we only require the existance of one accepting run
- It does not matter if there are also non-accepting runs on w

Example: NFA definition

Example (Formal definition of $\mathcal{B}$)



$\mathcal{B} = (Q, \Sigma, \Delta, q_0, F)$ with

$Q = \{q_0, q_1, q_2, q_3\}$

$\Sigma = \{\texttt{a}, \texttt{b}\}$

$F = \{q_3\}$

$$\Delta = \{(q_0, \texttt{a}, q_0), (q_0, \texttt{b}, q_0), (q_0, \texttt{b}, q_1), (q_1, \texttt{a}, q_2), (q_1, \texttt{b}, q_2), (q_2, \texttt{a}, q_3), (q_2, \texttt{b}, q_3)\}$$

Δ	a	b	ε
q_0	$\{q_0\}$	$\{q_0, q_1\}$	$\{\}$
q_1	$\{q_2\}$	$\{q_2\}$	$\{\}$
q_2	$\{q_3\}$	$\{q_3\}$	$\{\}$
q_3	$\{\}$	$\{\}$	$\{\}$

Exercise: NFA

Develop an NFA $\mathcal{A}$ whose language $L(\mathcal{A}) \subset \{\mathtt{a}, \mathtt{b}\}^*$ contains all those words featuring the substring $\mathtt{aba}$. Give:

- a regular expression representing $L(\mathcal{A})$,
- a graphical representation of $\mathcal{A}$,
- a formal definition of $\mathcal{A}$.

Equivalence of DFA and NFA

Theorem (Equivalence of DFA and NFA)

NFAs and DFAs recognize the same class of languages.

- *For every DFA $\mathcal{A}$ there is an an NFA $\mathcal{B}$ with $L(\mathcal{A}) = L(\mathcal{B})$.*
- *For every NFA $\mathcal{B}$ there is an an DFA $\mathcal{A}$ with $L(\mathcal{B}) = L(\mathcal{A})$.*

- The direction DFA to NFA is trivial:
  - Every DFA is (essentially) an NFA
  - ...since every function is a relation
- What about the other direction?

Equivalence of DFA and NFA

Equivalence of DFAs and NFAs can be shown by transforming

- an NFA $\mathcal{A}$
- into a DFA $\det(\mathcal{A})$ accepting the same language.

Method:

- states of $\det(\mathcal{A})$ represent sets of states of $\mathcal{A}$
- a transition from q_1 to q_2 with character c in $\det(\mathcal{A})$ is possible if
  - in $\mathcal{A}$ there is a transition with c
  - from one of the states that q_1 represents
  - to one of the states that q_2 represents.
- a state in $\det(\mathcal{A})$ is accepting if it contains an accepting state

To this end, we define three auxiliary functions.

- ec to compute the ε closure of a state
- δ^* to compute possible successors of a state
- $\hat{\delta}$, the extended transition function for NFAs

Step 1: ε closure of an NFA

The ε closure of a state q contains all states the NFA can change to by means of ε transitions starting from q.

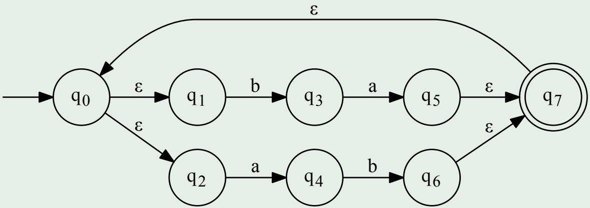
Definition (ε closure)

The function $ec : Q \to 2^Q$ is the smallest function with the properties:

- $q \in ec(q)$
- $p \in ec(q) \land (p, \varepsilon, r) \in \delta \quad \Rightarrow \quad r \in \; ec(q)$

Example: ε closure

Example



- $ec(q_0) = \{q_0, q_1, q_2\}$,
- $ec(q_1) = \{q_1\}$,
- $ec(q_2) = \{q_2\}$,
- $ec(q_3) = \{q_3\}$,
- $ec(q_4) = \{q_4\}$,
- $ec(q_5) = \{q_5, q_7, q_0, q_1, q_2\}$,
- $ec(q_6) = \{q_6, q_7, q_0, q_1, q_2\}$,
- $ec(q_7) = \{q_7, q_0, q_1, q_2\}$.

Step 2: Successor state function for NFAs

The function δ^* maps

- a pair (q, c)
- to the set of all states the NFA can change to from q with c
- followed by any number of ε transitions.

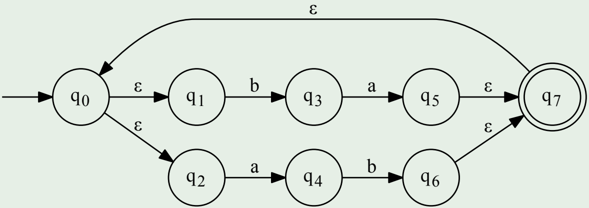
Definition (Successor state function)

The function $\delta^* : Q \times \Sigma \to 2^Q$ is defined as follows:

$$\delta^*(q, c) = \bigcup_{r \in Q:\ (q,c,r) \in \Delta} ec(r)$$

Example: successor state function

Example



$$\delta^*(q,c) = \bigcup_{r \in Q:\ (q,c,r) \in \Delta} ec(r)$$

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

Step 3: extended transition function

The function $\hat{\delta}$ maps

- a pair (M, c) consisting of a set of states M and a character c
- to the set N of states that are reachable from any state of M via Δ by reading the character c
- possibly followed by ε transitions.

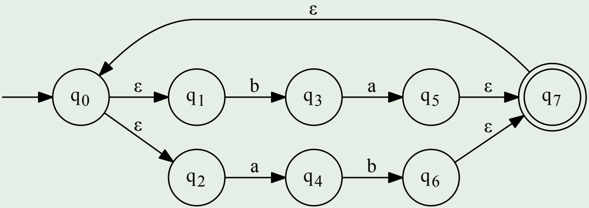
Definition (Extended transition function)

The function $\hat{\delta} : 2^Q \times \Sigma \to 2^Q$ is defined as follows:

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

Example: extended transition function

Example



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

- $\hat{\delta}(\{q_0, q_1, q_2\}, \mathtt{a}) = \{q_4\}$
- $\hat{\delta}(\{q_3\}, \mathtt{a}) = \{q_5, q_7, q_0, q_1, q_2\}$
- $\hat{\delta}(\{q_3\}, \mathtt{b}) = \{\}$
- …

Equivalence of DFA and NFA: formal definition

Using the three steps, we can define $\det(\mathcal{A})$.

Definition

For an NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$, the deterministic Automaton $\det(\mathcal{A})$ is defined as

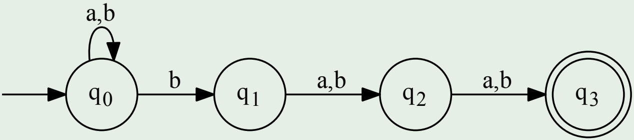
$$(2^Q, \Sigma, \hat{\delta}, ec(q_0), \hat{F})$$

with $\hat{F} = \{M \in 2^Q \mid M \cap F \neq \{\}\}$.

The set of final states $\hat{F}$ is the set of all subsets of Q containing a final state.

Example: transformation into DFA

Example (NFA $\mathcal{B}$ for $(\mathtt{a}+\mathtt{b})^*\mathtt{b}(\mathtt{a}+\mathtt{b})(\mathtt{a}+\mathtt{b})$)

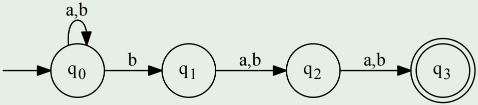


$$\begin{aligned}\mathcal{B} &= (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \Delta, q_0, \{q_3\}) \\ \det(\mathcal{B}) &= (\hat{Q}, \{a, b\}, \hat{\delta}, S_0, \hat{F})\end{aligned}$$

- Initial state: $S_0 := ec(q_0) = \{q_0\}$

Example: transformation into DFA (cont')

Example



- $\hat{\delta}(S_0, \texttt{a}) = \{q_0\} = S_0$
- $\hat{\delta}(S_0, \texttt{b}) = \{q_0, q_1\} =: S_1$
- $\hat{\delta}(S_1, \texttt{a}) = \{q_0, q_2\} =: S_2$
- $\hat{\delta}(S_1, \texttt{b}) = \{q_0, q_1, q_2\} =: S_4$
- $\hat{\delta}(S_2, \texttt{a}) = \{q_0, q_3\} =: S_3$
- $\hat{\delta}(S_2, \texttt{b}) = \{q_0, q_1, q_3\} =: S_5$
- $\hat{\delta}(S_4, \texttt{a}) = \{q_0, q_2, q_3\} =: S_6$
- $\hat{\delta}(S_4, \texttt{b}) = \{q_0, q_1, q_2, q_3\} =: S_7$
- $\hat{\delta}(S_3, \texttt{a}) = \{q_0\} = S_0$
- $\hat{\delta}(S_3, \texttt{b}) = \{q_0, q_1\} = S_1$
- $\hat{\delta}(S_5, \texttt{a}) = \{q_0, q_2\} = S_2$
- $\hat{\delta}(S_5, \texttt{b}) = \{q_0, q_1, q_2\} = S_4$
- $\hat{\delta}(S_6, \texttt{a}) = \{q_0, q_3\} = S_3$
- $\hat{\delta}(S_6, \texttt{b}) = \{q_0, q_1, q_3\} = S_5$
- $\hat{\delta}(S_7, \texttt{a}) = \{q_0, q_2, q_3\} = S_6$
- $\hat{\delta}(S_7, \texttt{b}) = \{q_0, q_1, q_2, q_3\} = S_7$

Example: transformation into DFA (cont')

Example

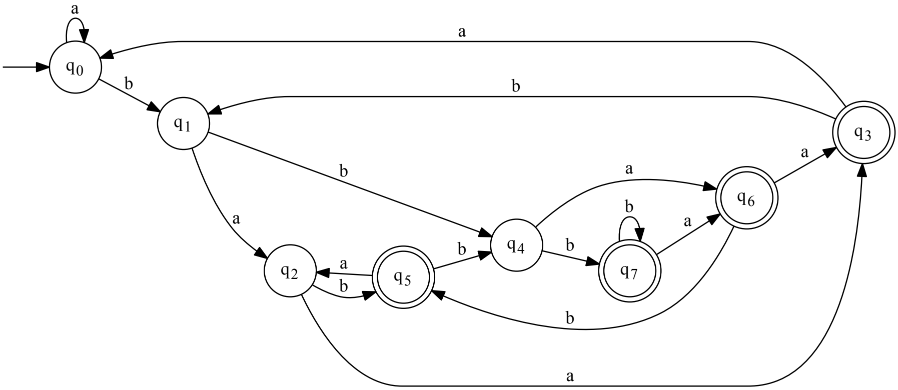
We can now define the DFA $\det(\mathcal{B}) = (\hat{Q}, \Sigma, \hat{\delta}, S_0, \hat{F})$ as follows:

- the set of states $\hat{Q} = \{S_0, \cdots, S_7\}$,
- the state transition function $\hat{\delta}$ is:

$\hat{\delta}$	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
a	S_0	S_2	S_3	S_0	S_6	S_2	S_3	S_6
b	S_1	S_4	S_5	S_1	S_7	S_4	S_5	S_7

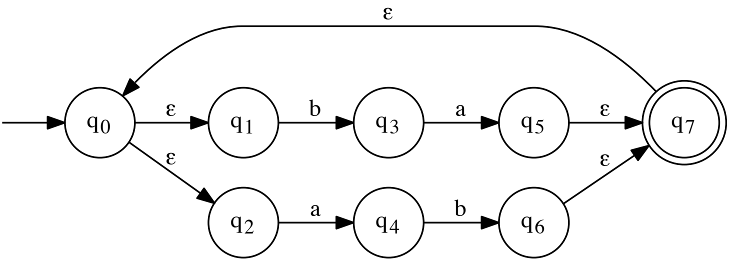
- and the set of final states $\hat{F} = \{S_3, S_5, S_6, S_7\}$.

Example: transformation into DFA (cont')



Exercise: Transformation into DFA

Given the following NFA $\mathcal{A}$:



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

b) Draw $\det(\mathcal{A})$'s graphical representation

c) Give a regular expression representing the same language as $\mathcal{A}$.

Solution

Outline

Regular expressions and Finite Automata

- Regular expressions describe regular languages
  - For each regular language L, there is an regular expression r with $L(r) = L$
  - For every regular expression r, $L(r)$ is a regular language
- Finite automata describe regular languages
  - For each regular language L, there is a FA $\mathcal{A}$ with $L(\mathcal{A}) = L$
  - For every finite automaton $\mathcal{A}$, $L(\mathcal{A})$ is a regular language
- Now: constructive proof of equivalence between REs and FAs
  - We already know that DFAs and NFAs are equivalent
  - Now: Equivalence of NFAs and REs

Transformation of regular expressions into NFAs

- For a regular expression r, derive NFA $\mathcal{A}(r)$ with $L(\mathcal{A}(r)) = L(r)$.
- Idea:
  - Construct NFAs for the elementary REs ($\emptyset, \varepsilon, c \in \Sigma$)
  - We combine NFAs for subexpressions to generate NFAs for composite REs
- All NFAs we construct have a number of special properties:
  - There are no transitions to the initial state.
  - There is only a single final state.
  - There are no transitions from the final state.

We can easily achieve this with ε-transitions!

Reminder: Regular Expression

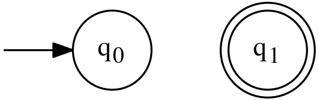
Let Σ be an alphabet.

- The elementary regular expressions over Σ are:
  - $\emptyset$ with $L(\emptyset) = \emptyset$
  - ε with $L(\varepsilon) = \{\varepsilon\}$
  - $c \in \Sigma$ with $L(c) = \{c\}$
- Let r_1 and r_2 be regular expressions over Σ.
  Then the following are also regular expressions over Σ:
  - $r_1 + r_2$ with $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $r_1 \cdot r_2$ with $L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2)$
  - r_1^* with $L(r_1^*) = (L(r_1))^*$

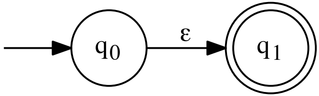
NFAs for elementary REs

Let Σ be the alphabet which r is based on.

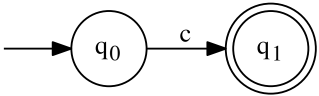
1. $\mathcal{A}(\emptyset) = (\{q_0, q_1\}, \Sigma, \{\}, q_0, \{q_1\})$



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

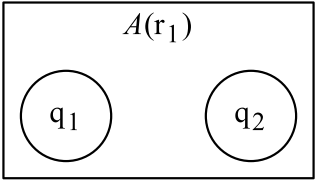


3. $\mathcal{A}(c) = (\{q_0, q_1\}, \Sigma, \{(q_0, c, q_1)\}, q_0, \{q_1\})$ for all $c \in \Sigma$



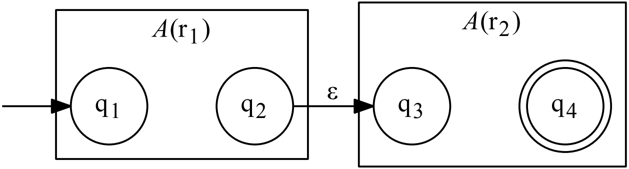
NFAs for composite REs (general)

- Assume in the following:
  - $\mathcal{A}(r_1) = (Q_1, \Sigma, \Delta_1, q_1, \{q_2\})$
  - $\mathcal{A}(r_2) = (Q_2, \Sigma, \Delta_2, q_3, \{q_4\})$
  - $Q_1 \cap Q_2 = \emptyset$
  - $q_0, q_5 \notin Q_1 \cup Q_2$
- $\mathcal{A}(r_1)$ is visualised by a square box with two explicit states
  - The initial state q_1 is on the left
  - The unique accepting state q_2 on the right
  - All other states and transitions are implicit
  - We mark initial/accepting states only for the composite automaton



NFAs for composite REs (concatenation)

4 $\mathcal{A}(r_1 \cdot r_2) = (Q_1 \cup Q_2, \Sigma, \Delta_1 \cup \Delta_2 \cup \{(q_2, \varepsilon, q_3)\}, q_1, \{q_4\})$

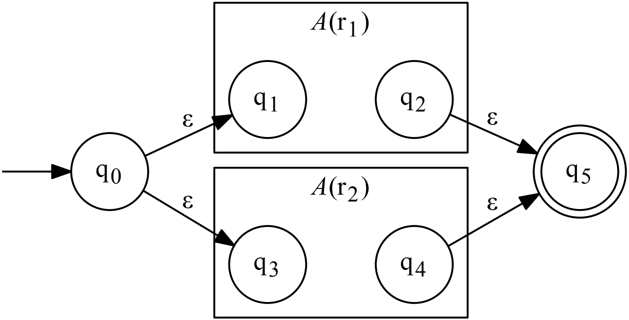


Reminder:

- $\mathcal{A}(r_1) = (Q_1, \Sigma, \Delta_1, q_1, \{q_2\})$
- $\mathcal{A}(r_2) = (Q_2, \Sigma, \Delta_2, q_3, \{q_4\})$

NFAs for composite REs (alternatives)

5 $\mathcal{A}(r_1 + r_2) = (\{q_0, q_5\} \cup Q_1 \cup Q_2, \Sigma, \Delta, q_0, \{q_5\})$
$\Delta = \Delta_1 \cup \Delta_2 \cup \{(q_0, \varepsilon, q_1), (q_0, \varepsilon, q_3), (q_2, \varepsilon, q_5), (q_4, \varepsilon, q_5)\}$

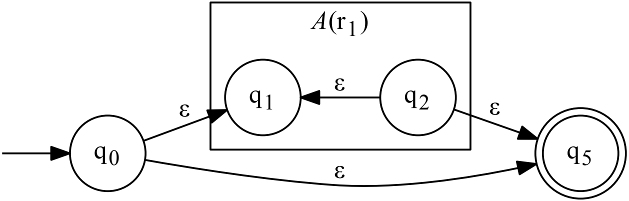


Reminder:

- $\mathcal{A}(r_1) = (Q_1, \Sigma, \Delta_1, q_1, \{q_2\})$
- $\mathcal{A}(r_2) = (Q_2, \Sigma, \Delta_2, q_3, \{q_4\})$

NFAs for composite REs (Kleene Star)

6. $\mathcal{A}(r_1^*) = (\{q_0, q_5\} \cup Q_1, \Sigma, \Delta, q_0, \{q_5\})$
   $\Delta = \Delta_1 \cup \{(q_0, \varepsilon, q_1), (q_2, \varepsilon, q_1), (q_0, \varepsilon, q_5), (q_2, \varepsilon, q_5)\}$



Reminder:

- $\mathcal{A}(r_1) = (Q_1, \Sigma, \Delta_1, q_1, \{q_2\})$

Result: NFAs can simulate REs

The previous construction produces for each regular expression r an NFA $\mathcal{A}$ with $L(\mathcal{A}) = L(r)$.

Corollary

Every language described by a regular expression can be accepted by a non-deterministic finite automaton.

Exercise: transformation of RE into NFA

- Systematically construct an NFA accepting the same language as the regular expression

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

Solution End lecture 5

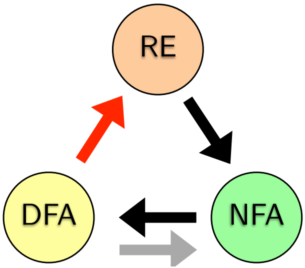
Overview and orientation

- Claim: NFAs, DFAs and REs all describe the same language class
- Previous transformations:
  - REs into equivalent NFAs
  - NFAs into equivalent DFAs
  - (DFAs to equivalent NFAs)

Todo: convert DFA to equivalent RE

- Given a DFA $\mathcal{A}$, derive a regular expression $r(\mathcal{A})$ accepting the same language:

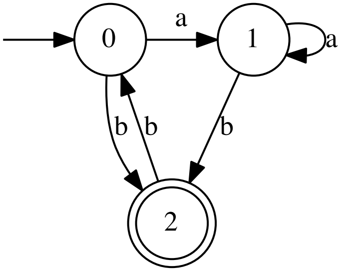
$$L(r(\mathcal{A})) = L(\mathcal{A})$$



Convert DFA into RE

- Goal: transform DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ into RE $r(\mathcal{A})$ with $L(r(\mathcal{A})) = L(\mathcal{A})$
- Idea
  - For each state q,
  - generate an equation describing the language L_q that is accepted when starting from q,
  - depending on the languages accepted at neighbouring states
  - For each transition with c to q': $c \cdot L_{q'}$
  - For final states: additionally ε
- Solve the resulting system for L_{q_0}
  - Result: RE describing $L_{q_0} = L(\mathcal{A})$
- Convention:
  - States are named $\{0, 1, \ldots, n\}$
  - Start state is 0

Convert DFA to RE: Example



- $L_0 = aL_1 + bL_2$
- $L_1 = aL_1 + bL_2$
- $L_2 = bL_0 + \varepsilon$

3 equations, 3 unknowns

What now?

Insert: Arden's Lemma

Lemma:

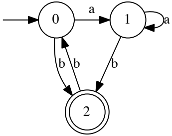
$$\varepsilon \notin L(s) \text{ and } r \doteq sr + t \longrightarrow r \doteq s^* t$$

Compare Arto Salomaa:

$$\varepsilon \notin L(s) \text{ and } r \doteq rs + t \longrightarrow r \doteq ts^*$$

Arden, Dean N.: *Delayed-logic and finite-state machines*, Proceedings of the Second Annual Symposium on Switching Circuit Theory and Logical Design, 1961, pp. 133–151, IEEE

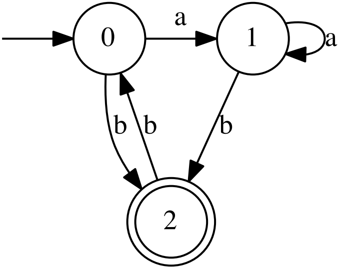
Convert DFA to RE: Example



- $L_0 = aL_1 + bL_2$
- $L_1 = aL_1 + bL_2$
- $L_2 = bL_0 + \varepsilon$

$$
\begin{array}{lcll}
L_1 & \doteq & aL_1 + b(bL_0 + \varepsilon) & \text{[replace } L_2\text{]} \\
 & \doteq & a^*b(bL_0 + \varepsilon) & \text{[Arden]} \\
L_0 & \doteq & a(a^*b(bL_0 + \varepsilon)) + b(bL_0 + \varepsilon) & \text{[replace } L_1, L_2\text{]} \\
 & \doteq & aa^*bbL_0 + aa^*b + bbL_0 + b & \text{[Dist.]} \\
 & \doteq & (aa^*bb + bb)L_0 + aa^*b + b & \text{[Comm.,Dist.]} \\
 & \doteq & (aa^*bb + bb)^*(aa^*b + b) & \text{[Arden]} \\
 & \doteq & ((aa^* + \varepsilon)bb)^*((aa^* + \varepsilon)b) & \text{[Dist.]} \\
 & \doteq & (a^*bb)^*(a^*b) & [rr^* + \varepsilon \doteq r^*]
\end{array}
$$

Convert DFA to RE: Example (continued)

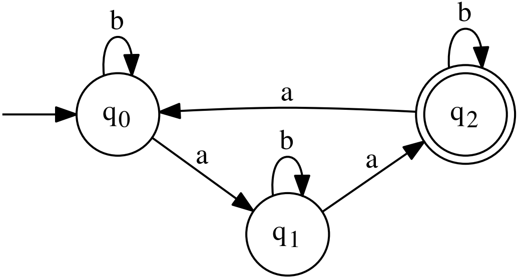


$$
\begin{array}{rcl}
L_0 & \doteq & \ldots \\
 & \doteq & (a^*bb)^*(a^*b)
\end{array}
$$

Therefore: $L(\mathcal{A}) = L((a^*bb)^*(a^*b))$

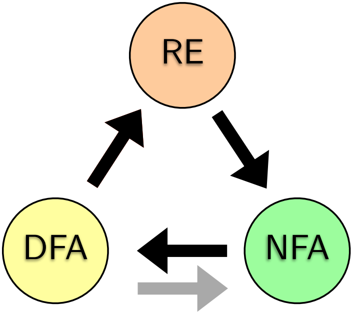
Exercise: conversion from DFA to RE

Transform the following DFA into a regular expression accepting the same language:

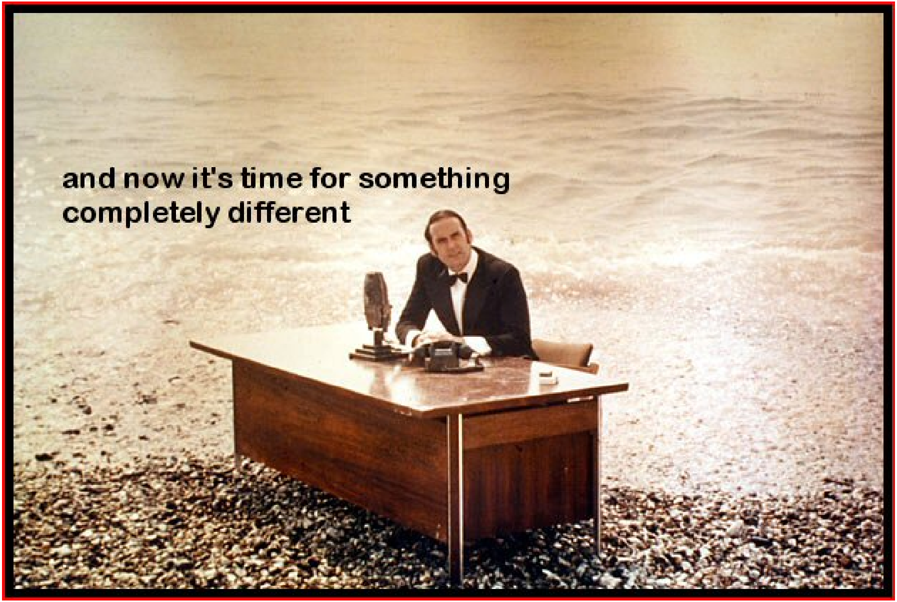


Resume: Finite automata and regular expressions

- We have learned how to convert
  - REs to equivalent NFAs
  - NFAs to equivalent DFAs
  - (DFAs to equivalent NFAs)
  - DFAs to equivalent REs



REs, NFAs and DFAs describe the same class of languages – regular languages!


and now it's time for something
completely different

Outline

Efficient Automata: Minimisation of DFAs

Given the DFA

$$\mathcal{A} = (Q, \Sigma, \delta, q_0, F),$$

we want to derive a DFA

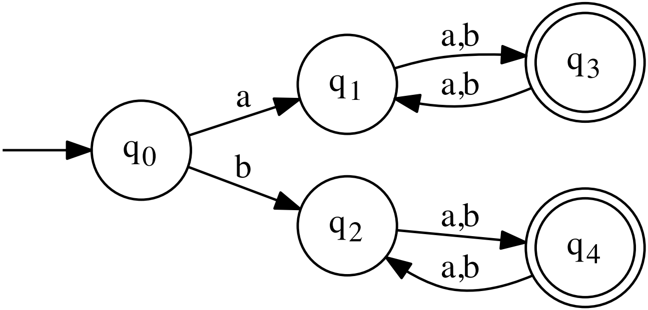
$$\mathcal{A}^- = (Q^-, \Sigma, \delta^-, q_0, F^-),$$

accepting the same language:

$$L(\mathcal{A}) = L(\mathcal{A}^-)$$

for which the number of states (elements of Q^-) is minimal, i.e. there is no DFA accepting $L(\mathcal{A})$ with fewer states.

Minimisation of DFAs: example/exercise



How small can we make it?

Minimisation of DFAs

Idea: For a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, identify pairs of necessarily distinct states

- Base case: Two states p, q are necessarily distinct if:
  - one of them is accepting, the other is not accepting
- Inductive case: Two states p, q are necessarily distinct if
  - there is a $c \in \Sigma$ such that $\delta(p, c) = p', \delta(q, c) = q'$
  - and p', q' are already necessarily distinct

Definition (Necessarily distinct states)

For a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, V is the smallest set of pairs with

- $\{(p, q) \mid p \in F, q \notin F\} \subseteq V$
- $\{(p, q) \mid p \notin F, q \in F\} \subseteq V$
- if $\delta(p, c) = p', \delta(q, c) = q', (p', q') \in V$ for some $c \in \Sigma$, then $(p, q) \in V$.

Minimisation of DFAs

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

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

2. While there exists
   - a new pair of states (p,q) and a symbol c
   - such that the states $\delta(p,c)$ and $\delta(q,c)$ are necessarily distinct,
   - add this pair and its inverse to V:

```
while(∃(p,q) ∈ Q × Q ∃c ∈ Σ | (δ(p,c), δ(q,c)) ∈ V ∧ (p,q) ∉ V)
{
  V = V ∪ {(p,q),(q,p)}
}
```

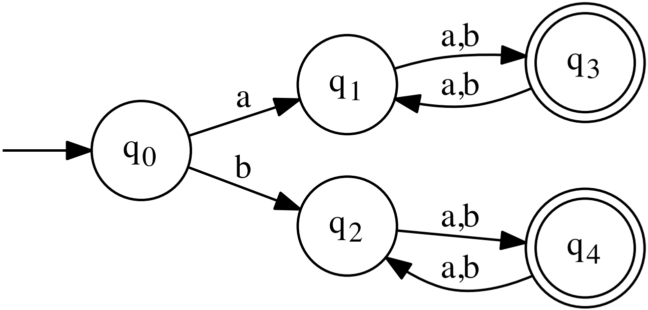
Minimisation of DFAs: merging States

- If there is a pair of states (p, q) such that for every word $w \in \Sigma^*$
  - reading w results in indistinguishable successor states,
  - then p and q are indistinguishable.

  $(p, q) \notin V \Rightarrow \forall w \in \Sigma^* : \delta(p, w)$ and $\delta(q, w))$ are indistinguishable.
- Indistinguishable states p, q can be merged
  - Replace all transitions to p by transitions to q
  - Remove p
- This process can be iterated to identify and merge all indistinguishable pairs of states

Minimisation of DFAs: example

We want to minimize this DFA with 5 states:



Minimisation of DFAs: example (cont.)

This is the formal definition of the DFA:

$$\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$$

with

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

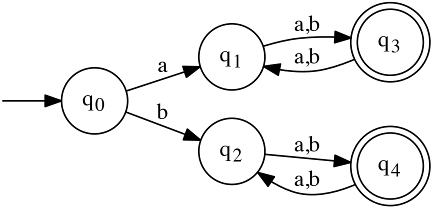
Represent the set V by means of a two-dimensional table with

- the elements of Q as columns and rows
- the elements of V are marked with $\times$
- pairs that are definitely not members of V are marked with $\circ$

Minimisation of DFAs: example (cont.)

1. the initial state of V is obtained by using $F = \{q_3, q_4\}$ and $Q \backslash F = \{q_0, q_1, q_2\}$:

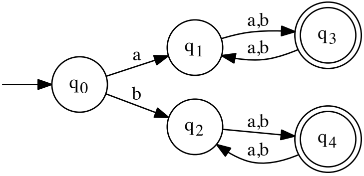
	q_0	q_1	q_2	q_3	q_4
q_0				$\times$	$\times$
q_1				$\times$	$\times$
q_2				$\times$	$\times$
q_3	$\times$	$\times$	$\times$		
q_4	$\times$	$\times$	$\times$		



Minimisation of DFAs: example (cont.)

2 The elements of $\{(q_i, q_i) | i \in \{0, \cdots, 4\}$ are not contained in V since every state is indistinguishable from itself:

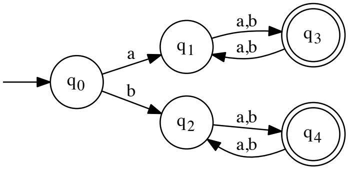
	q_0	q_1	q_2	q_3	q_4
q_0	$\circ$			$\times$	$\times$
q_1		$\circ$		$\times$	$\times$
q_2			$\circ$	$\times$	$\times$
q_3	$\times$	$\times$	$\times$	$\circ$	
q_4	$\times$	$\times$	$\times$		$\circ$



There are eight remaining empty fields. Since the table is symmetric, four pairs of states have to be checked.

Minimisation of DFAs: example (cont.)

3 Check the transitions of every remaining state-pair for every letter.



1. $\delta(q_0, \mathtt{a}) = q_1; \delta(q_1, \mathtt{a}) = q_3; (q_1, q_3) \in V \rightarrow (q_0, q_1), (q_1, q_0) \in V$
2. $\delta(q_0, \mathtt{a}) = q_1; \delta(q_2, \mathtt{a}) = q_4; (q_1, q_4) \in V \rightarrow (q_0, q_2), (q_2, q_0) \in V$
3. $\delta(q_1, \mathtt{a}) = q_3; \delta(q_2, \mathtt{a}) = q_4; (q_3, q_4) \notin V$ (as of yet)
   $\delta(q_1, \mathtt{b}) = q_3; \delta(q_2, \mathtt{b}) = q_4; (q_3, q_4) \notin V$ (as of yet)
4. $\delta(q_3, \mathtt{a}) = q_1; \delta(q_4, \mathtt{a}) = q_2; (q_1, q_2) \notin V$ (as of yet)
   $\delta(q_3, \mathtt{b}) = q_1; \delta(q_4, \mathtt{b}) = q_2; (q_1, q_2) \notin V$ (as of yet)

Minimisation of DFAs: example (cont.)

4 Mark the newly found distinguishable pairs with $\times$:

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

Two pairs remain to be checked.

Minimisation of DFAs: example (cont.)

5. Check the remaining pairs.
6. Since no additional distinguishable state pairs are found, fill empty cells with $\circ$:

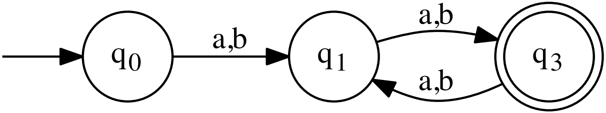
	q_0	q_1	q_2	q_3	q_4
q_0	$\circ$	$\times$	$\times$	$\times$	$\times$
q_1	$\times$	$\circ$	$\circ$	$\times$	$\times$
q_2	$\times$	$\circ$	$\circ$	$\times$	$\times$
q_3	$\times$	$\times$	$\times$	$\circ$	$\circ$
q_4	$\times$	$\times$	$\times$	$\circ$	$\circ$

From the table, we can derive the following indistinguishable state pairs (omitting trivial and symmetric ones):

- (q_1, q_2),
- (q_3, q_4).

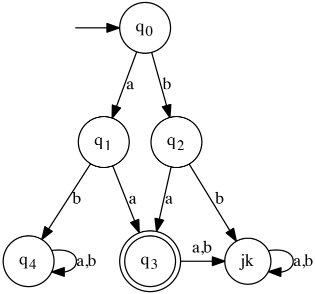
Minimisation of DFAs: example (cont.)

- This is the minimized DFA after merging indistinguishable states:



Handling Ω

- The algorithm does not handle missing transitions/Ω-transitions
  - A rejection due to an Ω-transition is indistinguiable from a rejection due to reachung a junk state
  - However, the algorithm treats these cases differently.
- Solution: If the automaton has Ω-transititions, add an explicit junk state and complete the transition function



Definition (Complete DFA)

A deterministic finite automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ is called *complete*, if δ is a total function, i.e. if $\mathcal{A}$ does not have any Ω-transitions.

Minimisation of DFAs: exercise

Derive a minimal DFA accepting the language

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

Solve the exercise in three steps:

1. Derive an NFA accepting L.
2. Transform the NFA into a DFA.
3. Minimize the DFA.

Uniqueness of minimal DFA

Theorem (The mininmal DFA is unique)

Assume an arbitrary regular language L. Then there is a unique (up to the the renaming of states) complete minimal DFA $\mathcal{A}$ with $L(\mathcal{A}) = L$.

- States can easily be systematically renamed to make equivalent minimal automata strictly equal
- The unique minimal DFA for L can be constructed by minimizing an arbitrary DFA that accepts L

Outline

Equivalence of regular expressions

- Different regular expressions can describe the same language
- Algebraic transformation rules can be used to prove equivalence
  - requires human interaction
  - can be very difficult
  - non-equivalence cannot be shown
- Now: straight-forward algorithm proving equivalence of REs based on FA
- The algorithm is described in the textbook by John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman: *Introduction to Automata Theory, Languages, and Computation (3rd edition)*, 2007 (and earlier editions)

Equivalence of regular expressions: algorithm

1. Given the REs r_1 and r_2, derive NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ accepting their respective languages:

$$L(r_1) = L(\mathcal{A}_1) \quad \text{and} \quad L(r_2) = L(\mathcal{A}_2).$$

2. Transform the NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ into the DFAs $\mathcal{D}_1$ and $\mathcal{D}_2$.
3. Minimize the DFAs $\mathcal{D}_1$ and $\mathcal{D}_2$ yielding the DFAs $\mathcal{M}_1$ and $\mathcal{M}_2$.
4. $r_1 \doteq r_2$ holds iff $\mathcal{M}_1$ and $\mathcal{M}_2$ are identical (modulo renaming of states)

Note: If equivalence can be shown in any intermediate stage of the algorithm, this is sufficient to prove $r_1 \doteq r_2$ (e.g. if $\mathcal{A}_1 = \mathcal{A}_2$).

Exercise: Equivalence of regular expressions

Reusing an exercise from an earlier section, prove the following equivalence (by conversion to minimal DFAs):

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

Solution End lecture 6

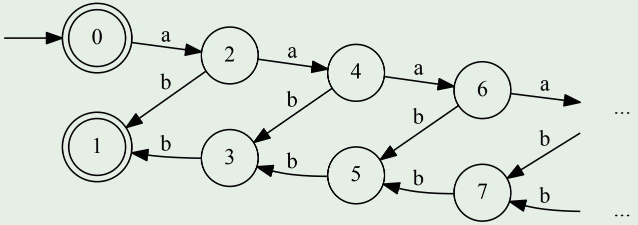
Outline

Non-regular languages

For some simple languages, there is no obvious FA:

Example (Naive automaton $\mathcal{A}$ for $L = \{a^n b^n \mid n \in \mathbb{N}\}$)

$\mathcal{A}$ has an infinite number of states:

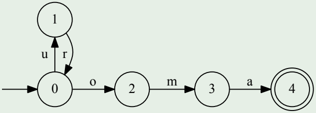


- Is there a better solution?
- If no, how can this be shown?

Pumping Lemma: Idea

1. Every regular language L is accepted by a deterministic finite Automaton $\mathcal{A}_L$.
2. If L contains arbitrarily long words, then $\mathcal{A}_L$ must contain a cycle.
   - L contains arbitrarily long words iff L is infinite.
3. If $\mathcal{A}_L$ contains a cycle, then the cycle can be traversed arbitrarily often (and the resulting word will be accecpted).

Example (Cyclic DFA $\mathcal{C}$)



- $\mathcal{C}$ accepts `uroma`
- $\mathcal{C}$ also accepts `ururur...oma`

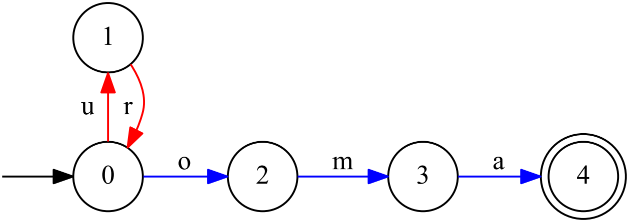
The Pumping Lemma

Lemma

Let L be a regular language.
Then there exists a $k \in \mathbb{N}$ such that for every word $s \in L$ with $|s| \geq k$ the following holds:

1. $\exists u, v, w \in \Sigma^*(s = u \cdot v \cdot w)$,
   i.e. s consists of prolog u, cycle v and epilog w,
2. $v \neq \varepsilon$,
   i.e. the cycle has a length of at least 1,
3. $|u \cdot v| \leq k$,
   i.e. prolog and cycle combined have a length of at most k,
4. $\forall h \in \mathbb{N}(u \cdot v^h \cdot w \in L)$,
   i.e. an arbitrary number of cycle transitions results in a word of the language L.

The Pumping Lemma visualised



- $\mathcal{C}$ has 5 states $\qquad k = 5$
- `uroma` has 5 letters $\qquad s = \texttt{uroma}$
- There is a segmentation $s = u \cdot v \cdot w$ $\qquad u = \varepsilon \quad v = \texttt{ur} \quad w = \texttt{oma}$
- such that $|v| \neq \varepsilon$ $\qquad v = \texttt{ur}$
- and $|u \cdot v| \leq k$ $\qquad |\varepsilon \cdot \texttt{ur}| = 2 \leq 5$
- and $\forall h \in \mathbb{N}(u \cdot v^h \cdot w \in L(\mathcal{C}))$ $\qquad (\texttt{ur})^*\texttt{oma} \subseteq L(\mathcal{C})$

Pumping Lemma: Idea II

- If L is regular, then there exists a DFA $\mathcal{A}$ with $L = L(\mathcal{A})$
- That DFA has (e.g.) $k - 1$ states
- For every $w \in L$ with $|w| \geq k$ the automaton must execute a loop
- u is the word read to the first state of the loop
- v is the word read in the loop
- w is the word read after the loop
- …so every word that traverses v zero or multiple times is also accepted by $\mathcal{A}$

Using the Pumping Lemma

- The Pumping Lemma describes a property of regular languages
  - *If L is regular, then some words can be pumped up.*
- Goal: proof of irregularity of a language
  - *If L has property X, then L is not regular.*
- How can the Pumping Lemma help?

Theorem (Contraposition)

$$A \rightarrow B \quad \Leftrightarrow \quad \neg B \rightarrow \neg A$$

Contraposition of the Pumping Lemma

The Pumping Lemma in formal logic:

$$\begin{aligned} reg(L) \quad \rightarrow \quad & \exists k \in \mathbb{N}\ \forall s \in L : (|s| \geq k \rightarrow \\ & \exists u, v, w : (s = u \cdot v \cdot w \wedge v \neq \varepsilon \wedge |u \cdot v| \leq k \wedge \\ & \forall h \in \mathbb{N} : (u \cdot v^h \cdot w \in L))) \end{aligned}$$

Contraposition of the PL:

$$\begin{aligned} \neg(\exists k \in \mathbb{N}\ \forall s \in L(|s| \geq k \rightarrow \\ \exists u, v, w(s = u \cdot v \cdot w \wedge v \neq \varepsilon \wedge |u \cdot v| \leq k \wedge \\ \forall h \in \mathbb{N}(u \cdot v^h \cdot w \in L)))) \quad \rightarrow \quad \neg reg(L) \end{aligned}$$

After pushing negation inward and doing some propositional transformations:

$$\begin{aligned} \forall k \in \mathbb{N}\ \exists s \in L(|s| \geq k \wedge \\ \forall u, v, w(s = u \cdot v \cdot w \wedge v \neq \varepsilon \wedge |u \cdot v| \leq k \rightarrow \\ \exists h \in \mathbb{N}(u \cdot v^h \cdot w \notin L))) \quad \rightarrow \quad \neg reg(L) \end{aligned}$$

What does it mean?

$$
\begin{aligned}
\forall k \in \mathbb{N}\ \exists s \in L(|s| \geq k \wedge \\
\forall u, v, w(s = u \cdot v \cdot w \ \wedge v \neq \varepsilon \wedge |u \cdot v| \leq k \ \rightarrow \\
\exists h \in \mathbb{N}\ (u \cdot v^h \cdot w \ \notin \ L))) \quad \rightarrow \quad \neg reg(L)
\end{aligned}
$$

If for every number k there is a word s with length at least k
and for every segmentation $u \cdot v \cdot w$ of s (with $v \neq \varepsilon$ and $|u \cdot v| \leq k$)
there is a number h such that $u \cdot v^h \cdot w$ does not belong to L,
then L is not regular.

Proving Irregularity for a Language

We have to show:

- For every natural number k
- For an unspecified arbitrary natural number k
- there is a word $s \in L$ that is longer than k
- such that every segmentation $u \cdot v \cdot w = s$ with $|u \cdot v| \leq k$ and $|v| \neq \varepsilon$
- can be pumped up into a word $u \cdot v^h \cdot w \notin L$.

Example ($L = a^n b^n$)

- Choose $s = a^k b^k$. It follows:

$$s = \underbrace{a^i}_{u} \cdot \underbrace{a^j}_{v} \cdot \underbrace{a^\ell \cdot b^k}_{w}$$

  - $i + j + \ell = k$
  - since $|u \cdot v| \leq k$ holds, u and v consist only of as
  - $v \neq \varepsilon$ implies $j \geq 1$
- Choose $h = 0$. It follows:
  - $u \cdot v^h \cdot w = u \cdot w = a^{i+\ell} b^k$
  - $j \geq 1$ implies $i + \ell < k$
  - $a^{i+\ell} b^k \notin L$

Regarding quantifiers

Four quantifiers:

- In the lemma:

$$\exists k \forall s \exists u, v, w \forall h (u \cdot v^h \cdot w \in L)$$

- To show irregularity:

$$\forall k \exists s \forall u, v, w \exists h (u \cdot v^h \cdot w \notin L)$$

To do:

1. Find a word s depending on the length k.
2. Find an h depending on the segmentation $u \cdot v \cdot w$.
3. Prove that $u \cdot v^h \cdot w \notin L$ holds.

Exercise: $a^n b^m$ with $n < m$

Use the pumping lemma to show that

$$L = \{a^n b^m \mid n < m\}$$

is not regular.

Reminder:

1. Find a word s depending on the length k.
2. Find an h depending on the segmentation $u \cdot v \cdot w$.
3. Prove that $u \cdot v^h \cdot w \notin L$ holds.

Solution

Challenging exercise / homework

Let L be the number containing all words of the form $\mathtt{a}^p$ where p is a prime number:

$$L = \{\mathtt{a}^p \mid p \in \mathbb{P}\}.$$

Prove that L is not a regular language.

Hint: let $h = p + 1$

Solution

Practical relevance of irregularity

Finite automata cannot count arbitrarily high.

Examples (Nested dependencies)

C	for every { there is a }
XML	for every `<token>` there is a `</token>`
LaTeX	for every `\begin{env}` there is a `\end{env}`
German	for every subject there is a predicate

```
Erinnern Sie sich,
   wie der Krieger,
      der die Botschaft,
         die den Sieg,
            den die Griechen bei Marathon
            errungen hatten,
         verkündete,
      brachte,
   starb!
```

Pumping Lemma: Summary

- Every regular language is accepted by a DFA $\mathcal{A}$ (with k states).
- Pumping lemma: words with at least k letters can be pumped up.
- If it is possible to pump up a word $w \in L$ and obtain a word $w' \notin L$, then L is not regular.
  - Make sure to handle quantifiers correctly!
- Practical relevance
  - FAs cannot count arbitrarily high.
  - Nested structures are not regular.
    - programming languages
    - natural languages
  - More powerful tools are needed to handle these languages.

Outline

Regular languages: Closure properties

Reminder:

- Formal languages are sets of words (over a finite alphabet)
- A formal language L is a *regular language* if any of the following holds:
  - There exists an NFA $\mathcal{A}$ with $L(\mathcal{A}) = L$
  - There exists a DFA $\mathcal{A}$ with $L(\mathcal{A}) = L$
  - There exists a regular expression r with $L(r) = L$
  - There exists a regular *grammar* G with $L(G) = L$
- Pumping lemma: not all languages are regular

Question
What can we do to regular languages and be sure the result is still regular?

Closure properties (Question)

Question: If L_1 and L_2 are regular languages, does the same hold for

$L_1 \cup L_2$?	(closure under union)
$L_1 \cap L_2$?	(closure under intersection)
$L_1 \cdot L_2$?	(closure under concatenation)
$\overline{L_1}$, i.e. $\Sigma^* \setminus L$?	(closure under complement)
L_1^*?	(closure under Kleene-star)

Closure properties (Theorem)

Theorem

Let L_1 and L_2 be regular languages. Then the following langages are also regular:

- $L_1 \cup L_2$
- $L_1 \cap L_2$
- $L_1 \cdot L_2$
- $\overline{L_1}$, *i.e.* $\Sigma^* \setminus L$
- L_1^* *?*

Proof.

Idea: using (disjoint) finite automata for L_1 and L_2, construct an automaton for the different languages above. □

Closure under union, concatenation, and Kleene-star

We use the same construction that was used to generate NFAs for regular expressions:

Let $\mathcal{A}_{L_1}$ and $\mathcal{A}_{L_2}$ be automata for L_1 and L_2.

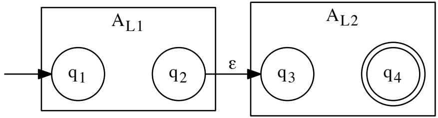
$L_1 \cup L_2$ new initial and final states, ε-transitions to initial/final states of $\mathcal{A}_{L_1}$ and $\mathcal{A}_{L_2}$

$L_1 \cdot L_2$ ε-transition from final state of $\mathcal{A}_{L_1}$ to initial state of $\mathcal{A}_{L_2}$

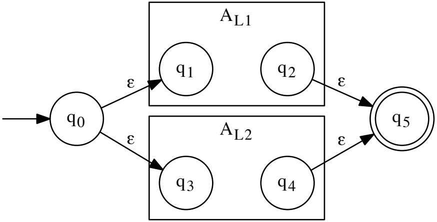
$(L_1)^*$
- new initial and final states (with ε-transitions),
- ε-transitions from the original final states to the original initial state,
- ε-transition from the new initial to the new final state.

Visual refresher

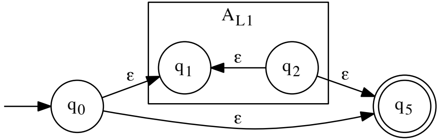
$L_1 \circ L_2$



$L_1 \cup L_2$



L_1^*



Closure under intersection

Let $\mathcal{A}_{L_1} = (Q_1, \Sigma, \delta_1, q_{0_1}, F_1)$ and $\mathcal{A}_{L_2} = (Q_2, \Sigma, \delta_2, q_{0_2}, F_2)$ be DFAs for L_1 and L_2.

An automaton $L = (Q, \Sigma, \delta, q_0, F)$ for $\mathcal{A}_{L_1} \cap \mathcal{A}_{L_2}$ can be generated as follows:

- if there are Ω transitions, add junk state(s).
- $Q = Q_1 \times Q_2$
- $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ for all $q_1 \in Q_1, q_2 \in Q_2, a \in \Sigma$
- $q_0 = (q_{0_1}, q_{0_2})$
- $F = F_1 \times F_2$

This so-called product automaton

- starts in state that corresponds to initial states of $\mathcal{A}_{L_1}$ and $\mathcal{A}_{L_2}$,
- simulates simultaneous processing in both automata
- accepts if both $\mathcal{A}_{L_1}$ and $\mathcal{A}_{L_2}$ accept.

Product automaton: exercise

Generate automata for

- $L_1 = \{w \in \{0,1\}^* \mid |w|_1$ is divisible by $2\}$
- $L_2 = \{w \in \{0,1\}^* \mid |w|_1$ is divisible by $3\}$

Then generate an automaton for $L_1 \cap L_2$.

Closure under complement

Let $\mathcal{A}_L$ be a complete DFA for the language L.
(If there are Ω transitions, add a junk state.)

Then $\overline{\mathcal{A}_L} = (Q, \Sigma, q_0, \delta, Q \setminus F)$ is an automaton accepting $\overline{L}$:

- if $w \in L(\mathcal{A})$ then $\delta'(q_0, w) \in F$, i.e. $\delta'(q_0, w) \notin Q \setminus F$, which implies $w \notin L(\overline{\mathcal{A}_L})$.
- if $w \notin L(\mathcal{A})$ then $\delta'(q_0, w) \notin F$, i.e. $\delta'(q_0, w) \in Q \setminus F$, which implies $w \in L(\overline{\mathcal{A}_L})$.

Reminder:

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

$\delta'(q_0, w)$ is the final state of the automaton after processing w

All we have to do is exchange final and non-final states.

Closure properties: exercise

Show that $L = \{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$ is not regular.

Hint: Use the following:

- $a^n b^n$ is not regular. (Pumping lemma)
- $a^* b^*$ is regular. (Regular expression)
- (one of) the closure properties shown before.

End lecture 7

Finite languages and automata

Theorem (Regularity of finite languages)

Every finite language, i.e. every language containing only a finite number of words, is regular.

Proof.

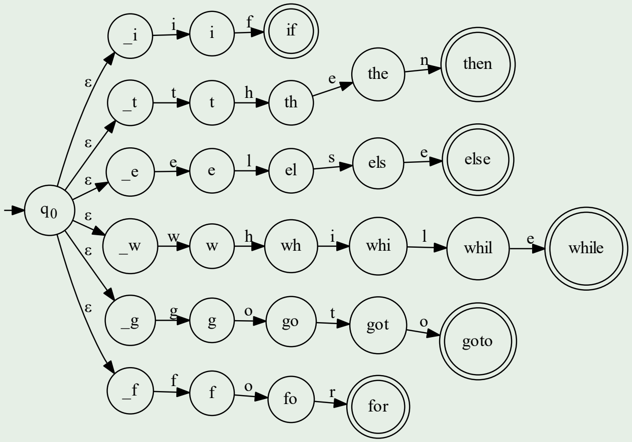
Let $L = \{w_1, \ldots, w_n\}$.

- For each w_i, generate an automaton $\mathcal{A}_i$ with initial state q_{0_i} and final state q_{f_i}.
- Let q_0 be a new state, from which there is an ε-transition to each q_{0_i}.

Then the resulting automaton, with q_0 as initial state and all q_{f_i} as final states, accepts L. $\square$

Example: finite language

Example ($L = \{if, then, else, while, goto, for\}$ over Σ_{ASCII})



Finite languages and regular expressions

Theorem (Regularity of finite languages)

Every finite language is regular.

Alternate proof.

Let $L = \{w_1, w_2, \ldots, w_n\}$.
Write L as the regular expression $w_1 + w_2 + \ldots + w_n$. $\square$

Corollary

The class of finite languages is characterised by

- *acyclic finite automata,*
- *regular expressions without Kleene star.*

Decision problems

For regular languages L_1 and L_2 and a word w, answer the following questions:

Is there a word in L_1?	emptiness problem
Is w an element of L_1?	word problem
Is L_1 equal to L_2?	equivalence problem
Is L_1 finite?	finiteness problem

Emptiness problem

Theorem (Emptiness problem for regular languages)

The emptiness problem for regular languages is decidable.

Proof.

Algorithm: Let $\mathcal{A}$ be an automaton accepting the language L.

- Starting with the initial state q_0, mark all states to which there is a transition from q_0 as reachable.
- Continue with transitions from states which are already marked as reachable until either a final state is reached or no further states are reachable.
- If a final state is reachable, then $L \neq \emptyset$ holds.

□

Group exercise: Emptiness problem

- Find an alternative proof for the emptiness problem!

Word problem

Theorem (Word problem for regular languages)

The word problem for regular languages is decidable.

Proof.

Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA accepting L and $w = c_1 c_2 \ldots c_n$.
Algorithm:

- $q_1 := \delta(q_0, c_1)$
- If $q_1 = \Omega$ holds, then w $\notin L$
- $q_2 := \delta(q_1, c_2)$
- $\ldots$
- If $q_n \in F$ holds, then $\mathcal{A}$ accepts w.

$\square$

All we have to do is simulate the run of $\mathcal{A}$ on w.

Equivalence problem

Theorem (Equivalence problem for regular languages)

The equivalence problem for regular languages is decidable.

We have already shown how to prove this using minimised DFAs for L_1 and L_2.

Alternative proof.

One can also use closure properties and decidability of the emptiness problem:

$$L_1 = L_2 \text{ iff } \underbrace{(L_1 \cap \overline{L_2})}_{\text{words that are in } L_1 \text{, but not in } L_2} \cup \underbrace{(\overline{L_1} \cap L_2)}_{\text{words that are not in } L_1 \text{, but in } L_2} = \emptyset$$

□

Finiteness problem

Theorem (Finiteness problem for regular languages)

The finiteness problem for regular languages is decidable.

Proof.

Idea: if there is a loop in an accepting run, words of arbitrary length are accepted.

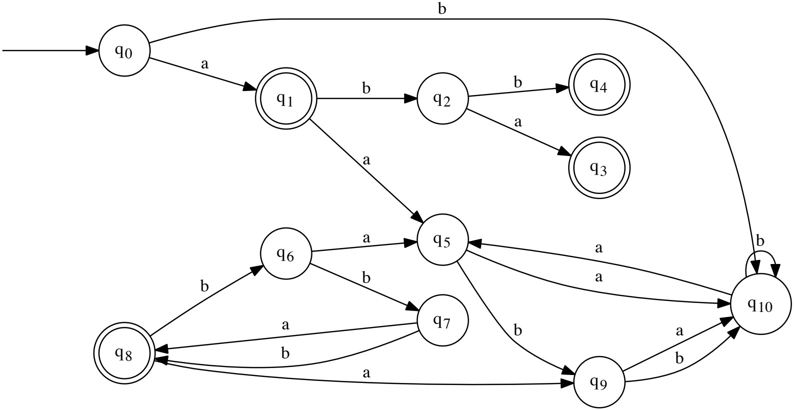
Let $\mathcal{A}$ be a DFA accepting L.

- Eliminate from $\mathcal{A}$ all states that are not reachable from the initial state, obtaining $\mathcal{A}_r$.
- Eliminate from $\mathcal{A}_r$ all states from which no final state is reachable, obtaining $\mathcal{A}_f$.
- L is infinite iff $\mathcal{A}_f$ contains a loop.

$\square$

Exercise: Finiteness

Consider the following DFA $\mathcal{A}$. Use to previous algorithm to decide if $L(\mathcal{A})$ is finite. Describe $L(\mathcal{A})$.



Regular languages: summary

Regular languages

- are characterised by
  - NFAs / DFAs
  - regular expressions
  - regular grammars
- can be transferred from one formalism to another one
- are closed under all operators (considered here)
- all decision problems (considered here) are decidable
- do not contain several interesting languages ($a^n b^n$, counting)
  - see chapter on grammars
- can express important features of programming languages
  - keywords
  - legal identifiers
  - numbers
- in compilers, these features are used by scanners (next chapter)

Outline

Scanners and Flex

Computing Environment

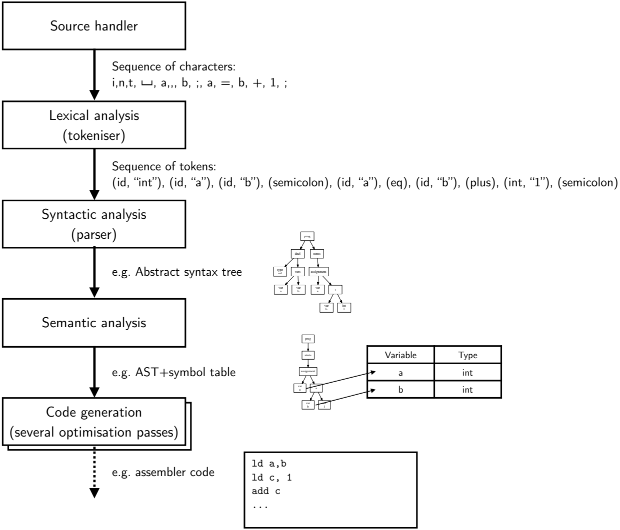
- For practical exercises, you will need a complete Linux/UNIX environment. If you do not run one natively, there are several options:
  - You can install VirtualBox (`https://www.virtualbox.org`) and then install e.g. Ubuntu (`http://www.ubuntu.com/`) on a virtual machine. Make sure to install the *Guest Additions*
  - For Windows, you can install the complete UNIX emulation package Cygwin from `http://cygwin.com`
  - For MacOS, you can install `fink` (`http://fink.sourceforge.net/`) or MacPorts (`https://www.macports.org/`) and the necessary tools
- You will need at least `flex`, `bison`, `gcc`, `grep`, `sed`, `AWK`, `make`, and a good text editor

Syntactic Structure of Programming Languages

Most computer languages are mostly context-free

- Regular: vocabulary
  - Keywords, operators, identifiers
  - Described by regular expressions or regular grammar
  - Handled by (generated or hand-written) scanner/tokenizer/lexer
- Context-free: program structure
  - Matching parenthesis, block structure, algebraic expressions, ...
  - Described by context-free grammar
  - Handled by (generated or hand-written) *parser*
- Context-sensitive: e.g. declarations
  - Described by human-readable constraints
  - Handled in an ad-hoc fashion (e.g. symbol table)

High-Level Architecture of a Compiler



Source Handler

- Handles input files
- Provides character-by-character access
- May maintain file/line/colum (for error messages)
- May provide look-ahead

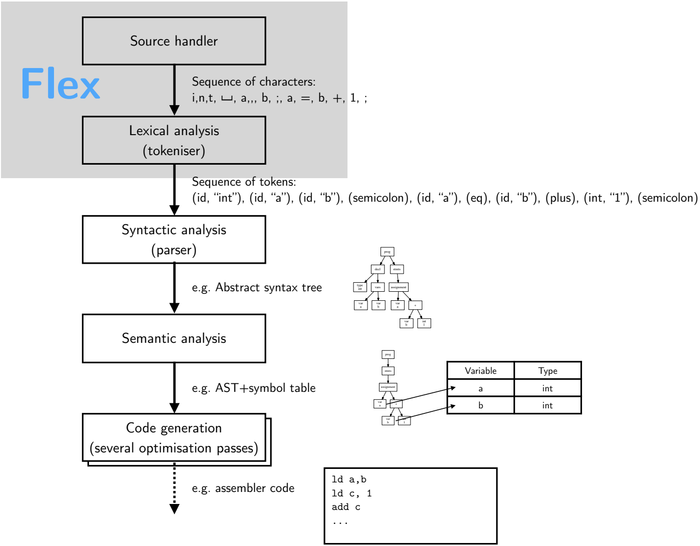
Result: Sequence of characters (with positions)

Lexical Analysis/Scanning

- Breaks program into tokens
- Typical tokens:
  - Reserved word (`if`, `while`)
  - Identifier (`i`, `database`)
  - Symbols (`{, }, (, ), +, -, ...`)

Result: Sequence of tokens

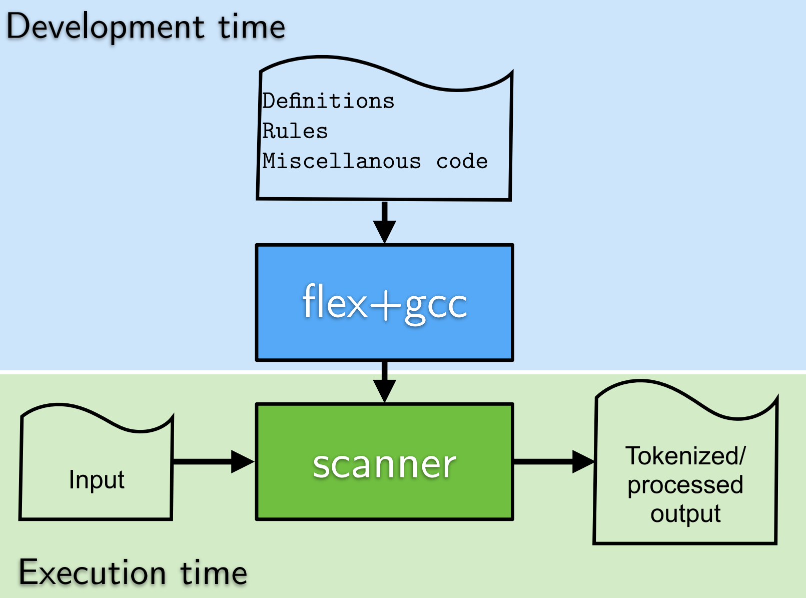
Automatisation with Flex



Flex Overview

- Flex is a scanner generator
- Input: Specification of a regular language and what to do with it
  - Definitions - named regular expressions
  - Rules - patterns+actions
  - (miscellaneous support code)
- Output: Source code of scanner
  - Scans input for patterns
  - Executes associated actions
  - Default action: Copy input to output
  - Interface for higher-level processing: `yylex()` function

Flex Overview



Flex Example Task

- Goal: Sum up all numbers in a file, separately for ints and floats
- Given: A file with numbers and commands
  - Ints: Non-empty sequences of digits
  - Floats: Non-empty sequences of digits, followed by decimal dot, followed by (potentially empty) sequence of digits
  - Command `print`: Print current sums
  - Command `reset`: Reset sums to 0.
- At end of file, print sums

Flex Example Output

Input

```
12 3.1415
0.33333
print reset
2 11
1.5 2.5 print
1
print 1.0
```

Output

```
int:   12 ("12")
float: 3.141500 ("3.1415")
float: 0.333330 ("0.33333")
Current: 12 : 3.474830
Reset
int:   2 ("2")
int:   11 ("11")
float: 1.500000 ("1.5")
float: 2.500000 ("2.5")
Current: 13 : 4.000000
int:   1 ("1")
Current: 14 : 4.000000
float: 1.000000 ("1.0")
Final  14 : 5.000000
```

Basic Structure of Flex Files

- Flex files have 3 sections
  - Definitions
  - Rules
  - User Code
- Sections are separated by `%%`
- Flex files traditionally use the suffix `.l`

Example Code (definition section)

```
%%option noyywrap

DIGIT    [0-9]

%{
    int intval   = 0;
    double floatval = 0.0;
%}

%%
```

Example Code (rule section)

```
{DIGIT}+    {
   printf( "int:   %d (\"%s\")\n", atoi(yytext), yytext );
   intval += atoi(yytext);
}
{DIGIT}+"."{DIGIT}*        {
   printf( "float: %f (\"%s\")\n", atof(yytext),yytext );
   floatval += atof(yytext);
}
reset {
   intval = 0;
   floatval = 0;
   printf("Reset\n");
}
print {
   printf("Current: %d : %f\n", intval, floatval);
}
\n|. {
   /* Skip */
}
```

Example Code (user code section)

```
%%
int main( int argc, char **argv )
{
   ++argv, --argc;  /* skip over program name */
   if ( argc > 0 )
      yyin = fopen( argv[0], "r" );
   else
      yyin = stdin;

   yylex();

   printf("Final  %d : %f\n", intval, floatval);
}
```

Generating a scanner

```
> flex  -t numbers.l > numbers.c
> gcc   -c -o numbers.o numbers.c
> gcc numbers.o -o scan_numbers
> ./scan_numbers Numbers.txt
int:   12 ("12")
float: 3.141500 ("3.1415")
float: 0.333330 ("0.33333")
Current: 12 : 3.474830
Reset
int:   2 ("2")
int:   11 ("11")
float: 1.500000 ("1.5")
float: 2.500000 ("2.5")
...
```

Flexing in detail

```
> flex -tv numbers.l > numbers.c
scanner options: -tvI8 -Cem
37/2000 NFA states
18/1000 DFA states (50 words)
5 rules
Compressed tables always back-up
1/40 start conditions
20 epsilon states, 11 double epsilon states
6/100 character classes needed 31/500 words
of storage, 0 reused
36 state/nextstate pairs created
24/12 unique/duplicate transitions
...
381 total table entries needed
```

Exercise: Building a Scanner

- Download the `flex` example and input from `http://wwwlehre.dhbw-stuttgart.de/~sschulz/fla2015.html`
- Build and execute the program:
  - Generate the scanner with `flex`
  - Compile/link the C code with `gcc`
  - Execute the resulting program in the input file

Definition Section

- Can contain `flex` options
- Can contain (C) initialization code
  - Typically `#include()` directives
  - Global variable definitions
  - Macros and type definitions
  - Initialization code is embedded in `%{` and `%}`
- Can contain definitions of regular expressions
  - Format: `NAME RE`
  - Defined NAMES can be referenced later

Regular Expressions in Practice (1)

- The minimal syntax of REs as discussed before suffices to show their equivalence to finite state machines
- Practical implementations of REs (e.g. in Flex) use a richer and more powerful syntax
- Regular expressions in Flex are based on the ASCII alphabet
- We distinguish between the set of operator symbols

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

  and the set of regular expressions
  1. $c \in \Sigma_{\mathrm{ASCII}} \backslash O \longrightarrow c \in R$
  2. "$\texttt{.}$"$\in R$
     any character but newline (`\n`)

Regular Expressions in Practice (2)

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 " ")

Regular Expressions in Practice (3)

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. $r_1, r_2 \in R \longrightarrow r_1 \mid r_2 \in R$
   infix operation using "$\mid$" rather than "$+$"
8. $r \in R \longrightarrow r\star \in R$
   Kleene star
9. $r \in R \longrightarrow r+ \in R$
   (one or more or r)
10. $r \in R \longrightarrow r? \in R$
   optional presence (zero or one r)

Regular Expressions in Practice (4)

11. $r \in R, n \in \mathbb{N} \longrightarrow r\{n\} \in R$
    concatenation of n times r
12. $r \in R;\ m, n \in \mathbb{N};\ m \leq n \longrightarrow r\{m,n\} \in R$
    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 $r_1 r_2$, however, only the contents of r_1 is consumed. The trailing context r_2 can be processed by the next rule.
16. $r \in R \longrightarrow (r) \in R$
    Grouping regular expressions with brackets.

Regular Expressions in Practice (5)

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. $w \in \{\Sigma_{\mathrm{ASCII}} \setminus \{\backslash, \texttt{"}\}\}^* \longrightarrow \texttt{"}w\texttt{"} \in R$
    verbatim text (no escape sequences)

Regular Expressions in Practice (6)

21. $r \in R \longrightarrow \tilde{}r \in R$
    The upto operator matches the shortest string ending with r.
22. predefined character classes
    - `[:alnum:]` `[:alpha:]` `[:blank:]`
    - `[:cntrl:]` `[:digit:]` `[:graph:]`
    - `[:lower:]` `[:print:]` `[:punct:]`
    - `[:space:]` `[:upper:]` `[:xdigit:]`

Regular Expressions in Practice (precedences)

I. “`(`”, “`)`” (strongest)
II. “$\star$”, “`+`”, “`?`”
III. concatenation
IV. “`|`” (weakest)

Example

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

Rule of thumb: `*`, `+`, `?` bind the smallest possible RE.
Use () if in doubt!

Regular Expressions in Practice (definitions)

- Assume definiton `NAME DEF`
  - In later REs. {`NAME`} is expanded to `(DEF)`
- Example:

```
DIGIT   [0-9]
INTEGER {DIGIT}+
PAIR    \({INTEGER},{INTEGER}\)
```

Exercise: extended regular expressions

Given the alphabet Σ_{ascii}, how would you express the following practical REs using only the simple REs we have used so far?

1. `[a-z]`
2. `[^0-9]`
3. `(r)+`
4. `(r){3}`
5. `(r){3,7}`
6. `(r)?`

Example Code (definition section) (revisited)

```
%%option noyywrap

DIGIT    [0-9]

%{
   int    intval   = 0;
   double floatval = 0.0;
%}


%%
```

Rule Section

- This is the core of the scanner!
- Rules have the form `PATTERN ACTION`
- Patterns are regular expressions
  - Typically use previous definitions
- There has to be white space between pattern and action
- Actions are C code
  - Can be embedded in { and }
  - Can be simple C statements
  - For a token-by-token scanner, must include `return` statement
  - Inside the action, the variable `yytext` contains the text matched by the pattern
  - Otherwise: Full input file is processed

Example Code (rule section) (revisited)

```
{DIGIT}+    {
   printf( "int:   %d (\"%s\")\n", atoi(yytext), yytext );
   intval += atoi(yytext);
}
{DIGIT}+"."{DIGIT}*        {
   printf( "float: %f (\"%s\")\n", atof(yytext),yytext );
   floatval += atof(yytext);
}
reset {
   intval = 0;
   floatval = 0;
   printf("Reset\n");
}
print {
   printf("Current: %d : %f\n", intval, floatval);
}
w\n|. {
   /* Skip */
}
```

User code section

- Can contain all kinds of code
- For stand-alone scanner: must include `main()`
- In `main()`, the function `yylex()` will invoke the scanner
- `yylex()` will read data from the file pointer `yyin`
  (so `main()` must set it up reasonably)

Example Code (user code section) (revisited)

```
%%
int main( int argc, char **argv )
{
   ++argv, --argc;  /* skip over program name */
   if ( argc > 0 )
      yyin = fopen( argv[0], "r" );
   else
      yyin = stdin;

   yylex();

   printf("Final  %d : %f\n", intval, floatval);
}
```

A comment on comments

- Comments in Flex are complicated
  - ...because nearly everything can be a pattern
- Simple rules:
  - Use old-style C comments `/* This is a comment */`
  - Never start them in the first column
  - Comments are copied into the generated code
  - Read the manual if you want the dirty details

Flex Miscellaneous

- Flex online:
  - `http://flex.sourceforge.net/`
  - Manual: `http://flex.sourceforge.net/manual/`
  - REs: `http://flex.sourceforge.net/manual/Patterns.html`
- `make` knows `flex`
  - Make will automatically generate `file.o` from `file.l`
  - Be sure to set `LEX=flex` to enable `flex` extensions
  - `Makefile` example:

```
LEX=flex
all: scan_numbers
numbers.o: numbers.l

scan_numbers: numbers.o
gcc numbers.o -o scan_numbers
```

Flexercise (1)

A security audit firm needs a tool that scans documents for the following:

- Email addesses
  - Fomat: String over [A-Za-z0-9_.∼-], followed by @, followed by a domain name according to RFC-1034, https://tools.ietf.org/html/rfc1034, Section 3.5 (we only consider the case that the domain name is not empty)
- (simplified) Web addresses
  - http:// followed by an RFC-1034 domain name, optionally followed by :<port> (where <port> is a non-empty sequence of digits), optionally followed by one or several parts of the form /<str>, where <str> is a non-empty sequence over [A-Za-z0-9_.∼-]

Flexercise (2)

- Bank account numbers
  - Old-style bank account numbers start with an identifying string, optionally followed by `.`, optionally followed by `:`, optionally followed by spaces, followed by a non-empty sequence of up to 10 digits. Identifying strings are `Konto`, `Kto`, `KNr`, `Ktonr`, `Kontonummer`
  - (German) IBANs are strings starting with `DE`, followed by exactly 20 digits. Human-readable IBANs have spaces after every 4 characters (the last group has only 2 characters)
- Examples:
  - `Rosenda@gidwd-39.at.z8o3rw2.zhv`
  - `http://jzl.j51g.m-x95.vi5/oj1g_i1/72zz_gt68f`
  - `http://iefbottw99.v4gy.zslu9q.zrc2es01nr.dy:8004`
  - `Ktonr.  241524`
  - `DE26959558703965641174`
  - `DE27 0192 8222 4741 4694 55`

Flexercise (3)

- Create a programm scanning for the data described above and printing the found items.
- Example data for Jan Hladik's lecture can be found in `http://wwwlehre.dhbw-stuttgart.de/~hladik/FLA/skim-source.txt`
- Example input/output data for Stephan Schulz's lecture can be found under `http://wwwlehre.dhbw-stuttgart.de/~sschulz/fla2015.html`

End lecture 8

Outline

Outline

Formal Grammars: Motivation

So far, we have seen

- regular expressions: compact description of regular languages
- finite automata: recognise words of a regular language

Another, more powerful formalism: formal grammars

- generate words of a language
- contain a set of rules allowing to replace symbols with different symbols

Grammars: examples

Example (Formal grammars)

$S \to aA, \quad A \to bB, \quad B \to \varepsilon$
generates ab (starting from S): $S \to aA \to abB \to ab$

$S \to \varepsilon, \quad S \to aSb$
generates $a^n b^n$

Grammars: definition

Definition (Grammar according to Chomsky)

A (formal) grammar is a quadruple

$$G = (N, \Sigma, P, S)$$

with

1. the set of non-terminal symbols N,
2. the set of terminal symbols Σ,
3. the set of production rules P of the form

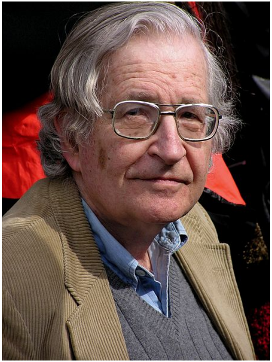
$$\alpha \to \beta$$

$$\text{with} \quad \alpha \in V^*NV^*, \beta \in V^*, V = N \cup \Sigma$$

4. the distinguished start symbol $S \in N$.

Noam Chomsky (*1928)

- Linguist, philosopher, logician, ...
- BA, MA, PhD (1955) at the Univeristy of Pennsylvania
- Mainly teaching at MIT (since 1955)
  - Also Harvard, Columbia University, Institute of Advanced Studie (Princeton), UC Berkely, ...
- Opposition to Vietnam War, Essay *The Responsibility of Intellectuals*
- Most cited academic (1980-1992)
- “World’s top public intellectual” (2005)
- More than 40 honorary degrees



Grammar for C identifiers

Example (C identifiers)

$G = (N, \Sigma, P, S)$ describes C identifiers:

- alpha-numeric words
- which must not start with a digit
- and may contain an underscore (_)

$$
\begin{aligned}
N &= \{S, R, L, D\} \text{ (start, rest, letter, digit)},\\
\Sigma &= \{\mathtt{a}, \ldots, \mathtt{z}, \mathtt{A}, \ldots, \mathtt{Z}, \mathtt{0}, \ldots, \mathtt{9}, \_\},\\
P &= \{ \quad \begin{array}{rcl}
S & \rightarrow & LR \mid \_R\\
R & \rightarrow & LR \mid DR \mid \_R \mid \varepsilon\\
L & \rightarrow & \mathtt{a} \mid \ldots \mid \mathtt{z} \mid \mathtt{A} \mid \ldots \mid \mathtt{Z}\\
D & \rightarrow & \mathtt{0} \mid \ldots \mid \mathtt{9}\}
\end{array}
\end{aligned}
$$

$\alpha \rightarrow \beta_1 | \ldots | \beta_n$ is an abbreviation for $\alpha \rightarrow \beta_1, \ldots, \alpha \rightarrow \beta_n$.

Formal grammars: derivation, language

Definition (Derivation, Language of a Grammar)

For a grammar $G = (N, \Sigma, P, S)$ and words $x, y \in (\Sigma \cup N)^*$, we say that

G derives y from x in one step $(x \Rightarrow_G y)$ iff

$$\exists u, v, p, q \in V^* : (x = upv) \wedge (p \to q \in P) \wedge (y = uqv)$$

Moreover, we say that

G derives y from x $(x \Rightarrow_G^* y)$ iff

$$\exists w_0, \ldots, w_n$$

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

The language of G is $L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$

Example (G_3)

Let $G_3 = (N, \Sigma, P, S)$ with

- $N = \{S\}$,
- $\Sigma = \{\mathtt{a}\}$,
- $P = \{S \to \mathtt{a}S, \quad S \to \varepsilon\}$.

Derivations of G_3 have the general form

$$S \Rightarrow \mathtt{a}S \Rightarrow \mathtt{aa}S \Rightarrow \cdots \Rightarrow \mathtt{a}^n S \Rightarrow \mathtt{a}^n$$

The language produced by G_3 is

$$L(G_3) = \{\mathtt{a}^n \mid n \in \mathbb{N}\}.$$

Grammars and derivations (cont')

Example (G_2)

Let $G_2 = (N, \Sigma, P, S)$ with

- $N = \{S\}$,
- $\Sigma = \{\mathtt{a}, \mathtt{b}\}$,
- $P = \{S \to \mathtt{a}S\mathtt{b}, \quad S \to \varepsilon\}$

Derivations of G_2:

$$S \Rightarrow \mathtt{a}S\mathtt{b} \Rightarrow \mathtt{aa}S\mathtt{bb} \Rightarrow \cdots \Rightarrow \mathtt{a}^n S \mathtt{b}^n \Rightarrow \mathtt{a}^n\mathtt{b}^n.$$

$$L(G_2) = \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}.$$

Reminder: $L(G_2)$ is not regular!

Grammars and derivations (cont')

Example (G_0)

Let $G_1 = (N, \Sigma, P, S)$ with

- $N = \{S, B, C\}$,
- $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$,
- P:

$$
\begin{array}{rcll}
S & \rightarrow & \mathtt{a}SBC & 1 \\
S & \rightarrow & \mathtt{a}BC & 2 \\
CB & \rightarrow & BC & 3 \\
\mathtt{a}B & \rightarrow & \mathtt{ab} & 4 \\
\mathtt{b}B & \rightarrow & \mathtt{bb} & 5 \\
\mathtt{b}C & \rightarrow & \mathtt{bc} & 6 \\
\mathtt{c}C & \rightarrow & \mathtt{cc} & 7
\end{array}
$$

Grammars and derivations (cont.)

Derivations of G_1:

$$S \Rightarrow_1 \mathtt{a}SBC \Rightarrow_1 \mathtt{aa}SBCBC \Rightarrow_1 \cdots \Rightarrow_1 \mathtt{a}^{n-1}S(BC)^{n-1} \Rightarrow_2 \mathtt{a}^n(BC)^n$$

$$\Rightarrow_3^* \mathtt{a}^n B^n C^n \Rightarrow_{4,5}^* \mathtt{a}^n \mathtt{b}^n C^n \Rightarrow_{6,7}^* \mathtt{a}^n \mathtt{b}^n \mathtt{c}^n$$

$$L(G_1) = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n | n \in \mathbb{N}; n > 0\}.$$

- 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 (1956).

Outline

The Chomsky hierarchy (0)

Definition (Grammar of type 0)

Every Chomsky grammar $G = (N, \Sigma, P, S)$ is of Type 0 or unrestricted.

The Chomsky hierarchy (1)

Definition (context-sensitive grammar)

A Chomsky grammar $G = (N, \Sigma, P, S)$ is of is Type 1 (context-sensitive) if all productions are of the form

$$\alpha \to \beta \quad \text{with} \quad |\alpha| \leq |\beta|$$

Exception: the rule $S \to \varepsilon$ is allowed if S does not appear on the right-hand side of any rule

- Rules never derive shorter words
  - except for the empty word in the first step

Context-sensitive vs. monotonic grammars

- The grammars defined previously were originally called monotonic or non-contracting by Chomsky
- Context-sensitive grammars addtionally have to satisfy:

$$\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2 \text{ with } A \in N; \alpha_1, \alpha_2 \in V^*, \beta \in VV^*$$

  - rule application can depend on a context α_1, α_2
  - context cannot be modified (or moved)
  - only one NTS can be modified
- every monotonic grammar can be rewritten as context-sensitive
  - $AB \to BA$ is not context-sensitive, but $AB \to AY \to XY \to XA \to BA$
  - if terminal symbols are involved: replace $S \to aB \to ba$ with $S \to N_aB \to \ldots N_bN_a \to bN_a \to ba$
- since context is irrelevant for the language class, we drop the context requirement for this lecture
- since the term “context-sensitive” is generally used in the literature, we stick with this term

The Chomsky hierarchy (2)

Definition (context-free grammar)

A Chomsky grammar $G = (N, \Sigma, P, S)$ is of is Type 2 (context-free) if all productions are of the form

$$A \to \beta \text{ with } A \in N; \beta \in V^*$$

- Only single non-terminals are replaced
  - independent of their context
- Contracting rules are allowed!
  - context-free grammars are not a subset of context-sensitive grammars
  - but: context-free languages are a subset of context-sensitive languages
  - reason: contracting rules can be removed from context-free grammars, but not from context-sensitive ones

The Chomsky hierarchy (3)

Definition (right-linear grammar)

A Chomsky grammar $G = (N, \Sigma, P, S)$ is of Type 3 (right-linear or regular) if all productions are of the form

$$A \to aB$$

$$\text{with } A \in N; B \in N \cup \{\varepsilon\}; a \in \Sigma \cup \{\varepsilon\}$$

- only one NTS on the left
- on the right: one TS, one NTS, both, or neither
- analogy with automata is obvious

Definition (language classes)

A language is called

recursively enumerable, context-sensitive, context-free, or regular,

if it can be generated by a

unrestricted, context-sensitive, context-free, or regular

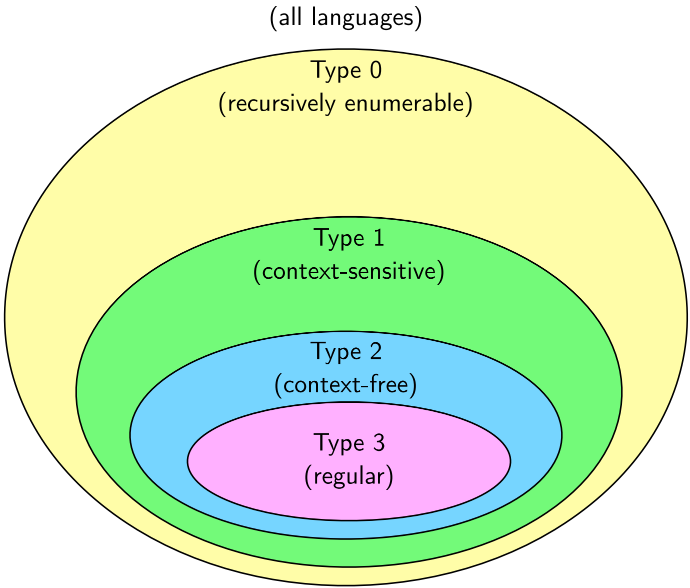
grammar, respectively.

Formal grammars vs. formal languages vs. machines

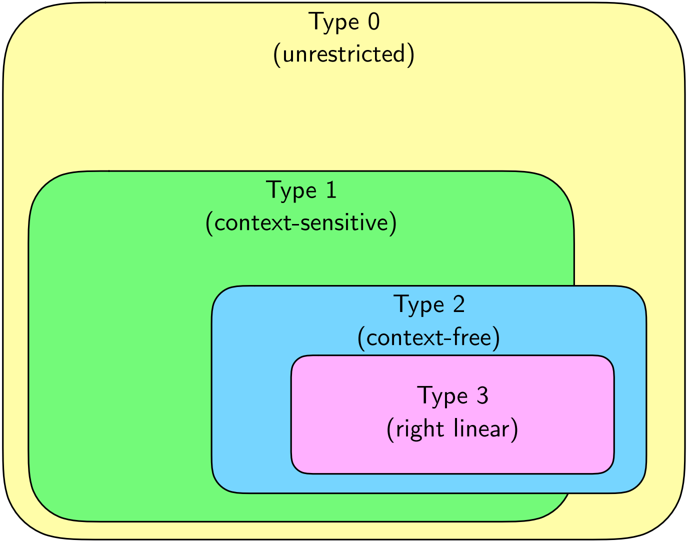
For each grammar/language type, there is a corresponding type of machine model:

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

The Chomsky Hierarchy for Languages



The Chomsky Hierarchy for Grammars



The Chomsky hierarchy: examples

Example (C identifiers revisited)

$$\begin{array}{rcl} S & \rightarrow & LR|\_R \\ R & \rightarrow & LR|DR|\_R|\varepsilon \\ L & \rightarrow & \mathtt{a}|\ldots|\mathtt{z}|\mathtt{A}|\ldots|\mathtt{Z} \\ D & \rightarrow & \mathtt{0}|\ldots|\mathtt{9} \end{array}$$

This grammar is context-free but not regular.
An equivalent regular grammar:

$$\begin{array}{rcl} S & \rightarrow & \mathtt{A}R|\cdots|\mathtt{Z}R|\mathtt{a}R|\cdots|\mathtt{z}R|\_R \\ R & \rightarrow & \mathtt{A}R|\cdots|\mathtt{Z}R|\mathtt{a}R|\cdots|\mathtt{z}R|\mathtt{0}R|\cdots|\mathtt{9}R|\_R|\varepsilon \end{array}$$

The Chomsky hierarchy: examples revisited

Returning to the three derivation examples:

- G_3 with $P = \{S \to aS, S \to \varepsilon\}$
  - G_3 is regular.
  - So is the produced language $L_3 = \{\mathtt{a}^n \mid n \in \mathbb{N}\}$.
- G_2 with $P = \{S \to \mathtt{a}S\mathtt{b}, S \to \varepsilon\}$
  - G_2 is context-free.
  - So is the produced language $L_2 = \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$.
- G_1 with $P = \{S \to \mathtt{a}SBC, S \to \mathtt{a}BC, CB \to BC, \ldots\}$
  - G_0 is context-sensitive.
  - So is the produced language $L_1 = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \in \mathbb{N}; n > 0\}$.

The Chomsky hierarchy: exercises

Let $G = (N, \Sigma, P, S)$ with

- $N = \{S, A, B\}$,
- $\Sigma = \{\mathtt{a}\}$,
- P :

$$
\begin{array}{rcll}
S & \rightarrow & \varepsilon & 1 \\
S & \rightarrow & ABA & 2 \\
AB & \rightarrow & \mathtt{aa} & 3 \\
\mathtt{a}A & \rightarrow & \mathtt{aaa}A & 4 \\
A & \rightarrow & \mathtt{a} & 5
\end{array}
$$

a) What is G's highest type?

b) Show how G derives the word $\mathtt{aaaaa}$.

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

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

The Chomsky hierarchy: exercises (cont.)

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.

The Chomsky hierarchy: exercises (cont.)

Let $G = (N, \Sigma, P, S)$ with

- $N = \{S, A, B\}$,
- $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{t}\}$,
- P :

$S \to \mathtt{a}AS$	1	$A\mathtt{a} \to \mathtt{a}A$	6
$S \to \mathtt{b}BS$	2	$A\mathtt{b} \to \mathtt{b}A$	7
$S \to \mathtt{t}$	3	$B\mathtt{a} \to \mathtt{a}B$	8
$A\mathtt{t} \to \mathtt{ta}$	4	$B\mathtt{b} \to \mathtt{b}B$	9
$B\mathtt{t} \to \mathtt{tb}$	5		

a) What is G's highest type?

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

Outline

Regular languages and right-linear grammars

Theorem (right-linear grammars and regular languages)

The class of regular languages (generated by regular expressions, accepted by finite automata) is exactly the class of languages generated by right-linear grammars.

Proof.

- Convert DFA to right-linear grammar
- Convert right-linear grammar to NFA

□

DFA $\rightsquigarrow$ right-linear grammar

Algorithm for transforming a DFA

$$\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$$

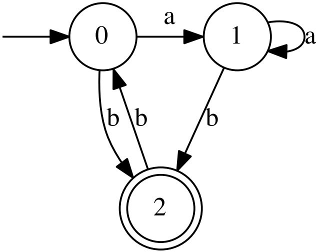
into a grammar

$$G = (N, \Sigma, P, S)$$

- $N = Q$
- $S = q_0$
- $P = \{p \to aq \mid (p, a, q) \in \delta\} \quad \cup \quad \{p \to \varepsilon \mid p \in F\}$

Regular grammars and FAs: exercise

Consider the following DFA $\mathcal{A}$:



a) Give a formal definition of $\mathcal{A}$

b) Generate a right-linear grammar G with $L(G) = L(\mathcal{A})$

Right-linear grammar $\rightsquigarrow$ NFA

Algorithm for transforming a grammar

$$G = (N, \Sigma, P, S)$$

into an NFA

$$\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$$

- $Q = N \cup \{q_f\} \quad (q_f \notin N)$
- $q_0 = S$
- $F = \{q_f\}$
- $\Delta = \{(A, c, B) \mid A \to cB \in P\} \quad \cup$
  $\{(A, c, q_f) \mid A \to c \in P\} \quad \cup$
  $\{(A, \varepsilon, B) \mid A \to B \in P\} \quad \cup$
  $\{(A, \varepsilon, q_f) \mid A \to \varepsilon \in P\}$

End lecture 9

Outline

Context-free grammars

- Reminder: $G = (N, \Sigma, P, S)$ is context-free if all rules are of the form $A \to \beta$ with $A \in N$.
- Context-free languages/grammars are highly relevant
  - Core of most programming languages
  - XML
  - Algebraic expressions
  - Many aspects of human language

Grammars: equivalence and normal forms

Definition (equivalence)

Two grammars are called equivalent if they generate the same language.

We will now compute grammars that are equivalent to some given context-free grammar G but have "nicer" properties

- Reduced grammars contain no unproductive symbols
- Grammars in Chomsky normal form support efficient decision of the word problem

Reduced grammars

Definition (reduced)

Let $G = (N, \Sigma, P, S)$ be a context-free grammar.

- $A \in N$ is called terminating if $A \Rightarrow_G^* w$ for some $w \in \Sigma^*$.
- $A \in N$ is called reachable if $S \Rightarrow_G^* uAv$ for some $u, v \in V^*$.
- G is called reduced if N contains only reachable and terminating symbols.

Terminating and reachable symbols

The terminating symbols can be computed as follows:

1. $T := \{A \in N \mid \exists w \in \Sigma^* : A \to w \in P\}$
2. add all symbols M to T with a rule $M \to D$ with $D \in (\Sigma \cup T)^*$
3. repeat step 2 until no further symbols can be added

Now T contains exactly the terminating symbols.

The reachable symbols can be computed as follows:

1. $R := \{S\}$
2. for every $A \in R$, add all symbols M with a rule $A \to V^* M V^*$
3. repeat step 2 until no further symbols can be added

Now R contains exactly the reachable symbols.

Reducing context-free grammars

Theorem (reduction of context-free grammars)

Every context-free grammar G can be transformed into an equivalent reduced context-free grammar G_r.

Proof.

1. generate the grammar G_T by removing all non-terminating symbols (and rules containing them) from G
2. generate the grammar G_r by removing all unreachable symbols (and rules containing them) from G_T

□

Sequence is important: symbols can become unreachable through removal of non-terminating symbols.

Reachable and terminating symbols

Example

Let $G = (N, \Sigma, P, S)$ with

- $N = \{S, A, B, C, T\}$,
- $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$,
- P :

$$\begin{array}{lcl} S & \rightarrow & T|B|C \\ T & \rightarrow & AB \\ A & \rightarrow & a \\ B & \rightarrow & bB \\ C & \rightarrow & c \end{array}$$

- terminating symbols in G: $C, A, S \quad \rightsquigarrow \quad G_T$
- reachable symbols in G_T: $S, C \quad \rightsquigarrow \quad G_r$
- note: A is still reachable in G!

Exercise: reducing grammars

Compute the reduced grammar $G = (N, \Sigma, P, S)$ for the following grammar $G' = (N', \Sigma, P', S)$:

1. $N' = \{S, A, B, C, D\}$,
2. $\Sigma = \{\mathtt{a}, \mathtt{b}\}$,
3. P' :

$$
\begin{array}{lll}
S & \rightarrow & A|aS|B \\
A & \rightarrow & a \\
A & \rightarrow & AS \\
A & \rightarrow & Ba
\end{array}
\qquad\qquad
\begin{array}{lll}
B & \rightarrow & Ba \\
C & \rightarrow & Da \\
D & \rightarrow & Cb \\
D & \rightarrow & a
\end{array}
$$

Chomsky normal form

Reduced grammars can be further modidified to allow for an efficient decision procedure for the word problem.

Definition (CNF)

A context-free grammar (N, Σ, P, S) is in Chomsky normal form if all rules are of the kind

- $N \to a$ with $a \in \Sigma$
- $N \to AB$ with $A, B \in N$
- $S \to \varepsilon$, if S does not appear on the right-hand side of any rule

Transformation into CNF:

1. remove ε-productions
2. remove chain rules ($A \to B$)
3. introduce auxiliary symbols

Removal of ε-productions

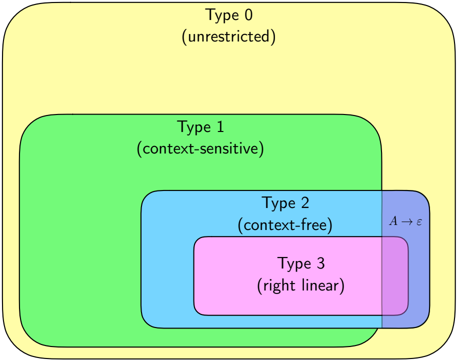
Theorem (ε-free grammar)

Every context-free grammar can be transformed into an equivalent cf. grammar that does not contain rules of the kind $A \to \varepsilon$ (except $S \to \varepsilon$ if S does not appear on the rhs).

Procedure:

1. let $E = \{A \in N \mid A \to \varepsilon \in P\}$
2. add all symbols B to E for which there is a rule $B \to \beta$ with $\beta \in E^*$
3. repeat step 2 until no further symbols can be added
4. for every rule $C \to \beta_1 \boldsymbol{B} \beta_2$ with $B \in E$
   - add a rule $C \to \beta_1\beta_2$ to P
5. remove all rules $A \to \varepsilon$ from P
6. if $S \in E$
   - use a new start symbol S_0
   - add rules $S_0 \to \varepsilon | S$

Interlude: Chomsky-Hierarchy for Grammars (again)



- For languages, Type-0, Type-1, Type-2, Type-3 form a real inclusion hierarchy
- Not quite true for grammars:
  - $A \to \varepsilon$ allowed in context-free/regular grammars, not in context-free languages
- Eliminating ε-productions removes this discrepancy!

Removal of chain rules

Theorem (chain rules)

Every context-free grammar can be transformed into an equivalent cf. grammar that does not contain rules of the kind $A \to B$.

Procedure:

1. for every $A \in N$, compute the set $N(A) = \{B \in N \mid A \Rightarrow_G^* B\}$ (this can be done iteratively, as shown previously)
2. remove $A \to C$ for any $C \in N$ from P
3. add the following production rules to P $\{A \to w \mid w \notin N \text{ and } B \to w \in P \text{ and } B \in N(A)\}$

Example

$A \to a|B; \quad B \to bb|C; \quad C \to ccc$

is equivalent to

$A \to a|bb|ccc; B \to bb|ccc; C \to ccc$

Chomsky normal form

Reminder: Chomsky normal form
A context-free grammar (N, Σ, P, S) is in CNF if all rules are of the kind

- $N \to a$ with $a \in \Sigma$
- $N \to AB$ with $A, B \in N$
- $S \to \varepsilon$, if S does not appear on the right-hand side of any rule

Theorem (transformation into Chomsky normal form)

Every context free grammar can be transformed into an equivalent cf. grammar in Chomsky normal form.

Algorithm for computing Chomsky normal form

1. remove ε rules
2. remove chain rules
3. compute reduced grammar
    1. remove non-terminating symbols
    2. remove unreachable symbols
4. for all rules $A \to w$ with $w \notin \Sigma$:
    - replace all occurrences of a with X_a for all $a \in \Sigma$
    - add rules $X_a \to a$
5. replace rules $A \to B_1 B_2 \ldots B_n$ for $n > 2$ with rules

$$\begin{aligned} A &\to B_1 C_1 \\ C_1 &\to B_2 C_2 \\ &\vdots \\ C_{n-2} &\to B_{n-1} B_n \end{aligned}$$

with new symbols C_i.

Exercise: tranformation into CNF

Compute the Chomsky normal form of the following grammar:

$$G = (N, \Sigma, P, S)$$

- $N = \{S, A, B, C, D, E\}$
- $\Sigma = \{\mathtt{a}, \mathtt{b}\}$
- P :

$$\begin{array}{lcl} S & \to & AB|SB|BDE \\ A & \to & Aa \\ B & \to & bB|BaB|ab \end{array} \qquad \begin{array}{lcl} C & \to & SB \\ D & \to & E \\ E & \to & \varepsilon \end{array}$$

Solution

Chomsky NF: purpose

Why transform G into Chomsky NF?

- in a context-free grammar, derivations can have arbitrary length
  - if there are contracting rules, a derivation of w can contain words longer than w
  - if there are chain rules ($C \to B; B \to C$), a derivation of w can contain arbitrarily many steps
- word problem is difficult to decide
- if G is in CNF, for a word of length n, a derivation has $2n - 1$ steps:
  - $n - 1$ rule applications $A \to BC$
  - n rule applications $A \to a$
- word problem can be decided by checking all derivations of length $2n - 1$
- That's still plenty of derivations!

More efficient algorithm: Cocke-Younger-Kasami (CYK)

CYK algorithm: idea

Decide the word problem for a context-free grammar G in Chomsky NF and a word w.

- find out which NTS are needed in the end to produce the TS for w (using production rules $A \to a$).
- iteratively find all NTS that can generate the required sequence of NTS (using production rules $A \to BC$).
- if S can produce the required sequence, $w \in L(G)$ holds.

Mechanism:

- operates on a table.
- field in row i and column j contains all NTS that can generate words from character i through j.

Example of dynamic programming!

CYK algorithm: example

$$
\begin{array}{rcl}
S & \rightarrow & a \\
B & \rightarrow & b \\
B & \rightarrow & c \\
S & \rightarrow & SA \\
A & \rightarrow & BS \\
B & \rightarrow & BB \\
B & \rightarrow & BS
\end{array}
$$

$i \setminus j$	1	2	3	4	5	6
1	S	$\emptyset$	S	$\emptyset$	$\emptyset$	S
2		B	A,B	B	B	A,B
3			S	$\emptyset$	$\emptyset$	S
4				B	B	A,B
5					B	A,B
6						S
$w =$	a	b	a	c	b	a

$w = abacba$

CYK: formal algorithm

for $i := 1$ to n **do**
  $N_{ii} := \{A \mid A \rightarrow a_i \in P\}$
for $d := 1$ to $n - 1$ **do**
  for $i := 1$ to $n - d$ **do**
    $j := i + d$
    $N_{ij} := \emptyset$
    for $k := i$ to $j - 1$ **do**
      $N_{ij} := N_{ij} \cup \{A \mid A \rightarrow BC \in P; B \in N_{ik}; C \in N_{(k+1)j}\}$

CYK algorithm: exercise

Consider the grammar $G = (N, \Sigma, P, S)$ from the previous exercise

- $N = \{S, A, B, D, X, Y\}$
- $\Sigma = \{\mathtt{a}, \mathtt{b}\}$

$$
P: \quad \begin{array}{lcl}
S & \to & SB|BD|YB|XY \\
B & \to & BD|YB|XY \\
D & \to & XB \\
X & \to & a \\
Y & \to & b
\end{array}
$$

Use the CYK algorithm to determine if the following words can be generated by G:

a) $w_1 = babaab$

b) $w_2 = abba$

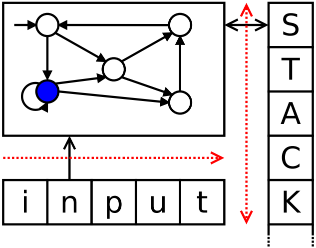
End lecture 10

Outline

Pushdown automata: motivation

- DFAs/NFAs are weaker than context-free grammars
- to accept languages like $a^n b^n$, an unlimited storage component is needed
- Pushdown automata have an unlimited stack
  - LIFO: last in, first out
  - only top symbol can be read
  - arbitrary amount of symbols can be added to the top

PDA: conceptual model



- extends FA by unlimited stack:
  - transitions can read and write stack
  - only a the top
  - stack alphabet Γ
  - LIFO: last in, first out
- acceptance condition
  - empty stack after reading input
  - no final states needed
- commonalities with FA:
  - read input from left to right
  - set of states, input alphabet
  - initial state

PDA transitions

$$\Delta \subseteq Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \times \Gamma^* \times Q$$

- PDA is in a state
- can read next input character or nothing
- must read (and remove) top stack symbol
- can write arbitrary amout of symbols on top of stack
- goes into a new state

A transition (p, c, A, BC, q) can be written as follows:

$$p \quad c \quad A \quad \rightarrow \quad BC \quad q$$

Pushdown automata: definition

Definition (pushdown automaton)

A pushdown automaton (PDA) is a 6-tuple $(Q, \Sigma, \Gamma, \Delta, q_0, Z_0)$ where

- Q, Σ, q_0 are defined as for NFAs.
- Γ is the stack alphabet
- Z_0 is the initial stack symbol
- $\Delta \subseteq Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \times \Gamma^* \times Q$ is the transition relation

A configuration of a PDA is a triple (q, w, γ) where

- q is the current state
- w is the input yet unread
- γ is the current stack content

A PDA $\mathcal{A}$ accepts a word $w \in \Sigma^*$ if, starting from the configuration (q_0, w, Z_0), $\mathcal{A}$ can reach the configuration $(q, \varepsilon, \varepsilon)$ for some q.

PDAs: important properties

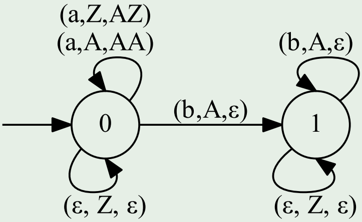
- PDAs defined above are non-deterministic
  - deterministic PDAs are weaker
- ε transitions are possible
- it is possible to define acceptance condition using final states
  - makes representation of PDAs more complex
  - makes proofs more difficult

Example: PDA for $a^n b^n$

Example (Automaton $\mathcal{A}$)

$\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, 0, Z)$

- $Q = \{0, 1\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \{A, Z\}$
- Δ :



0	ε	Z	$\rightarrow$	ε	0	accept empty word
0	a	Z	$\rightarrow$	AZ	0	read first a, store A
0	a	A	$\rightarrow$	AA	0	read further a, store A
0	b	A	$\rightarrow$	ε	1	read first b, delete A
1	b	A	$\rightarrow$	ε	1	read further b, delete A
1	ε	Z	$\rightarrow$	ε	1	accept if all As have been deleted

PDA: example (2)

0	ε	Z	$\rightarrow$	ε	0
0	a	Z	$\rightarrow$	AZ	0
0	a	A	$\rightarrow$	AA	0
0	b	A	$\rightarrow$	ε	1
1	b	A	$\rightarrow$	ε	1
1	ε	Z	$\rightarrow$	ε	1

Process $aabb$:

1. $(0, aabb, Z)$
2. $(0, abb, AZ)$
3. $(0, bb, AAZ)$
4. $(1, b, AZ)$
5. $(1, \varepsilon, Z)$
6. $(1, \varepsilon, \varepsilon)$

Process abb:

1. $(0, abb, Z)$
2. $(0, bb, AZ)$
3. $(1, b, Z)$
4. No rule applicable

PDA: exercise

Define a PDA detecting all palindromes over $\{a, b\}$, i.e. all words

$$\{w \cdot \overleftarrow{w} \mid w \in \{a, b\}\}$$

where

$$\overleftarrow{w} = a_n \dots a_1 \text{ if } w = a_1 \dots a_n$$

Can you define a deterministic automaton?

Equivalence of PDAs and Context-Free Grammars

Theorem

The class of languages that can be accepted by a PDA is exactly the class of languages that can be produced by a context-free grammar.

Proof.

- For a cf. grammar G, generate a PDA $\mathcal{A}_G$ with $L(\mathcal{A}_G) = L(G)$.
- For a PDA $\mathcal{A}$, generate a cf. grammar $G_\mathcal{A}$ with $L(G_\mathcal{A}) = L(\mathcal{A})$.

$\square$

From context-free grammars to PDAs

For a grammar $G = (N, \Sigma, P, S)$, an equivalent PDA is:

$$\mathcal{A}_G = (\{q\}, \Sigma, \Sigma \cup N, \Delta, q, S)$$

$$\begin{array}{rcll} \Delta & = & \{(q, \varepsilon, A, \gamma, q) \mid A \to \gamma \in P\} & \cup \\ & & \{(q, a, a, \varepsilon, q) \mid a \in \Sigma\} & \end{array}$$

$\mathcal{A}_G$ simulates the productions of G in the following way:

- a production rule is applied to the top stack symbol if it is an NTS
- a TS is removed from the stack if it corresponds to the next input character

Note:

- $\mathcal{A}_G$ is nondeterministic if there are several rules for one NTS.
- $\mathcal{A}_G$ only has one single state.
  - Corollary: PDAs need no states, could be written as $(\Sigma, \Gamma, \Delta, Z_0)$.

From context-free grammars to PDAs: exercise

For the grammar $G = (\{S\}, \{a, b\}, P, S)$ with

$$
\begin{aligned}
P = \{S &\rightarrow aSa \\
S &\rightarrow bSb \\
S &\rightarrow \varepsilon\}
\end{aligned}
$$

- create an equivalent PDA $\mathcal{A}_G$,
- show how $\mathcal{A}_G$ processes the input $abba$.

From PDAs to context-free grammars

Transforming a PDA $\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, q_0, Z_0)$ into a grammar $G_{\mathcal{A}} = (N, \Sigma, P, S)$ is more involved:

- N contains symbols $[pZq]$, meaning
  - $\mathcal{A}$ must go from p to q deleting Z from the stack
- for a transition $(p, a, Z, \varepsilon, q)$ that deletes a stack symbol:
  - $\mathcal{A}$ can switch from p to q and delete Z by reading input a
  - this can be expressed by a production rule $[pZq] \to a$.
- for transitions (p, a, Z, ABC, q) that produce stack symbols:
  - test all possible transitions for removing these symbols
  - $[p, Z, t] \to a[qAr][rBs][sCt]$ for all states r, s, t
  - intuitive meaning: in order to go from p to t and delete Z, you can
    1. read the input a
    2. go into state q
    3. find states r, s through which you can go from q to t and delete A, B, and C from the stack.

$G_{\mathcal{A}}$: formal definition

For $\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, q_0, Z_0)$ we define $G_{\mathcal{A}} = (N, \Sigma, P, S)$ as follows

- $N = \{S\} \cup \{[p, Z, q] \mid p, q \in Q, Z \in \Gamma\}$
- P contains the following rules:
  - for every $q \in Q$, P contains $\{S \to [q_0, Z_0, q]\}$
    meaning: $\mathcal{A}$ has to go from q_0 to any state q, deleting Z_0.
  - for each transition $(p, a, Z, Y_1 Y_2 \dots Y_n, q)$ with
    - $a \in \Sigma \cup \{\varepsilon\}$ and
    - $Z, Y_1, Y_2 \dots Y_n \in \Gamma$,

    P contains rules

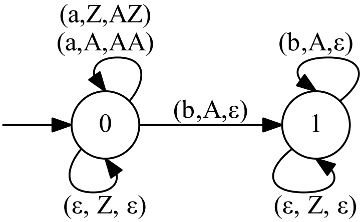
    $$[p, Z, q_n] \to a[qY_1q_1][q_1Y_2q_2] \dots [q_{n-1}Y_nq_n]$$

    for all possible combinations of states $q_1, q_2, \dots q_n \in Q$.

Exercise: transformation of PDA into grammar

$\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, 0, Z)$

- $Q = \{0, 1\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \{A, Z\}$
- Δ :



0	ε	Z	$\to$	ε	0
0	a	Z	$\to$	AZ	0
0	a	A	$\to$	AA	0
0	b	A	$\to$	ε	1
1	b	A	$\to$	ε	1
1	ε	Z	$\to$	ε	1

- Transform $\mathcal{A}$ into a grammar $G_{\mathcal{A}}$ (and reduce $G_{\mathcal{A}}$).
- Show how $\mathcal{A}_G$ produces the words ε, ab, and $aabb$.

Outline

Closure properties

Theorem (Closure under $\cup, \cdot, ^*$)

The class of context-free languages is closed under union, concatenation, and Kleene star.

For context-free grammars

$$G_1 = (N_1, \Sigma, P_1, S_1) \quad \text{and} \quad G_2 = (N_2, \Sigma, P_2, S_2)$$

with $N_1 \cap N_2 = \emptyset$ (rename NTSs if needed), let S be a new start symbol.

- for $L(G_1) \cup L(G_2)$, add productions $S \to S_1, S \to S_2$.
- for $L(G_1) \cdot L(G_2)$, add production $S \to S_1 S_2$.
- for $L(G_1)^*$, add productions $S \to \varepsilon, S \to T, T \to S_1 T, T \to S_1$.

Proving that a language is not context-free

Pumping-Lemma for cf. languages, similar to the PL for regular languages

- Commonalities:
  - If a grammar produces words of arbitrary length, there must be a repeated NTS.
  - This NTS produces itself (and possibly other symbols).
  - This cycle can be repeated arbitrarily often.
- Difference:
  - instead of pumping one part of the word, two are pumped in parallel.

The Lemma

Theorem (Pumping-Lemma for context-free languages)

Let L be a context-free language, generated by a context-free grammar $G_L = (N, \Sigma, P, S)$ without contracting rules or chain rules. Let $m = |N|$, r be the maximum length of the rhs of a rule in P, and $k = r^{m+1}$.
Then for every $s \in L$ with $|s| > k$ there exists a segmentation $u \cdot v \cdot w \cdot x \cdot y = s$ such that

1. $vx \neq \varepsilon$
2. $|vwx| \leq k$
3. $u \cdot v^h \cdot w \cdot x^h \cdot y \in L$ *for every* $h \in \mathbb{N}$.

- Cannot be applied to $\{a^n b^n\}$, but to $\{a^n b^n c^n\}$.
- $\{a^n b^n c^n\}$ is not context-free, but context-sensitive, as we have seen before.

Closure properties (cont.)

Theorem (Closure under $\cap$)

Context-free languages are not closed under intersection.

Otherwise, $\{a^n b^n c^n\}$ would be context-free:

- $\{a^n b^n c^m\}$ is context-free
- $\{a^m b^n c^n\}$ is context-free
- $\{a^n b^n c^n\} = \{a^n b^n c^m\} \cap \{a^m b^n c^n\}$

Exercise: closure properties

1. Define context-free grammars for $L_1 = \{a^n b^n c^m \mid n, m \geq 0\}$ and $L_2 = \{a^m b^n c^n \mid n, m \geq 0\}$.
2. Use the known closure properties to show that context-free languages are not closed under complement.

Decision problems: word problem

Theorem (Word problem for cf. languages)

For a word w and a context-free grammar G, it is decidable whether $w \in L(G)$ holds.

Proof.

The CYK algorithm decides the word problem. $\square$

Decision problems: emptiness problem

Theorem (Emptiness problem for cf. languages)

For a context-free grammar G, it is decidable if $L(G) = \emptyset$ holds.

Proof.

Let $G = (N, \Sigma, P, S)$.
Iteratively compute productive NTSs, i.e. symbols that produce terminal words as follows:

1. let $Z = \Sigma$
2. add all symbols A to Z for which there is a rule $A \to \beta$ with $\beta \in Z^*$
3. repeat step 2 until no further symbols can be added
4. $L(G) = \emptyset$ iff $S \notin Z$.

□

Decision problems: equivalence problem

Theorem (Equivalence problem for cf. languages)

For context-free grammars G_1, G_2, it is undecidable if $L(G_1) = L(G_2)$ holds.

This follows from undecidability of Post's Correspondence Problem.

Summary: context-free languages

- characterised by
  - context-free grammars
  - pushdown automata
- closure properties
  - closed under $\cup, ^*, \cdot$
  - not closed under $\cap, \overline{}$
- decision problems
  - decidable: $w \in L(G)$, $L(G) = \emptyset$ (Chomsky NF, CYK algorithm)
  - undecidable: $L(G_1) = L(G_2)$
- can describe nested dependencies
  - structure of programming languages
  - natural language processing
- in compilers, these features are used by parsers (next chapter)

Outline

Outline

Turing machines

Turing machine: Motivation

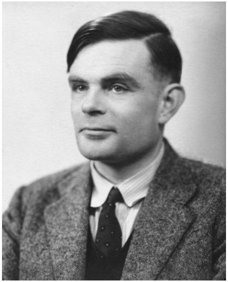
Four classes of languages described by grammars and equivalent machine models:

1. regular languages $\rightsquigarrow$ finite automata
2. context-free languages $\rightsquigarrow$ pushdown automata
3. context-sensitive languages $\rightsquigarrow$?
4. Type-0-languages $\rightsquigarrow$?

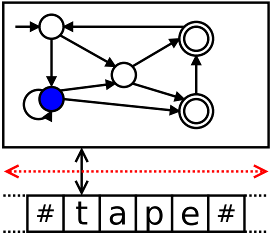
We need a machine model that is more powerful than PDAs:
Turing machines

Turing machine: history

- proposed in 1936 by Alan Turing
  - paper: *On computable numbers, with an application to the Entscheidungsproblem*
  - uses the TM to show that satisfiability of first-order formulas is undecidable
- model of a universal computer
  - very simple (and thus easy to describe formally)
  - but as powerful as any conceivable machine



Turing machine: conceptual model



- medium: unlimited tape (bidirectional)
  - initially contains input (and blanks #)
  - TM can read and write tape
  - TM can move arbitrarily over tape
  - serves for input, working, output
  - output possible
- transition relation
  - read and write current position
  - moving instructions (l, r, n)
- acceptance condition
  - final state is reached
  - no transitions possible
- commonalities with FA
  - control unit (finite set of states),
  - initial and final states
  - input alphabet

Transitions in Turing machines

$$\Delta \subseteq Q \times \Gamma \times \Gamma \times \{l, n, r\} \times Q$$

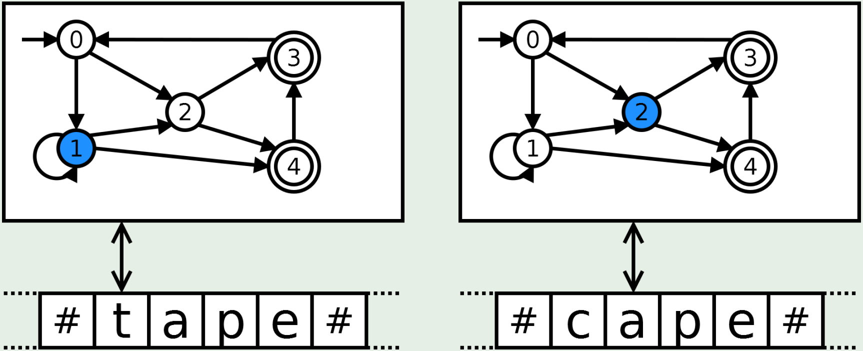
- TM is in state
- reads tape symbol from current position
- writes tape symbol on current position
- moves to left, right, or stays
- goes into a new state

A transition p, a, b, l, q can also be written as

$$p \quad a \quad \rightarrow \quad b \quad l \quad q$$

Example: transition

Example (transition $1, t \to c, r, 2$)



Turing machine: formal definition

Definition (Turing machine)

A Turing machine (TM) is a 6-tuple $(Q, \Sigma, \Gamma, \Delta, q_0, F)$ where

- Q, Σ, q_0, F are defined as for NFAs,
- $\Gamma \supseteq \Sigma \cup \{\#\}$ is the tape alphabet,
  including at least Σ and the blank symbol,
- $\Delta \subseteq Q \times \Gamma \times \Gamma \times \{l, n, r\} \times Q$ is the transition relation.

If Δ contains at most one transition (p, a, b, d, q) for each pair $(p, a) \in Q \times \Sigma$, the TM is called deterministic. The transition function is then denoted by δ.

Configurations of TMs

Definition (configuration)

A configuration $c = \alpha q \beta$ of a Turing machine is given by

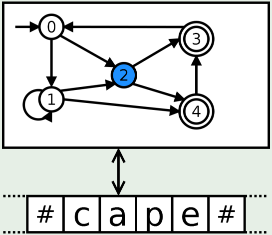
- the current state q
- the tape content α on the left of the read/write head (except unlimited # sequences)
- the tape content β starting with the position of the head (except unlimited # sequences)

A configuration $c = \alpha q \beta$ is accepting if $q \in F$.

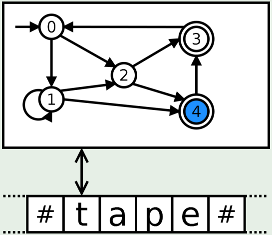
A configuration c is a stop configuration if there are no transitions from c.

Example: configuration

Example (configurations)



- This TM is in the configuration $c2ape$.



- The configuration $4tape$ is accepting.
- If there are no transitions $4, t \rightarrow \ldots$, $4tape$ also is a stop configuration.

Computations of TMs

Definition (computation, acceptance)

A computation of a TM $\mathcal{M}$ on a word w is a sequence of configurations (according to the transition function) of configurations of $\mathcal{M}$, starting from $q_0 w$.

$\mathcal{M}$ accepts w if there exists a computation of $\mathcal{M}$ on w that results in accepting stop configuration.

Exercise: Turing machines

Let $\Sigma = \{a, b\}$ and $L = \{w \in \Sigma^* \mid |w|_a \text{ is even}\}$.

- Give a TM $\mathcal{M}$ that accepts (exactly) the words in L.
- Give the computation of $\mathcal{M}$ on the words $abbab$ and $bbab$.

Example: TM for $a^n b^n c^n$

$\mathcal{M} = (Q, \Sigma, \Gamma, \Delta, \text{start}, \{f\})$ with

- $Q = \{$start, findb, findc, check, back, end, f$\}$
- $\Sigma = \{$a, b, c$\}$ and $\Gamma = \Sigma \cup \{$#, x, y, z$\}$

state	read	write	move	state
start	#	#	n	f
start	a	x	r	findb
findb	a	a	r	findb
findb	y	y	r	findb
findb	b	y	r	findc
findc	b	b	r	findc
findc	z	z	r	findc
findc	c	z	r	check
check	c	c	l	back
check	#	#	l	end

state	read	write	move	state
back	z	z	l	back
back	b	b	l	back
back	y	y	l	back
back	a	a	l	back
back	x	x	r	start
end	z	z	l	end
end	y	y	l	end
end	x	x	l	end
end	#	#	n	f

Exercise: Turing machines (2)

a) Simulate the computations of $\mathcal{M}$ on $aabbcc$ and $aabc$.

b) Develop a Turing machine $\mathcal{P}$ accepting $L_{\mathcal{P}} = \{wcw \mid w \in \{a, b\}^*\}$.

c) How do you have to modify $\mathcal{P}$ if you want to recognise inputs of the form ww?

Turing machines with several tapes

- A k-tape TM has k tapes on which the heads can move independently.
- $\Delta \subseteq Q \times \Gamma^k \times \Gamma^k \times \{r, l, n\}^k \times Q$
- It is possible to simulate a k-tape TM with a (1-tape) TM:
  - use alphabet $\Gamma^k \times \{X, \#\}^k$
  - the first k language elements encode the tape content, the remaining ones the positions of the heads.

Nondeterminism

Reminder

- just like FAs and PDAs, TMs can be deterministic or non-deterministic, depending on the transition relation.
- for non-deterministic TMs, the machine accepts w if there exists a sequence of transitions leading to an accepting stop configuration.

Simulating non-deterministic TMs

Theorem (equivalence of deterministic and non-deterministic TMs)

Deterministic TMs can simulate computations of non-deterministic TMs; i.e. they describe the same class of languages.

Proof.

Use a 3-tape TM:

- tape 1 stores the input w
- tape 2 enumerates all possible sequences of non-deterministic choices (for all non-deterministic transitions)
- tape 3 encodes the computation on w with choices stored on tape 2.

$\square$

Outline

Turing machines and unrestricted grammars

Theorem (equivalence of TMs and unrestricted grammars)

The class of languages that can be accepted by a Turing machine is exactly the class of languages that can be generated by unrestricted Chomsky grammars.

Proof.

1. simulate grammar derivations with a TM
2. simulate a TM computation with a grammar

□

Simulating a Type-0-grammar G with a TM

Use a non-deterministic 2-tape TM:

- tape 1 stores input word w
- tape 2 simulates the derivations of G, starting with S
  - (non-deterministically) choose a position
  - if the word starting at the position, matches α of a rule $\alpha \to \beta$, apply the rule
    - move tape content if necessary
    - replace α with β
  - compare content of tape 2 with tape 1
    - if they are equal, accept
    - otherwise continue

Simulating a TM with a Type-0-grammar

Goal: transform TM $\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, q_0, F)$ into grammar G
Technical difficulty:

- $\mathcal{A}$ receives word as input at the start, possibly modifies it, then possibly accepts.
- G starts with S, applies rules, possibly generating w at the end.

1. generate initial configuration $q_0 w \in \Sigma^*$ with blanks left and right
2. simulate the computation of $\mathcal{A}$ on w

$$
\begin{aligned}
(p, a, b, r, q) &\quad\rightsquigarrow\quad pa \to bq \\
(p, a, b, l, q) &\quad\rightsquigarrow\quad cpa \to qcb \text{ (for all } c \in \Gamma) \\
(p, a, b, n, q) &\quad\rightsquigarrow\quad pa \to qb
\end{aligned}
$$

3. if an accepting stop configuration is reached, recreate w
   - requires a "backup" tape or a more complex alphabet

Outline

Linear bounded automata

- context-sensitive grammars do not allow for contracting rules
- a linear bounded automaton (LBA) is a TM that only uses the space originally occupied by the input w.
- limits of w are indicated by markers that cannot be passed by the read/write head

...	>	i	n	p	u	t	<	...

Equivalence of cs. grammars and LBAs

Transformation of cs. grammar G into LBA:

- as for Type-0-grammar: use 2-tape-TM
  - input on tape 1
  - simulate operations of G on tape 2
- since the productions of G are non-contracting, words longer than w need not be considered

Transformation of LBA $\mathcal{A}$ into cs. grammar:

- similar to construction for TM:
  - generate w without blanks
  - simulate operation of $\mathcal{A}$ on w
    - rules are non-contracting ✓

Closure properties: regular operations

Theorem (closure under $\cup, \cdot, ^*$)

The class of languages described by context-sensitive grammars is closed under $\cup, \cdot, ^*$.

Proof.

Concatenation and Kleene-star are more complex than for cf. grammars because the context can influence rule applicability.

- rename NTSs (as for cf. grammars)
- only allow NTSs as context
- only allow productions of the kind
  - $N_1 N_2 \ldots N_k \to M_1 M_2 \ldots M_j$
  - $N \to a$

□

Closure properties: intersection and complement

Theorem (closure under $\cap$)

The class of context-sensitive languages is closed under intersection.

Proof.

- use a 2-tape-LBA
- simulate computation of $\mathcal{A}_1$ on tape 1, $\mathcal{A}_2$ on tape 2
- accept if both $\mathcal{A}_1$ and $\mathcal{A}_2$ accept $\square$

Theorem (closure under $\overline{}$)

The class of context-sensitive languages is closed under complement.

- shown in 1988

Context-sensitive grammars: decision problems

Theorem (Word problem for cs. languages)

The word problem for cs. languages is decidable.

Proof.

- N, Σ and P are finite
- rules are non-contracting
- for a word of length n only a finite number of derivations up to length n has to be considered.

$\square$

Context-sensitive grammars: decision problems (cont')

Theorem (Emptiness problem for cs. languages)

The emptiness problem for cs. languages is undecidable.

Proof.

Also follows from undecidability of Post's correspondence problem. □

Theorem (Equivalence problem for cs. languages)

The equivalence problem for cs. languages is undecidable.

Proof.

If this problem was decidable for cs. languages, ist would also be decidable for cf. languages (since every cf. language is also cs.). □

Outline

The universal Turing machine $\mathcal{U}$

- $\mathcal{U}$ is a TM that simulates other Turing machines
- since TMs have finite alphabets and state sets, they can be encoded by a (binary) alphabet by an encoding function $c()$
- Input:
  - encoding $c(\mathcal{A})$ of a TM $\mathcal{A}$ on tape 1
  - encoding $c(w)$ of an input word w for $\mathcal{A}$ on tape 2
- with input $c(\mathcal{A})$ and $c(w)$, $\mathcal{U}$ behaves exactly like $\mathcal{A}$ on w:
  - $\mathcal{U}$ accepts iff $\mathcal{A}$ acceptss
  - $\mathcal{U}$ halts iff $\mathcal{A}$ halts
  - $\mathcal{U}$ runs forever if $\mathcal{A}$ runs forever

Every solvable problem can be solved in software.

Operation of $\mathcal{U}$

1. encode initial configuration
   - tape on lhs of head
   - state
   - tape on rhs of head
2. use $c(\mathcal{A})$ to find a transition from the current configuration
3. modify the current configuration accordingly
4. accept if $\mathcal{A}$ accepts
5. stop if $\mathcal{A}$ stops
6. otherwise, continue with step 2

The Halting problem

Definition (halting problem)

For a TM $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$ and a word $w \in \Sigma^*$, does $\mathcal{A}$ halt (i.e. reach a stop configuration) with input w?

Wanted: TMs $\mathcal{H}1$ and $\mathcal{H}2$, such that with input $c(\mathcal{A})$ and $c(w)$

1. $\mathcal{H}1$ accepts iff $\mathcal{A}$ halts on w and
2. $\mathcal{H}2$ accepts iff $\mathcal{A}$ does not halt on w.

decision procedure for HP: let $\mathcal{H}1$ and $\mathcal{H}2$ run in parallel

1. $\mathcal{U}$ (almost) does what $\mathcal{H}1$ needs to do.
2. Difficult: $\mathcal{H}2$ needs to detect that that $\mathcal{A}$ does not terminate.
   - infinite tape $\leadsto$ infinite number possible configurations
   - recognising repeated configurations not sufficient.

Undecidability of the halting problem

Assumption: there is a TM $\mathcal{H}2$ which, given $c(\mathcal{A})$ and $c(w)$ as input

1. accepts if $\mathcal{A}$ does not halt with input w and
2. runs forever if $\mathcal{A}$ halts with input w.

If $\mathcal{H}2$ exists, then there is also a TM $\mathcal{S}$ accepting exactly those encodings of TMs that do not accept their own encoding

1. input: TM encoding $c(\mathcal{A})$ on tape 1
2. $\mathcal{S}$ copies $c(\mathcal{A})$ to tape 2
3. afterwards $\mathcal{S}$ operates like $\mathcal{H}2$

Computation of $\mathcal{S}$ with input $c(\mathcal{S})$

Reminder: $\mathcal{S}$ accepts $c(\mathcal{A})$ iff $\mathcal{A}$ does not accept $c(\mathcal{A})$.

Case 1 $\mathcal{S}$ accepts $c(\mathcal{S})$. This implies that $\mathcal{S}$ does not halt on the input $c(\mathcal{S})$. Therefore $\mathcal{S}$ does not accept $c(\mathcal{S})$. $\lightning$

Case 2 $\mathcal{S}$ rejects $c(\mathcal{S})$. Since $\mathcal{S}$ accepts exactly the encodings of those TMs that reject their own encoding, this implies that $\mathcal{S}$ accepts the input $c(\mathcal{S})$. $\lightning$

This implies:

1. There is no such TM $\mathcal{S}$.
2. There is no TM $\mathcal{H}2$.

Theorem (Turing 1936)

The halting problem is undecidable.

Decision problems

Theorem (Decision problems for Turing machines)

The word problem, the emptiness problem, and the equivalence problem are undecidable.

Proof.

If any of these problems were decidable, one could easily derive a decision procedure for the halting problem. □

Closure properties

Theorem (closure under $\overline{}$)

The class of languages accepted by Turing machines is not closed under complement.

Proof.

If it were closed under complement, $\mathcal{H}2$ would exist. $\square$

Theorem (closure under $\cup, \cdot, ^*, \cap$)

The class of languages accepted by TMs is closed under $\cup, \cdot, ^*, \cap$.

Proof.

Analogous to Type-1-grammars / LBAs. $\square$

Diagonalisation

Challenge of the proof:
show for all possible (infinitely many) TMs that none of them can decide the halting problem.

TM \ input	$c(\mathcal{A})$	$c(\mathcal{B})$	$c(\mathcal{C})$	$c(\mathcal{D})$	$c(\mathcal{E})$	$\ldots$
$\mathcal{A}$	✗					
$\mathcal{B}$		✗				
$\mathcal{C}$			✗			
$\mathcal{D}$				✗		
$\mathcal{E}$					✗	
$\vdots$						$\ddots$

Further diagonalisation arguments

Theorem (Cantor diagonalisation, 1891)

The set of real numbers is uncountable.

Theorem (Epimenides paradox, 6th century BC)

Epimenides [the Cretan] says: "[All] Cretans are always liars."

Theorem (Russell's paradox, 1903)

$R := \{T \mid T \notin T\}$ *Does* $R \in R$ *hold?*

Theorem (Gödel's incompleteness theorem, 1931)

Construction of a sentence in 2nd order predicate logic which states that itself cannot be proved.

Is this important?

- What is so bad about not being able to decide if a TM halts?
- Isn't this a purely academic problem?

Ludwig Wittgenstein:

> *It is very queer that this should have puzzled anyone. [...] If a man says "I am lying" we say that it follows that he is not lying, from which it follows that he is lying and so on. Well, so what? You ca go on like that until you are black in the face. Why not? It doesn't matter.*

(Lectures on the Foundations of Mathematics, Cambridge 1939)

Does it matter in practice?

It does not only affect halting

Halting is a fundamental property.
If halting cannot be decided, what can be?

Theorem (Rice, 1953)

Every non-trivial semantic property of TMs is undecidable.

non-trivial: satisfied by some TMs, not satisfied by others

semantic: referring to the accepted language

Undecidability of semantic properties

Example (Property E: TM accepts the set of prime numbers P)

If E is decidable, then so is the halting problem for $\mathcal{A}$ and an input $w_{\mathcal{A}}$.
Approach: Turing machine $\mathcal{E}$, input $w_{\mathcal{E}}$

1. simulate computation of $\mathcal{A}$ auf $w_{\mathcal{A}}$
2. decide if $w_{\mathcal{E}} \in P$

Check if $\mathcal{E}$ accepts the set of prime numbers:
yes $\rightsquigarrow$ $\mathcal{A}$ halts with input $w_{\mathcal{A}}$ no $\rightsquigarrow$ $\mathcal{A}$ does not halt on input $w_{\mathcal{A}}$

It does not only affect Turing machines

Church-Turing-thesis

Every effectively calculable function is a computable function.

computable	means calculable by a (Turing) machine
effectively calculable	refers to the intuitive idea without reference to a particular computing model

What holds for Turing machines also holds for

- unrestricted grammars,
- *while* programs,
- von Neumann architecture,
- Java/C++/Lisp/Prolog programs,
- future machines and languages

No interesting property is decidable
for any powerful programming language!

Undecidable problems in practice

software development	Does the program match the specification?
debugging	Does the program have a memory leak?
malware	Does the program harm the system?
education	Does the student's TM compute the same function as the teacher's TM?
formal languages	Do two cf. grammars generate the same language?
mathematics	Hilbert's tenth problem: find integer solutions for a polynomial with several variables
logic	Satisfiability of formulas in first-order predicate logic

Yes, it does matter!

Some things that are still possible

It is possible	because
to translate a program P from a language into an equivalent one in another language	one specific program is created for P.
to detect if a program contains a instruction to write to the hard disk	this is a syntactic property. Deciding if this instruction is eventually executed is impossible in general.
to check at runtime if a program accesses the hard disk	this corresponds to the simulation by $\mathcal{U}$. It is undecidable if the code is never executed.
to write a program that gives the correct answer in many "interesting" cases	there will always be cases in which an incorrect answer or none at all is given.

What can be done?

Can the Turing machine be “fixed”?

- undecidability proof does not use any specific TM properties
- only requirement: existence of universal machine $\mathcal{U}$
- TM is not to weak, but too powerful
- different machine models have the same problem (or are weaker)

Alternatives:

- If possible: use weaker formalisms (modal logic, dynamic logic)
- use heuristics that work well in many cases, solve remaining ones manually
- interactive programs

Turing machines: summary

- Halting problem: does TM $\mathcal{A}$ halt on input w?
- Turing: no TM can decide the halting problem.
- Rice: no TM can decide any non-trivial semantic property of TMs.
- Church-Turing: this holds for every powerful machine model.
- No interesting problem of programs in any powerful programming language is decidable.

Consequences:

☹ Computers cannot take all work away from computer scientists.

☺ Computers will never make computer scientists redundant.

Property overview

property	regular (Type 3)	context-free (Type 2)	context-sens. (Type 1)	unrestricted (Type 0)
closure				
$\cup, \cdot, ^*$	✓	✓	✓	✓
$\cap$	✓	✗	✓	✓
$\overline{}$	✓	✗	✓	✗
decidability				
word	✓	✓	✓	✗
emptiness	✓	✓	✗	✗
equiv.	✓	✗	✗	✗
deterministic equivalent to non-det.	✓	✗	?	✓

This is the End. . .

Lecture-specific material

Goals for Lecture 1

- (Getting acquainted)
- Clarifying practical issues
- Course outline and motivation
  - Formal languages
  - Language classes
  - Grammars
  - Automata
  - Questions
  - Applications
- Formal basics of formal languages

Practical Issues

- One lecture per week (on average)
  - Usually Wednesday, 10:00-13:15
  - Sometimes Tuesdays, 10:00-13:15 (see schedule for details)
  - 10 minute break around 11:30
  - I'll try to keep it entertaining...
- Important exception: 23.9.2015
  - Start at 9:30 with 45 minutes of tryout lecture by potential new faculty member
  - Please be there in time!
- Written exam
  - Calender week 48 (23.11.–27.11.)

Lecture 1

Summary

- Clarifying practical issues
  - You need running `flex`, `bison`, C compiler, editor!
- Course outline and motivation
  - Formal languages
  - Language classes
  - Grammars
  - Automata
  - Questions
  - Applications
- Formal basics of formal languages

Feedback round

- What was the best part of todays lecture?
- What part of todays lecture has the most potential for improvement?
  - Optional: how would you improve it?

Goals for Lecture 2

- Review of last lecture
- Formal languages and operations on them
- Understanding and applying regular expressions
  - Syntax - what is a valid RE?
  - Semantics - what language does it describe?
  - Applicaton - find REs for languages and vice versa

Review

- Introduction
  - Language classes
  - Grammars
  - Automata
  - Applications
- Formal languages
  - Finite alphabet Σ of symbols/letters
  - Words are finite sequences of letters from Σ
  - Languages are (finite or infinite) sets of words
- Words - properties and operations
  - $|w|, |w|_a, w[k]$
  - $w_1 \cdot w_2, w^n$
- Interesting languages
  - Binary representations of natural numbers
  - Binary representations of prime numbers
  - C functions (over strings)
  - C functions with input/output pairs

Summary

- Review of last lecture
- Formal languages and operations on them
- Understanding and applying regular expressions
  - Syntax - what is a valid RE?
  - Semantics - what language does it describe?
  - Applicaton - find REs for languages and vice versa

Feedback round

- What was the best part of todays lecture?
- What part of todays lecture has the most potential for improvement?
  - Optional: how would you improve it?

Goals for Lecture 3

- Review of last lecture
- Regular expression algebra
  - Equivalences on regular expressions
  - Simplifying REs
- Introduction to Finite Automata

Review (1)

- Operations on Languages
  - Product $L_1 \cdot L_2$: Concatenation of one word from each language
  - Power L^n: Concatenation of n words from L
  - Kleene Star: L^*: Concat any number of words from L
- Regular expressions R_Σ
  - Base cases:
    - $L(\emptyset) = \{\}$
    - $L(\epsilon) = \{\epsilon\}$
    - $L(a) = \{a\}$ for each $a \in \Sigma$
  - Complex cases:
    - $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
    - $L(r_1 \cdot r_2) = L(r_1 r_2) = L(r_1) \cdot L(r_2)$
    - $L(r^*) = L(r)^*$
    - $L((r)) = L(r)$ (brackets are used to group expressions)

Review (2)

- Equivalency: $r_1 \doteq r_2$ iff $L(r_1) = L(r_2)$
- Precedence of RE operators:
  - $(\ldots)$
  - $*$
  - $\cdot$
  - $+$

Warmup Exercise

- Assume $\Sigma = \{a, b\}$
  - Find a regular expression for the language L_1 of all words over Σ with at least 3 characters and where the third character is a a.
  - Describe L_1 formally (i.e. as a set)
  - Find a regular expression for the language L_2 of all words over Σ with at least 3 characters and where the third character is the same as the third-last character
  - Describe L_2 formally.

Lecture 3

Summary

- Regular expression algebra
  - Equivalences on regular expressions
  - Simplifying REs
- Introduction to Finite Automata
  - Graphical representation
  - Formal definition
  - Language recognized by an automata
  - Tabular representation
  - Exercises

Feedback round

- What was the best part of todays lecture?
- What part of todays lecture has the most potential for improvement?
  - Optional: how would you improve it?

Goals for Lecture 4

- Review of last lecture
- Finite Automata
  - Graphical representation
  - Formal definition
  - Language recognized by an automata
  - Tabular representation
  - Exercises

Review

- (Pumping lemma and its application)
- Review of regular expressions
- Regular expression algebra
  - Commutativity of $+$
  - Distributivity
  - $\varepsilon \notin L(s)$ and $r \doteq rs + t \longrightarrow r \doteq ts^*$ (Aarto)
  - ...for a total of 15 unconditional and 2 conditional equivalences
- Excercise: Simplifying REs

Last Weeks Exercise

1. Prove the equivalence using only algebraic operations

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

2. Simplify the following regular expression:

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

3. Prove the equivalence using only algebraic operations

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

Solution

1. Claim: $r^* \doteq \varepsilon + r^*$

   Proof:
$$\begin{aligned} \varepsilon + r^* &\doteq \varepsilon + \varepsilon + r^* r && (13) \\ &\doteq \varepsilon + r^* r && (9) \\ &\doteq r^* && (13) \end{aligned}$$

2. Simplify $r = \mathtt{0}(\varepsilon + \mathtt{0} + \mathtt{1})^* + (\varepsilon + \mathtt{1})(\mathtt{1} + \mathtt{0})^* + \varepsilon$
   - Exercise & Blackboard
3. Show $\mathtt{10}(\mathtt{10})^* \doteq \mathtt{1}(\mathtt{01})^*\mathtt{0}$
   - Exercise & Blackboard

Lecture 4

Summary

- Finite Automata
  - Graphical representation
  - Formal definition
  - Language recognized by an automata
  - Tabular representation
  - Exercises

Feedback round

- What was the best part of todays lecture?
- What part of todays lecture has the most potential for improvement?
  - Optional: how would you improve it?

Goals for Lecture 5

- Review of last lecture
  - Comment on Aarto
  - Comment on δ'
- Introduction to Nondeterministic Finite Automata
  - Definitions
  - Exercises
  - Equivalency of deterministic and nondeterministic finite automata
    - Converting NFAs to DFAs
    - Exercises
  - Equivalency of regular expressions and NFAs
    - Construction of an NFA from a regular expression

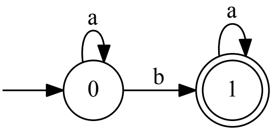
Review

- Solutions to algebraic exercises
- Finite Automata
  - Graphical representation
  - Formal definition
  - Language recognized by an automata
  - Tabular representation
  - Exercises

A note on Aarto/Arden

- Aarto: $\varepsilon \notin L(s)$ and $r \doteq rs + t \longrightarrow r \doteq ts^*$
- Why do we need $\varepsilon \notin L(s)$?
  - This guarantees that *only* words of the form ts^* are in $L(r)$
  - Example: $r \doteq rs + t$ mit $s = b^*$, $t = a$.
    - If we could apply Aarto, the result would be $r \doteq a(b^*)^* \doteq ab^*$
    - But $L = \{ab^*\} \cup \{b^*\}$ also fulfills the equation, i.e. there is no single unique solution in this case
  - Intuitively: $\varepsilon \in L(s)$ is a second escape from the recursion that bypasses t
- The case for Arden's lemma ($\varepsilon \notin L(s)$ and $r \doteq sr + t \longrightarrow r \doteq s^*t$) is analoguous

Note: Generalised Transition Function δ' (1)

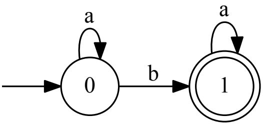


- We have defined the extended transition function for DFA's δ' to start the recursion at the front of the word:
  - $\delta'(q,\varepsilon) = q$
  - $\delta'(q, `w) = \begin{cases} \delta'(\delta(q,c),v) & \text{if} \quad \delta(q,c) \neq \Omega \\ \Omega & \text{otherwise} \end{cases}$

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

$$\begin{array}{rcl} \delta'(0, abaa) & = & \delta'(\delta(0,a), baa) \\ & = & \delta'(\delta(\delta(0,a),b)aa) \\ & = & \delta'(\delta(\delta(\delta(0,a),b),a),a) \\ & = & \delta'(\delta(\delta(\delta(\delta(0,a),b),a),a),\varepsilon) \\ & = & \delta'(\delta(\delta(\delta(0,b),a),a),\varepsilon) \\ & = & \delta'(\delta(\delta(1,a),a),\varepsilon) \\ & = & \delta'(\delta(1,a),\varepsilon) \\ & = & \delta'(1,\varepsilon) \\ & = & 1 \end{array}$$

Note: Generalised Transition Function δ' (2)



- Alternative definiton (dissassemble the word from the end):
  - $\delta' : Q \times \Sigma^* \to Q \cup \{\Omega\}$
  - $\delta'(q, \varepsilon) = q$
  - $\delta'(q, wc) = \begin{cases} \delta(\delta'(q, w), c) & \text{if} \quad \delta(q, c) \neq \Omega \\ \Omega & \text{otherwise} \end{cases}$

  with $\quad c \in \Sigma; w \in \Sigma^*$
- Thus:

$$
\begin{aligned}
\delta'(0, abaa) &= \delta(\delta'(0, a), baa) \\
&= \delta(\delta'(0, aba), a) \\
&= \delta(\delta(\delta'(0, ab), a), a) \\
&= \delta(\delta(\delta(\delta'(0, a), b), a), a) \\
&= \delta(\delta(\delta(\delta(\delta'(0, \varepsilon), a), b), a), a) \\
&= \delta(\delta(\delta(\delta(0, a), b), a), a) \\
&= \delta(\delta(\delta(0, b), a), a) \\
&= \delta(\delta(1, a), a) \\
&= \delta(1, a) \\
&= 1
\end{aligned}
$$

Note: Generalised Transition Function δ' (3)

Definition (Generalised transition function δ')

Assume a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$. The extended transition function $\delta' : Q \times \Sigma^* \to Q \cup \{\Omega\}$ is defined as follows:

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

  with $\quad c \in \Sigma; w \in \Sigma^*$

This is the definition we will use from now on!

Exercise (from last lecture)

- Assume $\Sigma = \{a, b\}$
- Find a DFA for $L((a+b)^*b(a+b)(a+b))$
- The language contains all words from Σ^* which at least three characters and where the third-last character is b

Lecture 5

Summary

- Review of last lecture
- Introduction to Nondeterministic Finite Automata
  - Definitions
  - Exercises
  - Equivalency of deterministic and nondeterministic finite automata
    - Converting NFAs to DFAs
    - Exercises
  - Equivalency of regular expressions and NFAs
    - Construction of an NFA from a regular expression

Feedback round

- What was the best part of todays lecture?
- What part of todays lecture has the most potential for improvement?
  - Optional: how would you improve it?

Goals for Lecture 6

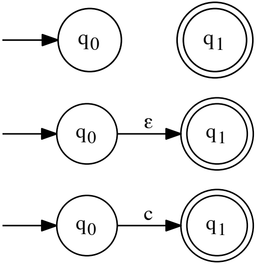
- Review of last lecture
- Warmup exercise
- Completing the circle: REs from DFAs
- Minimizing DFAs
  - . . . and a first application

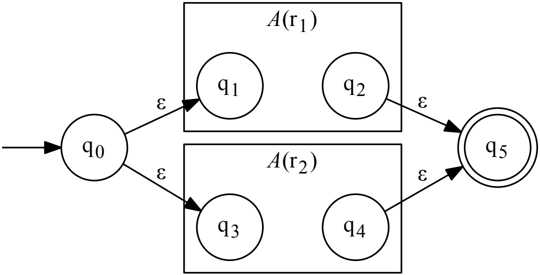
Review: NFAs

- NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$
  1. Q is the finite set of states.
  2. Σ is the input alphabet.
  3. Δ is a relation on $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$
  4. $q_0 \in Q$ is the initial state.
  5. $F \subseteq Q$ is the set of final states.
- Significant differences to DFAs:
  - Δ is a relation - the automaton can change to multiple successor states
  - Δ allows for ε-transistion - it can change states spontaneously
- DFAs are (in essence) already NFAs
- NFAs can be simulated by DFAs
  - States of $det(A)$ are sets of states of A
  - $\hat{\delta}$ goes from sets of A-states to sets of A
    - ...by combining the transistion of the individual states
    - ...and taking the ε-closure

Review (REs and NFAs)

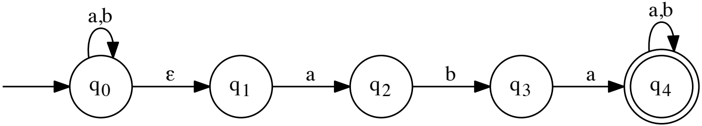
- Every language described by a regular expression can be accepted by and NFA!
- Proof: Construction of NFAs from REs
  - Simple NFAs for base cases
  - Glue NFAs together with ε-transition for complex REs





Warmup: NFA to DFA transformation

Convert the following NFA (over $\Sigma = \{a, b\}$) into an equivalent DFA:



Solution Lecture 6

Homework assignment

- Install an operational UNIX/Linux environment on your computer
  - You can install VirtualBox (`https://www.virtualbox.org`) and then install e.g. Ubuntu (`http://www.ubuntu.com/`) on a virtual machine
  - For Windows, you can install the complete UNIX emulation package Cygwin from `http://cygwin.com`
  - For MacOS, you can install `fink` (`http://fink.sourceforge.net/`) or MacPorts (`https://www.macports.org/`) and the necessary tools
- You will need at least `flex`, `bison`, `gcc`, `grep`, `sed`, `AWK`, `make`, and a good text editor of your choice

Summary

- Review of last lecture
- Warmup exercise
- Completing the circle: REs from DFAs
  - Find system of equations (easy)
  - Solve system of equations (harder)
    - Use substitution to get rid of variables
    - Use simplification to make expressions smaller and bring them into the right form ($sL+t$)
    - Use Arden's lemma to eliminate loops (s^*t)
- Minimizing DFAs
  - Identify and merge equivalent states
  - Result is unique (up to names of states)
  - Equivalency of REs can be decided by comparison of corresponding minimal DFAs
- Homework: Get ready for `flex`ing...

Feedback round

- What was the best part of todays lecture?
- What part of todays lecture has the most potential for improvement?
  - Optional: how would you improve it?

Goals for Lecture 7

- Review of last lecture
- Discussion of execise/homework Exercise: Equivalence of regular expressions
- Beyond regular languages: The Pumping Lemma
  - Motivation/Lemma
  - Application of the lemma
  - Implications
- Properties of regular languages
  - Closure properties (union, intersection, . . .)

Review

- Finding an RE for a given DFAs
  - Find system of equations (easy)
  - Solve system of equations (harder)
    - Use substitution to get rid of variables
    - Use simplification to make expressions smaller and bring them into the right form ($sL + t$)
    - Use Arden's lemma to eliminate loops (s^*t)
- Minimizing DFAs
  - Identify and merge equivalent states
  - Result is unique (up to names of states)
  - Equivalency of REs can be decided by comparison of corresponding minimal DFAs
  - Open exercise/homework!

Exercise: Equivalence of REs

Reusing an exercise from an earlier section, prove the following equivalence (by conversion to minimal DFAs):

$$\texttt{10(10)}^* \doteq \texttt{1(01)}^*\texttt{0}$$

1. Construct NFAs from the REs
2. Convert NFAs to DFAs
3. Minimize DFAs
4. Compare minimized DFAs (modulo state names)

Solution

Lecture 7

Reminder: Homework assignment

- Install an operational UNIX/Linux environment on your computer
  - You can install VirtualBox (`https://www.virtualbox.org`) and then install e.g. Ubuntu (`http://www.ubuntu.com/`) on a virtual machine
  - For Windows, you can install the complete UNIX emulation package Cygwin from `http://cygwin.com`
  - For MacOS, you can install `fink` (`http://fink.sourceforge.net/`) or MacPorts (`https://www.macports.org/`) and the necessary tools
- You will need at least `flex`, `bison`, `gcc`, `grep`, `sed`, `AWK`, `make`, and a good text editor of your choice

Summary

- Review of last lecture
- Discussion of execise/homework Exercise: Equivalence of regular expressions
- Beyond regular languages: The Pumping Lemma
  - Motivation/Lemma
  - Application of the lemma ($a^nb^n, a^nb^m, n < m$)
  - Implications (Nested structures are not regular)
- Properties of regular languages
  - Closure properties (union, intersection, . . .)

Feedback round

- What was the best part of todays lecture?
- What part of todays lecture has the most potential for improvement?
  - Optional: how would you improve it?

Goals for Lecture 8

- Review of last lecture
- Completing the theory of regular languages
  - Emptiness, finiteness, ...
  - Decision problems (word problem, equivalence, ...)
  - Wrap-up
- Scanning in practice
  - Scanners in context
  - Practical regular expressions
  - Flex

Review

- The Pumping Lemma
  - Motivation/Lemma
    - For every regular language L there exits a k such that any word s with $|s| \geq k$ can be split into $s = uvw$ with $|uv| \leq k$ and $v \neq \varepsilon$ and $uv^h w \in L$ for all $h \in \mathbb{N}$
    - Use in proofs by contradiction: Assume a language is regular, then derive contradiction
  - Application of the lemma ($a^n b^n, a^n b^m, n < m$)
  - Implications (Nested structures are not regular)
- Properties of regular languages
  - The union of two regular languages is regular
  - The intersection of two regular languages is regular (Product automaton!)
  - The concatenation of two regular languages is regular
  - The Kleene star of a regular language is regular
  - The complement of a regular language is regular

Closure under complement

Let $\mathcal{A}_L$ be a complete DFA for the language L.
(If there are Ω transitions, add a junk state.)

Then $\overline{\mathcal{A}_L} = (Q, \Sigma, q_0, \delta, Q \setminus F)$ is an automaton accepting $\overline{L}$:

- if $w \in L(\mathcal{A})$ then $\delta'(q_0, w) \in F$, i.e. $\delta'(q_0, w) \notin Q \setminus F$, which implies $w \notin L(\overline{\mathcal{A}_L})$.
- if $w \notin L(\mathcal{A})$ then $\delta'(q_0, w) \notin F$, i.e. $\delta'(q_0, w) \in Q \setminus F$, which implies $w \in L(\overline{\mathcal{A}_L})$.

Reminder:

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

$\delta'(q_0, w)$ is the final state of the automaton after processing w

All we have to do is exchange final and non-final states.

Closure properties: exercise

Show that $L = \{w \in \{a, b\}^* \mid |w|_a = |w|_b\}$ is not regular.

Hint: Use the following:

- $a^n b^n$ is not regular. (Pumping lemma)
- $a^* b^*$ is regular. (Regular expression)
- (one of) the closure properties shown before.

Lecture 8

Summary

- Completing the theory of regular languages
  - Emptiness, finiteness, ...
  - Decision problems (word problem, equivalence, ...)
  - Wrap-up
- Scanning in practice
  - Scanners in context
  - Practical regular expressions
  - Flex
    - Definition section
    - Rule section
    - User code section/`yylex()`

Feedback round

- What was the best part of todays lecture?
- What part of todays lecture has the most potential for improvement?
  - Optional: how would you improve it?

Goals for Lecture 9

- Review of last lecture
  - Short review of the homework exercise
- Formal grammars
  - Formal grammars and their languages
  - The Chomsky-Hierarchy
  - Regular grammars/Right-linear grammars and automata

Review

- Wrap-up of regular languages
  - Properties (closures under complement, finiteness)
  - Decision problems (emptiness, word, equivalence, finiteness)
- Practical scanning
  - Scanning in context
  - Scanning with `flex`
    - 3 sections (definitions, rules, user code)
    - Workflow (`flexx`, `gcc`, `gcc`)
    - Regular expressions in practice
    - Flexercise (`http://wwwlehre.dhbw-stuttgart.de/~sschulz/TEACHING/FLA2015/scammer.l`)

Lecture 9

Summary

- Formal grammars
  - Formal grammars and their languages
  - The Chomsky-Hierarchy
  - Regular grammars/Right-linear grammars and automata

Feedback round

- What was the best part of todays lecture?
- What part of todays lecture has the most potential for improvement?
  - Optional: how would you improve it?

Goals for Lecture 10

- Review of last lecture
- Context-Free grammars
  - Examples
  - Chomsky Normal Form
  - Parsing with Cocke-Younger-Kasami

Review

- Formal grammars
  - Formal grammars and their languages
  - The Chomsky-Hierarchy
    - Unrestricted
    - Context-sensitive ($\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, non-contracting)
    - Context-free ($A \rightarrow \beta$)
    - Regular/right-linear ($A \rightarrow aB$ (where a, B can be ϵ))
  - Regular grammars/Right-linear grammars and automata

Lecture 10

Summary

- Context-Free grammars
  - Examples
  - Chomsky Normal Form
  - Parsing with Cocke-Younger-Kasami

Feedback round

- What was the best part of todays lecture?
- What part of todays lecture has the most potential for improvement?
  - Optional: how would you improve it?

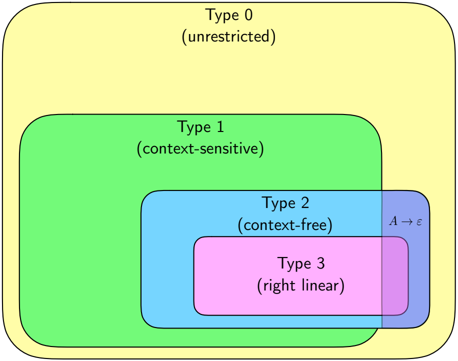
Goals for Lecture 11

- Review of last lecture
- Test exam
- Solutions

Review

- Context-Free grammars
  - Reduced grammar
    - Remove non-terminating symbols
    - Remove non-reachable symbols
  - Chomsky Normal Form
    - Remove ε-rules
    - Remove chain rules
    - Reduce grammar
    - Introduce new non-terminals to remove terminals from complex RHS
    - Intoduce new non-terminals to break up long RHS
  - Parsing with Cocke-Younger-Kasami
    - Dynamic programming

Interlude: Chomsky-Hierarchy for Grammars (again)



- For languages, Type-0, Type-1, Type-2, Type-3 form a real inclusion hierarchy
- Not quite true for grammars:
  - $A \to \varepsilon$ allowed in context-free/regular grammars, not in context-free languages
- Eliminating ε-productions removes this discrepancy!

Test Exam

Summary

- Review of last lecture
- Test exam
- Solutions

Final feedback round

- What was the best part of the course?
- What part of the course that has the most potential for improvement?
  - Optional: how would you improve it?

Selected Solutions

Solution to Exercise: Algebra on regular expressions (1)

- Claim: $r^* \doteq \varepsilon + r^*$

Proof:
$$\begin{aligned} \varepsilon + r^* &\doteq \varepsilon + \varepsilon + r^* r && (13) \\ &\doteq \varepsilon + r^* r && (9) \\ &\doteq r^* && (13) \end{aligned}$$

Simplification of regular expressions

Solution to Exercise: Algebra on regular expressions (2)

$$
\begin{aligned}
r &= \mathtt{0}(\varepsilon + \mathtt{0} + \mathtt{1})^* + (\varepsilon + \mathtt{1})(\mathtt{1} + \mathtt{0})^* + \varepsilon \\
&\overset{14,1}{\doteq} \mathtt{0}(\mathtt{0} + \mathtt{1})^* + (\varepsilon + \mathtt{1})(\mathtt{0} + \mathtt{1})^* + \varepsilon \\
&\overset{7}{\doteq} \mathtt{0}(\mathtt{0} + \mathtt{1})^* + \varepsilon(\mathtt{0} + \mathtt{1})^* + \mathtt{1}(\mathtt{0} + \mathtt{1})^* + \varepsilon \\
&\overset{5}{\doteq} \mathtt{0}(\mathtt{0} + \mathtt{1})^* + (\mathtt{0} + \mathtt{1})^* + \mathtt{1}(\mathtt{0} + \mathtt{1})^* + \varepsilon \\
&\overset{1,7}{\doteq} \varepsilon + (\mathtt{0} + \mathtt{1})(\mathtt{0} + \mathtt{1})^* + (\mathtt{0} + \mathtt{1})^* \\
&\overset{16}{\doteq} \varepsilon + (\mathtt{0} + \mathtt{1})^*(\mathtt{0} + \mathtt{1}) + (\mathtt{0} + \mathtt{1})^* \\
&\overset{13}{\doteq} (\mathtt{0} + \mathtt{1})^* + (\mathtt{0} + \mathtt{1})^* \\
&\overset{9}{\doteq} (\mathtt{0} + \mathtt{1})^*.
\end{aligned}
$$

Application of Aarto's lemma

Solution to Exercise: Algebra on regular expressions (3)

- Show that $u = \texttt{10}(\texttt{10})^* \doteq \texttt{1}(\texttt{01})^*\texttt{0}$
- Idea: u is of the form ts^* with:
  - $t = \texttt{10}$
  - $s = \texttt{10}$
  - This suggest Aarto's Lemma. To apply the lemma, we must show that $r = \texttt{1}(\texttt{01})^*\texttt{0} \doteq rs + t$
- So:

$$\begin{array}{rcll} rs + t & = & \texttt{1}(\texttt{01})^*\texttt{010} + \texttt{10} & \\ & \doteq & \texttt{1}((\texttt{01})^*\texttt{010} + \texttt{0}) & \text{(factor out } \texttt{1}\text{)} \\ & \doteq & \texttt{1}((\texttt{01})^*\texttt{01} + \varepsilon)\texttt{0} & \text{(factor out } \texttt{0}\text{)} \\ & \doteq & \texttt{1}(\texttt{01})^*\texttt{0} & \text{(14)} \\ & = & r & \end{array}$$

  - Since $L(s) = \{\texttt{10}\}$ (and hence $\varepsilon \notin L(s)$), we can apply Aarto and rewrite $r \doteq ts^* \doteq \texttt{10}(\texttt{10})^*$.

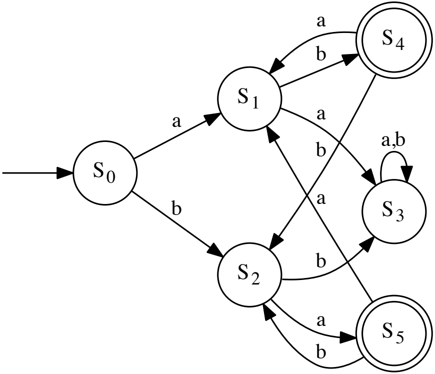
Transformation into DFA (1)

- Incremental computation of $\hat{Q}$ and $\hat{\delta}$:
  - Initial state $S_0 = ec(q_0) = \{q_0, q_1, q_2\}$
  - $\hat{\delta}(S_0, a) = \delta^*(q_0, a) \cup \delta^*(q_1, a) \cup \delta^*(q_2, a) = \{\} \cup \{\} \cup \{q_4\} = \{q_4\} = S_1$
  - $\hat{\delta}(S_0, b) = \{q_3\} = S_2$
  - $\hat{\delta}(S_1, a) = \{\} = S_3$
  - $\hat{\delta}(S_1, b) = ec(q_6) = \{q_6, q_7, q_0, q_1, q_2\} = S_4$
  - $\hat{\delta}(S_2, a) = \{q_5, q_7, q_0, q_1, q_2\} = S_5$
  - $\hat{\delta}(S_2, b) = \{\} = S_3$
  - $\hat{\delta}(S_3, a) = \{\} = S_3$
  - $\hat{\delta}(S_3, b) = \{\} = S_3$
  - $\hat{\delta}(S_4, a) = \{q_4\} = S_1$
  - $\hat{\delta}(S_4, b) = \{q_3\} = S_2$
  - $\hat{\delta}(S_5, a) = \{q_4\} = S_1$
  - $\hat{\delta}(S_5, b) = \{q_3\} = S_2$
- $\hat{F} = \{S_4, S_5\}$ (since $q_7 \in S_4, q_7 \in S_5$)

Transformation into DFA (2)

- $det(\mathcal{A}) = (\hat{Q}, \Sigma, \hat{\delta}, S_0, \hat{F})$
  - $\hat{Q} = \{S_0, S_1, S_2, S_3, S_4, S_5\}$
  - $\hat{F} = (S_4, S_5\}$
  - $\hat{\delta}$ given by the table below

$\hat{\delta}$	a	b
$\rightarrow S_0$	S_1	S_2
S_1	S_3	S_4
S_2	S_5	S_3
S_3	S_3	S_3
$*S_4$	S_1	S_2
$*S_5$	S_1	S_2



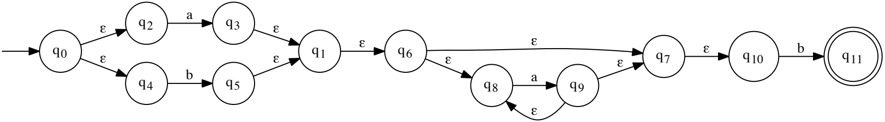
- Regexp:
  $L(\mathcal{A}) = L((ab + ba)(ab + ba)^*)$

Back to exercise

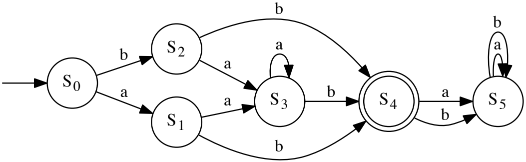
Transformation of RE into NFA

Systematically construct an NFA accepting the same language as the regular expression $(\mathtt{a}+\mathtt{b})\mathtt{a}^*\mathtt{b}$.

Solution:

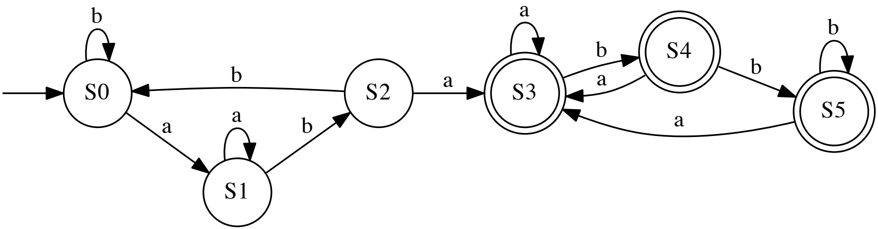


Corresponding DFA:



Back to exercise

Solution: NFA to DFA “aba”

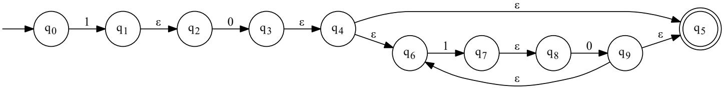


Back to exercise

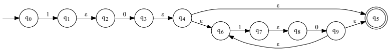
Show $\mathtt{10(10)}^* \doteq \mathtt{1(01)}^*\mathtt{0}$ via minimal DFAs (1)

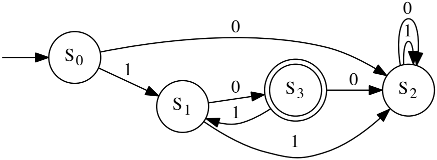
- Step 1: NFA for $\mathtt{10(10)}^*$:

```
       | epsilon     0      1
---------------------------
-> q0  |      {}    {}   {q1}
   q1  |    {q2}    {}     {}
   q2  |      {}  {q3}     {}
   q3  |    {q4}    {}     {}
   q4  | {q5,q6}    {}     {}
*  q5  |      {}    {}     {}
   q6  |      {}    {}   {q7}
   q7  |    {q8}    {}     {}
   q8  |      {}  {q9}     {}
   q9  | {q5,q6}    {}     {}
```



- Step 2: DFA $\mathcal{A}$ for $\texttt{10(10)}^*$:





- Step 3: Minimizing of $\mathcal{A}$

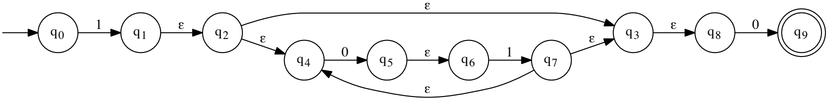
	S_0	S_1	S_2	S_3	S_4	S_5
S_0	o	x	x	x	x	x
S_1	x	o	x	x	o	x
S_2	x	x	o	x	x	x
S_3	x	x	x	o	x	o
S_4	x	o	x	x	o	x
S_5	x	x	x	o	x	o

Result: (S_1, S_4) and (S_3, S_5) can be merged

Show $\texttt{10(10)}^* \doteq \texttt{1(01)}^*\texttt{0}$ via minimal DFAs (3)

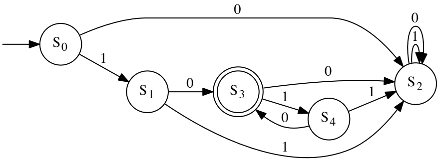
- Step 4: NFA zu $\texttt{1(01)}^*\texttt{0}$:

```
       | epsilon    0      1
---------------------------
-> q0 |       {}   {}   {q1}
   q1 |     {q2}   {}     {}
   q2 | {q3,q4}   {}     {}
   q3 |     {q8}   {}     {}
   q4 |       {} {q5}     {}
   q5 |     {q6}   {}     {}
   q6 |       {}   {}   {q7}
   q7 | {q4,q3}   {}     {}
   q8 |       {}   {}   {q9}
 * q9 |       {}   {}     {}
```



Show $\texttt{10(10)}^* \doteq \texttt{1(01)}^*\texttt{0}$ via minimal DFAs (4)

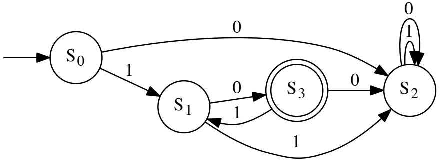
- Step 5: DFA $\mathcal{B}$ for $\texttt{1(01)}^*\texttt{0}$



- Step 6: Minimization of $\mathcal{B}$

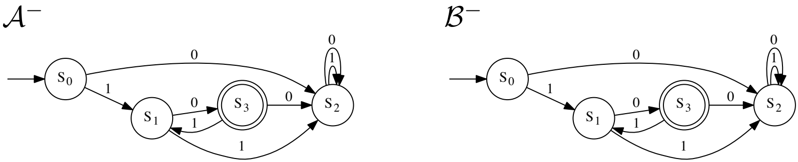
	S_0	S_1	S_2	S_3	S_4
S_0	o	x	x	x	x
S_1	x	o	x	x	o
S_2	x	x	o	x	x
S_3	x	x	x	o	x
S_4	x	o	x	x	o

Result: (S_1, S_4) can be merged



Show $\texttt{10(10)}^* \doteq \texttt{1(01)}^*\texttt{0}$ via minimal DFAs (5)

- Step 7: Comparision of $\mathcal{A}^-$ and $\mathcal{B}^-$



- Result: The two automata are identical, hence the two original regular expressions describe the same languages.

Back to exercise
Back to review

Pumping lemma

Solution to $a^n b^m$ with $n < m$

- Proposition: $L = \{a^n b^m \mid n < m\}$ is not regular.
- Proof by contradiction. We assume L is regular
- Then: $\exists k \in \mathbb{N}$ with:
  - $\forall s \in L$ with $|s| \geq k : \exists u, v, w \in \Sigma^*$ such that
    - $s = uvw$
    - $|uv| \leq k$
    - $v \neq \varepsilon$
    - $uv^h w \in L$ for all $h \in \mathbb{N}$
- We consider the word $s = a^k b^{k+1} \in L$
  - Since $|uv| \leq k$: $u = a^i, v = a^j, w = a^l b^{k+1}$ and $j > 0, i + j + l = k$
  - Now consider $s' = uv^2 w$. According to the pumping lemma, $s' \in L$. But $s' = a^i a^j a^j a^l b^{k+1} = a^{i+j+l+j} b^{k+1} = a^{k+j} b^{k+1}$. Since $j \in \mathbb{N}, j > 0$: $k + j \not< k + 1$. Hence $s' \notin L$. This is a contradiction. Hence the assumption is wrong, and the original proposition is true. q.e.d.

Solution: Pumping lemma (Prime numbers)

- Proposition: $L = \{a^p \mid p \in \mathbb{P}\}$ is not regular (where $\mathbb{P}$ is the set of all prime numbers)
- Proof: By contradiction, using the pumping lemma.
  - Assumption: L is regular. Then there exist a k such that all words in L with at least lenght k can be pumped.
- Consider the word $s = a^p$, where $p \in \mathbb{P}, p \geq k$
  - Then there are $u, v, w \in \Sigma^*$ with $uvw = s, |uv| \leq k, v \neq \varepsilon$, and $uv^h w \in L$ for all $h \in \mathbb{N}$.
  - We can write $u = a^i, v = a^j, w = a^l$ with $i + j + l = p$
  - So $s = a^i a^j a^l$ and $a^i a^{j \cdot h} a^l \in L$ for all $h \in \mathbb{N}$.
  - Consider $h = p + 1$. Then $a^i a^{j \cdot (p+1)} a^l \in L$
  - $a^i a^{j \cdot (p+1)} a^l = a^i a^{jp+j} a^l = a^i a^{jp} a^j a^l = a^i a^j a^l a^{jp} = a^p a^{jp} = a^{(j+1)p}$
  - But $(j+1)p \notin \mathbb{P}$, since $j + 1 > 1$ and $p > 1$, and $(j+1)p$ thus has (at least)two non-trivial divisors.
  - Thus $a^{(j+1)p} \notin L$. This violates the pumping lemma and hence contradicts the assumption. Thus the assumption is wrong and the proposition holds. *q.e.d.*

Solution: Transformation to Chomsky Normal Form (1)

Compute the Chomsky normal form of the following grammar:
$G = (N, \Sigma, P, S)$

- $N = \{S, A, B, C, D, E\}$
- $\Sigma = \{\mathtt{a}, \mathtt{b}\}$
- P :
$$\begin{array}{lcl} S & \to & AB|SB|BDE \\ A & \to & Aa \\ B & \to & bB|BaB|ab \end{array} \qquad \begin{array}{lcl} C & \to & SB \\ D & \to & E \\ E & \to & \varepsilon \end{array}$$

Step 1: ε-Elimination

- Nullable NTS: $N = \{E, D\}$
- New rules:

$S \to BD$ (from $S \to BDE, \beta_1 = BD, \beta_2 = \varepsilon$)
$S \to BE$ (from $S \to BDE, \beta_1 = B, \beta_2 = E$)
$S \to B$ (from $S \to BD$ or $S \to BE, \beta_1 = B, \beta_2 = \varepsilon$)
$D \to \varepsilon$ (from $D \to E, \beta_1 = \varepsilon, \beta_2 = \varepsilon$)

- Remove $E \to \varepsilon$, $D \to \varepsilon$

Solution: Transformation to Chomsky Normal Form (2)

Step 2: Elimination of Chain Rules.

- Current chain rules: $S \to B$, $D \to E$
- Eliminate $S \to B$:
  - $N(S) = \{B\}$
  - New rules: $S \to bB, S \to BaB, S \to ab$
- Eliminate $D \to E$
  - $N(D) = \{E\}$
  - E has no rule, therefore no new rules!
- Current state of P:

$$\begin{array}{lcl} S & \to & AB|SB|BDE|BD|BE|bB|BaB|ab \\ A & \to & Aa \end{array} \qquad \begin{array}{lcl} C & \to & SB \\ B & \to & bB|BaB|ab \end{array}$$

Solution: Transformation to Chomsky Normal Form (3)

Step 3: Reducing the grammar

- Terminating symbols: $T = \{S, B, C\}$ (A, D, E do not terminate)
  - Remove all rules that contain A, E, D. Remaining:

$$
\begin{array}{lllllll}
S & \rightarrow & SB|bB|BaB|ab & \quad & C & \rightarrow & SB \\
B & \rightarrow & bB|BaB|ab & & & &
\end{array}
$$

- Reachable symbols: $R = \{S, B\}$ (C is not reachable)
  - Remove all rules containing C. Remaining:

$$
\begin{array}{lll}
S & \rightarrow & SB|bB|BaB|ab \\
B & \rightarrow & bB|BaB|ab
\end{array}
$$

Solution: Transformation to Chomsky Normal Form (4)

Step 4: Introduce new non-terminals for terminals

- New rules: $X_a \to a, X_b \to b$. Result:

$$\begin{array}{lllllll} S & \to & SB|X_bB|BX_aB|X_aX_b & \quad & X_a & \to & a \\ B & \to & X_bB|BX_aB|X_aX_b & & X_b & \to & b \end{array}$$

Step 5: Introduce new non-terminals to break up long right hand sides:

- Problematic RHS: BX_aB (in two rules)
- New rule: $C_1 \to X_aB$. Result:

$$\begin{array}{lllllll} S & \to & SB|X_bB|BC_1|X_aX_b & \quad & X_a & \to & a \\ B & \to & X_bB|BC_1|X_aX_b & & X_b & \to & b \\ C_1 & \to & X_aB & & & & \end{array}$$

Solution: Transformation to Chomsky Normal Form (5)

Final grammar: $G' = (N', \Sigma, P', S)$ with

- $N' = \{S, B, C_1, X_a, X_b\}$
- $\Sigma = \{a, b\}$
- P' : $$\begin{array}{lcl} S & \rightarrow & SB|X_bB|BC_1|X_aX_b \\ B & \rightarrow & X_bB|BC_1|X_aX_b \\ C_1 & \rightarrow & X_aB \end{array} \qquad \begin{array}{lcl} X_a & \rightarrow & a \\ X_b & \rightarrow & b \end{array}$$

Back to exercise