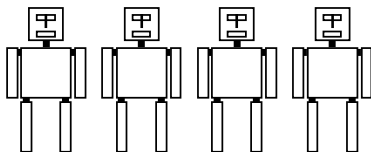# Formal Languages and Automata

Stephan Schulz & Jan Hladik

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

with contributions from David Suendermann

$$L \subseteq \Sigma^*$$

**DHBW** Stuttgart
Duale Hochschule
Baden-Württemberg

# Table of Contents

# Outline

# 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/fla2018.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, make, and a good text editor

# Outline of the Lecture

# Outline

# Formal language concepts

Alphabet: finite set $\Sigma$ 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            Jan sagt, dass wir
  let                         die Kinder
  the children                  dem Hans
    help                          das Haus
    Hans                          anstreichen
      paint                     helfen
      the house               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 (in first-order logic)
- ► 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 | $\rightsquigarrow$ | finite automaton/finite state machine |
| context-free | $\rightsquigarrow$ | pushdown automaton |
| context-sensitive | $\rightsquigarrow$ | linearly bounded Turing machine |
| recursively enumerable | $\rightsquigarrow$ | (unbounded) Turing machine |

# Example: Finite Automaton
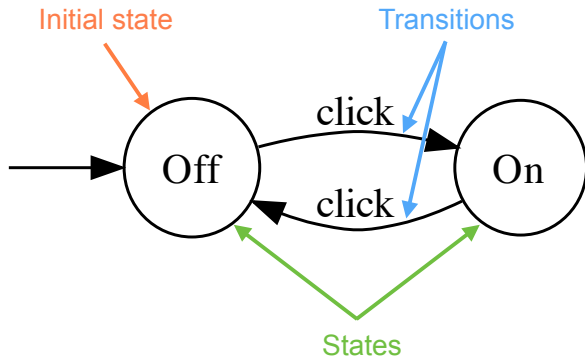
# Example: Finite Automaton

Formally:

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

  | $\delta$ | click |
  |----------|-------|
  | Off      | On    |
  | On       | Off   |

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

# ATC scenario



Aggregator

ATC Center
(controllers)
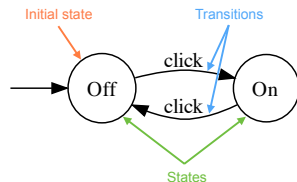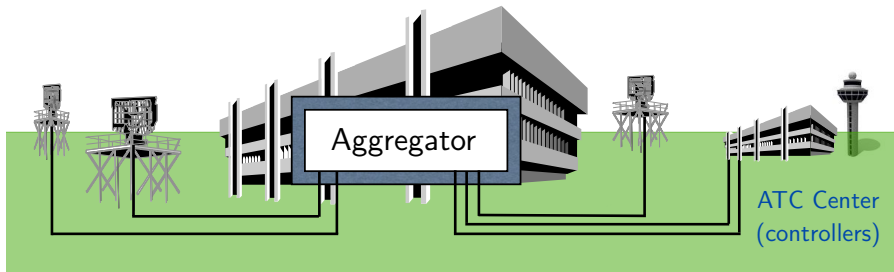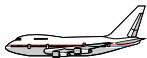
Aktive server:
- Accepts sensor data
- Provides ASP
- Sends "alive" messages
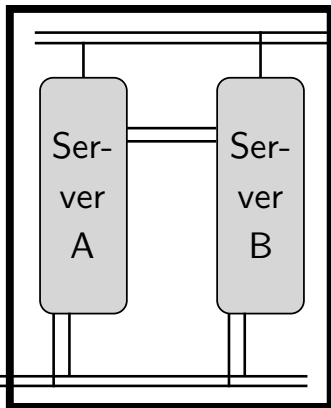
Passive server
- Ignores sensor data
- Monitors "alive" messages
- Takes over in case of failure

ATC

Server A

Server B

Sensors

# Finite automaton to the rescue
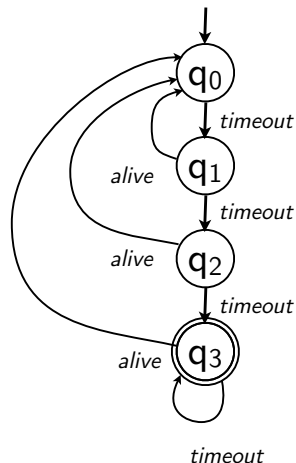


- ▶ Two events ("letters")
  - ▶ timeout: 0.1 seconds have passed
  - ▶ alive: message from active server
- ▶ States $q_0, q_1, q_2$: Server is passive
  - ▶ No processing of input
  - ▶ No sending of alive messages
- ▶ State $q_3$: Server becomes active
  - ▶ Process input, provide output to ATC
  -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

# Exercise: Finite automaton
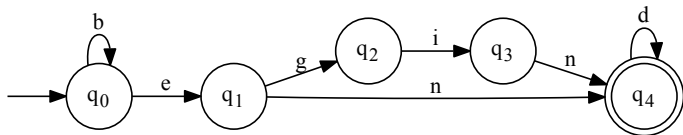


Does this automaton accept the words *begin*, *end*, *bind*, *bend*?

# Turing Machine

"Universal computer"

- ▶ Very simple model of a computer
  - ▶ Infinite tape, one read/write head
  - ▶ Tape can store letters from a alphabet
  - ▶ Finite automaton 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: $l \rightarrow r$ means: $l$ can be replaced by $r$ anywhere in the word

## Example

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

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

## Exercise: Grammars

Using

- the non-terminal symbols $S, B, D, E, G, I, N$
- the terminal symbols $b, d, e, g, i, n$
- the production rules $S \rightarrow BEGIN$,
  $BEG \rightarrow E, IN \rightarrow IND, IN \rightarrow N, EG \rightarrow EGG, GGG \rightarrow B$,
  $B \rightarrow b, D \rightarrow d, E \rightarrow e, G \rightarrow g, I \rightarrow i, N \rightarrow n$

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

End lecture 1

# Outline

29

# Alphabets

## Definition (Alphabet)

An alphabet $\Sigma$ 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 $\Sigma_{\text{en}} = \{a, \ldots, z, A, \ldots, Z\}$.
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 $\Sigma$ is a finite sequence (list) of characters of $\Sigma$:

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

▶ The empty word with no characters is written as $\varepsilon$.
▶ The set of all words over an alphabet $\Sigma$ is represented by $\Sigma^*$.

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

# Words

## Example

1. Using $\Sigma_{\text{bin}}$, 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 $\Sigma_{\text{en}}$, 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

- $|\text{example}| = 7$ and $|\varepsilon| = 0$
- $|\text{example}|_e = 2$ and $|\text{example}|_k = 0$
- $\text{example}[4] = \text{m}$

# 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 $w_1 = 01$ and $w_2 = 10$.
Then the following holds:

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

# Iterated concatenation

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

## Definition (Power of a word)

For $n \in \mathbb{N}$, the *n*-th power $w^n$ of a word $w$ concatenates the same word $n$ times:

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

## Example

Let $w = ab$. Then:

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

## 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 $\Sigma$, a formal language over $\Sigma$ is a subset $L \subseteq \Sigma^*$.

## Example

Let $L_{\mathbb{N}} = \{1w \mid w \in \Sigma_{\text{bin}}^*\} \cup \{0\}$.
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):

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

# 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) $n(0) = 0$,

(b) $n(1) = 1$,

(c) $n(w0) = 2 \cdot n(w)$ for $|w| > 0$,

(d) $n(w1) = 2 \cdot n(w) + 1$ 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

$$\mathbb{P} = \{p \in \mathbb{N}^{\geq 2} \mid \{t \in \mathbb{N} \mid \exists k \in \mathbb{N} : k \cdot t = p\} = \{1, p\}\}.$$

# 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.

## Examples

- ► `char* f(char* x){return x;}` $\in L_C$
- ► `char* f(char* x){return "";}` $\in L_C$
- ► `char* f(char* x, int y){return "";}` $\notin L_C$
- ► `Harakiri` $\notin L_C$

# C function evaluations as a language

## Definition

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

$$L_u = \{f \dagger x \dagger y \mid \text{(a) and (b) and (c)}\} \quad \text{with}$$

(a) $f \in L_C$ (i.e. $f$ is a C function mapping strings to strings),
(b) $x, y \in \Sigma_{\text{ASCII}}^*$,
(c) applying $f$ to $x$ terminates and returns $y$.

## Examples

▶ `char* f(char* x){return x;}`†aaa†aaa $\in L_u$
▶ `char* f(char* x){return x;}`†aaa†bbb $\notin L_u$
▶ `char* f(char* x){return "";}`†aaa† $\in L_u$

42

## Formal languages can be highly complex

Formal languages have a wide scope:

- ▶ Testing whether a word belongs to $L_{\mathbb{N}}$ is straightforward
- ▶ The same test for $L_{\mathbb{P}}$ is more complex
- ▶ The test for $L_C$ requires a proper C parser
- ▶ Later we will see that there is no algorithm that will correctly perform the membership test for $L_u$

```python
def inN(x):
    if x=="0":
        return True
    if x=="":
        return False
    if x[0]!="1":
        return False
    for c in x[1:]:
        if c!="0" and c!="1":
            return False
    return True
```

# Product

## Definition (Product of formal languages)

Given an alphabet $\Sigma$ 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 $\Sigma_{\mathrm{en}}$, we define the languages

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

The product is

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

# Power

## Definition (Power of a language)

Given an alphabet $\Sigma$, 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 $\Sigma_{\mathrm{en}}$, 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 $\Sigma$ 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 $\Sigma_{\text{en}}$, we define the language $L = \{a\}$. Thus:

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

## Exercise: formal languages

Given

- the alphabet $\Sigma_{\text{bin}}$,
- the function $n(w)$ mapping binary words to numbers as defined on page 39, and
- the language $L = \{1\}$,

formally describe the following:

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

# Outline

# Outline

# Regular expressions

Compact and convenient way to represent a set of strings

- ▶ Characterize tokens for compilers
- ▶ Describe search terms for a database
- ▶ 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**

## Introductory Examples

Consider $\Sigma_{bin} = \{0, 1\}$. With regular expressions we can conveniently and rigorously describe many languages:

- $1(0 + 1)^*$ – all words beginning with a 1 (also known as $L_{\mathbb{N}} \setminus \{0\}$)
- $11^*$ – all words consisting of one ore more letters $1$ (*M* from the last exercise)
- $0(10)^* + (10)^* + 1(01)^* + (01)^*$ – the language of all words where no two subsequent characters are the same

# 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 $\Sigma$ is an alphabet, $2^{\Sigma^*}$ is the set of all subsets of $\Sigma^*$ and hence the set of all possible formal languages over $\Sigma$.

## Example

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

# Regular expressions and formal languages

A regular expression over $\Sigma$...

- ▶ ...is a word over the extended alphabet $\Sigma \cup \{\emptyset, \varepsilon, +, \cdot, *, (, )\}$
  - ▶ Note that we implicitly assume that $\{\emptyset, \varepsilon, +, \cdot, *, (, )\} \cap \Sigma = \{\}$
  - ▶ $\emptyset$ denotes a regular expression (syntax)
  - ▶ $\{\}$ denotes the empty set (semantics)
- ▶ ...describes a formal language over $\Sigma$

## Terminology

The following terms are defined on the next slides:

- ▶ $R_\Sigma$ is the set of all regular expressions over the alphabet $\Sigma$.
- ▶ 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 $R_\Sigma$ over the alphabet $\Sigma$ is defined as follows:

1. The regular expression $\emptyset$ denotes the empty language.
   $\emptyset \in R_\Sigma$   and   $L(\emptyset) = \{\}$
2. The regular expression $\varepsilon$ denotes the language containing only the empty word.
   $\varepsilon \in R_\Sigma$   and   $L(\varepsilon) = \{\varepsilon\}$
3. Each symbol in the alphabet $\Sigma$ 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:
  $(\ldots) \;\; > \;\; * \;\; > \;\; \cdot \;\; > \;\; +$
- ▶ The product operator $\cdot$ can be omitted.

## Example

$$a + b \cdot c^* \;\; \doteq \;\; a + (b \cdot (c^*))$$
$$ac + bc^* \;\; \doteq \;\; a \cdot c + b \cdot c^*$$

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

# Examples for regular expressions

## Example

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

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

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

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

$$L(r_2) = \{w \in \Sigma_{\text{abc}}^* \mid |w| \geq 1\}$$

## Exercise: regular expressions

1. Using the alphabet $\Sigma_{abc} = \{a, b, c\}$, give a regular expression $r_1$ for all the words $w \in \Sigma_{abc}^*$ containing exactly one $a$ or exactly one $b$.

2. Formally describe $L(r_1)$ as a set.

3. Using the alphabet $\Sigma_{abc} = \{a, b, c\}$, give a regular expression $r_2$ for all the words containing at least one $a$ and at least one $b$.

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

5. Using the alphabet $\Sigma_{bin}$, give a regular expression for all the words not containing the string $110$.

6. Which language is described by the regular expression

$$r_6 = (1 + \varepsilon)(00^*1)^*0^*?$$

# 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 ($\emptyset r \doteq \emptyset$).

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

$\square$

# 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)

$$u^*u \doteq uu^*$$

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

$$
\begin{aligned}
u^*u &\doteq (\varepsilon + u^*u)u && \text{(by 13. } (r)^* \doteq \varepsilon + (r)^*r) \\
&\doteq (u^*u + \varepsilon)u && \text{(by 1. } r_1 + r_2 \doteq r_2 + r_1) \\
&\doteq u^*uu + u && \text{(by 7. } (r_1 + r_2)r_3 \doteq r_1 r_3 + r_2 r_3) \\
&\doteq uu^* && \text{(by 15. } r \doteq rs + t \text{ with } r = u^*u, s = u, t = u)
\end{aligned}
$$

$\square$

## Lemma: Kleene Star (2)

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

We show $L(u^*u) = L(u^*) = L(uu^*)$

a) Proof of $L(u^*u) \subseteq L(u^*)$
$L(u^*u) = L(u^*) \cdot L(u)$
$= (L(u))^* \cdot L(u)$
$= (\bigcup_{i \geq 0} L(u)^i) \cdot L(u)$
$= \bigcup_{i \geq 0} (L(u)^i \cdot L(u))$
$= \bigcup_{i \geq 1} L(u)^i$
$\subseteq L(u^*)$

b) Proof of $L(u^*u) \supseteq L(u^*)$
$L(u^*u) = \{tv \mid t \in L(u^*), v \in L(u)\}$
$\supseteq \{tv \mid t \in L(u^*), v = \varepsilon\}$
$= \{t \mid t \in L(u^*)\}$
$= L(u^*)$

▶ a) and b) imply $L(u^*u) = L(u^*)$
▶ $L(uu^*) = L(u^*)$: strictly analoguous

$\square$

# A note on Arto/Arden

- Arto: $\varepsilon \notin L(s)$ 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 Arto, 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 regular expression $s$ using only algebraic operations:

$$s = 0(\varepsilon + 0 + 1)^* + (\varepsilon + 1)(1 + 0)^* + \varepsilon.$$

**3** Prove the equivalence using only algebraic equivalences:

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

# 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
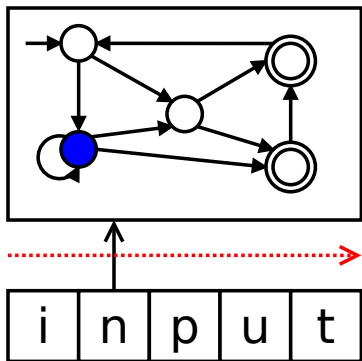


Deterministic finite automaton (DFA)

- ▶ 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 $\mathtt{a^*ba^*}$.
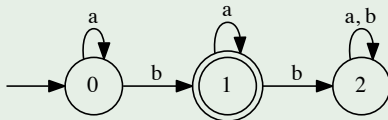


- ▶ $\mathcal{A}$ has three states, 0, 1 and 2.
- ▶ It operates on the alphabet $\{a, b\}$.
- ▶ The transition function is indicated by the arrows.
- ▶ 0 is the initial state (with an arrow "pointing at it from anywhere").
- ▶ 1 is an accepting state (represented as a double circle).
- ▶ 2 is also called a junk state (once it is reached, the word can never be accepted).

# 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.
- $\Sigma$ is the (finite) input alphabet.
- $\delta : Q \times \Sigma \to Q$ is the transition function.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of final (or accepting) states.

Notes:

- $\delta$ is a total function – there has to be a transition from every state for every letter
- ... but automata with partial transition functions can be "repaired" into a proper DFA by adding a junk state

# Formal definition of $\mathcal{A}$

## Example
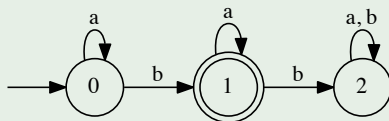


$\mathcal{A}$ is expressed as $(Q, \Sigma, \delta, q_0, F)$ with

- $Q = \{0, 1, 2\}$
- $\Sigma = \{\mathtt{a}, \mathtt{b}\}$
- $\delta(0, \mathtt{a}) = 0; \delta(0, \mathtt{b}) = 1, \delta(1, \mathtt{a}) = 1; \delta(1, \mathtt{b}) = \delta(2, \mathtt{a}) = \delta(2, \mathtt{b}) = 2$
- $q_0 = 0$
- $F = \{1\}$

# Language accepted by an DFA

## Definition (Language accepted by an automaton)

The state transition function $\delta$ is generalised to a function $\delta'$ whose second argument is a word as follows:
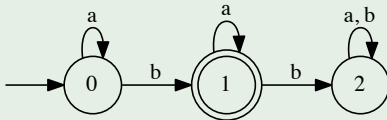
- $\delta' : Q \times \Sigma^* \to Q$
- $\delta'(q, \varepsilon) = q$ for every $q \in Q$
- $\delta'(q, wc) = \delta(\delta'(q, w), c)$   with   $c \in \Sigma; w \in \Sigma^*$

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

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

## Example



- $\delta'(0, aa) = \delta(\delta'(0, a), a) = \delta(\delta(\delta'(0, \varepsilon), a), a) = 0$
- $\delta'(1, aaa) = 1$
- $\delta'(0, bb) = \delta'(1, b) = 2$
- $L(\mathcal{A}) = \{w \in \{a, b\}^* \mid w = a^n b a^m \text{ and } n, m \in \mathbb{N}\}$

# Run of a DFA

## Definition (Configuration, Run)

Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA.

A configuration of $\mathcal{A}$ is a pair $(q, w)$ with $q \in Q$ and $w \in \Sigma^*$.

A run of $\mathcal{A}$ on a word $w = c_1 \cdot c_2 \cdots c_n$ is a sequence of configurations:

$$((q_0, c_1 \cdot c_2 \cdots c_n), (q_1, c_2 \cdots c_n), \ldots, (q_n, \varepsilon))$$
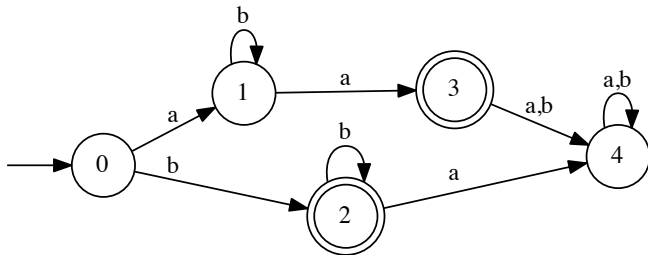
where

- $q_i \in Q$ holds for $1 \le i \le n$ and
- $\delta(q_{i-1}, c_i) = q_i$ holds for $1 \le i \le n$.

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 $\mathcal{A}$.
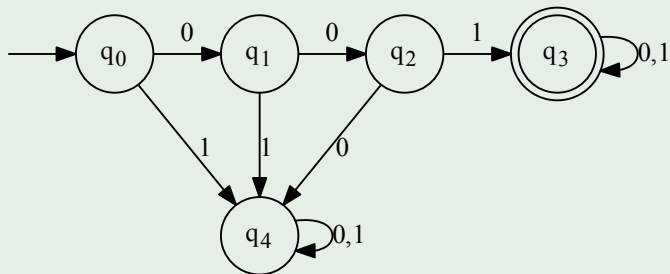
# Exercise: DFA

**2** Give

- ► a regular expression,
- ► a graphical representation of a DFA, and
- ► a formal definition

for the language $L \subset \{a, b\}^*$ containing all those words featuring the substring ab

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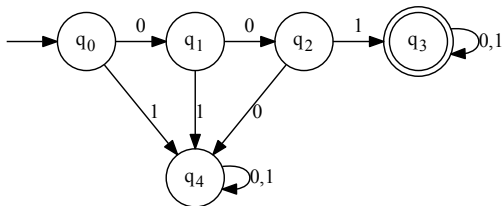b) at an arbitrary position,
c) at the end.

# 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: $q_0$

- $F = \{q_3\}$

| | $\delta$ | 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
```

- 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
   ▶ $\Sigma = \{a, b, c\}$
   ▶ $L_1 = \{ubw \mid u \in \Sigma^*, w \in \Sigma\}$
   ▶ $L_2 = \{ubw \mid u \in \Sigma, w \in \Sigma^*\}$

▶ 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):
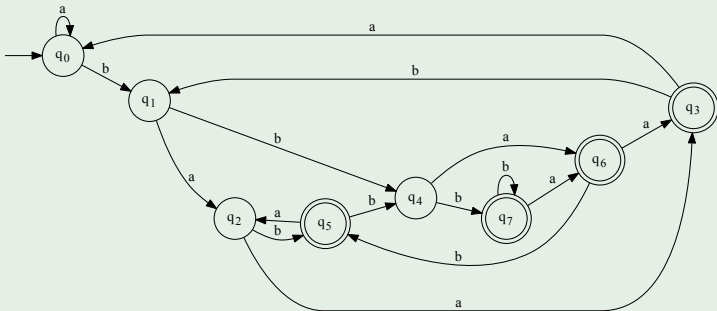  Find a DFA $\mathcal{A}$ with $L(\mathcal{A}) = L_2$

End lecture 4

# Outline

# Drawbacks of deterministic automata

Deterministic automata:
- ► Transition function $\delta$
  - ► exactly one transition from every configuration
- ► can be complex even for simple languages

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

# 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
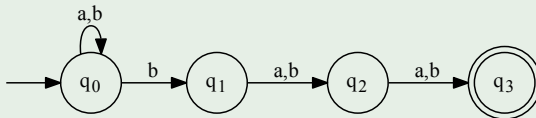  ▶ and always chooses the "right" one
▶ This is called a non-deterministic finite automaton (NFA)

## Example (NFA $\mathcal{B}$ for $(a + b)^*b(a + b)(a + b)$)
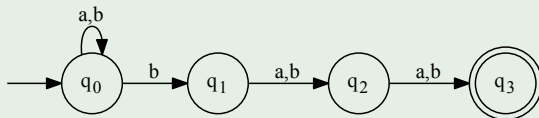
# Non-Deterministic automata

## 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 $\varepsilon$-transitions

- ▶ Non-deterministic transitions allow an NFA to go to more than one successor state
  - ▶ Instead of a function $\delta$, an NFA has a transition relation $\Delta$
- ▶ 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 $\varepsilon$-transition
  - ▶ Thus, $\Delta$ is a relation on $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$

## Example (NFA with $\varepsilon$-transitions)

# NFA: Formal definition
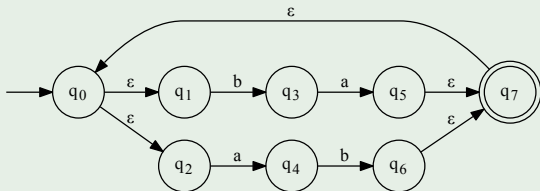
## Definition (NFA)

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. $\Sigma$ is the input alphabet.
3. $\Delta$ is a relation over $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$.
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 (Configuration, Run of an NFA)

Let $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ be an NFA.

A configuration of $\mathcal{A}$ is a pair $(q, w)$ as for a DFA.

A run of $\mathcal{A}$ on a word $w_0 = c_1 \cdot c_2 \cdots c_n$ is a sequence of transitions

$$r = ((q_0, w_0), (q_1, w_1), \ldots, (q_m, \varepsilon))$$

such that the following conditions are satisfied:

- $q_i \in Q$ for all $1 \leq i \leq m$,
- if $r$ contains the configurations $(q_i, w_i), (q_{i+1}, w_{i+1})$, then
  - there is a transition $(q_i, c, q_{i+1}) \in \Delta$ with $c \in \Sigma \cup \{\varepsilon\}$
  - $w_i = c \cdot w_{i+1}$.

The run $r$ is accepting if $q_m$ is a final state.

The slightly more complex definition is necessary to handle $\varepsilon$-transitions.

# Language recognized by an NFA

## Definition (Language recognized by an NFA)

Let $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ be an NFA. The language accepted by $\mathcal{A}$ is
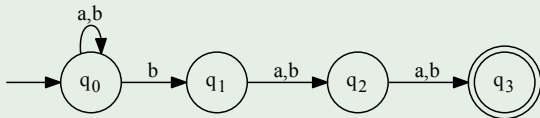
$$L(\mathcal{A}) = \{w \mid \text{there is an accepting run of } \mathcal{A} \text{ on } w\}$$

Note:

▶ Only existence of one accepting run is required
▶ 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

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

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

$F = \{q_3\}$

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

| $\Delta$ | $\mathtt{a}$ | $\mathtt{b}$ | $\varepsilon$ |
|---|---|---|---|
| $q_0$ | $\{q_0\}$ | $\{q_0, q_1\}$ | $\{\}$ |
| $q_1$ | $\{q_2\}$ | $\{q_2\}$ | $\{\}$ |
| $q_2$ | $\{q_3\}$ | $\{q_3\}$ | $\{\}$ |
| $q_3$ | $\{\}$ | $\{\}$ | $\{\}$ |

## Exercise: NFA

Develop an NFA $\mathcal{A}$ whose language $L(\mathcal{A}) \subset \{a, b\}^*$ contains all those words featuring the substring `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 $\varepsilon$ closure of a state
- $\delta^*$ to compute possible successors of a state
- $\hat{\delta}$, the new transition function for the generated DFA

# Step 1: $\varepsilon$ closure of an NFA

The $\varepsilon$ closure of a state $q$ contains all states the NFA can change to by means of $\varepsilon$ transitions starting from $q$.
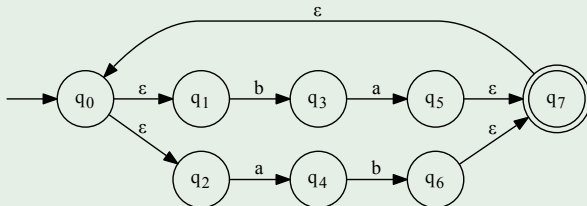
### Definition ($\varepsilon$ closure)

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: $\varepsilon$ 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 $\delta^*$ maps

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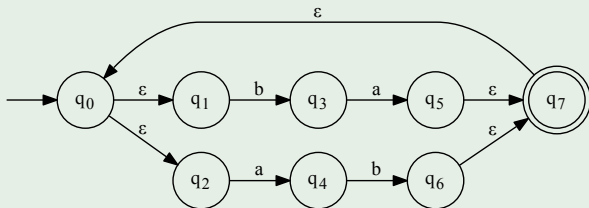- followed by any number of $\varepsilon$ transitions.

## 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



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

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

# 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 $\Delta$ by reading the character $c$
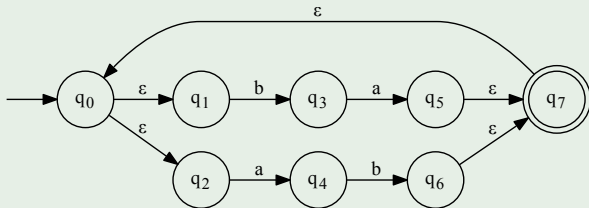- ▶ possibly followed by $\varepsilon$ transitions.

## 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 auxiliary functions $ec$, $\hat{\delta}$, 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

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

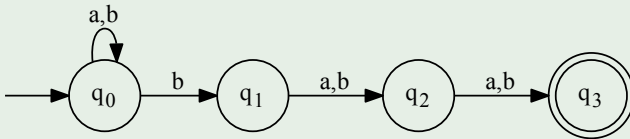with $\hat{F} = \{M \in 2^Q \mid M \cap F \neq \{\}\}$.

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

## Remark

In practice, we use a more efficient stepwise construction that only builds the *reachable* states, not all of $2^Q$!

# Example: transformation into DFA

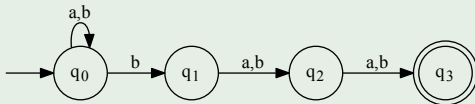## Example (NFA $\mathcal{B}$ for $(a + b)^*b(a + b)(a + b)$)



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

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

# Example: transformation into DFA (cont')

## Example



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

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

# 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 $\hat{F} = \{S_3, S_5, S_6, S_7\}$.
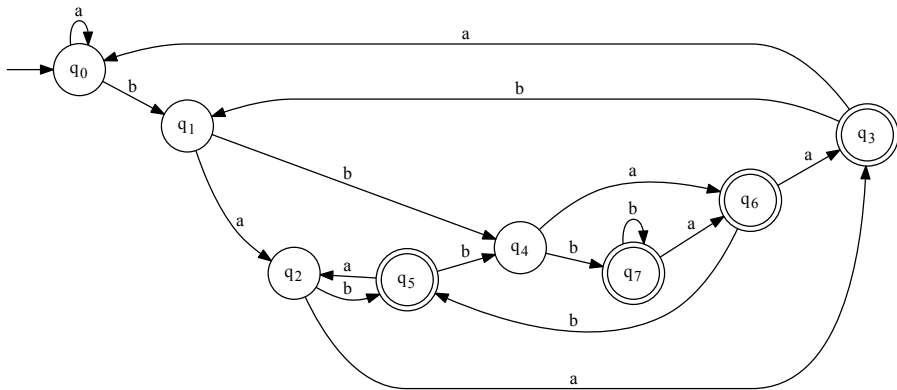
## Exercise: Transformation into DFA

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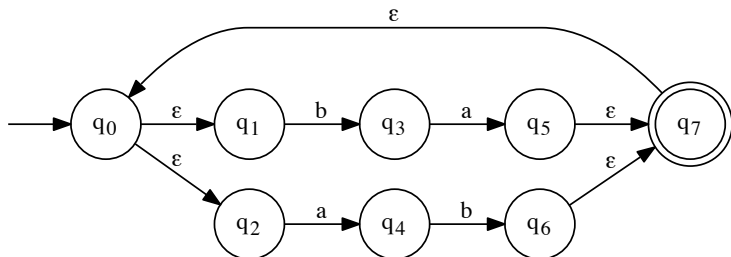


a) Determine $\det(\mathcal{A})$.

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

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

Solution   End lecture 5

106

# 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 $\varepsilon$-transitions!**

# Reminder: Regular Expression

Let $\Sigma$ be an alphabet.

- The elementary regular expressions over $\Sigma$ are:
    - $\emptyset$ with $L(\emptyset) = \{\}$
    - $\varepsilon$ with $L(\varepsilon) = \{\varepsilon\}$
    - $c \in \Sigma$ with $L(c) = \{c\}$
- Let $r_1$ and $r_2$ be regular expressions over $\Sigma$.
  Then the following are also regular expressions over $\Sigma$:
    - $r_1 + r_2$ 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 $\Sigma$ be the alphabet which $r$ is based on.

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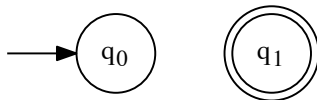


**2** $\mathcal{A}(\varepsilon) = (\{q_0, q_1\}, \Sigma, \{(q_0, \varepsilon, q_1)\}, q_0, \{q_1\})$
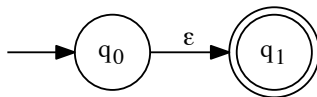


**3** $\mathcal{A}(c) = (\{q_0, q_1\}, \Sigma, \{(q_0, c, q_1)\}, q_0, \{q_1\})$ for all $c \in \Sigma$
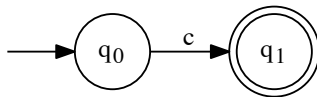
# NFAs for composite REs (general)

▶ Assume in the following:
  ▶ $\mathcal{A}(r_1) = (Q_1, \Sigma, \Delta_1, q_1, \{q_2\})$
  ▶ $\mathcal{A}(r_2) = (Q_2, \Sigma, \Delta_2, q_3, \{q_4\})$
  ▶ $Q_1 \cap Q_2 = \{\}$
  ▶ $q_0, q_5 \notin Q_1 \cup Q_2$

▶ $\mathcal{A}(r_1)$ 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
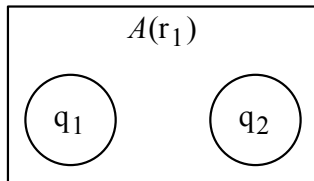
# NFAs for composite REs (concatenation)

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



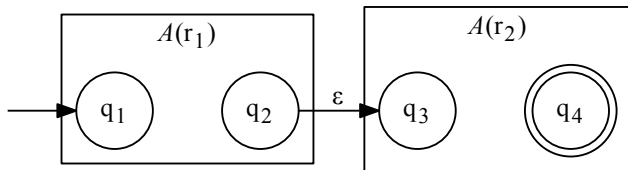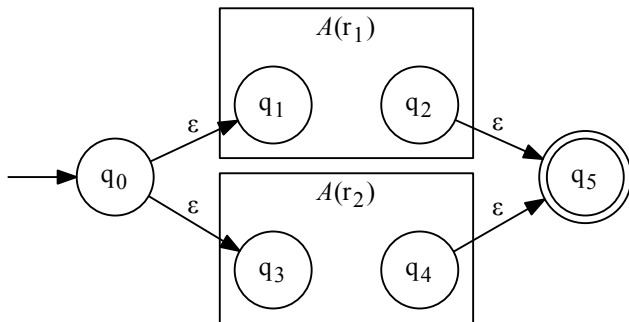Reminder:

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

# NFAs for composite REs (Kleene Star)

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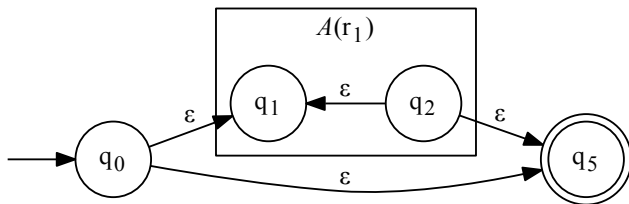


Reminder:

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

# 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

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

Solution

# 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:

$$L(r(\mathcal{A})) = L(\mathcal{A})$$
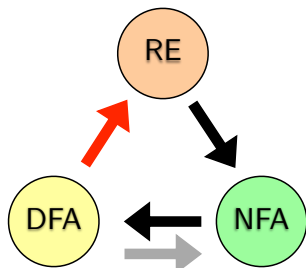
# Convert DFA into RE

- ▶ 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.
    - ▶ Method:
        - ▶ For each transition with $c$ to $q'$: generate alternative $c \cdot L_{q'}$
        - ▶ For final states: additionally $\varepsilon$
- ▶ Solve the resulting system for $L_{q_0}$
    - ▶ Result: RE describing $L_{q_0} = L(\mathcal{A})$
- ▶ Convention:
    - ▶ States are named $\{0, 1, \ldots, n\}$
    - ▶ Initial state is $0$

# Convert DFA to RE: Example
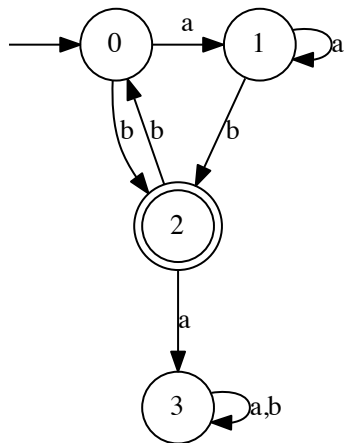


- $L_0 \doteq aL_1 + bL_2$
- $L_1 \doteq aL_1 + bL_2$
- $L_2 \doteq aL_3 + bL_0 + \varepsilon$
- $L_3 \doteq (a + b)L_3$

4 equations, 4 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
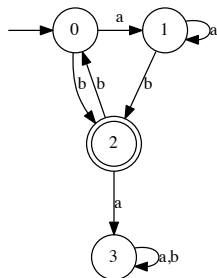


- $L_0 \doteq aL_1 + bL_2$
- $L_1 \doteq aL_1 + bL_2$
- $L_2 \doteq aL_3 + bL_0 + \varepsilon$
- $L_3 \doteq (a+b)L_3$

$$
\begin{aligned}
L_3 &\doteq (a+b)L_3 + \emptyset && \text{[neutral el.]} \\
&\doteq (a+b)^*\emptyset && \text{[Arden]} \\
&\doteq \emptyset && \text{[absorbing el.]} \\
L_2 &\doteq a\emptyset + bL_0 + \varepsilon && \text{[replace } L_3] \\
&\doteq \emptyset + bL_0 + \varepsilon && \text{[absorbing el.]} \\
&\doteq bL_0 + \varepsilon && \text{[neutral el.]} \\
L_1 &\doteq aL_1 + b(bL_0 + \varepsilon) && \text{[replace } L_2] \\
&\doteq a^*b(bL_0 + \varepsilon) && \text{[Arden]}
\end{aligned}
$$
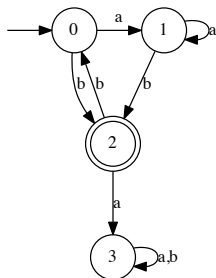
- $L_0 \doteq aL_1 + bL_2$
- $L_1 \doteq a^*b(bL_0 + \varepsilon)$
- $L_2 \doteq bL_0 + \varepsilon$
- $L_3 \doteq \emptyset$

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

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:
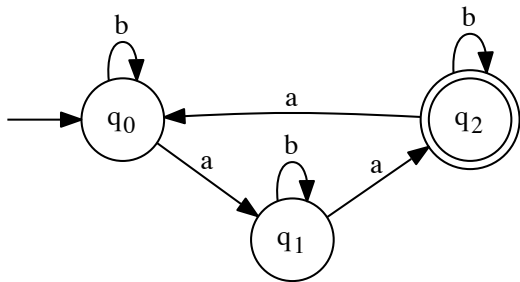
# Resume: Finite automata and regular expressions

- ▶ 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!**
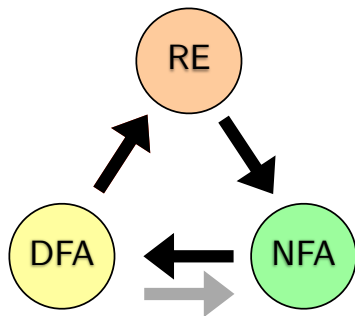
and now it's time for something completely different

End lecture 6

# 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.

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

- ▶ $\{(p, q) \in (Q \times Q) \mid p \in F, q \notin F\} \subseteq V$
- ▶ $\{(p, q) \in (Q \times Q) \mid p \notin F, q \in F\} \subseteq V$
- ▶ 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:

$$V = \{(p, q) \in Q \times Q \mid (p \in F \land q \notin F) \lor (p \notin F \land q \in F)\}.$$

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:
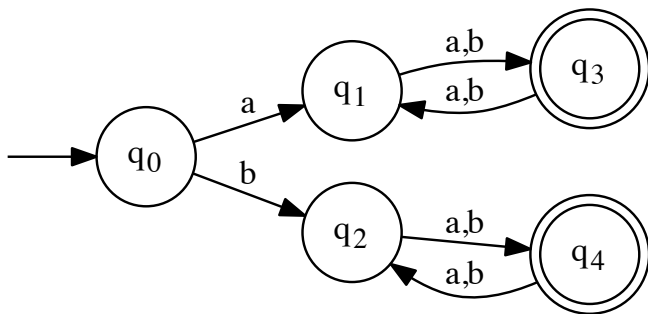
This is the formal definition of the DFA:

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

with

1. $Q = \{q_0, q_1, q_2, q_3, q_4\}$
2. $\Sigma = \{\mathtt{a}, \mathtt{b}.\}$
3. $\delta = \dots$ (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 $\circ$

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

|       | $q_0$    | $q_1$    | $q_2$    | $q_3$    | $q_4$    |
|-------|----------|----------|----------|----------|----------|
| $q_0$ |          |          |          | $\times$ | $\times$ |
| $q_1$ |          |          |          | $\times$ | $\times$ |
| $q_2$ |          |          |          | $\times$ | $\times$ |
| $q_3$ | $\times$ | $\times$ | $\times$ |          |          |
| $q_4$ | $\times$ | $\times$ | $\times$ |          |          |
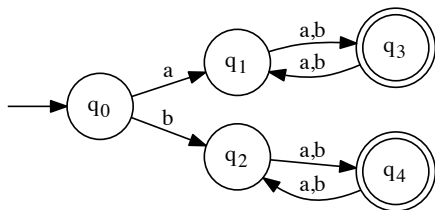
2. The elements of $\{(q_i, q_i) \mid i \in \{0, \cdots, 4\}$ are not contained in $V$ since every state is indistinguishable from itself:

|       | $q_0$ | $q_1$ | $q_2$ | $q_3$ | $q_4$ |
|-------|-------|-------|-------|-------|-------|
| $q_0$ | ○     |       |       | ×     | ×     |
| $q_1$ |       | ○     |       | ×     | ×     |
| $q_2$ |       |       | ○     | ×     | ×     |
| $q_3$ | ×     | ×     | ×     | ○     |       |
| $q_4$ | ×     | ×     | ×     |       | ○     |
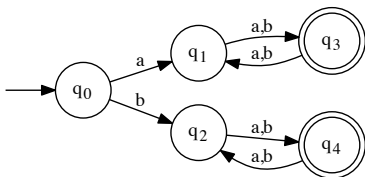


There are eight remaining empty fields. Since the table is symmetric, four pairs of states have to be checked.

**3** Check the transitions of every remaining state-pair for every letter.
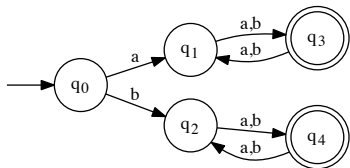
**1** $\delta(q_0, \mathtt{a}) = q_1; \delta(q_1, \mathtt{a}) = q_3; (q_1, q_3) \in V \to (q_0, q_1), (q_1, q_0) \in V$

**2** $\delta(q_0, \mathtt{a}) = q_1; \delta(q_2, \mathtt{a}) = q_4; (q_1, q_4) \in V \to (q_0, q_2), (q_2, q_0) \in V$

**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)

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 ○:

|       | $q_0$ | $q_1$ | $q_2$ | $q_3$ | $q_4$ |
|-------|-------|-------|-------|-------|-------|
| $q_0$ | ○ | × | × | × | × |
| $q_1$ | × | ○ | ○ | × | × |
| $q_2$ | × | ○ | ○ | × | × |
| $q_3$ | × | × | × | ○ | ○ |
| $q_4$ | × | × | × | ○ | ○ |

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

▶ $(q_1, q_2)$,
▶ $(q_3, q_4)$.

► This is the minimized DFA after merging indistinguishable states:
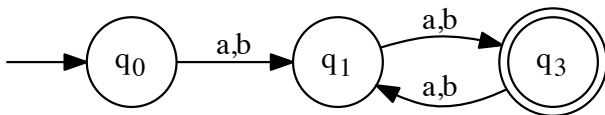
Derive a minimal DFA accepting the language

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

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) 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):

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

Solution    End lecture 7

# 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:
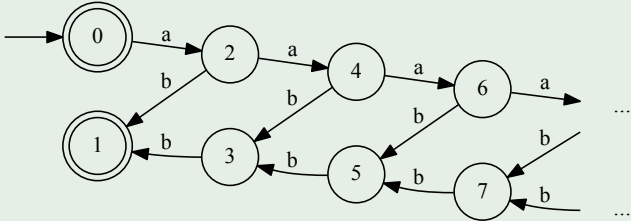


- ▶ Is there a better solution?
- ▶ If no, how can this be shown?

# Pumping Lemma: Idea

1. Every regular language $L$ is accepted by a 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 Automaton $\mathcal{C}$)
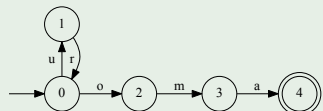


- $\mathcal{C}$ accepts `uroma`
- $\mathcal{C}$ also accepts `ururur...oma`

# 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
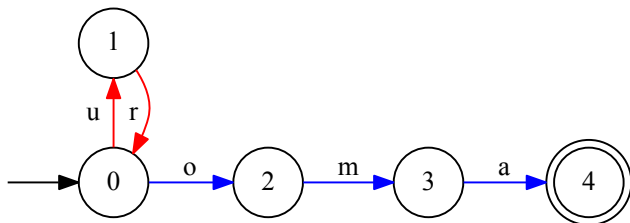


- ▶ $\mathcal{C}$ has 5 states $\hfill k = 5$
- ▶ uroma has 5 letters $\hfill s = \texttt{uroma}$
- ▶ There is a segmentation $s = u \cdot v \cdot w$ $\quad u = \varepsilon \quad v = \texttt{ur} \quad w = \texttt{oma}$
- ▶ such that $|v| \neq \varepsilon$ $\hfill v = \texttt{ur}$
- ▶ and $|u \cdot v|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

# 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)

$$A \to B \quad \Leftrightarrow \quad \neg B \to \neg A$$

## Contraposition of the Pumping Lemma

The Pumping Lemma in formal logic:

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

Contraposition of the PL:

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

After pushing negation inward and doing some propositional transformations:

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

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

If for each natural 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 $a$s
  - $v \neq \varepsilon$ 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: The Pumping Game

Play the pumping game at `http://weitz.de/pump/`.

# 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 $a^p$ where $p$ is a prime number:

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

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. $\Sigma^* \setminus L_1$? | (closure under complement) |
| $L_1^*$? | (closure under Kleene-star) |

**Meta-Question: How do we answer these questions?**

# Closure properties (Theorem)

## Theorem (Closure properties of regular languages)

*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.* $\Sigma^* \setminus L_1$
- $L_1^*$

## 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,
$\varepsilon$-transitions to initial/final states of $\mathcal{A}_{L_1}$ and $\mathcal{A}_{L_2}$

$L_1 \cdot L_2$  $\varepsilon$-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 $\varepsilon$-transitions),
- ▶ $\varepsilon$-transitions from the original final states to the original initial state,
- ▶ $\varepsilon$-transition from the new initial to the new final state.

# Visual refresher



$L_1 \circ L_2$
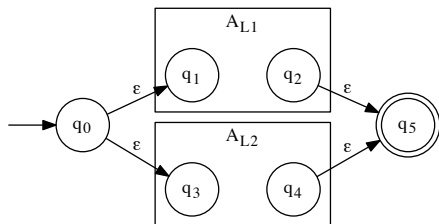
$L_1 \cup L_2$
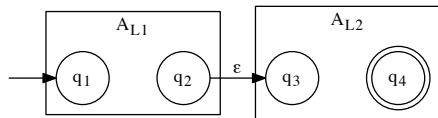
$L_1^*$
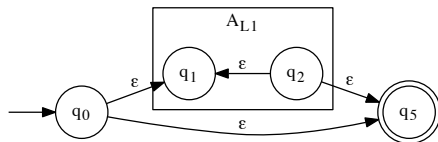
# Closure under intersection

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:

- $Q = Q_1 \times Q_2$
- $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ 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 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.

# Exercise: Product automaton

Generate automata for

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

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

# Closure under complement

## Theorem (Closure under complement)

*Let $L$ be a regular language over $\Sigma$. Then $\overline{L} = \Sigma^* \backslash L$ is regular.*

Let $\mathcal{A}_L = (Q, \Sigma, q_0, \delta, F)$ be a DFA for the language $L$.

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 accepting and non-accepting states.**

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.

# 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 $\varepsilon$-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 $L$. $\qquad\square$

# Example: finite language

## Example ($L = \{if, then, else, while, goto, for\}$ over $\Sigma_{\mathsf{ASCII}}$)
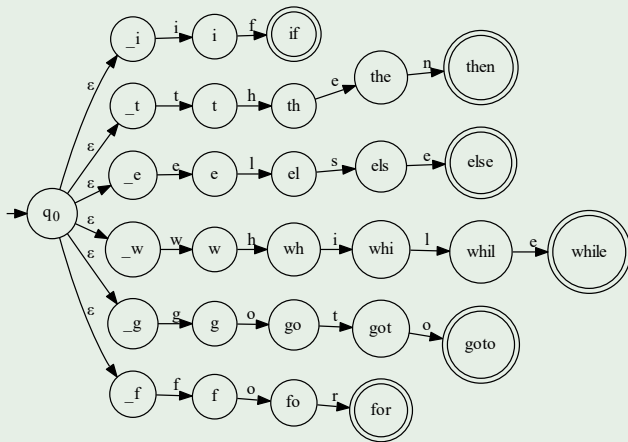
# Finite languages and regular expressions

## 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 $w_1 + w_2 + \ldots + w_n$. $\qquad\Box$

## Corollary

*The class of finite languages is characterised by*

▶ *acyclic NFAs (or DFAs that have no cycles on any path from the initial state to an accepting state)*

▶ *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 (accepting) state is reachable, then $L \neq \{\}$ holds.

□

# Group exercise: Emptiness problem

► Find an alternative algorithm for the checking emptiness, using the results from the chapter on equivalence.

# 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:

- $q_1 := \delta(q_0, c_1)$
- $q_2 := \delta(q_1, c_2)$
- $\ldots$
- 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:

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

$\square$

# 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.

$\square$

Note that $\mathcal{A}_f$ may be a NFA (because it misses some transitions)

## Exercise: Finiteness

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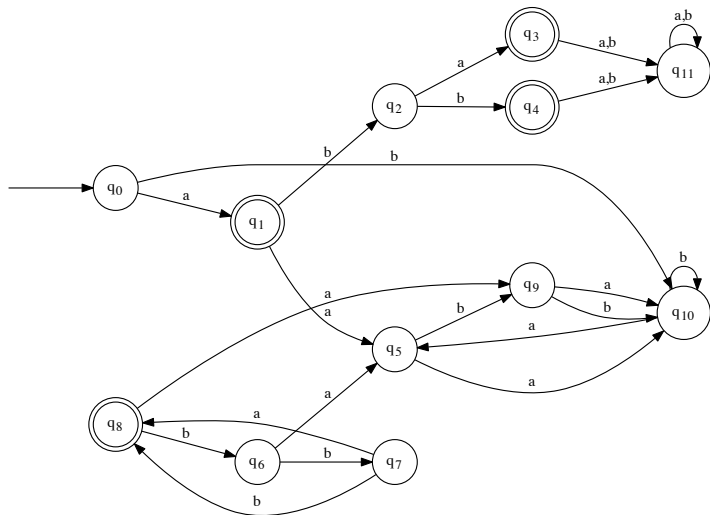
## Regular languages: summary

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)

End lecture 9

# Outline

# 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, 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

Sequence of characters:
i,n,t, ␣, a,,, b, ;, a, =, b, +, 1, ;

Lexical analysis
(tokeniser)

Sequence of tokens:
(id, "int"), (id, "a"), (id, "b"), (semicolon), (id, "a"), (eq), (id, "b"), (plus), (int, "1"), (semicolon)

Syntactic analysis
(parser)

e.g. Abstract syntax tree

Semantic analysis

e.g. AST+symbol table

| Variable | Type |
| --- | --- |
| a | int |
| b | int |

Code generation
(several optimisation passes)

e.g. assembler code

```
ld a,b
ld c, 1
add c
...
```
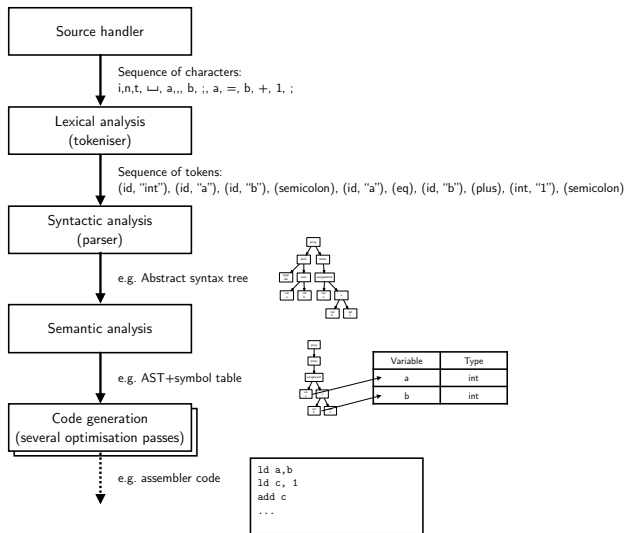
# Source Handler

- Handles input files
- Provides character-by-character access
- May maintain file/line/column (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
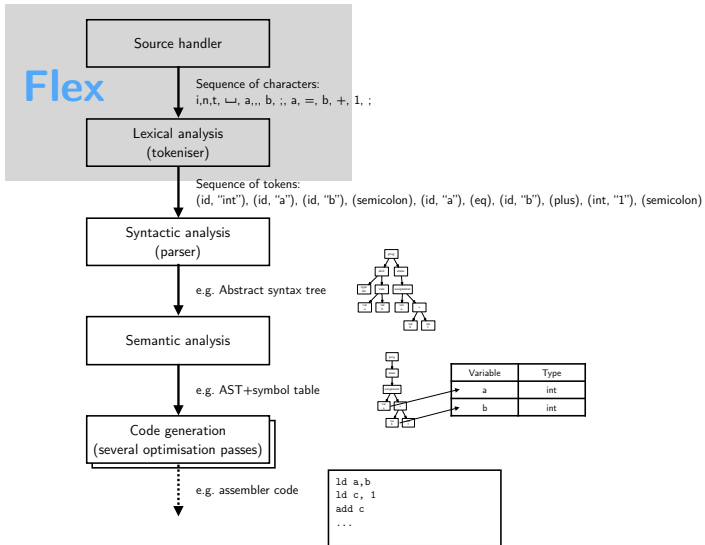
# Flex Overview

- ▶ 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

Development time

```
Definitions
Rules
Miscellanous code
```

flex+gcc

Input → scanner → Tokenized/processed output
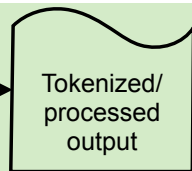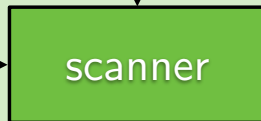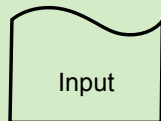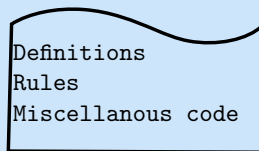
Execution time

# 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/`
  `fla2018.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

$$O = \{ ., \star, +, ?, -, \,\tilde{}\,, |, (, ), [, ], \{, \}, <, >, /, \backslash, \hat{}\,, \$, " \}$$

and the set of regular expressions

1. $c \in \Sigma_{\text{ASCII}} \backslash O \longrightarrow c \in R$
2. "."$\in R$
   any character but newline ($\backslash$n)

## Regular Expressions in Practice (2)

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

   > $\backslash \mathtt{a}$ (alert)
   > $\backslash \mathtt{b}$ (backspace)
   > $\backslash \mathtt{f}$ (form feed)
   > $\backslash \mathtt{n}$ (newline)
   > $\backslash \mathtt{r}$ (carriage return)
   > $\backslash \mathtt{t}$ (tabulator)
   > $\backslash \mathtt{v}$ (vertical tabulator)

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

5. $c \in O \longrightarrow \backslash c \in R$
   escaping operator symbols
6. $r_1, r_2 \in R \longrightarrow r_1 r_2 \in R$
   concatenation
7. $r_1, r_2 \in R \longrightarrow r_1 \,|\, r_2 \in R$
   infix operation using "|" rather than "+"
8. $r \in R \longrightarrow r\star \in R$
   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. $r \in R; \ m, n \in \mathbb{N}; \ m \le n \longrightarrow r\{m, n\} \in R$
    concatenation of between $m$ and $n$ times $r$
13. $r \in R \longrightarrow \ {}^\wedge r \in R$
    $r$ 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. $w \in \{\Sigma_{\text{ASCII}} \backslash \{\backslash, \texttt{"}\}\}^* \longrightarrow \texttt{"}w\texttt{"} \in R$
    verbatim text (no escape sequences)

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. "*", "+", "?"
    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 $\Sigma_{ascii}$, 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);
}
\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:
  ▶ https://github.com/westes/flex
  ▶ Manual: https://westes.github.io/flex/manual/
  ▶ REs: https://westes.github.io/flex/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/fla2018.html`

End lecture 10

220

# 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 $\Sigma$,
3. the set of production rules $P$ of the form

$$\alpha \rightarrow \beta$$

with $\alpha \in V^*NV^*, \beta \in V^*, V = N \cup \Sigma$

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

# Noam Chomsky (*1928)

- ▶ Linguist, philosopher, logician, . . .
- ▶ BA, MA, PhD (1955) at the University of Pennsylvania
- ▶ Mainly teaching at MIT (since 1955)
  - ▶ Also Harvard, Columbia University, Institute of Advanced Studies (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
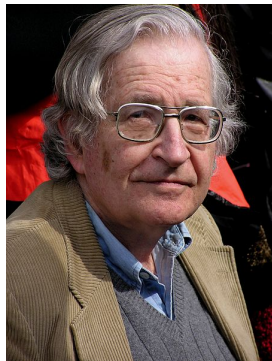
# Grammar for `C` identifiers

## 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 (_)

$$N = \{S, R, L, D\} \text{ (start, rest, letter, digit)},$$
$$\Sigma = \{\texttt{a}, \ldots, \texttt{z}, \texttt{A}, \ldots, \texttt{Z}, \texttt{0}, \ldots, \texttt{9}, \_\},$$
$$P = \{ \qquad S \quad \rightarrow \quad LR|\_R$$
$$R \quad \rightarrow \quad LR|DR|\_R|\varepsilon$$
$$L \quad \rightarrow \quad \texttt{a}|\ldots|\texttt{z}|\texttt{A}|\ldots|\texttt{Z}$$
$$D \quad \rightarrow \quad \texttt{0}|\ldots|\texttt{9}\}$$

$\alpha \rightarrow \beta_1|\ldots|\beta_n$ 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

$$G \text{ derives } y \text{ from } x \text{ in one step} \quad (x \Rightarrow_G y) \text{ iff}$$

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

Moreover, we say that

$$G \text{ derives } y \text{ from } x \quad (x \Rightarrow_G^* y) \text{ iff}$$

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

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

# Grammars and derivations

## Example ($G_3$)

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

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

Derivations of $G_3$ have the general form

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow \cdots \Rightarrow a^n S \Rightarrow a^n$$

The language produced by $G_3$ is

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

# Grammars and derivations (cont')

## Example ($G_2$)

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

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

Derivations of $G_2$:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow \cdots \Rightarrow a^n S b^n \Rightarrow a^n b^n.$$

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

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

## Example ($G_1$)

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

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

$$
\begin{array}{rll}
S & \to aSBC & 1 \\
S & \to aBC & 2 \\
CB & \to BC & 3 \\
aB & \to ab & 4 \\
bB & \to bb & 5 \\
bC & \to bc & 6 \\
cC & \to cc & 7
\end{array}
$$

### Example ($G_1$)

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

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

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

- ▶ Give derivations for 3 different words in $L(G_1)$
- ▶ Can you characterize $L(G_1)$?

Derivations of $G_1$:

$$S \Rightarrow_1 aSBC \Rightarrow_1 aaSBCBC \Rightarrow_1 \cdots \Rightarrow_1 a^{n-1}S(BC)^{n-1} \Rightarrow_2 a^n(BC)^n$$

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

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

▶ 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

## 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

▶ Grammars of the type defined on the last slide were originally called monotonic or non-contracting by Chomsky

▶ Context-sensitive grammars addtionally have to satisfy:

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

  ▶ rule application can depend on a context $\alpha_1$, $\alpha_2$
  ▶ context cannot be modified (or moved)
  ▶ only one NTS can be modified

▶ every monotonic grammar can be rewritten as context-sensitive
  ▶ $AB \rightarrow BA$ is not context-sensitive, but $AB \rightarrow AY \rightarrow XY \rightarrow XA \rightarrow BA$
  ▶ if terminal symbols are involved: replace $S \rightarrow aB \rightarrow ba$ with $S \rightarrow N_a B \rightarrow \ldots N_b N_a \rightarrow b N_a \rightarrow ba$

▶ 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 (for both grammars and languages)

# 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

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

- ▶ 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

$$A \to aB$$

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

# Formal grammars and formal languages

## 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

Type 0
(unrestricted)

Type 1
(context-sensitive)

Type 2
(context-free)

Type 3
(right linear)
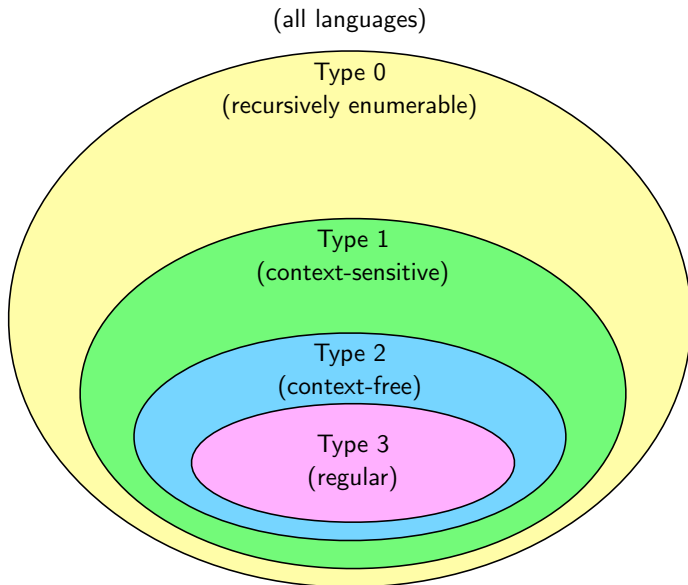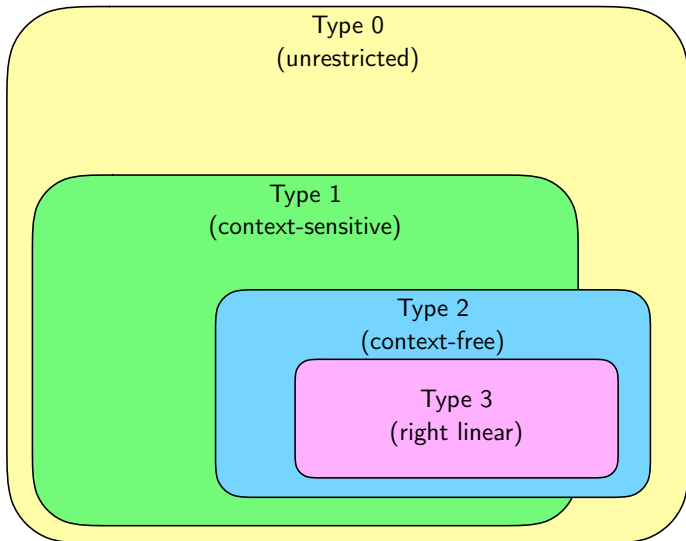
# The Chomsky hierarchy: examples

## Example (C identifiers revisited)

$$
\begin{aligned}
S &\rightarrow LR \mid \_R \\
R &\rightarrow LR \mid DR \mid \_R \mid \varepsilon \\
L &\rightarrow \text{a} \mid \ldots \mid \text{z} \mid \text{A} \mid \ldots \mid \text{Z} \\
D &\rightarrow \text{0} \mid \ldots \mid \text{9}
\end{aligned}
$$

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

$$
\begin{aligned}
S &\rightarrow \text{A}R \mid \cdots \mid \text{Z}R \mid \text{a}R \mid \cdots \mid \text{z}R \mid \_R \\
R &\rightarrow \text{A}R \mid \cdots \mid \text{Z}R \mid \text{a}R \mid \cdots \mid \text{z}R \mid \text{0}R \mid \cdots \mid \text{9}R \mid \_R \mid \varepsilon
\end{aligned}
$$

# 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 $L_3 = \{a^n \mid n \in \mathbb{N}\}$.
- ▶ $G_2$ with $P = \{S \to aSb, S \to \varepsilon\}$
  - ▶ $G_2$ is context-free.
  - ▶ So is the produced language $L_2 = \{a^n b^n \mid n \in \mathbb{N}\}$.
- ▶ $G_1$ with $P = \{S \to aSBC, S \to aBC, CB \to BC, \ldots\}$
  - ▶ $G_1$ is context-sensitive.
  - ▶ So is the produced language $L_1 = \{a^n b^n c^n \mid n \in \mathbb{N}; n > 0\}$.

# The Chomsky hierarchy: exercises

1. Let $G = (N, \Sigma, P, S)$ with
   - $N = \{S, A, B\}$,
   - $\Sigma = \{a\}$,
   - $P$ :

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

   a) What is $G$'s highest type?
   b) Show how $G$ derives the word `aaaaa`.
   c) Formally describe the language $L(G)$.
   d) Define a regular grammar $G'$ equivalent to $G$.

2 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.

3. Let $G = (N, \Sigma, P, S)$ with
   - $N = \{S, A, B\}$,
   - $\Sigma = \{a, b, t\}$,
   - $P$ :

| | | | | | |
|---|---|---|---|---|---|
| $S \rightarrow aAS$ | 1 | | $Aa \rightarrow aA$ | 6 |
| $S \rightarrow bBS$ | 2 | | $Ab \rightarrow bA$ | 7 |
| $S \rightarrow t$ | 3 | | $Ba \rightarrow aB$ | 8 |
| $At \rightarrow ta$ | 4 | | $Bb \rightarrow bB$ | 9 |
| $Bt \rightarrow tb$ | 5 | | | |

   a) What is $G$'s highest type?
   b) Formally describe the language $L(G)$.

End lecture 11

248

# Outline

# Right-linear grammars and regular languages
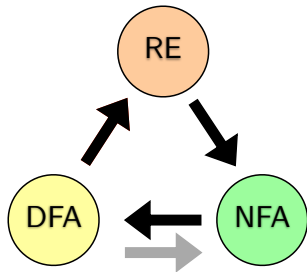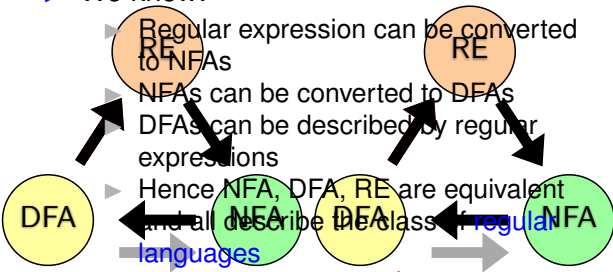


- We know:
  - Regular expression can be converted to NFAs
  - NFAs can be converted to DFAs
  - DFAs can be described by regular expressions
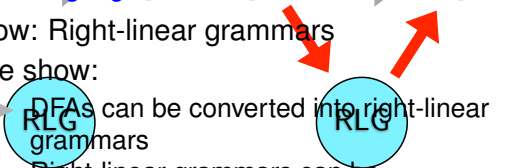  - Hence NFA, DFA, RE are equivalent and all describe the class of regular languages
- Now: Right-linear grammars
- We show:
  - DFAs can be converted into right-linear grammars
  - Right-linear grammars can be converted into NFAs

# 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.

We constructively prove the theorem by providing algorithms to...

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

...such that the languages of grammar and automaton are the same. □

Algorithm for transforming a DFA

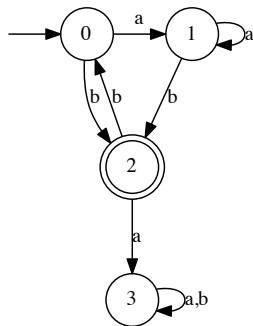$$\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$$

into a grammar

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

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

# Exercise: DFA to right-linear grammar

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



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

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

Algorithm for transforming a grammar

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

into an NFA

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

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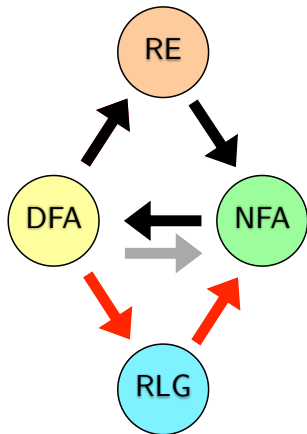  $\{(A, \varepsilon, q_f) \mid A \to \varepsilon \in P\}$

# Exercise: right-linear grammar to NFA

Transform the grammar $G = (\{S, A, B\}, \{a, b\}, P, S)$ into an NFA.

$$
\begin{aligned}
P: \quad S &\rightarrow aB \mid \varepsilon \\
A &\rightarrow aB \mid b \\
B &\rightarrow A
\end{aligned}
$$

Which language is generated by $G$?

# Right-linear grammars and regular languages



- We now know:
    - Regular expression can be converted to NFAs
    - NFAs can be converted to DFAs
    - DFAs can be described by regular expressions
    - Hence NFA, DFA, RE are equivalent and all describe the class of regular languages
    - DFAs can be converted into right-linear grammars
    - Right-linear grammars can be converted into NFAs

REs, DFAs, NFAs, Right-linear grammars are all equivalent formalisms do describe the class of regular languages!

# Outline

# Context-free grammars

- ► Reminder: $G = (N, \Sigma, P, S)$ is context-free if all rules in $P$ are of the form $A \to \beta$ with
  - ► $A \in N$ and
  - ► $\beta \in (\Sigma \cup N)^*$
- ► Context-free languages/grammars are highly relevant for practical applications
  - ► 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

I.e. grammars in CNF allow efficient parsing of arbitrary context-free languages!

# 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 become unreachable if they only appear together with non-terminating symbols.

# Reachable and terminating symbols

## Example

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

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

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

- terminating symbols in $G$: $C, A, S \quad \leadsto \quad G_T$
- reachable symbols in $G_T$: $S, C \quad \leadsto \quad G_r$
- 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. $P'$ :

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

# Chomsky normal form

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

## Definition (CNF)

A context-free grammar $(N, \Sigma, P, S)$ 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 of a reduced grammar into CNF:

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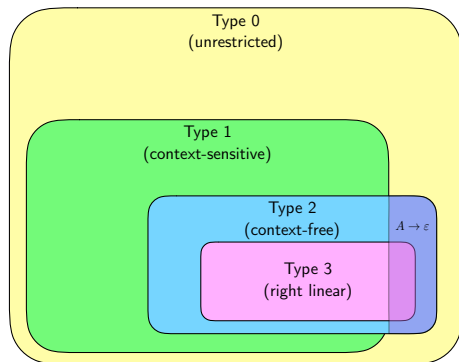3. introduce auxiliary symbols

# Removal of $\varepsilon$-productions

## Theorem ($\varepsilon$-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 $C \to \beta_1 B \beta_2$ with $B \in E$
   - ▶ add rule $C \to \beta_1 \beta_2$ to $P$
   - ▶ repeat this process until no new rules are added
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:
  - ▶ $A \rightarrow \varepsilon$ allowed in context-free/regular grammars, not in context-sensitive grammars
- ▶ Eliminating $\varepsilon$-productions removes this discrepancy!

267

# 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 $(N, \Sigma, P, S)$ 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 $\varepsilon$ 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 $w \notin \Sigma$:
   - for all $a \in \Sigma$ replace all occurrences of $a$ in $w$ by a new non-terminal symbol $X_a$
   - add rules $X_a \to a$
5. replace rules of the form $A \to B_1 B_2 \ldots B_n$ with $n > 2$ with rules

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

where the $C_i$ are new non-terminals

# Exercise: tranformation into CNF

Compute the Chomsky normal form of the following grammar:

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

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

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

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 ($C \rightarrow B; B \rightarrow C$), 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 $A \rightarrow BC$
  - ▶ $n$ rule applications $A \rightarrow a$
- ▶ 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)**

# Cocke-Younger-Kasami algorithm

- ▶ Eficient algorithm to decide the work problem for context-free grammars
- ▶ Core ideas independently developed by
    - ▶ John Cocke (1925—2002): *Programming languages and their compilers: Preliminary notes*, 1970 (with Jacob T. Schwartz)
    - ▶ Daniel H. Younger (??– ): *Recognition and parsing of context-free languages in time $n^3$*, 1967
    - ▶ Tadao Kasami (1930–2007) *An efficient recognition and syntax-analysis algorithm for context-free languages*, 1965
- ▶ Complexity: $O(|w|^3 \cdot |G|)$
- ▶ Can provide *all* ways to parse/generate a word
- ▶ Extends to probabilistic parsing

# 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 the target word from character $i$ through $j$.

**Example of dynamic programming!**

# CYK algorithm: example

$S \rightarrow a$

$B \rightarrow b$

$B \rightarrow c$

$S \rightarrow SA$

$A \rightarrow BS$

$B \rightarrow BB$

$B \rightarrow BS$

| $i \setminus j$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | $S$ | $\{\}$ | $S$ | $\{\}$ | $\{\}$ | $S$ |
| 2 | | $B$ | $A, B$ | $B$ | $B$ | $A, B$ |
| 3 | | | $S$ | $\{\}$ | $\{\}$ | $S$ |
| 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**
    $j := i + d$
    $N_{ij} := \{\}$
    **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

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

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

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

a) $w_1 = babaab$

b) $w_2 = abba$

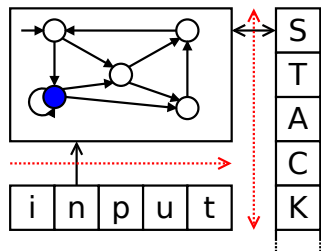# Outline

# 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 $\Gamma$
    - ▶ 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

- $Q, \Sigma, q_0$ are defined as for NFAs.
- $\Gamma$ 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 $(q, w, \gamma)$ where

- $q$ is the current state
- $w$ is the input yet unread
- $\gamma$ 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 $q$.
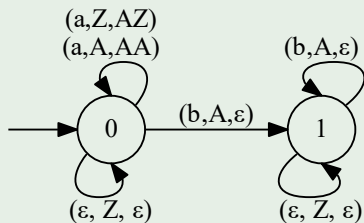
# Example: PDA for $a^n b^n$

## Example (Automaton $\mathcal{A}$)

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

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



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

## PDA: example (2)

Process *aabb*:

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

$$
\begin{array}{ccccc}
0 & \varepsilon & Z & \rightarrow & \varepsilon & 0 \\
0 & a & Z & \rightarrow & AZ & 0 \\
0 & a & A & \rightarrow & AA & 0 \\
0 & b & A & \rightarrow & \varepsilon & 1 \\
1 & b & A & \rightarrow & \varepsilon & 1 \\
1 & \varepsilon & Z & \rightarrow & \varepsilon & 1
\end{array}
$$

Process *abb*:

1. $(0, abb, Z)$
2. $(0, bb, AZ)$
3. $(1, b, Z)$
4. $(1, b, \varepsilon)$
5. No rule applicable, input not read entirely

# PDAs: important properties

- $\Gamma$ and $\Sigma$ do not need to be disjoint
  - Not uncommon: $\Sigma \subseteq \Gamma$
  - ... but convention: $\Sigma$ lowercase letters, $\Gamma$ uppercase letters
- $\varepsilon$ transitions are possible
  - ... and can modify the stack
- PDAs as defined above are non-deterministic
  - Deterministic PDA: For each situation there is only one applicable transition (either $\varepsilon$ or $c$)
  - deterministic PDAs are strictly weaker
- We can also define PDAs with acceptance via final states, but...
  - this makes representation of PDAs more complex
  - makes proofs more difficult

## PDA: exercise

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

$$\{w \cdot \overleftarrow{w} \mid w \in \Sigma^*\}$$

where

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

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 $L(G_\mathcal{A}) = L(\mathcal{A})$.

□

## From context-free grammars to PDAs

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

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

$$\Delta = \{(q, \varepsilon, A, \gamma, q) \mid A \to \gamma \in P\} \quad \cup$$
$$\{(q, a, a, \varepsilon, q) \mid a \in \Sigma\}$$

$\mathcal{A}_G$ 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 $(\Sigma, \Gamma, \Delta, Z_0)$.

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
  - $[pZt] \to a[qAr][rBs][sCt]$ 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

- $N = \{S\} \cup \{[pZq] \mid p, q \in Q, Z \in \Gamma\}$
- $P$ 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 $(p, a, Z, Y_1 Y_2 \ldots Y_n, q)$ with
        - $a \in \Sigma \cup \{\varepsilon\}$ and
        - $Z, Y_1, Y_2 \ldots Y_n \in \Gamma$,
      
      $P$ contains rules

      $$[pZq_n] \to a[qY_1q_1][q_1Y_2q_2] \ldots [q_{n-1}Y_nq_n]$$

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

# PDA to grammar illustrated

$$p, a, Z \rightarrow Y_1 Y_2 \ldots Y_n, q$$

$q_1, \ldots q_n$ can be chosen freely from $Q$!

$\rightsquigarrow$ $|Q|^n$ rules!

$$[pZq_n] \rightarrow a[qY_1q_1][q_1Y_2q_2]\ldots[q_{n-1}Y_nq_n]$$

---

Special case: $n = 0$

$$p, a, Z \rightarrow \varepsilon, q$$

$$[pZq] \rightarrow a$$
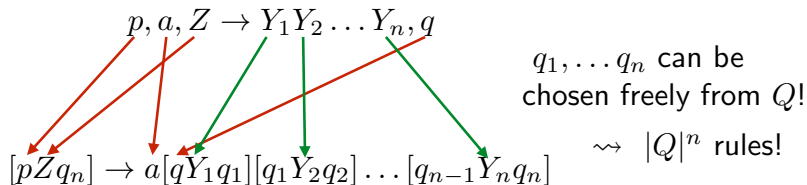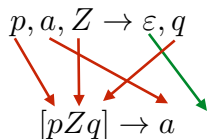
$\rightsquigarrow$ 1 rule!

Special case: $n = 1$

$$p, a, Z \rightarrow Y_1, q$$

$$[pZq_1] \rightarrow a[qY_1q_1]$$
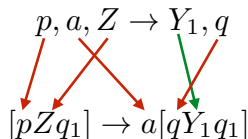
$\rightsquigarrow$ $|Q|$ rules!

# Exercise: transformation of PDA into grammar

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

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



$$
\begin{array}{ccccc}
0 & \varepsilon & Z & \to & \varepsilon & 0 \\
0 & a & Z & \to & AZ & 0 \\
0 & a & A & \to & AA & 0 \\
0 & b & A & \to & \varepsilon & 1 \\
1 & b & A & \to & \varepsilon & 1 \\
1 & \varepsilon & Z & \to & \varepsilon & 1 \\
\end{array}
$$
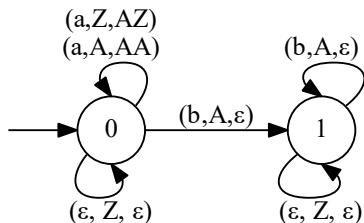
- Transform $\mathcal{A}$ into a grammar $G_{\mathcal{A}}$ (and reduce $G_{\mathcal{A}}$).
- Show how $\mathcal{A}_G$ produces the words $\varepsilon$, $ab$, and $aabb$.

# Bonus Exercises/Homework

- Assume $\Sigma = \{a, b\}$.
  - Find a PDA $A_1$ that accepts $L_1 = \{w \in \Sigma^* \mid |w|_a = |w|_b\}$.
  - Give an accepting run of $A_1$ on *abbbaa*.
- Assume $\Sigma = \{a, b\}$.
  - Find a PDA $A_2$ that accepts $L_2 = \{w \in \Sigma^* \mid |w|_a < |w|_b\}$.
  - Give an accepting run of $A_2$ on *bbbaaaabb*.
- Assume $\Sigma = \{a, b, c\}$.
  - Find a PDA $A_3$ that accepts
    $L_3 = \{w \in \Sigma^* \mid |w|$ is odd and $w[(|w| + 1)/2] = a\}$ (the middle symbol is an *a*)
  - Give an accepting run of $A_3$ on *cccaabb*.
- Assume $\Sigma = \{a, b, c\}$.
  - Find a PDA $A_4$ that accepts $L_4 = \{a^n b^m c^o \mid n, m, o \in \mathbb{N}, n = m + o\}$
  - Give an accepting run of $A_4$ on *aacc*.

End lecture 14

294

# Comparison: Regular vs. context-free languages



**Regular languages**

RE

DFA    NFA

RLG

**Context-free languages**

PDA

CFG

CNF
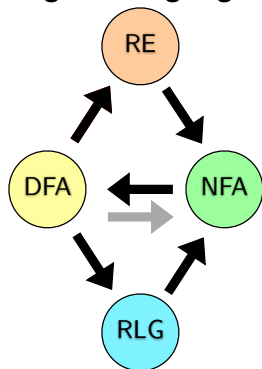
For both language classes there are different but equivalent formal descriptions, supporting different arguments about the classes!

# Outline

# Beyond context-free languages

## Theorem (Existence of non-context-free languages)

*Let $\Sigma$ be an alphabet. There are languages $L \subseteq \Sigma^*$ such that there is no context-free grammar $G$ with $L = L(G)$.*

- ▶ There are languages that are not context-free
  - ▶ e.g. $\{a^n b^n c^n \mid n \in \mathbb{N}\}$
  - ▶ ... but how do we show this?
- ▶ For regular languages: Pumping lemma
  - ▶ Finite automata must loop on long words
  - ▶ Loops can be repeated
  - ▶ Hence: A language than cannot be pumped is not regular
- ▶ For context-free languages?

# Pumping-lemma for context-free languages

**Idea:**

- ▶ If a context-free grammar $G$ can produce words of arbitrary length, there is least one repeated NTS in the derivation
    - ▶ If there are $n$ rules in the grammar, at least one rule has to be used more than once in a derivation of length $n + 1$
    - ▶ Slightly stronger arguments (based on $|N|$ instead of $|P|$ are possible)
- ▶ Hence there is a derivation $A \Longrightarrow^* vAx$ for $v, x \in (\Sigma \cup \Gamma)^*$
    - ▶ ... and we can repeat this part of the derivation
      $A \Longrightarrow^* vAx \Longrightarrow^* vvAxx \Longrightarrow^* vvvAxxx$
    - ▶ If $G$ has no chain rules, at least one of $v, x$ is non-empty

# Pumping Lemma I vs. Pumping Lemma II:

- ▶ PL I (regular languages)
  - ▶ Argument based on accepting automaton
  - ▶ Finite automaton must loop on sufficiently long word
  - ▶ One non-empty segment can be pumped
- ▶ PL II (context-free languages)
  - ▶ Argument based on generating grammar
  - ▶ At least on non-terminal must repeat in sufficiently long derivation
  - ▶ Two segments around this NTS can be pumped in parallel, at least one of which is non-empty

# Pumping Lemma II

## Theorem (Pumping-Lemma for context-free languages)

*Let $L$ be a 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 $k = r \cdot (m + 1)$.*
*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 $h \in \mathbb{N}$.*

- ▶ Holds for $\{a^n b^n\}$, but not for $\{a^n b^n c^n\}$.
- ▶ $\{a^n b^n c^n\}$ is not context-free, but context-sensitive, as we have seen before.

Use the Pumping Lemma II to show that $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not context-free.

# 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 $N_1 \cap N_2 = \{\}$ (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 $S \to \varepsilon, S \to S_1 S$.

# Closure properties (cont.)

## Theorem (Closure under ∩)

*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 $L_1$, $L_2$ and 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. $\qquad \square$

# Decision problems: emptiness problem

## Theorem (Emptiness problem for cf. languages)

*For a context-free grammar $G$, it is decidable if $L(G) = \{\}$ 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 $A \rightarrow \beta$ with $\beta \in Z^*$
3. repeat step 2 until no further symbols can be added
4. $L(G) = \{\}$ 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.

- ▶ A PCP can be encoded in grammars such that the PCP has a solution if and only if the two grammars are equivalent
- ▶ Since the PCP is undecidable, so is the equivalence problem
- ▶ Detail? Who needs details?

# Summary: context-free languages

- ▶ characterised by
  - ▶ context-free grammars
  - ▶ pushdown automata
- ▶ closure properties
  - ▶ closed under $\cup, ^*, \cdot$
  - ▶ not closed under $\cap, \overline{\phantom{-}}$
- ▶ decision problems
  - ▶ decidable: $w \in L(G)$, $L(G) = \{\}$ (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)

End lecture 15

# Outline

## Parsing: Motivation

Formal grammars describe formal languages

- ▶ A grammar has a set of rules
- ▶ Rules replace words with words
- ▶ A word that can be derived from the start symbol belongs to the language of the grammar

**In the concrete case of programming languages, "words of the language" are syntactically correct programs!**

# Grammars in Practice

- Most programming languages are described by context-free grammars (with extra "semantic" constraints)
- Grammars generate languages
- PDAs and e.g. CYK-Parsing recognize words
- For compiler we need to ...
  - identify correct programs
  - and understand their structure!

# Lexing and Parsing

- ▶ Lexer: Breaks programs into tokens
  - ▶ Smallest parts with semantic meaning
  - ▶ Can be recognized by regular languages/patterns
  - ▶ Example: `1, 2, 53` are all Integers
  - ▶ Example: `i, handle, stream` are all Identifiers
  - ▶ Example: `>, >=, *` are all individual operators
- ▶ Parser: Recognizes program structure
  - ▶ Language described by a grammar that has token types as terminals, not individual characters
  - ▶ Parser builds parse tree

# Automatisation with Bison

# YACC/Bison

- ▶ Yacc - Yet Another Compiler Compiler
  - ▶ Originally written ≈1971 by Stephen C. Johnson at AT&T
  - ▶ LALR parser generator
  - ▶ Translates grammar into syntax analyser





- ▶ GNU Bison
  - ▶ Written by Robert Corbett in 1988
  - ▶ Yacc-compatibility by Richard Stallman
  - ▶ Output languages now C, C++, Java
- ▶ Yacc, Bison, BYacc, …mostly compatible (POSIX P1003.2)

# Yacc/Bison Background

- By default, Bison constructs a 1 token Look-Ahead Left-to-right Rightmost-derivation or LALR(1) parser
  - Input tokens are processed left-to-right
  - Shift-reduce parser:
    - Stack holds tokens (terminals) and non-terminals
    - Tokens are shifted from input to stack. If the top of the stack contains symbols that represent the right hand side (RHS) of a grammar rule, the content is reduced to the LHS
    - Since input is reduced left-to-right, this corresponds to a rightmost derivation
    - Ambiguities are solved via look-ahead and special rules
    - If input can be reduced to start symbol, success!
    - Error otherwise
- LALR(1) is efficient in time and memory
  - Can parse "all reasonable languages"
  - For unreasonable languages, Bison (but not Yacc) can also construct GLR (General LR) parsers
    - Try all possibilities with back-tracking
    - Corresponds to the non-determinism of stack machines

## Yacc/Bison Overview

- ▶ Bison reads a specification file and converts it into (C) code of a parser
- ▶ Specification file: Declarations, grammar rules with actions, support code
    - ▶ Declarations: C declarations and data model, token names, associated values, includes
    - ▶ Grammar rules: Non-terminal with alternatives, action associated with each alternative
    - ▶ Support code: e.g. `main()` function, error handling. . .
    - ▶ Syntax similar to (F)lex
        - ▶ Sections separated by `%%`
        - ▶ Special commands start with `%`
- ▶ Bison generates function `yyparse()`
- ▶ Bison needs function `yylex()`
    - ▶ Usually provided via (F)lex

# Yacc/Bison workflow
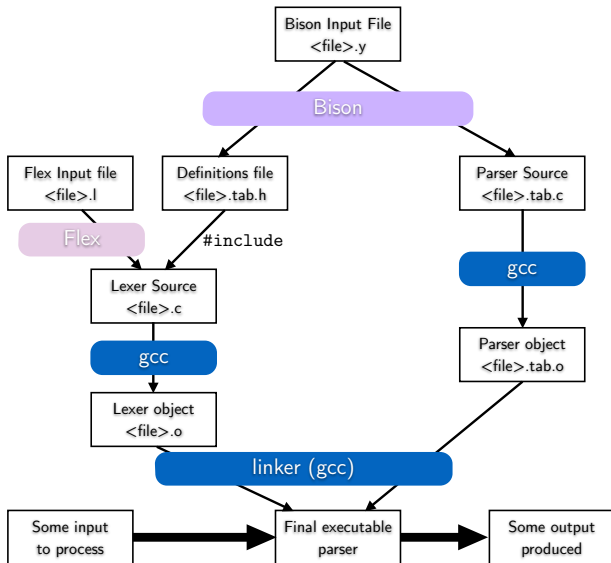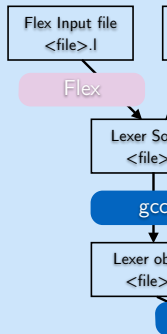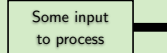
## Example task: Desk calculator

- ▶ Desk calculator
  - ▶ Reads algebraic expressions and assignments
  - ▶ Prints result of expressions
  - ▶ Can store values in registers R0-R99
- ▶ Example session:

```
[Shell] ./scicalc
R10=3*(5+4)
> RegVal: 27.000000

(3.1415*R10+3)
> 87.820500

R9=(3.1415*R10+3)
> RegVal: 87.820500

R9+R10
> 114.820500
...
```

## Abstract grammar for desk calculator (partial)

$G_{DC} = (N, \Sigma, P, S)$

- $\Sigma = \{$PLUS, MULT, ASSIGN, OPENPAR, CLOSEPAR, REGISTER, FLOAT, $\dots\}$

  - ▶ Some terminals are single characters $(+, =, \dots)$
  - ▶ Others are complex R10, 1.3e7
  - ▶ Terminals ("tokens") are generated by the lexer

- $N = \{$stmt, assign, expr, term, factor, $\dots\}$

▶ $P$:

```
stmt    →   assign
        |   expr
assign  →   REGISTER ASSIGN expr
expr    →   expr PLUS term
        |   term
term    →   term MULT factor
        |   factor
factor  →   REGISTER
        |   FLOAT
        |   OPENPAR expr CLOSEPAR
```

▶ $S$:

  - ▶ For a single statement, $S =$ stmt
  - ▶ In practice, we need to handle sequences of statements and empty input lines (not reflected in the grammar)

319

## Parsing statements (1)

- Example string: R10 = (4.5+3*7)
- Tokenized: REGISTER ASSIGN OPENPAR FLOAT PLUS FLOAT MULT FLOAT CLOSEPAR
  - In the following abbreviated R, A, O, F, P, F, M, F, C
- Parsing state:
  - Unread input (left column)
  - Current stack (middle column, top element on right)
  - How state was reached (right column)
- Parsing:

| Input | Stack | Comment |
|---|---|---|
| R A O F P F M F C | | Start |
| A O F P F M F C | R | Shift R to stack |
| O F P F M F C | R A | Shift A to stack |
| F P F M F C | R A O | Shift O to stack |
| P F M F C | R A O F | Shift F to stack |
| P F M F C | R A O factor | Reduce F |
| P F M F C | R A O term | Reduce factor |

. . .

## Parsing statements (2)

```
R A O F P F M F C                                              Start
A O F P F M F C      R                                         Shift R to stack
O F P F M F C        R A                                       Shift A to stack
F P F M F C          R A O                                     Shift O to stack
P F M F C            R A O F                                   Shift F to stack
P F M F C            R A O factor                              Reduce F
P F M F C            R A O term                                Reduce factor
P F M F C            R A O expr                                LA! Reduce term
F M F C              R A O expr P                              Shift P
M F C                R A O expr P F                            Shift F
M F C                R A O expr P factor                      Reduce F
M F C                R A O expr P term                        Reduce factor
F C                  R A O expr P term M                       LA! Shift M
C                    R A O expr P term M F                     Shift F
C                    R A O expr P term M factor                Reduce F
C                    R A O expr P term                         Reduce tMf
C                    R A O expr                                Reduce ePt
                     R A O expr C                              Shift C
                     R A factor                                Reduce OeC
                     R A term                                  Reduce factor
                     R A expr                                  Reduce term
                     assign                                    Reduce RAe
                     stmt                                      Reduce assign
```

321

# Lexer interface

- ▶ Bison parser requires `yylex()` function
- ▶ `yylex()` returns token
    - ▶ Token text is defined by regular expression pattern
    - ▶ Tokens are encoded as integers
    - ▶ Symbolic names for tokens are defined by Bison in generated header file
        - ▶ By convention: Token names are all CAPITALS
- ▶ `yylex()` provides optional semantic value of token
    - ▶ Stored in global variable `yylval`
    - ▶ Type of `yylval` defined by Bison in generated header file
        - ▶ Default is `int`
        - ▶ For more complex situations often a `union`
        - ▶ For our example: Union of double (for floating point values) and integer (for register numbers)

# Lexer for desk calculator (1)

```
/*
Lexer for a minimal "scientific" calculator.

Copyright 2014 by Stephan Schulz, schulz@eprover.org.

This code is released under the GNU General Public Licence
Version 2.
*/

%option noyywrap

%{
  #include "scicalcparse.tab.h"
%}
```

# Lexer for desk calculator (2)

```
DIGIT     [0-9]
INT       {DIGIT}+
PLAINFLOAT {INT}|{INT}[.]|{INT}[.]{INT}|[.]{INT}
EXP       [eE](\+|-)?{INT}
NUMBER    {PLAINFLOAT}{EXP}?
REG       R{DIGIT}{DIGIT}?

%%

"*" {return MULT;}
"+" {return PLUS;}
"=" {return ASSIGN;}
"(" {return OPENPAR;}
")" {return CLOSEPAR;}
\n  {return NEWLINE;}
```

```
{REG}    {
  yylval.regno = atoi(yytext+1);
  return REGISTER;
}

{NUMBER} {
  yylval.val = atof(yytext);
  return FLOAT;
}

[ ] { /* Skip  whitespace*/ }

/* Everything else is an invalid character. */
.    { return ERROR;}

%%
```

# Data model of desk calculator

- Desk calculator has simple state
  - 100 floating point registers
  - R0-R99
- Represented in C as array of doubles:

```
#define MAXREGS 100

double regfile[MAXREGS];
```

- Needs to be initialized in support code

```
%{
  #include <stdio.h>

  #define MAXREGS 100

  double regfile[MAXREGS];

  extern int yyerror(char* err);
  extern int yylex(void);
%}

%union {
  double val;
  int    regno;
  }
```

## Bison code for desk calculator: Tokens

```
%start stmtseq

%left   PLUS
%left   MULT
%token  ASSIGN
%token  OPENPAR
%token  CLOSEPAR
%token  NEWLINE
%token  REGISTER
%token  FLOAT
%token  ERROR

%%
```

## Actions in Bison

- ▶ Bison is based on syntax rules with associated actions
  - ▶ Whenever a reduce is performed, the action associated with the rule is executed
- ▶ Actions can be arbitrary C code
- ▶ Frequent: semantic actions
  - ▶ The action sets a semantic value based on the semantic value of the symbols reduced by the rule
  - ▶ For terminal symbols: Semantic value is `yylval` from Flex
  - ▶ Semantic actions have "historically valuable" syntax
    - ▶ Value of reduced symbol: `$$`
    - ▶ Value of first symbol in syntax rule body: `$1`
    - ▶ Value of second symbol in syntax rule body: `$2`
    - ▶ . . .
    - ▶ Access to named components: `$<val>1`

# Bison code for desk calculator: Grammar (1)

```
stmtseq: /* Empty */
      | NEWLINE stmtseq        {}
      | stmt  NEWLINE stmtseq {}
      | error NEWLINE stmtseq {};  /* After an error,
      start afresh */
```

► Head: sequence of statements
► First body line: Skip empty lines
► Second body line: separate current statement from rest
► Third body line: After parse error, start again with new line

```
stmt: assign  {printf("> RegVal: %f\n", $<val>1);}
      |expr   {printf("> %f\n", $<val>1);};

assign: REGISTER ASSIGN expr {regfile[$<regno>1] = $<val>3;
         $<val>$ = $<val>3;} ;

expr: expr PLUS term {$<val>$ = $<val>1 + $<val>3;}
      | term {$<val>$ =  $<val>1;};

term: term MULT factor {$<val>$ = $<val>1 * $<val>3;}
      | factor {$<val>$ =  $<val>1;};

factor: REGISTER {$<val>$ = regfile[$<regno>1];}
      | FLOAT  {$<val>$ =  $<val>1;}
      | OPENPAR expr CLOSEPAR {$<val>$ =  $<val>2;};
```

```
int yyerror(char* err)
{
  printf("Error: %s\n", err);
  return 0;
}

int main (int argc, char* argv[])
{
  int i;

  for(i=0; i<MAXREGS; i++)
  {
    regfile[i] = 0.0;
  }
  return yyparse();
}
```

# Bison workflow and dependencies

# Building the calculator

1. Generate parser C code and include file for lexer
   - ▶ `bison -d scicalcparse.y`
   - ▶ Generates `scicalcparse.tab.c` and `scicalcparse.tab.h`
2. Generate lexer C code
   - ▶ `flex -t scicalclex.l > scicalclex.c`
3. Compile lexer
   - ▶ `gcc -c -o scicalclex.o scicalclex.c`
4. Compile parser and support code
   - ▶ `gcc -c -o scicalcparse.tab.o scicalcparse.tab.c`
5. Link everything
   - ▶ `gcc scicalclex.o scicalcparse.tab.o -o scicalc`
6. Fun!
   - ▶ `./scicalc`

## Exercise: calculator

- Exercise 1:
  - Go to `http://wwwlehre.dhbw-stuttgart.de/~sschulz/fla2018.html`
  - Download `scicalcparse.y` and `scicalclex.l`
  - Build the calculator
  - Run and test the calculator
- Exercise 2:
  - Add support for division and subtraction `/`, `-`
  - Add support for unary minus (the negation operator `-`)

# Derivations

## Definition (derivation)

For a grammar $G$, a derivation of a word $w_n$ in $L(G)$ is a sequence $S \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n$ where $S$ is the start symbol, and each $w_i$ is generated from its predecessor by application of a production of the grammar

# Example: derivation

## Example (well-formed expressions over $x, +, *.(,)$)

Let $G_E = (N, \Sigma, P, E)$ with

- $N = \{E\}$
- $\Sigma = \{(,), +, *, x\}$
- P:
  - 1 $E \to x$
  - 2 $E \to (E)$
  - 3 $E \to E + E$
  - 4 $E \to E * E$

The following is a derivation of $x + x + x * x$ (with the replaced symbol printed bold):

$$
\begin{aligned}
& \mathbf{E} \\
\Rightarrow\ & \mathbf{E} + E \\
\Rightarrow\ & E + E + \mathbf{E} \\
\Rightarrow\ & \mathbf{E} + E + E * E \\
\Rightarrow\ & x + \mathbf{E} + E * E \\
\Rightarrow\ & x + x + \mathbf{E} * E \\
\Rightarrow\ & x + x + x * \mathbf{E} \\
\Rightarrow\ & x + x + x * x
\end{aligned}
$$

# Rightmost and leftmost Derivations

## Definition (rightmost/leftmost)

▶ A derivation is called a rightmost derivation, if at any step it replaces the rightmost non-terminal in the current word.

▶ A derivation is called a leftmost derivation, if at any step it replaces the leftmost non-terminal in the current word.

## Example

▶ The derivation on the previous slide is neither leftmost nor rightmost.

▶ A rightmost derivation is:
$\mathbf{E} \Rightarrow E + \mathbf{E} \Rightarrow E + E + \mathbf{E} \Rightarrow E + E + E * \mathbf{E} \Rightarrow E + E + \mathbf{E} * x \Rightarrow E + \mathbf{E} + x * x \Rightarrow \mathbf{E} + x + x * x \Rightarrow x + x + x * x$

# Parse trees

## Definition (parse tree)

A parse tree for a derivation in a grammar $G = (N, \Sigma, P, S)$ is an ordered, labelled tree with the following properties:

- ► Each node is labelled with a symbol from $N \cup \Sigma$
- ► The root of the tree is labelled with the start symbol $S$.
- ► Each inner node is labelled with a single non-terminal symbol from $V_N$
- ► If an inner node with label $A$ has children labelled with symbols $\alpha_1, \ldots, \alpha_n$, then there is a production $A \to \alpha_1 \ldots \alpha_n$ in $P$.

<br>

- ► The parse tree represents a derivation of the word formed by the labels of the leaf nodes
- ► It abstracts from the order in which productions are applied.

# Example: parse trees

## Example

The derivation $\mathbf{E} \Rightarrow E + \mathbf{E} \Rightarrow E + E + \mathbf{E} \Rightarrow E + E + E * \mathbf{E} \Rightarrow$
$E + E + \mathbf{E} * x \Rightarrow E + \mathbf{E} + x * x \Rightarrow \mathbf{E} + x + x * x \Rightarrow x + x + x * x$
can be represented by a sequence of parse trees:

# Ambiguity

## Definition (ambiguous)

A grammar $G = (N, \Sigma, P, S)$ is ambiguous if it has multiple different parse trees for a word $w$ in $L(G)$.

## Example

$G_E$ is ambiguous since it has several parse trees for $x + x + x$.

1. $E \to x$
2. $E \to (E)$
3. $E \to E + E$
4. $E \to E * E$

Consider the grammar $G_E$ and its parse trees for $x + x + x$.

1  $E \rightarrow x$

2  $E \rightarrow (E)$

3  $E \rightarrow E + E$

4  $E \rightarrow E * E$



► Provide a rightmost derivation for the right tree.

► Provide a rightmost derivation for the left tree.

► Provide a leftmost derivation for the left tree.

► Provide a leftmost derivation for the right tree.

## Exercise: Eliminating Ambiguity

Consider $G_E$ with the following productions:

1. $E \rightarrow x$
2. $E \rightarrow (E)$
3. $E \rightarrow E + E$
4. $E \rightarrow E * E$

Define a grammar $G'$ with $L(G) = L(G')$ that is not ambiguous, that respects that $*$ has a higher precedence than $+$, and that respects left-associativity for all operators.

# Abstract Syntax Trees

▶ Abstract Syntax Trees represent the structure of a derivation without the specific details
▶ Think: "Parse trees without redundant syntactic elements"

## Example

Parse Tree:

Corresponding AST:

# High-Level Architecture of a Compiler



Source handler

Sequence of characters:
i,n,t, ⎵, a,,, b, ;, a, =, b, +, 1, ;

Lexical analysis
(tokeniser)

Sequence of tokens:
(id, "int"), (id, "a"), (id, "b"), (semicolon), (id, "a"), (eq), (id, "b"), (plus), (int, "1"), (semicolon)

Syntactic analysis
(parser)

e.g. Abstract syntax tree

Semantic analysis

e.g. AST+symbol table

| Variable | Type |
|----------|------|
| a        | int  |
| b        | int  |

Code generation
(several optimisation passes)

e.g. assembler code

```
ld a,b
ld c, 1
add c
...
```

# Syntactic Analysis/Parsing

- ▶ Description of the language with a context-free grammar
- ▶ Parsing:
  - ▶ Try to build a parse tree/abstract syntax tree (AST)
  - ▶ Parse tree unambiguously describes structure of a program
  - ▶ AST reflects abstract syntax (can e.g. drop parenthesis)
- ▶ Methods:
  - ▶ Manual recursive descent parser
  - ▶ Automatic with a table-driven bottom-up parser

**Result:** Abstract Syntax Tree

# Semantic Analysis

- ► Analyze static properties of the program
  - ► Which variable has which type?
  - ► Are all expressions well-typed?
  - ► Which names are defined?
  - ► Which names are referenced?
- ► Core tool: Symbol table

**Result:** Annotated AST

# Optimization

- ▶ Transform Abstract Syntax Tree to generate better code
  - ▶ Smaller
  - ▶ Faster
  - ▶ Both
- ▶ Mechanisms
  - ▶ Common sub-expression elimination
  - ▶ Loop unrolling
  - ▶ Dead code/data elimination
  - ▶ . . .

**Result:** Optimized AST

# Code Generation

- ▶ Convert optimized AST into low-level code
- ▶ Target languages:
  - ▶ Assembly code
  - ▶ Machine code
  - ▶ VM code (z.B. JAVA byte-code, p-Code)
  - ▶ C (as a "portable assembler")
  - ▶ . . .

**Result:** Program in target language

# Summary: parsers

Parsers

- ► recognise structure of programs
- ► receive tokens from scanner
- ► construct parse tree and symbol table
- ► common: shift-reduce parsing

Bison

- ► receives tokens and their semantic values from Flex
- ► uses grammar rules to perform semantic actions
- ► uses look-ahead to resolve conflicts

End lecture 16

# 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 $\rightsquigarrow$ ?



(all languages)

Type 0
(?)

Type 1
(?)

Type 2
(PDA)

Type 3
(DFA/NFA)

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 $1, t \rightarrow c, r, 2$)

# Turing Machine: formal definition

## Definition (Turing Machine)

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

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

If $\Delta$ 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 $\delta$.

Note:

- $\Gamma$ (the tape alphabet) can contain additional characters
- This does not increase the power, but makes finding TMs easier

# 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 $\alpha$ on the left of the read/write head (except unlimited # sequences)
- the tape content $\beta$ 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 4*tape* is accepting.
▶ If there are no transitions $4, t \to \ldots$, 4*tape* also is a stop configuration.

## 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 leads to an accepting stop configuration.

## Exercise: Turing Machines

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

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

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

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.

□

# 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 $\alpha$ of a rule $\alpha \to \beta$, apply the rule
        - ▶ move tape content if necessary
        - ▶ replace $\alpha$ with $\beta$
    - ▶ 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 $w$

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

3. if an accepting stop configuration is reached, recreate $w$
   - ▶ requires a more complex alphabet with a "backup track"

# Outline

373

# 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
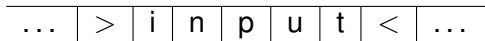
| ... | > | i | n | p | u | t | < | ... |
|-----|---|---|---|---|---|---|---|-----|

# 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 ∪, ·, *)

*The class of languages described by context-sensitive grammars is closed under ∪, ·, \*.*

## 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 $\qquad\square$

## Theorem (closure under $\overline{\phantom{-}}$)

*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.

- $N$, $\Sigma$ 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.

□

# 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 $\mathcal{A}$ accepts
    - ▶ $\mathcal{U}$ 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 $\rightsquigarrow$ 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 $c(\mathcal{S})$.    ↯

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 $c(\mathcal{S})$.    ↯

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.*

385

# 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. $\qquad\square$

# Closure properties

## Theorem (closure under $\overline{\phantom{x}}$)

*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. □

## 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. □

# Diagonalisation

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

| TM | input | $c(\mathcal{A})$ | $c(\mathcal{B})$ | $c(\mathcal{C})$ | $c(\mathcal{D})$ | $c(\mathcal{E})$ | ... |
|---|---|---|---|---|---|---|---|
| $\mathcal{A}$ | | ✗ | | | | | |
| $\mathcal{B}$ | | | ✗ | | | | |
| $\mathcal{C}$ | | | | ✗ | | | |
| $\mathcal{D}$ | | | | | ✗ | | |
| $\mathcal{E}$ | | | | | | ✗ | |
| ⋮ | | | | | | | ⋱ |

# 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 can 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?**

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 $\mathcal{A}$ on $w_{\mathcal{A}}$

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 | | | | |
| $\cup, \cdot, {}^*$ | ✓ | ✓ | ✓ | ✓ |
| $\cap$ | ✓ | ✗ | ✓ | ✓ |
| $\underline{\phantom{\cap}}$ | ✓ | ✗ | ✓ | ✗ |
| decidability | | | | |
| word | ✓ | ✓ | ✓ | ✗ |
| emptiness | ✓ | ✓ | ✗ | ✗ |
| equiv. | ✓ | ✗ | ✗ | ✗ |
| deterministic equivalent to non-det. | ✓ | ✗ | ? | ✓ |

**This is the End. . .**

# Outline

**Lecture-specific material**

# Outline

# Goals for Lecture 1

- ▶ Getting acquainted
- ▶ Clarifying practical issues
- ▶ Course outline and motivation
  - ▶ Formal languages
  - ▶ Language classes
  - ▶ Grammars
  - ▶ Automata
  - ▶ Questions
  - ▶ Applications

# Practical Issues

▶ Lecture times (usually)
  ▶ Tuesday 12:30-14:00
  ▶ Wednesday 14:00-15:45
  ▶ Plus irregular extra times (usually Monday)
  ▶ No lectures:
    ▶ KW39 (September 24.-28.)
    ▶ September 26th, October 4th, October 25th, October 27th
    ▶ KW44-KW45 (29.10.-11.11.)
▶ Written exam
  ▶ Calender week 47 (19.11.–23.11.)

Lecture 1

# Summary Lecture 1

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

# 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?

# Outline

# Goals for Lecture 2

- ▶ Review of last lecture
- ▶ Formal basics of formal languages
- ▶ Operations on words
- ▶ Operations on languages
- ▶ Introduction to regular expressions
  - ▶ Examples
  - ▶ Formal definition

# Review

- Introduction
  - Language classes
  - Grammars
  - Automata
  - Applications

# Summary

- ▶ Formal languages
  - ▶ Alphabets
  - ▶ Words
  - ▶ Languages
  - ▶ Examples of languages
- ▶ Operations on words and languages
  - ▶ Concatenation
  - ▶ Power
  - ▶ Kleene star
- ▶ Introduction to regular expressions

Remark: Formal languages are *sets*, and hence we can also apply set operations like ∪ (union), ∩ (intersection), ⁻ (complement) to them!

# 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?

# Outline

# Goals for Lecture 3

- ▶ Review of last lecture
- ▶ Understanding regular expressions
- ▶ Regular expression algebra
  - ▶ Equivalences on regular expressions
  - ▶ Simplifying REs

# Review (1)

- ▶ Formal languages
  - ▶ Finite alphabet $\Sigma$ of symbols/letters
  - ▶ Words are finite sequences of letters from $\Sigma$
  - ▶ Languages are (finite or infinite) sets of words
- ▶ Words - properties and operations
  - ▶ $|w|, |w|_a, w[k]$
  - ▶ $w_1 \cdot w_2, w^n$

# Review (2)

- ▶ Interesting languages
  - ▶ Binary representations of natural numbers
  - ▶ Binary representations of prime numbers
  - ▶ C functions (over strings)
  - ▶ C functions with input/output pairs
- ▶ 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 $L$

Remark: Formal languages are *sets*, and hence we can also apply set operations like ∪ (union), ∩ (intersection), ⁻ (complement) to them!

# Review (3)

- Regular expressions $R_\Sigma$
  - 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)

Lecture 3

416

# Summary

- ▶ Regular expression algebra
  - ▶ REs are equivalent if they describe the same language
  - ▶ REs can be simplified applying equivalences
  - ▶ Recursive definitions: Lemmas of Arden/Salomaa

# 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?

# Outline

# Goals for Lecture 4

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

# Review (1)

▶ Regular expressions $R_\Sigma$
  ▶ 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)

- Regular expression algebra
  - REs are equivalent if they describe the same language
  - REs can be simplified applying equivalences
  - 17 equivalences
    - Commutativity of $+$, associativity of $+, \cdot$, distributivity (!)
    - Arden/Salomaa
    - . . .

# Warmup Exercise

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

Lecture 4

# Summary

- REs revisited (with exercise)
- 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?

# Outline

# Goals for Lecture 5

- ▶ Review of last lecture
- ▶ Introduction to Nondeterministic Finite Automata
  - ▶ Definitions
  - ▶ Exercises
- ▶ Equivalency of deterministic and nondeterministic finite automata
  - ▶ Converting NFAs to DFAs
  - ▶ Exercises

# Review

- Finite Automata
    - Graphical representation
    - Formal definition
        - $Q$: Set of states
        - $\Sigma$: Alphabet
        - $\delta$: Transition function
        - $q_0$: Initial state
        - $F$: Final (accepting) states
    - Language recognized by an automata
        - $w \in L(A)$ if $\delta'(w) \in F$ or there exist an accepting run of $A$ for $w$
    - Tabular representation
    - Exercises

Lecture 5

# Summary

- Nondeterministic Finite Automata
  - $Q$: Set of states
  - $\Sigma$: Alphabet
  - $\Delta$: Transition *relation* (with $\varepsilon$-transitions!)
  - $q_0$: Initial state
  - $F$: Final (accepting) states
- NFAs and DFAs accept the same class of languages!
- Converting NFA to DFA
  - States of $det(\mathcal{A})$ are elements of $2^Q$
  - . . .

# 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?

# Outline

# Goals for Lecture 6

- ▶ Review of last lecture
- ▶ Equivalency of regular expressions and NFAs
  - ▶ Construction of an NFA from a regular expressione
  - ▶ Extraction of an RE from a DFA

# Review

- ▶ Nondeterministic Finite Automata
  - ▶ $Q$: Set of states
  - ▶ $\Sigma$: Alphabet
  - ▶ $\Delta$: Transition *relation* (with $\varepsilon$-transitions!)
  - ▶ $q_0$: Initial state
  - ▶ $F$: Final (accepting) states
- ▶ NFAs and DFAs accept the same class of languages!
- ▶ Converting NFA to DFA
  - ▶ States of $det(\mathcal{A})$ are elements of $2^Q$
  - ▶ . . .

Lecture 6

# Summary

- ▶ Equivalency of regular expressions and NFAs
    - ▶ Construction of an NFA from a regular expression
        - ▶ Base cases each result in a trivial 2-state automaton
        - ▶ Compose automata for composite regular expressions
        - ▶ *Glue* automata together with $\varepsilon$-Transitions
    - ▶ Extraction of an RE for a DFA
        - ▶ Determine system of equations
        - ▶ For each state add one alternative for each transition
        - ▶ For accepting states add $\varepsilon$.
        - ▶ Solve the system of equations, handling loops with Arden's lemma

# 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?

# Outline

# Goals for Lecture 7

- ▶ Review of last lecture
- ▶ Minimizing DFAs
  - ▶ . . . and a first application
- ▶ Exercises

# Review

- ▶ Central result: REs, NFAs and DFAs describe the same class of lanuguages (namely regular languages)
    - ▶ Proof via construcive algorithms
        - ▶ NFA to DFA already done
        - ▶ RE to NFA and DFA to RE new
- ▶ Construction of an NFA from a regular expression
    - ▶ Base cases each result in a trivial 2-state automaton
    - ▶ Compose automata for composite regular expressions
    - ▶ *Glue* automata together with $\varepsilon$-Transitions
- ▶ Extraction of an RE for a DFA
    - ▶ Determine system of equations describing language accepted at each state
        - ▶ For each state add one alternative for each transition
        - ▶ For accepting states add $\varepsilon$.
    - ▶ Solve the system of equations, handling loops with Arden's lemma

## Homework assignment

► Get access to an operational Linux/UNIX enviroment
  ► 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

- ▶ 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?

# Outline

# Goals for Lecture 8

- Review of last lecture
- Beyond regular languages: The Pumping Lemma
  - Motivation/Lemma
  - Application of the lemma
  - Implications
- Properties of regular languages
  - Which operations leave regular languages regular?

# Review

- 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

# 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

- ▶ Beyond regular languages: The Pumping Lemma
  - ▶ Motivation/Lemma
  - ▶ Application of the lemma ($a^n b^n, a^n b^m, n < m$)
  - ▶ Implications (Nested structures are not regular)
- ▶ Properties of regular languages
  - ▶ Closure properties (union, intersection, . . . )
  - ▶ Proof per construction of suitable NFA (union, concatenation, Kleene Star)
  - ▶ Proof per construction of product automaton (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?

# Outline

# Goals for Lecture 9

- Completing the theory of regular languages
  - Complement
  - Finite languages
  - Decision problems: Emptiness, word problem, . . .
  - Decision problems: Equivalence, finiteness, . . .
  - Wrap-up

# 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 (Solution: 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)

# Summary

▶ Completing the theory of regular languages
  ▶ Complement
  ▶ All finite languages are regular
  ▶ Decidable for regular languages: Emptiness, word problem
  ▶ Decidable for regular languages: Equivalence, finiteness
  ▶ Wrap-up

# 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?

# Outline

# Goals for Lecture 10

- ▶ Review of last lecture
- ▶ Scanning in practice
  - ▶ Scanners in context
  - ▶ Practical regular expressions
  - ▶ Automatic scanner creation with `flex`
  - ▶ `Flex` exercise

# Review

- Completing the theory of regular languages
  - Complement
  - All finite languages are regular
  - Decidable for regular languages: Emptiness, word problem
  - Decidable for regular languages: Equivalence, finiteness
  - Use of properties to prove statements (e.g. non-regularity, e.g. reducing emptyness question to equivalency)
  - Wrap-up

Lecture 10

# Summary

- ▶ Scanners in context
- ▶ Practical regular expressions
  - ▶ Basic characters, escapes, ranges, escape with \
  - ▶ Richer operators, z.B. $+$, $\{4\}$,?...
- ▶ Flex
  - ▶ Definition section
  - ▶ Rule section
  - ▶ User code section/`yylex()`
- ▶ Exercise

# 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?

# Outline

# Goals for Lecture 11

- ▶ Review of last lecture
- ▶ Grammars and languages
  - ▶ Formal grammars
  - ▶ Derivations
  - ▶ Languages
- ▶ The Chomsky-Hierarchy

# Review

- ▶ Scanners in context
- ▶ Practical regular expressions
    - ▶ Basic characters, escapes, ranges, escape with \
    - ▶ Richer operators, z.B. +, {4},?...
- ▶ Flex
    - ▶ Definition section
    - ▶ Rule section
    - ▶ User code section/`yylex()`
- ▶ `make`
- ▶ Exercise

# Summary

- Grammars and languages
  - Formal grammars
  - Derivations
  - Languages
  - Grammars can describe non-regular languages
- The Chomsky-Hierarchy
  - Type 0: Unrestricted grammars
  - Type 1: Context-sensitive/monotonic grammars
  - Type 2: Context-free grammars
  - Type 3: Right-linear/regular grammars

## 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?

# Outline

# Goals for Lecture 12

- ▶ Review of last lecture
- ▶ Right-linear grammars
  - ▶ Equivalence with finite automata
- ▶ Context-free grammars
  - ▶ Equivalency for grammars
  - ▶ Reducing grammars
  - ▶ Eliminating $\varepsilon$-Productions
  - ▶ Chomsky-hierarchy revisited

# Reviews

- ▶ Grammars and languages
  - ▶ Formal grammars $(N, \Sigma, P, S)$
  - ▶ Derivations
  - ▶ Languages $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$
  - ▶ Grammars can describe non-regular languages
- ▶ The Chomsky-Hierarchy
  - ▶ Type 0: Unrestricted grammars
  - ▶ Type 1: Context-sensitive/monotonic grammars (non-shortening rules)
  - ▶ Type 2: Context-free grammars (Only one non-terminal on LHS)
  - ▶ Type 3: Right-linear/regular grammars (limited RHS)

Lecture 12

# Summary

- Right-linear grammars: $N \to aB$, $a \in \Sigma \cup \{\epsilon\}$
  - Convert DFA to right-linear grammar
  - Convert right-linear grammar to NFA
- Context-free grammars
  - Chomsky Normal Form
  - Towards CNF
    - Remove non-terminating symbols and corresponding rules
    - Remove non-reachable symbols and corresponding rules
    - Inline $\varepsilon$-rules

# 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?

# Outline

# Goals for Lecture 13

- ▶ Review of last lecture
- ▶ Completing Chomsky Normal Form tranformation
  - ▶ Eliminating chain rules
  - ▶ Bring right hand sides into normalform
- ▶ Solving the word problem for context-free grammars
  - ▶ Derivations in CNF
  - ▶ Parsing with Dynamic Programming: Cocke-Younger-Kasami

# Review

- ▶ Right-linear grammars: $N \to aB$, $a \in \Sigma \cup \{\epsilon\}$
  - ▶ Convert DFA to right-linear grammar ($N \sim Q, \delta \sim P$)
  - ▶ Convert right-linerar grammar to NFA
- ▶ Context-free grammars
  - ▶ Chomsky Normal Form
  - ▶ Towards CNF
    - ▶ Remove non-terminating symbols and corresponding rules
    - ▶ Remove non-reachable symbols and corresponding rules
    - ▶ Inline $\varepsilon$-rules
    - ▶ . . .

Lecture 13

# Summary

- Chomsky Normal Form tranformation
  - Inline $\varepsilon$-rules
  - Inline chain rules
  - Reduce grammar
  - Introduce NTS names for terminals
  - Break long RHS via introduction of definitions/intermediate rules
- Solving the word problem for context-free grammars
  - Derivations in CNF
  - Cocke-Younger-Kasami
    - Systematically consider all decompositions of $w$
    - Tabulate all NTS for given subwords bottom-up
    - Example of *dynamic programming*

# 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?

# Outline

# Goals for Lecture 14

- ▶ Review of last lecture
- ▶ Pushdown automata
  - ▶ NFA+unlimited stack
- ▶ Equivalence of context-free grammars and PDAs
  - ▶ From grammar to PDA
  - ▶ From PDA to grammar

# Review

▶ Chomsky Normal Form tranformation
  ▶ Inline $\varepsilon$-rules
  ▶ Inline chain rules
  ▶ Reduce grammar
  ▶ Introduce NTS names for terminals
  ▶ Break long RHS via introduction of definitions/intermediate rules
▶ Solving the word problem for context-free grammars
  ▶ Derivations in CNF
  ▶ Cocke-Younger-Kasami
    ▶ Systematically consider all decompositions of $w$
    ▶ Tabulate all NTS for given subwords bottom-up
    ▶ Complexity: $O(n^3)$ with $n = |w|$

# Summary

- ▶ Pushdown automata
    - ▶ Transitions must read/can write unlimited stack
    - ▶ Transition can read characters from word (left-to-right only)
    - ▶ Acceptance: Empty stack, empty word
- ▶ Equivalence of context-free grammars and PDAs
    - ▶ From grammar to PDA
        - ▶ Simulate grammar rules on the stack
        - ▶ Highly non-deterministic
    - ▶ From PDA to grammar
        - ▶ Non-terminals represent state changes with removal of a symbols from stack
        - ▶ Complex encoding of transition relation into grammar rules

# 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?

# Outline

# Goals for Lecture 15

- ▶ Review of last lecture
- ▶ Limits of context-free languages
  - ▶ Pumping Lemma II
- ▶ Closure properties of context-free languages
- ▶ Decision problems for context-free languages

# Review

- ▶ Pushdown automata
  - ▶ Transitions must read/can write unlimited stack
  - ▶ Transition can read characters from word (left-to-right only)
  - ▶ Acceptance: Empty stack, empty word
- ▶ Equivalence of context-free grammars and PDAs
  - ▶ From grammar to PDA
    - ▶ Simulate grammar rules on the stack
    - ▶ Highly non-deterministic
  - ▶ From PDA to grammar
    - ▶ Non-terminals represent state changes with removal of a symbols from stack
    - ▶ Complex encoding of transition relation into grammar rules

Lecture 15

# Summary

- ▶ Limits of context-free languages
  - ▶ Pumping Lemma II
- ▶ Closure properties of context-free languages
  - ▶ The class of context-free languages is closed under $\cup, \cdot, ^*$
  - ▶ The class of context-free languages is not closed under $\cap$, complement
- ▶ Decision problems for context-free languages
  - ▶ Decidable: Word problem, emptiness problem
  - ▶ Undeciable: Equivalence

# 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?

# Outline

# Goals for Lecture 16

- ▶ Review of last lecture
- ▶ Introduction to parsing
- ▶ YACC/Bison
  - ▶ Automatic parser generation
  - ▶ Example: desk calculator
- ▶ LALR(1) shift/reduce parsing
- ▶ Parse trees and abstract syntax trees

# Review

- Limits of context-free languages
  - Pumping Lemma II
    - Sufficiently long words require correspondingly long derivations
    - Sufficiently long derivations will contain at least one duplicate NTS $A$
    - The partial derivation from $A$ to $uAv$ can be repeated
- Closure properties of context-free languages
  - The class of context-free languages is closed under $\cup, \cdot, *$
  - The class of context-free languages is not closed under $\cap$, complement
- Decision problems for context-free languages
  - Decidable: Word problem, emptiness problem
  - Undeciable: Equivalence

# Summary

- ▶ Introduction to parsing
  - ▶ Recognize words (programs) in $L(G)$
  - ▶ Understand structure of words
- ▶ YACC/Bison
  - ▶ Automatic parser generation
    - ▶ Core: Syntax rules with actions (C code)
    - ▶ Whenever input can be *reduced*, action is executed
    - ▶ Often: Manipulation of *semantic values*
  - ▶ Example: desk calculator
- ▶ LALR(1) shift/reduce parsing
  - ▶ Shift input to stack
  - ▶ Reduce RHS to LHS
  - ▶ Use lookahead to reduce ambiguity
- ▶ Parse trees and abstract syntax trees
  - ▶ Parse trees: Represents not just word but derivation
  - ▶ AST: "parse tree without syntactic details

# 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?

# Outline

# Goals for Lecture 17

- ▶ Review of last lecture
- ▶ Beyond context-free languages: Turing Machines
    - ▶ Idea
    - ▶ Definition
    - ▶ Examples
    - ▶ Variants
- ▶ Context-sensitive languages
    - ▶ Linear bounded Turing Machine (LBA)
    - ▶ Closure properties
    - ▶ Decision problems
- ▶ Type-0 languages
    - ▶ The Universal Turing Machine
    - ▶ The Halting Problem
- ▶ Thesis of Church-Turing
    - ▶ . . . and practical applications

# Review

- ▶ Introduction to parsing
  - ▶ Recognize words (programs) in $L(G)$
  - ▶ Understand structure of words
- ▶ YACC/Bison
  - ▶ Automatic parser generation
    - ▶ Core: Syntax rules with actions (C code)
    - ▶ Whenever input can be *reduced*, action is executed
    - ▶ Often: Manipulation of *semantic values*
  - ▶ Example: desk calculator
- ▶ LALR(1) shift/reduce parsing
  - ▶ Shift input to stack
  - ▶ Reduce RHS to LHS
  - ▶ Use lookahead to reduce ambiguity
- ▶ Parse trees and abstract syntax trees
  - ▶ Parse trees: Represents not just word but derivation
  - ▶ AST: "parse tree without syntactic details

# Summary

- ▶ Turing Machines
  - ▶ Unbounded tape memory
  - ▶ Finite automaton controls head movement, writing based on current state and input at current tape location
  - ▶ Configurations (general, stopping and accepting)
  - ▶ TM for $a^n b^n c^n$
- ▶ Context-sensitive languages
  - ▶ Linear bounded Turing Machine (LBA)
  - ▶ Closure properties: Closed under $\cup, \cdot, ^*, \cap, \overline{\phantom{a}}$
  - ▶ Decision problems: Word yes, emptiness, equivalence no
- ▶ Type-0 languages
  - ▶ The Universal Turing Machine
  - ▶ The Halting Problem
- ▶ Thesis of Church-Turing: All "strong" computing models are equivalent
  - ▶ . . . and so may be humans (!)

▶ 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?

# Outline

# 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?

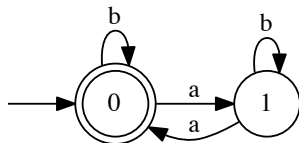# Outline

# Bonus Exercises

Assume $\Sigma = \{a, b\}$. Consider the automaton $A$.

- ▶ Give a formal description of $A$.

- ▶ Which language $L(A)$ is accepted by A? Give a formal description.

- ▶ Where does $L(A)$ reside in the Chomsky-hierarchy?

- ▶ Manually find a regular expression $R$ with $L(R) = L(A)$.

- ▶ Generate a system of equations for $A$ and generate a regular expression $R_S$ for $L(A)$ by solving this. Compare your result the the previous one.

- ▶ Convert $A$ into a regular grammar $G$ with $L(A) = L(G)$.



- ▶ Can $L(A)$ be pumped? If yes, provide an example of a pumpable word.

- ▶ Generalize $A$ to recognize $L_3 = \{w \in \Sigma^* \mid |w|_a \text{ modulo } 3 = 0\}$

- ▶ Form the product automaton $P$ to find $L(A) \cap L_3$.

- ▶ Minmize $P$.

- ▶ Systematically construct a NFA $D_A$ for $R_S$.

- ▶ Convert $D_R$ to a DFA.

498

# Outline

# Equivalence of regular expressions

Solution to Exercise: Algebra on regular expressions (1)

▶ Claim: $r^* \doteq \varepsilon + r^*$

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

# Simplification of regular expressions

Solution to Exercise: Algebra on regular expressions (2)

$$
\begin{aligned}
r &= 0(\varepsilon + 0 + 1)^* + (\varepsilon + 1)(1 + 0)^* + \varepsilon \\
&\overset{14,1}{\doteq} 0(0 + 1)^* + (\varepsilon + 1)(0 + 1)^* + \varepsilon \\
&\overset{7}{\doteq} 0(0 + 1)^* + \varepsilon(0 + 1)^* + 1(0 + 1)^* + \varepsilon \\
&\overset{5}{\doteq} 0(0 + 1)^* + (0 + 1)^* + 1(0 + 1)^* + \varepsilon \\
&\overset{1,7}{\doteq} \varepsilon + (0 + 1)(0 + 1)^* + (0 + 1)^* \\
&\overset{16}{\doteq} \varepsilon + (0 + 1)^*(0 + 1) + (0 + 1)^* \\
&\overset{13}{\doteq} (0 + 1)^* + (0 + 1)^* \\
&\overset{9}{\doteq} (0 + 1)^*.
\end{aligned}
$$

# Application of Arto's lemma

Solution to Exercise: Algebra on regular expressions (3)

- ► Show that $u = 10(10)^* \doteq 1(01)^*0$
- ► Idea: $u$ is of the form $ts^*$ with:
  - ► $t = 10$
  - ► $s = 10$
  - ► This suggest Arto's Lemma. To apply the lemma, we must show that $r = 1(01)^*0 \doteq rs + t$

$$
\begin{aligned}
rs + t &= 1(01)^*010 + 10 \\
&\doteq 1((01)^*010 + 0) \quad \text{(factor out 1)} \\
\text{► So:} \qquad &\doteq 1((01)^*01 + \varepsilon)0 \quad \text{(factor out 0)} \\
&\doteq 1(01)^*0 \qquad\qquad (14) \\
&= r
\end{aligned}
$$

  - ► Since $L(s) = \{10\}$ (and hence $\varepsilon \notin L(s)$), we can apply Arto and rewrite $r \doteq ts^* \doteq 10(10)^*$.
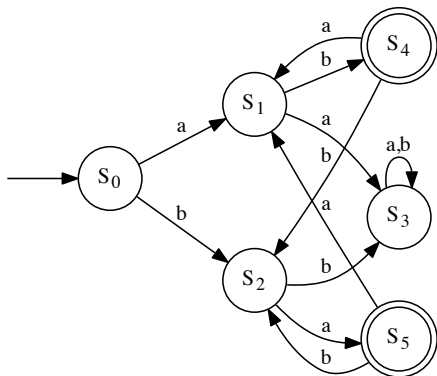
# Transformation into DFA (1)

- 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:
  $L(\mathcal{A}) = L((ab + ba)(ab + ba)^*)$

504
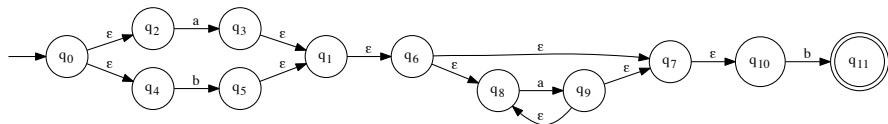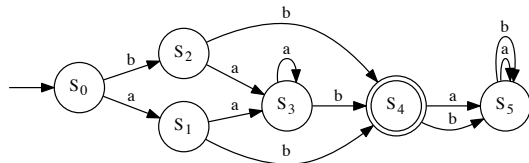
# Transformation of RE into NFA

Systematically construct an NFA accepting the same language as the regular expression $(a + b)a^*b$.
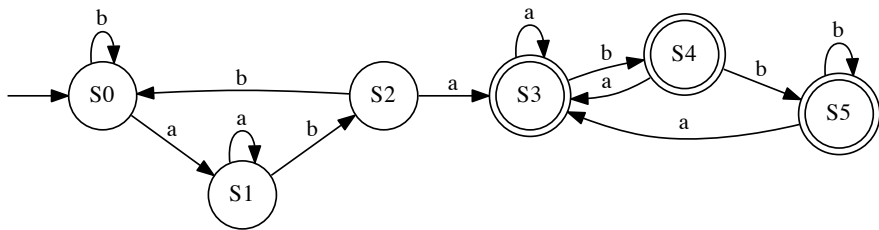
Solution:



Corresponding DFA:

# Solution: NFA to DFA "aba"
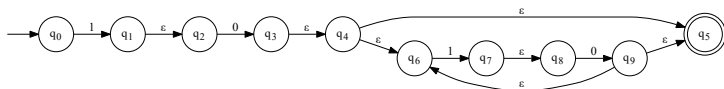
# Show $10(10)^* \doteq 1(01)^*0$ via minimal DFAs (1)

▶ Step 1: NFA for $10(10)^*$:

```
     | epsilon      0        1
-----------------------------
-> q0 |      {}     {}     {q1}
   q1 |   {q2}      {}       {}
   q2 |      {}   {q3}       {}
   q3 |   {q4}      {}       {}
   q4 | {q5,q6}     {}       {}
*  q5 |      {}     {}       {}
   q6 |      {}     {}     {q7}
   q7 |   {q8}      {}       {}
   q8 |      {}   {q9}       {}
   q9 | {q5,q6}     {}       {}
```
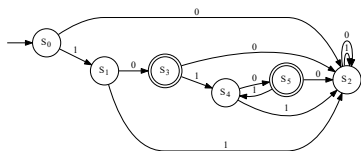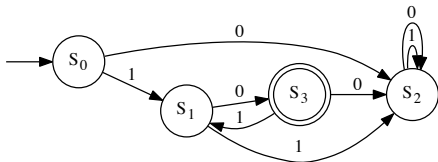
# Show $10(10)^* \doteq 1(01)^*0$ via minimal DFAs (2)

▶ Step 2: DFA $\mathcal{A}$ for $10(10)^*$:



▶ Step 3: Minimizing of $\mathcal{A}$

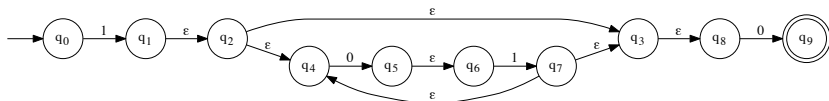|       | $S_0$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ |
|-------|-------|-------|-------|-------|-------|-------|
| $S_0$ | o     | x     | x     | x     | x     | x     |
| $S_1$ | x     | o     | x     | x     | o     | x     |
| $S_2$ | x     | x     | o     | x     | x     | x     |
| $S_3$ | x     | 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 $10(10)^* \doteq 1(01)^*0$ via minimal DFAs (3)

▶ Step 4: NFA zu $1(01)^*0$:

```
      | epsilon    0      1
---------------------------
-> q0 |      {}    {}   {q1}
   q1 |   {q2}     {}     {}
   q2 | {q3,q4}    {}     {}
   q3 |   {q8}     {}     {}
   q4 |      {}  {q5}     {}
   q5 |   {q6}     {}     {}
   q6 |      {}    {}   {q7}
   q7 | {q4,q3}    {}     {}
   q8 |      {}    {}   {q9}
 * q9 |      {}    {}     {}
```
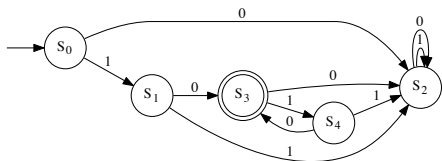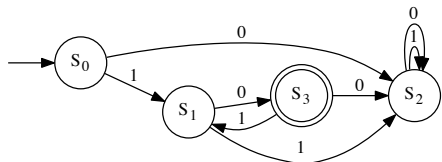
# Show $10(10)^* \doteq 1(01)^*0$ via minimal DFAs (4)

- Step 5: DFA $\mathcal{B}$ for `1(01)*0`



- Step 6: Minimization of $\mathcal{B}$

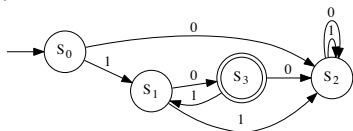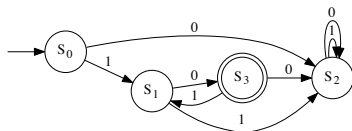|       | $S_0$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|-------|-------|-------|-------|-------|-------|
| $S_0$ | o     | x     | x     | x     | x     |
| $S_1$ | x     | o     | x     | x     | o     |
| $S_2$ | x     | x     | o     | x     | x     |
| $S_3$ | x     | x     | x     | o     | x     |
| $S_4$ | x     | o     | x     | x     | o     |

Result: $(S_1, S_4)$ can be merged

# Show $10(10)^* \doteq 1(01)^*0$ via minimal DFAs (5)

► Step 7: Comparision of $\mathcal{A}^-$ and $\mathcal{B}^-$

$\mathcal{A}^-$



$\mathcal{B}^-$



► Result: The two automata are identical, hence the two original regular expressions describe the same languages.

# 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.*
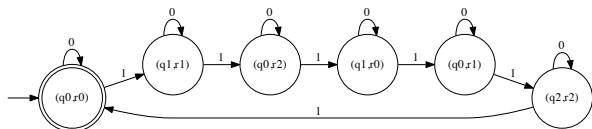
# Solution: Product automaton for $L_1 \cap L_2$

Solution to Exercise: Product automaton

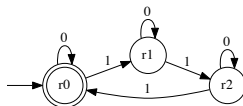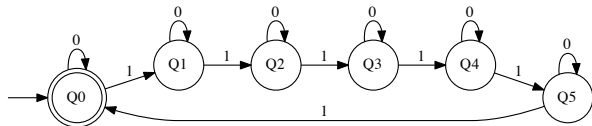$A_1$:
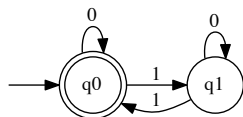
$A_1 \times A2$ for $L_1 \cap L_2$:



$A_2$:

Äquivalent Automaton for $L_1 \cap L_2$:

# Solution: Transformation to Chomsky Normal Form (1)

Compute the Chomsky normal form of the following grammar:
$G = (N, \Sigma, P, S)$

- $N = \{S, A, B, C, D, E\}$
- $\Sigma = \{a, b\}$
- $P$ :
  $$
  \begin{array}{llll}
  S & \to & AB|SB|BDE & \quad C \to SB \\
  A & \to & Aa & \quad D \to E \\
  B & \to & bB|BaB|ab & \quad E \to \varepsilon
  \end{array}
  $$

Step 1: $\varepsilon$-Elimination

- Nullable NTS: $N = \{E, D\}$
- New rules:
  $$
  \begin{array}{ll}
  S \to BD & \text{(from } S \to BDE, \beta_1 = BD, \beta_2 = \varepsilon) \\
  S \to BE & \text{(from } S \to BDE, \beta_1 = B, \beta_2 = E) \\
  S \to B & \text{(from } S \to BD \text{ or } S \to BE, \beta_1 = B, \beta_2 = \varepsilon) \\
  D \to \varepsilon & \text{(from } D \to E, \beta_1 = \varepsilon, \beta_2 = \varepsilon)
  \end{array}
  $$
- Remove $E \to \varepsilon$, $D \to \varepsilon$

# Solution: Transformation to Chomsky Normal Form (2)

Step 2: Elimination of Chain Rules.

- ▶ Current chain rules: $S \rightarrow B$, $D \rightarrow E$
- ▶ Eliminate $S \rightarrow B$:
  - ▶ $N(S) = \{B\}$
  - ▶ New rules: $S \rightarrow bB, S \rightarrow BaB, S \rightarrow ab$
- ▶ Eliminate $D \rightarrow E$
  - ▶ $N(D) = \{E\}$
  - ▶ $E$ has no rule, therefore no new rules!
- ▶ Current state of $P$:

$$
\begin{aligned}
S &\rightarrow AB|SB|BDE|BD|BE|bB|BaB|ab & C &\rightarrow SB \\
A &\rightarrow Aa & B &\rightarrow bB|BaB|ab
\end{aligned}
$$

# 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:

  $$
  \begin{aligned}
  S &\rightarrow SB|bB|BaB|ab & C &\rightarrow SB \\
  B &\rightarrow bB|BaB|ab
  \end{aligned}
  $$

▶ Reachable symbols: $R = \{S, B\}$ ($C$ is not reachable)

  ▶ Remove all rules containing $C$. Remaining:

  $$
  \begin{aligned}
  S &\rightarrow SB|bB|BaB|ab \\
  B &\rightarrow bB|BaB|ab
  \end{aligned}
  $$

Step 4: Introduce new non-terminals for terminals

▶ New rules: $X_a \to a, X_b \to b$. Result:

$$
\begin{array}{llll}
S & \to & SB|X_bB|BX_aB|X_aX_b & \quad X_a \to a \\
B & \to & X_bB|BX_aB|X_aX_b & \quad X_b \to b
\end{array}
$$

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:

$$
\begin{array}{llll}
S & \to & SB|X_bB|BC_1|X_aX_b & \quad X_a \to a \\
B & \to & X_bB|BC_1|X_aX_b & \quad X_b \to b \\
C_1 & \to & X_aB
\end{array}
$$

Final grammar: $G' = (N', \Sigma, P', S)$ with

▶ $N' = \{S, B, C_1, X_a, X_b\}$

▶ $\Sigma = \{a, b\}$

▶ $P'$ :
$$
\begin{array}{llll}
S & \to & SB|X_bB|BC_1|X_aX_b \qquad & X_a \to a \\
B & \to & X_bB|BC_1|X_aX_b & X_b \to b \\
C_1 & \to & X_aB
\end{array}
$$

Back to exercise

519

## Solution: PDA to Grammar (1)

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

▶ $Q = \{0, 1\}$

▶ $\Sigma = \{a, b\}$

▶ $\Gamma = \{A, Z\}$

$$\Delta : \quad \begin{matrix} (1) & 0 & \varepsilon & Z & \to & \varepsilon & 0 \\ (2) & 0 & a & Z & \to & AZ & 0 \\ (3) & 0 & a & A & \to & AA & 0 \\ (4) & 0 & b & A & \to & \varepsilon & 1 \\ (5) & 1 & b & A & \to & \varepsilon & 1 \\ (6) & 1 & \varepsilon & Z & \to & \varepsilon & 1 \end{matrix}$$

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

▶ $N = \{S, [0A0], [0A1], [1A0], [1A1], [0Z0], [0Z1], [1Z0], [1Z1]\}$

▶ $\Sigma$ and $S$ as given

▶ Start rules for $P$:
  ▶ $S \to [0Z0]$
  ▶ $S \to [0Z1]$

▶ From transitions:
  ▶ From (1): $[0Z0] \to \varepsilon$
  ▶ From (4): $[0A1] \to b$
  ▶ From (5): $[1A1] \to b$
  ▶ From (6): $[1Z1] \to \varepsilon$
  ▶ (2) and (3): Next page

## Solution: PDA to Grammar (2)

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

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

$$\Delta : \begin{array}{llllll} (1) & 0 & \varepsilon & Z & \to & \varepsilon & 0 \\ (2) & 0 & a & Z & \to & AZ & 0 \\ (3) & 0 & a & A & \to & AA & 0 \\ (4) & 0 & b & A & \to & \varepsilon & 1 \\ (5) & 1 & b & A & \to & \varepsilon & 1 \\ (6) & 1 & \varepsilon & Z & \to & \varepsilon & 11 \end{array}$$

Computing of $P$ continued:

- From (2):
  - $[0Z0] \to a[0A0][0Z0]$
  - $[0Z0] \to a[0A1][1Z0]$
  - $[0Z1] \to a[0A0][0Z1]$
  - $[0Z1] \to a[0A1][1Z1]$

- From (3):
  - $[0A0] \to a[0A0][0A0]$
  - $[0A0] \to a[0A1][1A0]$
  - $[0A1] \to a[0A0][0A1]$
  - $[0A1] \to a[0A1][1A1]$

521

## Solution: PDA to Grammar (3)

Full grammar $G = \{N, \Sigma, P, S\}$

- $N = \{S, [0A0], [0A1], [1A0], [1A1], [0Z0], [0Z1], [1Z0], [1Z1]\}$

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

- $P$:
  - $S \rightarrow [0Z0]$
  - $S \rightarrow [0Z1]$
  - $[0Z0] \rightarrow \varepsilon$
  - $[0A1] \rightarrow b$
  - $[1A1] \rightarrow b$
  - $[1Z1] \rightarrow \varepsilon$
  - $[0Z0] \rightarrow a[0A0][0Z0]$
  - $[0Z0] \rightarrow a[0A1][1Z0]$
  - $[0Z1] \rightarrow a[0A0][0Z1]$
  - $[0Z1] \rightarrow a[0A1][1Z1]$
  - $[0A0] \rightarrow a[0A0][0A0]$
  - $[0A0] \rightarrow a[0A1][1A0]$
  - $[0A1] \rightarrow a[0A0][0A1]$
  - $[0A1] \rightarrow a[0A1][1A1]$

- Terminating: $T = \{[0Z0], [0A1], [1A1], 1Z1], S, [0Z1]\}$

- Remaining rules:
  1. $S \rightarrow [0Z0]$
  2. $S \rightarrow [0Z1]$
  3. $[0Z0] \rightarrow \varepsilon$
  4. $[0A1] \rightarrow b$
  5. $[1A1] \rightarrow b$
  6. $[1Z1] \rightarrow \varepsilon$
  7. $[0Z1] \rightarrow a[0A1][1Z1]$
  8. $[0A1] \rightarrow a[0A1][1A1]$

- Reachable:
  $R = \{S, [0Z0], [0Z1], [0A1], [1Z1], [1A1]\}$
  - No change!

522

## Solution: PDA to Grammar (4)

- *P*:
  1. $S \to [0Z0]$
  2. $S \to [0Z1]$
  3. $[0Z0] \to \varepsilon$
  4. $[0A1] \to b$
  5. $[1A1] \to b$
  6. $[1Z1] \to \varepsilon$
  7. $[0Z1] \to a[0A1][1Z1]$
  8. $[0A1] \to a[0A1][1A1]$

- Derivation of $\varepsilon$:
  - $S \Rightarrow_1 [0Z0] \Rightarrow_3 \varepsilon$
- Derivation of *ab*:
  - $S \Rightarrow_2 [0Z1] \Rightarrow_7 a[0A1][1Z1] \Rightarrow_4$
    $ab[1Z1] \Rightarrow_6 ab$
- Derivation of *aabb*:
  - $S \Rightarrow_2 [0Z1] \Rightarrow_7 a[0A1][1Z1] \Rightarrow_8$
    $aa[0A1][1A1][1Z1] \Rightarrow_4$
    $aab[1A1][1Z1] \Rightarrow_5 aabb[1Z1] \Rightarrow_6$
    $aabb$