Formal Languages and Automata

Stephan Schulz & Jan Hladik
stephan.schulz@dhbw-stuttgart.de
jan.hladik@dhbw-stuttgart.de
with contributions from David Suendermann
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
Goals for Today

- Getting acquainted
- Practical issues
- Course outline and motivation
- Basics of formal languages
- Regular expressions
Practical Issues

▶ One lecture per week
  ► Wednesday, 8:45-12:15
  ► 10 minute break around 10:15
  ► I’ll try to keep it entertaining...

▶ Exceptions
  ► 24.9. (T2000 examination)
  ► 19.11. (Tag der Informatik)
    ► This is the review class, need to reschedule

▶ Written exam
  ► Calendar week 48 (24.11.–28.11.)
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/fla2014.html

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

http://wwwlehre.dhbw-stuttgart.de/~stroetma/Formal-Languages/formal-languages.pdf

▶ Books

► John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman: Introduction to Automata Theory, Languages, and Computation
► Michael Sipser: Introduction to the Theory of Computation
► Dirk W. Hoffmann: Theoretische Informatik
► Ulrich Hedtstück: Einführung in die theoretische Informatik
Computing Environment

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

- **Phase 1 (individual)**
  - 3 minutes
  - List at least 3 topics/results you want(expect from this course

- **Phase 2 (partners)**
  - 3 minutes
  - Condense to “top 3” choices

- **Phase 3**
  - Presentation
Outline

Introduction

Regular Languages and Finite State Automata
Formal languages

- Sets of words (strings) over a finite alphabet

Examples

- All names in a phone directory
- All phone numbers in a phone directory
- All legal C identifiers
- All legal C programs
- All legal HTML 4.01 Transitional documents
- The empty set
- The set of all ASCII strings
- The set of all Unicode strings
Questions on languages

► Language description?
  ► What are the legal word in a language?
  ► What are syntactically correct LISP programs?
  ► How can we describe languages in general?

Formal grammars – regular expressions

► Language recognition/understanding?
  ► Is this a legal word in a language?
  ► How is this JAVA program constructed?
  ► How should I translate this C program/render this HTML page?

Finite state machines – Push-down automata – Syntax trees – Universal computers
More questions on languages

- For a given language, can I decide if a word is in the language?
  - ... with a finite and fast machine?
  - ... with a simple but infinite machine?
  - ... with arbitrary but known resources?
  - ... at all?
Abandon all hope...
Finite Automata - Example

Initial state:

Transitions:

States:

Off

On

Off
Finite Automata - Example

- Formally:
  - \( Q = \{ \text{Off, On} \} \) is the set of states
  - \( \Sigma = \{ \text{click} \} \) is the alphabet
  - The transition function \( \delta \) is given by
    
    \[
    \begin{array}{c|c}
    \text{off} & \text{on} \\
    \hline
    \text{click} & \text{on} \\
    \text{on} & \text{off}
    \end{array}
    \]
  - The initial state is Off
  - There are no accepting states
ATC scenario

Aggregator

ATC Center (controllers)
ATC redundancy

Aktive server:
- Accepts sensor data
- Provides ASP
- Sends “alive” messages

Passive server
- Ignores sensor data
- Monitors “alive” messages
- Takes over in case of failure
DFA to the rescue

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

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

- Phase 1 (individual)
  - 3 minutes
  - List at least 3 topics/results you want/expect from this course

- Phase 2 (different partners)
  - 3 minutes
  - Condense to “top 3” choices

- Phase 3
  - Presentation
Basics of formal languages
An alphabet $\Sigma$ is a finite, non-empty set of characters (symbols, letters):

$$\Sigma = \{c_1, \cdots, c_n\}. \quad (1)$$

Examples:

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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>SOH</td>
<td>STX</td>
<td>ETX</td>
<td>EOT</td>
<td>ENQ</td>
</tr>
<tr>
<td>1</td>
<td>DLE</td>
<td>DC1</td>
<td>DC2</td>
<td>DC3</td>
<td>DC4</td>
<td>NAK</td>
</tr>
<tr>
<td>2</td>
<td>Syn</td>
<td>ETB</td>
<td>CAN</td>
<td>EM</td>
<td>SUB</td>
<td>ESC</td>
</tr>
<tr>
<td>3</td>
<td>FS</td>
<td>GS</td>
<td>RS</td>
<td>US</td>
<td>FS</td>
<td>GS</td>
</tr>
<tr>
<td>4</td>
<td>RS</td>
<td>US</td>
<td>FS</td>
<td>GS</td>
<td>RS</td>
<td>US</td>
</tr>
<tr>
<td>5</td>
<td>FS</td>
<td>GS</td>
<td>RS</td>
<td>US</td>
<td>FS</td>
<td>GS</td>
</tr>
<tr>
<td>6</td>
<td>FS</td>
<td>GS</td>
<td>RS</td>
<td>US</td>
<td>FS</td>
<td>GS</td>
</tr>
<tr>
<td>7</td>
<td>FS</td>
<td>GS</td>
<td>RS</td>
<td>US</td>
<td>FS</td>
<td>GS</td>
</tr>
</tbody>
</table>

```
0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0   1  2  3  4  5  6  7  8  9   0   1  2  3  4
@  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _
`  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~  DEL
```
Words

- A word of the alphabet $\Sigma$ is a sequence (list) of characters of $\Sigma$:
  \[ w = c_1 \cdots c_n \quad \text{with} \quad c_1, \ldots, c_n \in \Sigma. \quad (2) \]
- The empty word is written as
  \[ w = \varepsilon. \quad (3) \]
- The set of all words of an alphabet $\Sigma$ is represented by $\Sigma^*$.  
- In programming languages, words are also referred to as strings.
- Examples:
  1. Using the aforementioned set $\Sigma_{\text{bin}}$, we can define the words
     \[ w_1 = 01100 \quad \text{and} \quad w_2 = 11001 \quad \text{with} \quad w_1, w_2 \in \Sigma_{\text{bin}}^*. \quad (4) \]
  2. Using the aforementioned set $\Sigma_{\text{en}}$, we can define the word
     \[ w = \text{example} \quad \text{with} \quad w \in \Sigma_{\text{en}}^*. \quad (5) \]
Length, character access

- We refer to the length of a word \( w \) as \( |w| \), e.g.:

  \[
  w = \text{example} \quad \text{with} \quad w \in \Sigma^* \quad \rightarrow \quad |w| = 7. \quad (6)
  \]

- We refer to the number of occurrences of a symbol \( l \) in \( w \) as \( |w|_l \), e.g.

  \[
  |\text{example}|_e = 2 \quad (7)
  \]

  and

  \[
  |\text{example}|_k = 0 \quad (8)
  \]

- We access individual characters within words using the terminology \( w[i] \) with \( i \in \{1, 2, \cdots, |w|\} \), e.g.

  \[
  \text{example}[4] = m \quad (9)
  \]
Concatenation

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

  $$w = w_1 w_2 \cdots w_n.$$  

  (10)

- Concatenation example:

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

  $$\rightarrow$$

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

  (11)
In the following, we will be frequently using the set of natural numbers
\[ \mathbb{N} = \{0, 1, \ldots \} \] (12)

The \( n \)th power of a word \( w \) concatenates the same word \( n \) times:
\[ w^n = w^{n-1}w \text{ with } w^0 = \varepsilon \text{ and } n \in \mathbb{N}, n \neq 0. \] (13)
Given the alphabet $\Sigma$, we refer to a subset $L \subseteq \Sigma^*$ as a formal language.
We define

\[ L_N = \{1w \mid w \in \Sigma_{\text{bin}}^* \} \cup \{0\}. \]  

(14)

Then, \( L_N \) is the set of all those words that represent integers using the binary system (all words starting with 1 and the word 0). Hence, we have

\[ 100 \in L_N \quad \text{but} \quad 010 \not\in L_N. \]  

(15)
We define the function

\[ d : L_\mathbb{N} \rightarrow \mathbb{N} \]  

as the function returning the numeric value of a word in the language \( L_\mathbb{N} \). This gives us

(a) \( d(0) = 0 \),
(b) \( d(1) = 1 \),
(c) \( d(w^0) = 2d(w) \) for \( |w| > 0 \),
(d) \( d(w^1) = 2d(w) + 1 \) for \( |w| > 0 \).
Formal languages - examples (3)

- We define the language $L_P$ as the language representing prime numbers in the binary system:

\[
L_P = \{ w \in L_N \mid d(w) \in \mathbb{P} \}. \tag{17}
\]

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

\[
\mathbb{P} = \{ p \in \mathbb{N} \mid \{ t \in \mathbb{N} \mid \exists k \in \mathbb{N} : kt = p \} = \{1, p\} \}. \tag{18}
\]
We define the language $L_C \subseteq \Sigma^{*}_{\text{ASCII}}$ as the set of all C functions with a declaration of the form

$$\text{char}\ast \ f(\text{char}\ast \ x);$$

(19)

that is, $L_C$ contains the ASCII code of all those C functions processing and returning a string.
Using the alphabet $\Sigma_{\text{ASCII}^+} = \Sigma_{\text{ASCII}} \cup \{\dagger\}$, we define the universal language

$$L_u = \{f\dagger x\dagger y\} \quad \text{with} \quad (20)$$

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

These examples show that formal languages have a wide scope.

Testing whether a word belongs to $L_N$ is straightforward whereas the same test for $L_P$ or $L_C$ is more complicated.

Later, we will see that there is no algorithm to do this test for $L_u$. 
Abandon all hope...
Given an alphabet $\Sigma$ and the formal languages $L_1, L_2 \subseteq \Sigma^*$, we define the **product**

$$L_1 \cdot L_2 = \{w_1 w_2 | w_1 \in L_1, w_2 \in L_2\}.$$  \hfill (21)

**Example:**
Using the alphabet $\Sigma_{en}$, we define the languages

$$L_1 = \{ab, bc\} \quad \text{and} \quad L_2 = \{ac, cb\}.$$ \hfill (22)

The product is

$$L_1 \cdot L_2 = \{abac, abcb, bcac, bccb\}.$$ \hfill (23)
Power of a language

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

$$L^n = L^{n-1} \cdot L \quad \text{with} \quad L^0 = \{ \varepsilon \}. \quad (24)$$

Using the alphabet $\Sigma_{en}$, we define the language $L = \{ab, ba\}$.

This gives us

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

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

$$L^2 = \{ab, ba\} \cdot \{ab, ba\} = \{abab, abba, baab, baba\}. \quad (26)$$
The Kleene Star

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

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

(27)

- Example:
  Using the alphabet $\Sigma_{en}$, we define the language

$$L = \{a\}.$$  

(28)

This gives us

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

(29)
Given the alphabet $\Sigma_{\text{bin}}$ and the language $L = \{1\}$. (30)

a) Formally describe the language $L' = L^* \setminus \{\varepsilon\}$. (31)

b) Formally describe the set $D = \{d(w) | w \in L'\}$. (32)

c) Formally describe the language $L'_- = \{w | d(w) - 1 \in D\}$. (33)

d) Formally describe the language $L'_+ = \{w | d(w) + 1 \in D\}$. (34)
Think!
Regular Expressions
Regular expressions

- Regular expressions
  - Compact way to represent a set of strings
  - Convenient way to represent a set of strings
- Widely used, e.g.
  - Characterize tokens for compilers
  - Describe search terms for a data base
  - Filter through genomic data
  - Extract URLs from web pages
  - Extract email addresses from web pages

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

Each single regular expression describes a formal language
Regular expressions and formal languages

- Using the alphabet $\Sigma$, we refer to the set of all regular expressions as $R$.
- We introduce a function
  \[
  L : R \rightarrow 2^{\Sigma^*}
  \]
  assigning a formal language $L(r) \subseteq \Sigma^*$ to each regular expression $r$.
- Here, $2^S$ denotes the power set of a set $S$.
- E.g.,
  \[
  2^{\Sigma_{\text{bin}}} = 2^{\{0,1\}} = \{\emptyset, \{0\}, \{1\}, \{0, 1\}\},
  \]
  and
  \[
  2^{\Sigma^*_{\text{bin}}} = 2^{\{\varepsilon, 0, 1, 00, 01, \ldots\}}
  \]
  \[
  = \{\emptyset, \{\varepsilon\}, \{0\}, \{1\}, \{00\}, \{01\}, \ldots \}
  \]
  \[
  \ldots \{\varepsilon, 0\}, \{\varepsilon, 1\}, \{\varepsilon, 00\}, \{\varepsilon, 01\}, \ldots \]
  \[
  \ldots \{010, 1110, 10101\}, \ldots \}.
  \]
The set of regular expressions

- The set of regular expressions \((R)\) is defined as follows:
  1. The regular expression \(\emptyset\) is associated with the empty language:
     \[
     L(\emptyset) = \emptyset \quad \text{with} \quad \emptyset \in R.
     \]
     (38)
  2. The regular expression \(\varepsilon\) is associated with the language containing only the empty word:
     \[
     L(\varepsilon) = \{\varepsilon\} \quad \text{with} \quad \varepsilon \in R.
     \]
     (39)
  3. Each symbol in the alphabet \(\Sigma\) is also a regular expression:
     \[
     c \in \Sigma \implies c \in R;
     \]
     \[
     L(c) = \{c\}.
     \]
     (40)
  4. We define the infix operator “+” generating new regular expressions by merging the languages of the regular expressions \(r_1\) and \(r_2\):
     \[
     r_1 \in R, r_2 \in R \implies r_1 + r_2 \in R;
     \]
     \[
     L(r_1 + r_2) = L(r_1) \cup L(r_2).
     \]
     (41)
The set of regular expressions (cont.)

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

$$r_1 \in R, r_2 \in R \rightarrow r_1 \cdot r_2 \in R;$$

$$L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2).$$  (42)

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

$$r \in R \rightarrow r^* \in R;$$

$$L(r^*) = L^*(r).$$  (43)

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

$$r \in R \rightarrow (r) \in R;$$

$$L((r)) = L(r).$$  (44)
Operator precedences

- To save brackets, we introduce the following operator precedences:
  1. “(”, “)” (strongest)
  2. “∗”
  3. “·”
  4. “+” (weakest)

- Example:
  \[ a + b \cdot c^* = a + (b \cdot (c^*)) \]  
  (45)

- For the sake of further simplicity, the product operator “·” can also be omitted, e.g.:
  \[ a + b \cdot c^* = a + bc^* \]  
  (46)

- Note: Some authors (and tools) use | instead of + to denote alternatives
For all the following examples, we are using the alphabet

\[ \Sigma_{abc} = \{a, b, c\}. \] (47)

1. The regular expression

\[ r_1 = (a + b + c)(a + b + c) \] (48)

describes all the words of exactly two symbols:

\[ L(r_1) = \{ w \in \Sigma_{abc} \mid |w| = 2 \}. \] (49)

2. The regular expression

\[ r_2 = (a + b + c)(a + b + c)^* \] (50)

describes all the words of one or more symbols:

\[ L(r_1) = \{ w \in \Sigma_{abc} \mid |w| \geq 1 \}. \] (51)
Regular expressions: exercises

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

b) Which language is expressed by $r_a$?

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

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

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

f) Which language is expressed by the regular expression

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

- Install an operational UNIX/Linux environment (per slide 7) on your computer.
- To test your installation, download and execute the program `miu.py` from the course web page [http://www.lehre.dhbw-stuttgart.de/~sschulz/fla2014.html](http://www.lehre.dhbw-stuttgart.de/~sschulz/fla2014.html)
- Read the source code of the program. What does it do? Try different parameter combinations.
Review of Goals

- Getting acquainted
- Practical issues
- Course outline and motivation
- Basics of formal languages
- Regular expressions
Feedback round

► What was the best part of today's lecture?
► What part of today's lecture has the most potential for improvement?
  ► Optional: how would you improve it?
Introduction Review
Goals for Today

- Review and mental warm-up
- Proofs
- Regular expression algebra
- Deterministic finite automata
Formal languages: Basics

- Alphabet $\Sigma$: Finite, nonempty set of characters (symbols/letters)
- Words: Finite sequences of characters
  - $\varepsilon$ is the empty word ($|\varepsilon| = 0$)
  - $abcab[3] = c$
  - $|abcab|_a = 2$
- $\Sigma^*$: Set of all words over $\Sigma$
- A formal language $L$ over $\Sigma$ is a (finite or infinite) set of words $L \subseteq \Sigma^*$
Give 5 examples each of formal languages with a suitable alphabet from the areas of
- Computer programming
- Human communication
- Data/knowledge collections

Formally describe the following languages (if they are languages):
- The set of all square numbers in decimal representation
- The set of all square roots of natural numbers in decimal representation
Regular expressions: Basics

► Elementary REs (over a given alphabet $\Sigma$)
  ► $L(\emptyset) = \{\}$
  ► $L(\varepsilon) = \{\varepsilon\}$
  ► $L(a) = \{a\}$ for $a \in \Sigma$

► Composite REs (with existing REs $r, r_1, r_2$):
  ► $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  ► $L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2)$
  ► $L(r^*) = L^*(r) (= \bigcup_{i \in \mathbb{N}} L^i(r_1))$

► Parentheses can be used to group subexpressions

► Operator precedence: ( ) > * > · > +
Regular expressions: Warm-up

- Characterise the languages described by the following REs over $\Sigma = \{0, 1\}$:
  - $(0 + 1)^*111(0 + 1)(0 + 1)$
  - $(ba)^*(a + b)^*(ab)^*$

- Find regular expressions for the following languages over $\Sigma = \{a, b\}$ (if possible):
  - $\{w \in \Sigma^* \mid w \text{ contains } ab\}$
  - $\{w \in \Sigma^* \mid |w| \geq 3 \text{ and } w[3] = a\}$
  - $\{w \in \Sigma^* \mid |w| < 3 \text{ or } w[2] = w[|w| - 2] = a\}$
  - $\{a^n b^m \mid n, m \in \mathbb{N}\}$
  - $\{a^n b^n \mid n \in \mathbb{N}\}$
  - $\{a^n a^n \mid n \in \mathbb{N}\}$
Excursion: proofs

A proof is a (semi-)formal argument that necessarily convinces an open-minded, rational, educated being of the truth of a statement.

- argument – a chain of logically connected steps
- necessarily – the argument is sound and complete
- open-minded – the recipient must be willing to consider the argument
- rational – the recipient must be able to follow the logic
- educated – the recipient must understand the concepts involved

Corollary: The form of a (semi-formal) proof depends on the audience!
More on Regular Expressions
We define two regular expressions $r_1$ and $r_2$ as equivalent, if $L(r_1) = L(r_2)$.

In that case, we write $r_1 \equiv r_2$.

Formally:

\[ r_1 \equiv r_2 \text{ if and only if } L(r_1) = L(r_2) \quad (53) \]
Algebraic operations on regular expressions

1. $r_1 + r_2 \equiv r_2 + r_1$ (commutative law)  
   This equivalence can be proven using the commutativity of set union:
   \[ L(r_1 + r_2) = L(r_1) \cup L(r_2) = L(r_2) \cup L(r_1) = L(r_2 + r_1). \]  
   (54)

2. $(r_1 + r_2) + r_3 \equiv r_1 + (r_2 + r_3)$ (associative law)

3. $(r_1 r_2) r_3 \equiv r_1 (r_2 r_3)$ (associative law)

4. $\emptyset r \equiv \emptyset$

5. $\varepsilon r \equiv r$

6. $\emptyset + r \equiv r$

7. $(r_1 + r_2) r_3 \equiv r_1 r_3 + r_2 r_3$ (distributive law)

8. $r_1 (r_2 + r_3) \equiv r_1 r_2 + r_1 r_3$ (distributive law)
We want to prove that
\[
\emptyset r \equiv \emptyset. \quad (55)
\]
According to Equation 53, to prove Equation 55, we have to show that
\[
L(\emptyset r) = L(\emptyset). \quad (56)
\]
One way to do so is:
\[
\begin{align*}
L(\emptyset r) & \overset{\text{Eq.}42}{=} L(\emptyset) \cdot L(r) \\
& \overset{\text{Eq.}38}{=} \emptyset \cdot L(r) \\
& \overset{\text{Eq.}21}{=} \{ w_1 w_2 | w_1 \in \emptyset, w_2 \in L(r) \} \\
& = \emptyset \\
& \overset{\text{Eq.}38}{=} L(\emptyset)
\end{align*}
\]
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. \(\varepsilon \not\in L(s)\) and \(r \doteq rs + t \rightarrow r \doteq ts^*\)  
   (proof by Arto Salomaa)
16. \(a^* a \doteq aa^*\) (see Lemma: Kleene Star below)
17. \(\varepsilon \not\in L(s)\) and \(r \doteq sr + t \rightarrow r \doteq s^* t\) (Arden’s Lemma)
a) Simplify the following regular expression:

\[ r = 0(\varepsilon + 0 + 1)^* + (\varepsilon + 1)(1 + 0)^* + \varepsilon. \]  

(58)

b) Prove the equivalence using only algebraic operations

\[ r^* \equiv \varepsilon + r^*. \]  

(59)

c) Prove the equivalence using only algebraic operations

\[ 10(10)^* \equiv 1(01)^*0. \]  

(60)
Group exercise: being Arto Salomaa

- Prove: $\varepsilon \not\in L(s)$ and $r \doteqdot rs + t \implies r \doteqdot ts^*$
- Group phase (groups of 3-4, 5-10 minutes)
- Discussion
- Group phase (5-10 minutes)
- Proof assembly
Finite Automata/Finite State Machines
Finite Automata: Motivation

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

- Automaton is in one of a finite number of states
- Words processed letter by letter
- State transitions triggered by letters read
- Words are accepted or rejected based on final state reached
DFA: Example

- A simple DFA recognizing the regular expression $a^*ba^*$

![DFA Diagram]

- This DFA has two states, 0 and 1.
- 0 is the initial state (with an arrow “pointing at it from anywhere” (Sipser, 2006))
- 1 is an accepting state (represented as a double circle)
A deterministic finite automaton (DFA) is a quintuple

\[ A = \langle Q, \Sigma, \delta, q_0, F \rangle \] (61)

with the following components

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

\[ A = \langle Q, \Sigma, \delta, q_0, F \rangle \]  \hspace{1cm} (62)

with

1. \( Q = \{0, 1\} \)
2. \( \Sigma = \{a, b\} \)
3. \( \delta(0, a) = 0; \delta(0, b) = 1; \)
   \hspace{1cm} \delta(1, a) = 1; \delta(1, b) = \Omega \)
4. \( q_0 = 0 \)
5. \( F = \{1\} \)
Language accepted by an DFA

- In order to formally define the language accepted by an DFA, we generalize the state transition function $\delta$ to a function

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

(63)

whose second argument is a string.

- We define

$$\delta'(q, \varepsilon) = q$$

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

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

- The **language accepted by a DFA** $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ is defined as

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

(64)
1. We are given this graphical representation of a DFA $A$:

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

b) Give a formal definition of $A$. 
2. Give
   - a regular expression,
   - a graphical representation, and
   - a formal definition
   of a DFA $A$ whose language $L(A) \subset \{a, b\}^*$ contains all those words featuring the substring $ab$
     a) at the beginning,
     b) at arbitrary position,
     c) at the end.
Which language is recognized by the DFA?
\( A = \langle Q, \Sigma, \delta, q_0, F \rangle \)

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

\[ \begin{array}{c|cccc}
\delta & 0 & 0 & 1 & 1 \\
\hline
\rightarrow q_0 & q_0 & q_1 & q_4 \\
q_1 & q_1 & q_2 & q_4 \\
q_2 & q_2 & q_4 & q_3 \\
q_3 & q_3 & q_3 & q_3 \\
q_4 & q_4 & q_4 & q_4 \\
\end{array} \]
### DFA: Tabular representation in practice

<table>
<thead>
<tr>
<th>Delta</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ q0</td>
<td>q1</td>
<td>q4</td>
</tr>
<tr>
<td>q1</td>
<td>q2</td>
<td>q4</td>
</tr>
<tr>
<td>q2</td>
<td>q4</td>
<td>q3</td>
</tr>
<tr>
<td>* q3</td>
<td>q3</td>
<td>q3</td>
</tr>
<tr>
<td>q4</td>
<td>q4</td>
<td>q4</td>
</tr>
</tbody>
</table>

```
> 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
q4 :: 0 -> q4
q4 :: 1 -> q4
Rejected
```
DFAs in tabular form: exercise

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

<table>
<thead>
<tr>
<th>parity</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>even</td>
<td>even</td>
<td>odd</td>
</tr>
<tr>
<td>odd</td>
<td>odd</td>
<td>even</td>
</tr>
</tbody>
</table>

- Characterize the language accepted by the DFA
Homework assignment

- Write (in a language of your choice) a program that reads a finite automaton in tabular form from a file (name given on the command line) and simulates its processing of a word (also given on the command line). Example files for input and output are given on the course web site, http://www.lehre.dhbw-stuttgart.de/~sschulz/fla2014.html

- Deadline: Next session (2014-10-01!)
Review of Goals

- Review and mental warm-up
- Proofs
- Regular expression algebra
- Deterministic finite automata
Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?
Goals for Today

- Refresh Deterministic Finite Automata
- Discuss homework and open points
- Non-deterministic FAs
Regular expression algebra

Definition: \( r_1 \vdash r_2 \) iff \( L(r_1) = L(r_2) \)

1. \( r_1 + r_2 \vdash r_2 + r_1 \)
2. \( (r_1 + r_2) + r_3 \vdash r_1 + (r_2 + r_3) \)
3. \( (r_1 r_2)r_3 \vdash r_1 (r_2 r_3) \)
4. \( \emptyset r \vdash \emptyset \)
5. \( \varepsilon r \vdash r \)
6. \( \emptyset + r \vdash r \)
7. \( (r_1 + r_2)r_3 \vdash r_1 r_3 + r_2 r_3 \)
8. \( r_1 (r_2 + r_3) \vdash r_1 r_2 + r_1 r_3 \)
9. \( r + r \vdash r \)
10. \( (r^*)^* \vdash r^* \)
11. \( \emptyset^* \vdash \varepsilon \)
12. \( \varepsilon^* \vdash \varepsilon \)
13. \( r^* \vdash \varepsilon + r^* r \)
14. \( r^* \vdash (\varepsilon + r)^* \)
15. \( \varepsilon \not\in L(s) \) and \( r \vdash rs + t \) \( \rightarrow \)
\[ r \vdash ts^* \]
Open exercise

Simplify the following regular expression:

\[ r = 0(\varepsilon + 0 + 1)^* + (\varepsilon + 1)(1 + 0)^* + \varepsilon \]
Lemma: Kleene Star

Lemma:

\[ a^* a = aa^* \]  \hspace{1cm} (65)

Proof: By case distinction.

Case 1: \( \varepsilon \in L(a) \). We show \( L(a^* a) = L(a^*) = L(aa^*) \)

a. \( L(a^* a) \subseteq L(a^*) \) by definition

b. \( L(a^* a) = \{ uv \mid u \in L(a^*), v \in L(a) \} \)

\[ \supseteq \{ uv \mid u \in L(a^*), v = \varepsilon \} \]

\[ = \{ u \mid u \in L(a^*) \} \]

\[ = L(a^*) \]

\[ \Longleftarrow \text{a. und b. imply } L(a^* a) = L(a^*) \]

\[ \Longleftarrow L(aa^*) = L(a^*) \text{: Strictly analogous} \]

\[ \Longleftarrow \text{Hence case 1 holds.} \]
Lemma: Kleene Star

Case 2: \( \varepsilon \notin L(a) \). Then

\[
a^*a \doteq (\varepsilon + a^*a)a \quad (\text{by 13. } a^* \doteq \varepsilon + a^*a)
\]
\[
\doteq (a^*a + \varepsilon)a \quad (\text{by 1. } r_1 + r_2 \doteq r_2 + r_1)
\]
\[
\doteq a^*aa + a \quad (\text{by 7. } (r_1 + r_2)r_3 \doteq r_1r_3 + r_2r_3)
\]
\[
\doteq aa^* \quad (\text{by 15. with } r = a^*a, s = a, t = a)
\]

Since cases 1 and 2 hold, the lemma holds. q.e.d.
Solution to open exercise

\[ r = 0(\varepsilon + 0 + 1)^* + (\varepsilon + 1)(1 + 0)^* + \varepsilon \]

14,1
\[ \equiv 0(0 + 1)^* + (\varepsilon + 1)(0 + 1)^* + \varepsilon \]

7
\[ \equiv 0(0 + 1)^* + \varepsilon(0 + 1)^* + 1(0 + 1)^* + \varepsilon \]

5
\[ \equiv 0(0 + 1)^* + (0 + 1)^* + 1(0 + 1)^* + \varepsilon \]

1,7
\[ \equiv \varepsilon + (0 + 1)(0 + 1)^* + (0 + 1)^* \]

Eq.65
\[ \equiv \varepsilon + (0 + 1)^*(0 + 1) + (0 + 1)^* \]

13
\[ \equiv (0 + 1)^* + (0 + 1)^* \]

9
\[ \equiv (0 + 1)^*. \]
DFAs

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

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

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

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

- Group 1 (your family name starts with A-M):
  Find a DFA $A$ with $L(A) = L_1$

- Group 2 (your family name does not start with A-M):
  Find a DFA $A$ with $L(A) = L_2$
Homework Assignment

> Write (in a language of your choice) a program that reads a finite automaton in tabular form from a file (name given on the command line) and simulates its processing of a word (also given on the command line).

```
A3 | 0 1
---|----
...|...

> ./easim.py ea03.txt 100010
Processing: 100010
q0 :: 1 -> q2
q2 :: 0 -> q3
q3 :: 0 -> q2
q2 :: 0 -> q3
q3 :: 1 -> q1
q1 :: 0 -> q0
Accepted
```
Student Experiences?
Language of (my) choice: Python

- Modern scripting language, widely used
- Good collection of built-in abstract data types
  - Lists/arrays/stacks/queues
  - Dictionaries/hashe
  - Sets
- Object-oriented features
  - Classes, objects
  - Inheritance
- Functional features
  - Functions as first-class members
  - map and lambda
- Good library support
  - Strings and regexps
  - “All of UNIX/POSIX”
Python peculiarities

- Statement blocks marked by indentation

```python
for c in string:
    newstate = self.delta_fun(state, c)
    print state, "::", c, "->", newstate
    state = newstate
```

- Methods use explicit `self` parameter

```python
def delta_fun(self, state, letter):
    return self.delta[(state, letter)]
```

```
...  
newstate = self.delta_fun(state, c)
```
DFAs in Python

- DFA is a class
  - Individual DFAs are objects/instances
  - Class constructor extracts DFA from string table
- Direkt mapping of \( A = \langle Q, \Sigma, \delta, q_0, F \rangle \)
  - \( Q \): Python set states
  - \( \Sigma \): Python list sigma
  - \( \delta \): Python dictionary delta mapping \((q, c)\) onto successor state
    - Also exposed as a method \( \text{delta_fun} \)
  - \( q_0 \): Python variable start
  - \( F \): Python set accept
DFAs in Python – structure

class DFA(object):
    """
    Object representing a (deterministic) finite automaton
    """

def __init__(self, spec):
    ...

def delta_fun(self, state, letter):
    ...

def proc_string(self, string):
    ...

def parse(self, spec):
    ...

def __str__(self):
    ...

def dotify(self):
    ...
```python
def __init__(self, spec):
    self.states = set()
    self.sigma = []
    self.delta = {}
    self.start = None
    self.accept = set()
    self.parse(spec)
```
DFAs in Python – processing

```python
def delta_fun(self, state, letter):
    return self.delta[(state, letter)]

def proc_string(self, string):
    print "Processing:", string

    state = self.start
    for c in string:
        newstate = self.delta_fun(state, c)
        print state, "::", c, "->", newstate
        state=newstate

    if state in self.accept:
        print "Accepted\n"
    else:
        print "Rejected\n"
```

if __name__ == '__main__':
    opts, args = getopt.gnu_getopt(sys.argv[1:], "hdp", ["help", "dot", "print"])
    # Options and error-handling omitted
    file = open(args[0], "r")
    str = file.read()
    file.close()

    ea = DFA(str)

    ....

    for arg in args[1:]:
        ea.proc_string(arg)
def parse(self, spec):
    lines = spec.split("\n")
    # Find sigma
    while True:
        i = lines.pop(0)
        l = i.strip()
        if l:
            sigmastr = l.split("|") [1]
            self.sigma = sigmastr.split()
            break
    # Skip
    while True:
        i = lines.pop(0)
        l = i.strip()
        if l:
            break
    ...
    ...

while lines:
    i = lines.pop(0)
    l = i.strip()
    if l:
        state, values = l.split("|")
        state = state.strip()
        start = False
        accept = False
        if state.startswith("->"):
            start = True
            state = state[2:].strip()
        if state.startswith("*"):
            accept = True
            state = state[1:].strip()
... 
self.states.add(state)
if start:
    self.start = state
if accept:
    self.accept.add(state)
dvals = values.split()
for i in xrange(len(self.sigma)):
    self.delta[[(state, self.sigma[i])]] = dvals[i]
Non-determinism
So far, we have discussed deterministic FAs, i.e. every state has exactly one transition for every possible input.

Often, DFAs can be rather complex as in the following example accepting a language specified by the regular expression

\[(a + b)^*b(a + b)(a + b)\]  (66)
Non-Deterministic FAs – motivation (2)
We can simplify such an FA if we permit that an input can lead to
one transition, multiple transitions, or no transition.

That is, an FA selects its next state from a set of states where
the set depends on the current state and the input.

We call this a non-deterministic finite automaton (NFA)

For the same example with the regular expression

\[(a + b)^* b (a + b)(a + b)\] (67)

...
This FA is non-deterministic, since, in state $q_0$ with the input $b$, the FA has to “guess” the next state.

An example string $abab$ can be read in three ways:

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

An NFA accepts a string, if one of the possible computations leads to an accepting state!
Non-deterministic transitions allow an NFA to go to more than one successor state
  ▶ \( \delta \) is a transition relation, not a transition function

In addition to allow the automaton to go to more than one state on a given symbol, we also allow it to change state without reading a symbol:

\[
q_1 \xrightarrow{\varepsilon} q_2. \tag{68}
\]

This is called the spontaneous transition or \( \varepsilon \)-transition

Thus, \( \delta \) is a relation on \( Q \times (\Sigma \cup \{\varepsilon\}) \times Q \)
▶ Even though NFAs seem to be based on guessing, in the following, we will see that they are exactly as powerful as DFAs. We can generate an equivalent DFA from any NFA!
An NFA is a quintuple

\[ A = \langle Q, \Sigma, \delta, q_0, F \rangle \] (69)

with the following components

1. \( Q \) is the finite set of states.
2. \( \Sigma \) is the input alphabet.
3. \( \delta \) is a relation on \( Q \times (\Sigma \cup \{\varepsilon\}) \times Q \). I.e.,

\[ \delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q \] (70)

4. \( q_0 \in Q \) is the initial state.
5. \( F \subseteq Q \) is the set of final states.
The above mentioned NFA example is expressed as

\[ A = \langle Q, \Sigma, \delta, q_0, F \rangle \]  \hspace{1cm} (71)

with

1. \( Q = \{ q_0, q_1, q_2, q_3 \} \)
2. \( \Sigma = \{ a, b \} \)
3. \( \delta = \{ \langle q_0, a, q_0 \rangle, \langle q_0, b, q_0 \rangle, \langle q_0, b, q_1 \rangle, \langle q_1, a, q_2 \rangle, \langle q_1, b, q_2 \rangle, \langle q_2, a, q_3 \rangle, \langle q_2, b, q_3 \rangle \} \)
4. Initial state \( q_0 \)
5. \( F = \{ q_3 \} \)
Develop an NFA $A$ whose language $L(A) \subseteq \{a,b\}^*$ contains all those words featuring the substring $aba$. Give:

- a regular expression representing $L(A)$,
- a graphical representation of $A$,
- a formal definition of $A$
Equivalence of DFA and NFA

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

\[
L(A) = L(\text{det}(A))
\]  

(72)

The core idea is that the states of \( \text{det}(A) \) are sets of states of \( A \).

Deterministic transitions in \( \text{det}(A) \) simulate transitions in \( A \).

A set \( M \) of states of \( A \) is a final state of \( \text{det}(A) \) if \( M \) contains a final state of \( A \).

To show this, we define three auxiliary functions.

- \( \varepsilon \) closure
- Successor states function \( \delta^* \)
- Extended transition function \( \Delta^* \) for NFAs
NFA: $\varepsilon$ closure

- The $\varepsilon$ closure

\[
ec : Q \rightarrow 2^Q
\]  

(73)

returns the set of all states the NFA can change to by means of an $\varepsilon$ transition coming from state $q$.

- Formal definition: $ec$ is the smallest function with the properties:

\[
q \in ec(q);
\]

(74)

\[
p \in ec(q) \land \langle p, \varepsilon, r \rangle \in \delta \rightarrow r \in ec(q).
\]

(75)
NFA: \( \varepsilon \) closure example (1)
NFA: $\varepsilon$ closure example (2)

- calculating the $\varepsilon$ closure for all states:
  - $ec(q_0) = \{q_0, q_1, q_2\}$,
  - $ec(q_1) = \{q_1\}$,
  - $ec(q_2) = \{q_2\}$,
  - $ec(q_3) = \{q_3\}$,
  - $ec(q_4) = \{q_4\}$,
  - $ec(q_5) = \{q_5, q_7, q_0, q_1, q_2\}$,
  - $ec(q_6) = \{q_6, q_7, q_0, q_1, q_2\}$,
  - $ec(q_7) = \{q_7, q_0, q_1, q_2\}$. 
Second, we transform the relation $\delta$ into a function

$$\delta^* : Q \times \Sigma \rightarrow 2^Q.$$ (76)

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

Formally, we have

$$\delta^*(q_1, c) = \bigcup_{q_2 \in Q: \langle q_1, c, q_2 \rangle \in \delta} ec(q_2).$$ (77)
Successor state function for NFAs (2)

\[ \delta^*(q_1, c) = \bigcup_{q_2 \in Q: \langle q_1, c, q_2 \rangle \in \delta} ec(q_2) \]

- examples (based on the above NFA):
  1. \( \delta^*(q_0, a) = \{\} \),
  2. \( \delta^*(q_1, b) = \{q_3\} \),
  3. \( \delta^*(q_3, a) = \{q_5, q_7, q_0, q_1, q_2\} \).
Extended transition function $\Delta^*$

- Third, we transform the function $\delta^*$ into a function

$$\Delta^* : 2^Q \times \Sigma \rightarrow 2^Q.$$  \hspace{1cm} (78)

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

- Formally, we have

$$\Delta^*(M, c) = \bigcup_{q \in M} \delta^*(q, c).$$ \hspace{1cm} (79)

- examples (based on the above NFA):

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


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

$$\text{det}(A) = \langle 2^Q, \Sigma, \Delta^*, ec(q_0), \hat{F} \rangle$$  \hspace{1cm} (80)$$

with

$$\hat{F} = \{ M \in 2^Q | M \cap F \neq \{ \} \}.$$  \hspace{1cm} (81)$$

That is, the set of final states $\hat{F}$ is the set of all subsets of $Q$ containing a final state.
returning to the example FSM expressing the regular expression

\[(a + b)^*b(a + b)(a + b)\]  \hfill (82)

The initial state:

\[S_0 = ec(q_0) = \{q_0\}.\]  \hfill (83)

The state transition function: Starting with the initial state...

\[\Delta^*(\{q_0\}, a) = \{q_0\} = S_0.\]
Exploring the set of states...

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

\[ \Delta^*([q_0, q_3], a) = \{q_0\} = S_0. \]
\[ \Delta^*([q_0, q_3], b) = \{q_0, q_1\} = S_1. \]
\[ \Delta^*([q_0, q_1, q_3], a) = \{q_0, q_2\} = S_2. \]
\[ \Delta^*([q_0, q_1, q_3], b) = \{q_0, q_1, q_2\} = S_4. \]
\[ \Delta^*([q_0, q_2, q_3], a) = \{q_0, q_3\} = S_3. \]
\[ \Delta^*([q_0, q_2, q_3], b) = \{q_0, q_1, q_3\} = S_5. \]
\[ \Delta^*([q_0, q_1, q_2, q_3], a) = \{q_0, q_2, q_3\} = S_6. \]
\[ \Delta^*([q_0, q_1, q_2, q_3], b) = \{q_0, q_1, q_2, q_3\} = S_7. \]
Now, we can define the DFA

$$\operatorname{det}(A) = \langle \hat{Q}, \Sigma, \Delta^*, S_0, \hat{F} \rangle$$ (84)

with

- the set of states
  \[ \hat{Q} = \{ S_0, \ldots, S_7 \} \] (85)
- the state transition function $\Delta^*$ as summarized as follows:

<table>
<thead>
<tr>
<th>$\Delta^*$</th>
<th>$S_0$</th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
<th>$S_5$</th>
<th>$S_6$</th>
<th>$S_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$S_0$</td>
<td>$S_2$</td>
<td>$S_3$</td>
<td>$S_0$</td>
<td>$S_6$</td>
<td>$S_2$</td>
<td>$S_3$</td>
<td>$S_6$</td>
</tr>
<tr>
<td>b</td>
<td>$S_1$</td>
<td>$S_4$</td>
<td>$S_5$</td>
<td>$S_1$</td>
<td>$S_7$</td>
<td>$S_4$</td>
<td>$S_5$</td>
<td>$S_7$</td>
</tr>
</tbody>
</table>

- and the set of final states (each DFA state containing the NFA final state $q_3$)
  \[ \hat{F} = \{ S_3, S_5, S_6, S_7 \} \] (86)
Equivalence of DFA and NFA: example (5)
We are given the following NFA $A$:

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

b) Draw $\text{det}(A)$’s graphical representation.

c) Give a regular expression representing the same language as $A$. 
Solution to exercise (1)

- Incremental computation of $\hat{Q}$ and $\Delta^*$:
  - Initial state $S_0 = ec(q_0) = \{q_0, q_1, q_2\}$
  - $\Delta^*(S_0, a) = \delta^*(q_0, a) \cup \delta^*(q_1, a) \cup \delta^*(q_2, a) = \{\} \cup \{\} \cup \{q_4\} = \{q_4\} = S_1$
  - $\Delta^*(S_0, b) = \{q_3\} = S_2$
  - $\Delta^*(S_1, a) = \{\} = S_3$
  - $\Delta^*(S_1, b) = ec(q_6) = \{q_6, q_7, q_0, q_1, q_2\} = S_4$
  - $\Delta^*(S_2, a) = \{q_5, q_7, q_0, q_1, q_2\} = S_5$
  - $\Delta^*(S_2, b) = \{\} = S_3$
  - $\Delta^*(S_3, a) = \{\} = S_3$
  - $\Delta^*(S_3, b) = \{\} = S_3$
  - $\Delta^*(S_4, a) = \{q_4\} = S_1$
  - $\Delta^*(S_4, b) = \{q_3\} = S_2$
  - $\Delta^*(S_5, a) = \{q_4\} = S_1$
  - $\Delta^*(S_5, b) = \{q_3\} = S_2$

- $\hat{F} = \{S_4, S_5\}$ (since $q_7 \in S_4$, $q_7 \in S_5$)
Solution to exercise (2)

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

\[
\hat{Q} = \{ S_0, S_1, S_2, S_3, S_4, S_5 \}
\]

\[
\hat{F} = \langle S_4, S_5 \rangle
\]

\[\Delta^* \text{ given by the table below}\]

<table>
<thead>
<tr>
<th>$\Delta^*$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>$S_1$</td>
<td>$S_2$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>$S_3$</td>
<td>$S_4$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$S_5$</td>
<td>$S_3$</td>
</tr>
<tr>
<td>$S_3$</td>
<td>$S_3$</td>
<td>$S_3$</td>
</tr>
<tr>
<td>$S_4^*$</td>
<td>$S_1$</td>
<td>$S_2$</td>
</tr>
<tr>
<td>$S_5^*$</td>
<td>$S_1$</td>
<td>$S_2$</td>
</tr>
</tbody>
</table>

**RE:**

\[L(A) = L((ab + ba)(ab + ba)^*)\]
Review of Goals

- Refresh Deterministic Finite Automata
- Discuss homework and open points
- Non-deterministic FAs
Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?
Goals for Today

- Refresh and warm-up
- Completing the circle
  - REs can be simulated by NFAs
  - DFAs can be simulated by REs
- Minimization of DFAs
Refresher: NFAs

- **NFA** \( A = \langle Q, \Sigma, \delta, q_0, F \rangle \)
  1. \( Q \) is the finite set of states.
  2. \( \Sigma \) is the input alphabet.
  3. \( \delta \) is a relation on \( Q \times (\Sigma \cup \{ \varepsilon \}) \times Q \)
  4. \( q_0 \in Q \) is the initial state.
  5. \( F \subseteq Q \) is the set of final states.

- **Significant differences to DFAs:**
  - \( \delta \) is a relation - the automaton can change to multiple successor states
  - \( \delta \) allows for \( \varepsilon \)-transition - it can change states spontaneously

- **NFAs can be simulared by DFAs**
  - States of \( \text{det}(A) \) are sets of states of \( A \)
  - \( \Delta^* \) goes from sets of \( A \)-states to sets of \( A \)
    - ...by combining the transistion of the individual states
    - ...and taking the \( \varepsilon \)-closure
Warmup: NFA to DFA transformation

- Convert the following NFA (over $\Sigma = \{a, b\}$) into an equivalent DFA:
Regular expressions and NFAs
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 $A$ with $L(A) = L$
  - For every finite automaton $A$, $L(A)$ is a regular language

- We will now (constructively) show this equivalence between REs and FAs
  - We already know that DFAs and NFAs are equivalent
  - Now: Equivalence of NFAs and REs
Given a regular expression \( r \), we want to derive an NFA \( A(r) \) accepting the same language:

\[
L(A(r)) = L(r). \tag{87}
\]

Ideas:
- We construct NFAs for the elementary REs \( (\emptyset, \varepsilon, c \in \Sigma) \)
- We combine NFAs for subexpressions to generate NFAs for composite regular expressions

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!
Let $\Sigma$ be an alphabet.

- The elementary regular expressions over $\Sigma$ are:
  - $\emptyset$ with $L(\emptyset) = \emptyset$
  - $\varepsilon$ with $L(\varepsilon) = \{\varepsilon\}$
  - $c \in \Sigma$ with $L(c) = \{c\}$

- Assume $r_1$ and $r_2$ are 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 $L(r_1^*) = L(r_1)^*$
Assuming $\Sigma$ is the alphabet which $r$ is based on, we define

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

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

3. $A(c) = \langle \{q_0, q_1\}, \Sigma, \{\langle q_0, c, q_1 \rangle\}, q_0, \{q_1\} \rangle$ for all $c \in \Sigma$
NFAs for composite REs (general)

- In the following we assume:
  - $A(r_1) = \langle Q_1, \Sigma, \delta_1, q_1, \{q_2\} \rangle$
  - $A(r_2) = \langle Q_2, \Sigma, \delta_2, q_3, \{q_4\} \rangle$
  - $Q_1 \cap Q_2 = \emptyset$
  - $q_0, q_5 \notin Q_1 \cup Q_2$

- We visualise an NFA for RE $r_1$ by a square box with two explicit states
  - The initial state is on the left
  - The unique accepting state on the right
  - All other states and transitions are implicit
  - We mark initial/accepting states only for the composite automaton

![NFA Diagram](image_url)
4. $A(r_1 r_2) = \langle Q_1 \cup Q_2, \Sigma, \{ \langle q_2, \varepsilon, q_3 \rangle \} \cup \delta_1 \cup \delta_2, q_1, \{ q_4 \} \rangle$

Reminder:

- $A(r_1) = \langle Q_1, \Sigma, \delta_1, q_1, \{ q_2 \} \rangle$
- $A(r_2) = \langle Q_2, \Sigma, \delta_2, q_3, \{ q_4 \} \rangle$
NFAs for composite REs (alternatives)

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

Reminder:
- \( A(r_1) = \langle Q_1, \Sigma, \delta_1, q_1, \{q_2\}\rangle \)
- \( A(r_2) = \langle Q_2, \Sigma, \delta_2, q_3, \{q_4\}\rangle \)
NFAs for composite REs (Kleene Star)

6. $A(r_1^*) = \langle \{q_0, q_5\} \cup Q_1, \Sigma, \{\langle q_0, \epsilon, q_1 \rangle, \langle q_2, \epsilon, q_1 \rangle, \langle q_0, \epsilon, q_5 \rangle, \langle q_2, \epsilon, q_5 \rangle \} \cup \delta_1, q_0, \{q_5\} \rangle$

Reminder:

$A(r_1) = \langle Q_1, \Sigma, \delta_1, q_1, \{q_2\} \rangle$
Fact: NFAs can simulate REs

▶ The previous construction produces for each regular expression $r$ an NFA $A$ with $L(A) = L(r)$

Corollary: Every language described by a regular expression can be accepted by a non-deterministic finite automaton
Determine an NFA accepting the same language as the regular expression

\[(a + b)a^*b\]  \(88\)
DFAs and Regular expressions
Overview and orientation

- We have claimed that NFAs, DFAs and REs all describe the same class of regular languages
- We have learned how to convert
  - regular expressions to equivalent NFAs
  - NFAs to equivalent DFAs
  - (DFAs to equivalent NFAs)

**Todo: convert DFA to equivalent RE**

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

\[ L(r(A)) = L(A) \quad (89) \]
Convert DFA into RE

- Given: DFA \( A = \langle Q, \Sigma, \delta, q_0, F \rangle \)
- Goal: RE \( r(A) \) with \( L(r(A)) = L(A) \)
- Idea: For each state \( q \), generate an equation describing the language \( L(q) \) that is accepted from that state, depending on the languages accepted at neighboring states
  - For each transition with \( c \) to \( q' \): \( c \cdot L(q') \)
  - Accepting states: \( \varepsilon \)
- Solve the resulting system for \( L(q_0) \)
  - Result: RE describing \( L(q_0) = L(A) \)
- Convention:
  - States are named \( \{0, 1, \ldots, n\} \)
  - Start state is 0
  - We write \( L_k \) instead of \( L(k) \) to describe the language accepted at state \( k \)
Convert DFA to RE: Example

\[ \begin{align*}
L_0 &= aL_1 + bL_2 \\
L_1 &= aL_1 + bL_2 \\
L_2 &= bL_0 + \varepsilon
\end{align*} \]

3 equations, 3 unknowns

What now?
Lemma:

\[ \varepsilon \notin L(s) \text{ and } r \doteq sr + t \rightarrow r \doteq s^*t \quad (90) \]

Compare Arto Salomaa:

\[ \varepsilon \notin L(s) \text{ and } r \doteq rs + t \rightarrow r \doteq ts^* \]
Convert DFA to RE: Example

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

$L_1 \quad \doteq \quad aL_1 + b(bL_0 + \varepsilon)$
\[= \quad a^* b(bL_0 + \varepsilon) \quad [\text{Arden}]\]

$L_0 \quad \doteq \quad a(a^* b(bL_0 + \varepsilon)) + b(bL_0 + \varepsilon)$
\[= \quad aa^* bbL_0 + aa^* b + bbL_0 + b \quad [\text{Dist.}]\]
\[= \quad (aa^* bb + bb)L_0 + aa^* b + b \quad [\text{Comm.,Dist.}]\]
\[= \quad (aa^* bb + bb)^*(aa^* b + b) \quad [\text{Arden}]\]
\[= \quad ((aa^* + \varepsilon)bb)^*((aa^* + \varepsilon)b) \quad [\text{Dist.}]\]
\[= \quad (a^* bb)^*(a^* b) \quad [rr^* + \varepsilon = r^*] \quad (91)\]
Convert DFA to RE: Example (continued)

\[ L_0 = \ldots \]
\[ = (a^*bb)^*(a^*b) \]

\[ \text{Ergo: } L(A) = L((a^*bb)^*(a^*b)) \]
We have learned how to convert
- regular expressions 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
Minimization of DFAs

Given the DFA

\[ A = \langle Q, \Sigma, \delta, q_0, F \rangle \], \quad (92)

we want to derive a DFA

\[ A^- = \langle Q^-, \Sigma, \delta^-, q_0, F^- \rangle \], \quad (93)

accepting the same language, i.e.,

\[ L(A) = L(A^-) \] \quad (94)

for which the number of states (elements of \( Q^- \)) is minimal.
Minimization of DFAs: example/exercise

How small can we make it?
Minimization of DFAs

Assume the DFA \( A = \langle Q, \Sigma, \delta, q_0, F \rangle \)

- The idea is to identify the set \( V \) comprising all the 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

- Formally: \( V \) is the smallest set of tuples with
  - \( \{ \langle p, q \rangle | p \in F, q \notin F \} \subset V \)
  - \( \{ \langle p, q \rangle | p \notin F, q \in F \} \subset V \)
  - \( \delta(p, c) = p', \delta(q, c) = q', \langle p', q' \rangle \in V \) for some \( c \in \Sigma \rightarrow \langle p, q \rangle \in V \)
Minimization of DFAs: the algorithm

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

$$V = \{ \langle p, q \rangle \in Q \times Q | (p \in F \land q \notin F) \lor (p \notin F \land q \in F) \}.$$  \hspace{1cm} (95)

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

$$\text{while} \left( \exists \langle p, q \rangle \in Q \times Q \forall c \in \Sigma | \langle \delta(p, c), \delta(q, c) \rangle \in V \land \langle p, q \rangle \notin V \right)$$

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

$$\}$$
Minimization of DFAs: Merging States

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

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

- 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
Minimization of DFAs: example

We want to minimize this DFA with 5 states:
Minimization of DFAs: example (cont.)

This is the formal definition of the DFA:

\[ A = \langle Q, \Sigma, \delta, q_0, F \rangle \] (98)

with

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

- For the sake of practicality, we represent the set \( V \) by means of a two-dimensional table with the elements of \( Q \) as columns and rows and \( V \)'s elements as cells featuring the symbol \( \times \).
- Analogously, we represent state pairs that are definitely not members of \( V \) using the symbol \( \circ \).
1. By determining all combinations of states in $F = \{q_3, q_4\}$ and $Q \setminus F = \{q_0, q_1, q_2\}$, we get the following initial state of $V$:

<table>
<thead>
<tr>
<th></th>
<th>$q_0$</th>
<th>$q_1$</th>
<th>$q_2$</th>
<th>$q_3$</th>
<th>$q_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td></td>
<td></td>
<td></td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td></td>
<td>$\times$</td>
<td></td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td></td>
<td></td>
<td>$\times$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_4$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td></td>
</tr>
</tbody>
</table>
2. Furthermore, the cases $\langle q_i, q_i \rangle | i \in \{0, \cdots, 4\}$ are naturally indistinguishable since they are identical:

<table>
<thead>
<tr>
<th></th>
<th>$q_0$</th>
<th>$q_1$</th>
<th>$q_2$</th>
<th>$q_3$</th>
<th>$q_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>〇</td>
<td></td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>$q_1$</td>
<td>〇</td>
<td></td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>$q_2$</td>
<td></td>
<td>〇</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>$q_3$</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>〇</td>
</tr>
<tr>
<td>$q_4$</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>〇</td>
</tr>
</tbody>
</table>
Minimization of DFAs: example (cont.)

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

\[
\begin{align*}
\delta(q_0, a) &= q_1; \delta(q_1, a) = q_3; \langle q_1, q_3 \rangle \in V \rightarrow \langle q_0, q_1 \rangle, \langle q_1, q_0 \rangle \in V \\
\delta(q_0, a) &= q_1; \delta(q_2, a) = q_4; \langle q_1, q_4 \rangle \in V \rightarrow \langle q_0, q_2 \rangle, \langle q_2, q_0 \rangle \in V \\
\delta(q_1, a) &= q_3; \delta(q_2, a) = q_4; \langle q_3, q_4 \rangle \notin V \text{ (as of yet)} \\
\delta(q_1, b) &= q_3; \delta(q_2, b) = q_4; \langle q_3, q_4 \rangle \notin V \text{ (as of yet)} \\
\delta(q_3, a) &= q_1; \delta(q_4, a) = q_2; \langle q_1, q_2 \rangle \notin V \text{ (as of yet)} \\
\delta(q_3, b) &= q_1; \delta(q_4, b) = q_2; \langle q_1, q_2 \rangle \notin V \text{ (as of yet)}
\end{align*}
\]
Since no other distinguishable state pairs could be found, we fill empty cells with \( \circ \):

\[
\begin{array}{c|ccccc}
& q_0 & q_1 & q_2 & q_3 & q_4 \\
\hline
q_0 & \circ & \times & \times & \times & \times \\
q_1 & \times & \circ & \circ & \times & \times \\
q_2 & \times & \circ & \circ & \times & \times \\
q_3 & \times & \times & \times & \circ & \circ \\
q_4 & \times & \times & \times & \circ & \circ \\
\end{array}
\]

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

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

b) \( \langle q_3, q_4 \rangle \).
Minimization of DFAs: example (cont.)

- This is the minimized DFA after merging indistinguishable states:
The algorithm does not handle missing transitions/Ω-transitions.

- A rejection due to a missing transition is indistinguishable from a rejection due to reaching a junk state.

Solution: If the automaton has a missing transition, add an explicit junk state and complete the transition function.
Minimization of DFAs: exercise

Derive a minimal DFA accepting the language

\[ L(aba^*). \]  \hspace{1cm} (99)

Solve the exercise in three steps:
1. Derive an NFA accepting \( L \).
2. Transform the NFA into a DFA.
3. Minimize the DFA.
Consider $\Sigma = \{a, b\}$ and $L = \{aba, bab\} \cup \{wbb|w \in \Sigma^*\}$

- Find an RE for this language
- Convert the RE into an NFA
- Convert the NFA to a DFA
- Minimize the DFA
- Convert the minimal DFA back into an RE

- Give a graphical representation of the 3 automata (NFA, DFA, minimized DFA)
Review of Goals

- Completing the circle
  - REs can be simulated by NFAs
  - DFAs can be simulated by REs
- Minimization of DFAs
Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?
Goals for Today

- Refresher & Homework
- Equivalence of regular expressions
- Properties of regular languages
  - Closure properties
  - Decision problems
- Non-regular languages and the pumping lemma
- Simulation of REs via NFAs: Composition of NFAs
- Simulation of DFAs via REs: Solve system of equations
  - May need Arden’s Lemma to handle loops!
- Important: NFAs, DFAs, REs are all equivalent!
- Minimization of DFAs:
  - Compute necessarily distinct states
  - Merge indistinguishable states
Consider $\Sigma = \{a, b\}$ and $L = \{aba, bab\} \cup \{wbb \mid w \in \Sigma^*\}$

- Find an RE for this language
- Convert the RE into an NFA
- Convert the NFA to a DFA
- Minimize the DFA
- Convert the minimal DFA back into an RE

- Give a graphical representation of the 3 automata (NFA, DFA, minimized DFA)

I’ve underestimated the effort!
Consider $\Sigma = \{a, b\}$ and $L = \{aba, bab\} \cup \{wbb | w \in \Sigma^*\}$

An RE $R$ with $L(R) = L$ is:

$$R = (aba + bab) + (a + b)^* bb$$
Homework: NFA

- Straightforward construction
- 28 states (ouch!)
### How Long Can You Work on Making a Routine Task More Efficient Before You’re Spending More Time Than You Save? (Across Five Years)

<table>
<thead>
<tr>
<th>Time You Shave Off</th>
<th>50/day</th>
<th>5/day</th>
<th>Daily</th>
<th>Weekly</th>
<th>Monthly</th>
<th>Yearly</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 second</td>
<td>1 day</td>
<td>2 hours</td>
<td>30 minutes</td>
<td>4 minutes</td>
<td>1 minute</td>
<td>5 seconds</td>
</tr>
<tr>
<td>5 seconds</td>
<td>5 days</td>
<td>12 hours</td>
<td>2 hours</td>
<td>21 minutes</td>
<td>5 minutes</td>
<td>25 seconds</td>
</tr>
<tr>
<td>30 seconds</td>
<td>4 weeks</td>
<td>3 days</td>
<td>12 hours</td>
<td>2 hours</td>
<td>30 minutes</td>
<td>2 minutes</td>
</tr>
<tr>
<td>1 minute</td>
<td>8 weeks</td>
<td>6 days</td>
<td>1 day</td>
<td>4 hours</td>
<td>1 hour</td>
<td>5 minutes</td>
</tr>
<tr>
<td>5 minutes</td>
<td>9 months</td>
<td>4 weeks</td>
<td>6 days</td>
<td>21 hours</td>
<td>5 hours</td>
<td>25 minutes</td>
</tr>
<tr>
<td>30 minutes</td>
<td>6 months</td>
<td>5 weeks</td>
<td>5 days</td>
<td>1 day</td>
<td>2 hours</td>
<td></td>
</tr>
<tr>
<td>1 hour</td>
<td>10 months</td>
<td>2 months</td>
<td>10 days</td>
<td>2 days</td>
<td>5 hours</td>
<td></td>
</tr>
<tr>
<td>6 hours</td>
<td>2 months</td>
<td>2 weeks</td>
<td>1 day</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 day</td>
<td>8 weeks</td>
<td>5 days</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Image credit: Randal Munroe, [http://xkcd.com/1205/](http://xkcd.com/1205/)
Homework: DFA (1)

S0 = frozenset(['q0', 'q2', 'q4', 'q6', 's2', 's0', 's7', 's6', 's4', 't2'])
Delta(S0, a) = frozenset(['q1', 'p0', 's2', 's1', 's0', 't2', 's5', 's4', 's7'])
S1 = frozenset(['q1', 'p0', 's2', 's1', 's0', 't2', 's5', 's4', 's7'])
Delta(S0, b) = frozenset(['p2', 'q3', 's3', 's2', 's7', 's0', 't2', 't3', 's5', 'u2', 's4'])
S2 = frozenset(['p2', 'q3', 's3', 's2', 's7', 's0', 't2', 't3', 's5', 'u2', 's4'])
Delta(S1, a) = frozenset(['s2', 's1', 's0', 't2', 's7', 's5', 's4'])
S3 = frozenset(['s2', 's1', 's0', 't2', 's7', 's5', 's4'])
Delta(S1, b) = frozenset(['p1', 'r0', 's3', 's2', 's7', 's0', 't2', 't3', 's5', 'u2', 's4'])
S4 = frozenset(['p1', 'r0', 's3', 's2', 's7', 's0', 't2', 't3', 's5', 'u2', 's4'])
Delta(S2, a) = frozenset(['p3', 'r2', 's2', 's1', 's0', 't2', 's5', 's4', 's7'])
S5 = frozenset(['p3', 'r2', 's2', 's1', 's0', 't2', 's5', 's4', 's7'])
Delta(S2, b) = frozenset(['q7', 's3', 's2', 's0', 's7', 't3', 's5', 'u2', 't2', 'u3', 's4'])
S6 = frozenset(['q7', 's3', 's2', 's0', 's7', 't3', 's5', 'u2', 't2', 'u3', 's4'])
Delta(S3, a) = frozenset(['s2', 's1', 's0', 't2', 's7', 's5', 's4'])
State is equal to S3
Delta(S3, b) = frozenset(['s3', 's2', 's0', 's7', 't3', 's5', 'u2', 't2', 's4'])
S7 = frozenset(['s3', 's2', 's0', 's7', 't3', 's5', 'u2', 't2', 's4'])
Delta(S4, a) = frozenset(['q5', 'r1', 'q7', 's2', 's1', 's0', 't2', 's5', 's4', 's7'])
S8 = frozenset(['q5', 'r1', 'q7', 's2', 's1', 's0', 't2', 's5', 's4', 's7'])
Delta(S4, b) = frozenset(['q7', 's3', 's2', 's0', 's7', 't3', 's5', 'u2', 't2', 'u3', 's4'])
State is equal to S6
Delta(S5, a) = frozenset(['s2', 's1', 's0', 't2', 's7', 's5', 's4'])
State is equal to S3
Delta(S5, b) = frozenset(['q5', 'q7', 'r3', 's3', 's2', 's7', 's0', 't2', 't3', 's5', 'u2', 's4'])
...
... 

S9 = frozenset(['q5', 'q7', 'r3', 's3', 's2', 's7', 's0', 't2', 't3', 's5', 'u2', 's4'])
Delta(S6, a) = frozenset(['s2', 's1', 's0', 't2', 's7', 's5', 's4'])
State is equal to S3
Delta(S6, b) = frozenset(['q7', 's3', 's2', 's0', 's7', 't3', 'u3', 's4', 't2', 's5', 'u2'])
State is equal to S6
Delta(S7, a) = frozenset(['s2', 's1', 's0', 't2', 's7', 's5', 's4'])
State is equal to S3
Delta(S7, b) = frozenset(['q7', 's3', 's2', 's0', 's7', 't3', 'u3', 's4', 't2', 's5', 'u2'])
State is equal to S6
Delta(S8, a) = frozenset(['s2', 's1', 's0', 't2', 's7', 's5', 's4'])
State is equal to S3
Delta(S8, b) = frozenset(['s3', 's2', 's0', 's7', 't3', 's5', 'u2', 't2', 's4'])
State is equal to S7
Delta(S9, a) = frozenset(['s2', 's1', 's0', 't2', 's7', 's5', 's4'])
State is equal to S3
Delta(S9, b) = frozenset(['q7', 's3', 's2', 's0', 's7', 't3', 's5', 'u2', 't2', 'u3', 's4'])
State is equal to S6
Homework: DFA (3)
## Homework: DFA minimisation

<table>
<thead>
<tr>
<th></th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
<th>S8</th>
<th>S9</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>S9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
## Homework: DFA minimisation

<table>
<thead>
<tr>
<th></th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
<th>S8</th>
<th>S9</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>o</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>x</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>S9</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Homework: DFA minimisation

<table>
<thead>
<tr>
<th></th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
<th>S8</th>
<th>S9</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>S1</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>S2</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>S3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>S4</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>S5</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>S6</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>o</td>
</tr>
<tr>
<td>S7</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>o</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>S8</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>o</td>
</tr>
<tr>
<td>S9</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>o</td>
<td>x</td>
<td>x</td>
<td>o</td>
</tr>
</tbody>
</table>

- S6 und S9 sind ununterscheidbar
- S5 und S7 sind ununterscheidbar
Homework: DFA (minimised)
Homework: System of equations

- \( L_0 = aL_1 + bL_2 \)
- \( L_1 = aL_3 + bL_4 \)
- \( L_2 = aL_5 + bL_6 \)
- \( L_3 = aL_3 + bL_5 = a^*bL_5 \)
- \( L_4 = aL_8 + bL_6 \)
- \( L_5 = aL_3 + bL_6 \)
- \( L_6 = aL_3 + bL_6 + \varepsilon = b^*(aL_3 + \varepsilon) \)
- \( L_8 = aL_3 + bL_5 + \varepsilon \)

If someone has solved this, I’d like the solution...
Equivalence of regular expressions
Earlier in this lecture, we have seen that there can be multiple regular expressions describing the same language.

We have also learned that using algebraic transformation rules to prove equivalence of regular expressions can be very difficult or even impossible.

In the following, we will learn a straight-forward algorithm proving equivalence of regular expressions based on FSMs.

The algorithm involves four steps and 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)
1. Given the regular expressions $r_1$ and $r_2$, derive NFAs $A_1$ and $A_2$ accepting their respective languages:

$$L(r_1) = L(A_1) \quad \text{and} \quad L(r_2) = L(A_2). \quad (100)$$

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

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

4. If $r_1 \equiv r_2$, then $M_1$ and $M_2$ must be identical modulo renaming of states.

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

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

\[ 10(10)^* \equiv 1(01)^*0 \]
Properties of regular languages
Regular languages: Closure properties

- Reminders:
  - 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 \( A \) with \( L(A) = L \)
    - There exists a DFA \( A \) with \( L(A) = L \)
    - There exists a regular expression \( R \) with \( L(R) = L \)
    - There exists a regular grammar \( G \) with \( L(G) = L \)

- Fact: Not all languages are regular
  - Proof later today

**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

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

Are regular languages closed under union, intersection, concatenation, complement, and Kleene-star?
Theorem: Let \( L_1 \) and \( L_2 \) be regular languages. Then the following languages are all regular:

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

Proof?

- Idea: We postulate (disjoint) finite automata for \( L_1 \) and \( L_2 \) and 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 $A_{L_1}$ and $A_{L_2}$ be automata for $L_1$ and $L_2$.

$L_1 \cup L_2 :$ new initial state, $\varepsilon$-transitions to the initial states of $A_{L_1}$ and $A_{L_2}$

$L_1 \cdot L_2 :$ $\varepsilon$-transition from the final state(s) of $A_{L_1}$ to the initial state of $A_{L_2}$

$(L_1)^* :$

- new initial and final states,
- $\varepsilon$-transitions from the original final states to the original initial state,
- $\varepsilon$-transition from the new initial to the new final state.
Visual refresher

$L_1 \cup L_2$

$L_1 \circ L_2$

$L_1^*$
Closure under intersection

Let $A_{L_1} = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ and $A_{L_2} = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ be DFAs for $L_1$ and $L_2$.

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

\begin{itemize}
  \item $Q = Q_1 \times Q_2$
  \item $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ for all $q_1 \in Q_1, q_2 \in Q_2, a \in \Sigma$
  \item $q_0 = (q_{01}, q_{02})$
  \item $F = F_1 \times F_2$
\end{itemize}

This so-called **product automaton**

\begin{itemize}
  \item starts in a state that corresponds to the initial states of $A_{L_1}$ and $A_{L_2}$,
  \item simulates simultaneous processing in both automata
  \item accepts if both $A_{L_1}$ and $A_{L_2}$ accept.
\end{itemize}
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

Let $A_L$ be an DFA for the language $L$.
(including a junk state, i.e. there is a transition from every state for every alphabet symbol, no $\Omega$ transitions)

Then $\overline{A_L} = \langle Q, \Sigma, q_0, \delta, Q \setminus F \rangle$ is an automaton accepting $\overline{L}$:

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

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

Reminder:

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

$\delta'(q_0, w)$ is the final state of the automaton after processing $w$
Show that \( L = \{ w \in \{0, 1\}^* \mid |w|_0 = |w|_1 \} \) is not regular.

Hint: Use the following:

- \( 0^n1^n \) is not regular.
- \( 0^*1^* \) is regular.
- (one of) the closure properties shown before.
Finite languages

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 \( A_i \) with initial state \( q_{0i} \) and final state \( q_{fi} \).
- Let \( q_0 \) be a new state, from which there is an \( \varepsilon \)-transition to each \( q_{0i} \).

Then the resulting automaton, with \( q_0 \) as initial state and all \( q_{fi} \) as final states accepts \( L \).
Finite languages: Example

- Assume \( L = \{\text{if, then, else, while, goto, for}\} \) over \( \Sigma_{\text{ascii}} \)
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

The emptiness problem for regular languages is decidable.

Algorithm: Let $A$ be an automaton accepting the language $L$.

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

The word problem for regular languages is decidable.

Algorithm: Let $A$ be an NFA accepting the language $L$ and $w = c_1 c_2 \ldots c_n$.

- Let $Q_1$ be the set of all states of $A$ for which there is a transition from $(q_0, c_1)$
- Let $Q'_1 = ec(Q_1)$
- Let $Q_2$ be the set of all states for which there is a transition $(q, c_2)$ from a state $q \in Q'_1$
- Continue until $Q'_n$ is computed.
- If $Q'_n$ contains a final state, $A$ accepts $w$.

All we have to do is simulate the run of $A$ on $w$. 
Equivalence problem

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 using closure properties and decidability of the emptiness problem:

\[ L_1 = L_2 \text{ iff } (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2) = \emptyset \]

words that are in $L_1$, but not in $L_2$  words that are not in $L_1$, but in $L_2$
Finiteness problem

We have already seen that every finite language is regular. Now we want to find out if a given regular language $L$ is finite.

Finiteness problem
The Finiteness problem for regular languages is decidable.

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

Let $A$ be a DFA accepting $L$.

- Eliminate from $A$ all states that are not reachable from the initial state, obtaining $A_r$.
- Eliminate from $A_r$ all states from which no final state is reachable, obtaining $A_f$.
- $A_f$ contains a loop iff $L$ is infinite.
Disproving regularity: the pumping lemma

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

206
Pumping lemma - intuition

- **Case 1: \( L \) is finite**
  - Then there exists a longest word \( w \in L \)
  - Then \( n = |w| + 1 \) works trivially

- **Case 2: \( L \) is infinite**
  - There is a DFA \( A \) with \( L(A) = A \) (because \( L \) is regular)
  - \( A \) has at most \( m \) states (it’s a **finite automaton**)
  - When accepting a word \( w \) with more than \( m \) letters, \( A \) has to visit at least one state \( q \) more than once
  - Ergo there is a loop from \( q \) to \( q \) . . .
  - . . . and we can go through this loop any number of times!
The pumping lemma can be written in a single formula as follows:

\[
\text{reg}(L) \rightarrow \exists n \in \mathbb{N} \ \forall s \in L(|s| \geq n \rightarrow \exists u, v, w \in \Sigma^* \\
(s = uvw \land v \neq \varepsilon \land |uv| \leq n \land \\
\forall h \in \mathbb{N}(uv^hw \in L)))
\] (101)

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

\[
\forall n \in \mathbb{N} \ \exists s \in L(|s| \geq n \land \forall u, v, w \in \Sigma^* \exists h \in \mathbb{N} \\
(\neg(s = uvw \land v \neq \varepsilon \land |uv| \leq n \land \\
uv^hw \in L))) \rightarrow \neg\text{reg}(L)
\] (102)
The pumping lemma: example

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

$$L = \{(^k)^k | k \in \mathbb{N}\}. \quad (103)$$

According to Eq. 102, for all possible integers $n$, we need to find an $s \in L$ whose length is greater than or equal to $n$, e.g.

$$s = (^{n})^n. \quad (104)$$

Now, we just have to show that there is no way to satisfy Conditions 1 to 4 with this $s$. 
Considering that $s = uvw$ (1), $|uv| \leq n$ (3), and $v \neq \varepsilon$ (2), we know that

$$u = (l, \quad v = (m, \quad w = (p)^n$$

(105)

with

$$l + m + p = n; \quad m \geq 1$$

(106)

i.e.

$$l + p \leq n - 1.$$ 

(107)

Now, if we are able to show that Condition 4 cannot be fulfilled, we are done.
The pumping lemma: example (cont.)

We need to show that

\[ \neg \forall h \in \mathbb{N} (uv^h w \in L) \text{ or } \exists h \in \mathbb{N} (uv^h w \notin L). \quad (108) \]

For \( h = 0 \), we would obtain the word

\[ uw = (l+p)^n \quad (109) \]

According to Eq. 107, \( l + p \neq n \), hence \( uw \notin L \) which completes the proof that

\[ \neg \text{reg}(L). \quad (110) \]
In conclusion, we see that the language

\[ L = \{ \langle k \rangle^k | k \in \mathbb{N} \}. \]  \hspace{1cm} (111)

is \textbf{not regular}. This means that

- regular languages are not capable of counting brackets;
- for most common programming languages, regular languages/grammars/expressions are not powerful enough.

In the following, we will learn more about context-free languages which are able to cope with most common programming languages.
The pumping lemma: exercise

We are given the language \( L \) comprising all the words of the form \( a^p \) where \( p \) is a prime number:

\[
L = \{a^p \mid p \in \mathbb{P}\}. \quad (112)
\]

Prove that \( L \) is not a regular language.

Hint: let \( h = p + 1 \)
Assume $L_1 = \{ a^n b^m \mid n, m \in \mathbb{N}, n > m \}$ and $L_2 = \{ a^n b^m \mid n, m \in \mathbb{N} \}$

- Is $L_1$ regular?
- Is $L_2$ regular?
- Prove your claims!

- Solve the exercise on page 86

- Bonus: Solve the equations on page 82
Review of Goals

- Refresher & Homework
- Equivalence of regular expressions
- Properties of regular languages
  - Closure properties
  - Decision problems
- Non-regular languages and the pumping lemma
Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?
Goals for Today

- ERASMUS+
- Refresher & Homework
- Real-world scanner
  - Compiler structure
  - Flex
  - Regular expressions - theory and practice
ERASMUS+

- ERASMUS+ fördert Praxisphasen im europäischen Ausland
  - Mindestens 60 Tage
  - Selbst gesuchte Praktika ok
  - Praktika bei Dualen Partnern oder Partnerfirmen ok

- Anmeldung bei geplanten Aufenthalten bis Juli 2015:
  - Ab sofort
  - Spätestens 15.12.2014!

- Weitere Information:
  - https://eu.daad.de/neu/studierende/studierendenmobilitaet/de/14998-studierendenmobilitaet/
  - Auslandsamt DHBW-Stuttgart, Frau Dorte Süchting
Refresher

- Equivalence of REs
  - $\text{RE} \Rightarrow \text{NFA} \Rightarrow \text{DFA} \Rightarrow \text{Unique Minimal DFA}$

- If $L_1, L_2$ are regular, then $L_1 \cup L_2, L_1 \cap L_2, L_1 \circ L_2, \overline{L_1}, L_1^*$ are all regular
  - Proof: Construction of NFSs as per REs ($\cup, \circ, *$)
  - Proof: Product automaton ($\cap, \cup$)
  - Proof: Swap accepting/non-accepting states ($\overline{\phantom{L}}$)

- Finite languages are regular (tree-like NFA)

- The following are decidable for regular languages:
  - Emptiness (reachability analysis of DFA/NFA)
  - Word problem (just run automaton)
  - Equivalence (as per above)
  - Finiteness (check for loops in DFA)

- Pumping lemma
  - Regular languages can be pumped
  - Normally used to show that languages are not regular
Assume $L_1 = \{a^n b^m | n, m \in \mathbb{N}, n > m\}$ and $L_2 = \{a^n b^m | n, m \in \mathbb{N}\}$

- Is $L_1$ regular?
- Is $L_2$ regular?

Solution?

- $L_1$ not regular, proof later
- $L_2$ regular. E.g. $L_2 = L(a^* b^*)$

Pumping lemma: Given a regular language $L$, there exists an integer $n \in \mathbb{N}$ such that all words $s \in L$ with $|s| \geq n$ can be split into three parts $u$, $v$, and $w$ satisfying the following conditions:

1. $s = uvw$,
2. $v \neq \varepsilon$,
3. $|uv| \leq n$,
4. $\forall h \in \mathbb{N}(uv^h w \in L)$. 
Homework: $L_1 = \{ a^n b^m | n, m \in \mathbb{N}, n > m \}$ is not regular

- Proof: By contradiction (using the pumping lemma).
- Assumption: $L_1$ is regular.
- Then: $\exists n \in \mathbb{N}$ such that $\forall s \in L_1$ with $|s| \geq n \exists u, v, w \in \Sigma^*$ with
  1. $s = uvw$,
  2. $v \neq \varepsilon$,
  3. $|uv| \leq n$,
  4. $\forall h \in \mathbb{N} (uv^h w \in L_1)$.
- Consider $s = a^n b^{n-1} \in L_1$
  - We know $|uv| \leq n$. Hence $u = a^i, v = a^j, w = a^k b^{n-1}$ and $i + j + k = n$, and $j \geq 1$ (because $v \neq \varepsilon$)
  - Now consider $s' = uv^0 w = a^i a^k b^{n-1} = a^{i+k} b^{n-1}$. Per pumping-lemma, $s' \in L_2$, and per definition of $L_2$ then $i + k > n - 1$, hence $i + k + 1 > n$
  - But $i + j + k = n$ and $j \geq 1$. Hence $i + k + 1 \leq n$
  - In summary: $i + k + 1 > n$ and $i + k + 1 \leq n\n
- The assumption leads to a contradiction, hence the assumption is wrong. Ergo: $L_1$ is not regular. q.e.d.
Lexical Analysis in Practice/Flex
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)

**Cautionary tale: ALGOL-68**
Conway’s Law

Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.

Melvin Conway, 1968

If you have four groups working on a compiler, you’ll get a 4-pass compiler.

The Jargon File

If a group of N persons implements a COBOL compiler, there will be N-1 passes. Someone in the group has to be the manager.

Tom Cheatham
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

Code generation (several optimisation passes)

e.g. assembler code

ld a, b
ld c, 1
add c
...
Flex Overview

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

Definitions
Rules
Miscellaneous code

flex + gcc

scanner

Tokenized/processed output

Development time
Execution time

Definitions
Rules
Miscellaneous code
flex + gcc
scanner

Input

Tokenized/processed output

Development time
Execution time

Input
Flex Example Task

- (Artificial) 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

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 3.1415</td>
<td>int: 12 (&quot;12&quot;)</td>
</tr>
<tr>
<td>0.33333</td>
<td>float: 3.141500 (&quot;3.1415&quot;)</td>
</tr>
<tr>
<td>print reset</td>
<td>float: 0.333330 (&quot;0.33333&quot;)</td>
</tr>
<tr>
<td>2 11</td>
<td>Current: 12 : 3.474830</td>
</tr>
<tr>
<td>1.5 2.5 print</td>
<td>Reset</td>
</tr>
<tr>
<td>1</td>
<td>int: 2 (&quot;2&quot;)</td>
</tr>
<tr>
<td>print 1.0</td>
<td>int: 11 (&quot;11&quot;)</td>
</tr>
<tr>
<td></td>
<td>float: 1.500000 (&quot;1.5&quot;)</td>
</tr>
<tr>
<td></td>
<td>float: 2.500000 (&quot;2.5&quot;)</td>
</tr>
<tr>
<td></td>
<td>Current: 13 : 4.000000</td>
</tr>
<tr>
<td></td>
<td>int: 1 (&quot;1&quot;)</td>
</tr>
<tr>
<td></td>
<td>Current: 14 : 4.000000</td>
</tr>
<tr>
<td></td>
<td>float: 1.000000 (&quot;1.0&quot;)</td>
</tr>
<tr>
<td></td>
<td>Final 14 : 5.000000</td>
</tr>
</tbody>
</table>
Flex Facts & Opinions

- **Flex**: *Fast Lexical Analyser*
  - Original: `lex` (described 1975)
  - `flex` is `lex` compatible

- **Flex** input syntax seems wonky…
  - Basics are 40 years old!
  - Represents regular expressions and C code with the same alphabet in the same file

- On the other hand: It would not have survived 40 years if it didn’t work!
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`
%%option noyywrap

DIGIT [0-9]

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

%%
Example Code (rule section)

```c
{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 */
    }
```
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);
}
> 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")
...

Generating a scanner
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
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 regular expressions as discussed before was introduced to be able to show their equivalence to finite state machines.
- Practical implementations of regular expressions (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

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

and the set of regular expressions

1. \( c \in \Sigma_{ASCII} \setminus O \rightarrow c \in R \)
2. \( " \cdot " \in R \)
   any character but newline (\n)
3. \( x \in \{a, b, f, n, r, t, v\} \rightarrow \backslash x \in R \) defines the following control characters

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

\( a, b, c \in \{0, \cdots, 7\} \rightarrow \backslash abc \in R \) octal representation of a character’s ASCII code (e.g. \040 represents the empty space “ ”)
5. \( c \in O \rightarrow \backslash c \in R \)

- escaping operator symbols

\( r_1, r_2 \in R \rightarrow r_1 r_2 \in R \)

- concatenation

\( r_1, r_2 \in R \rightarrow r_1 | r_2 \in R \)

- infix operation using “|” rather than “+”

\( r \in R \rightarrow r^* \in R \)

- Kleene star

\( r \in R \rightarrow r^+ \in R \)

- (one or more or \( r \))

\( r \in R \rightarrow r? \in R \)

- optional presence (zero or one \( r \))
11. \( r \in R, n \in \mathbb{N} \rightarrow r\{n\} \in R \)
concatenation of \( n \) times \( r \)
\( r \in R; \ m, n \in \mathbb{N}; \ m \leq n \rightarrow r\{m, n\} \in R \)
concatenation of between \( m \) and \( n \) times \( r \)
\( r \in R \rightarrow ^r \in R \)
\( r \) has to be at the beginning of line
\( r \in R \rightarrow r$ \in R \)
\( r \) has to be at the end of line
\( r_1, r_2 \in R \rightarrow r_1 / r_2 \in R \)
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.
\( r \in R \rightarrow (r) \in R \)
Grouping regular expressions with brackets.
17. Ranges

- \[\text{[aeiou]} \equiv a|e|i|o|u\]
- \[\text{[a-z]} \equiv a|b|c|\cdots|z\]
- \[\text{[a-zA-Z0-9]} : \text{alphanumeric characters}\]
- \[\text{[^0-9]} : \text{all ASCII characters w/o digits}\]

\[
\text{[ ] } \in R
\]

empty space

\[
\text{w } \in \left\{ \Sigma_{\text{ASCII}} \backslash \{\text{\textbackslash, }\text{"}"\} \right\}^* \rightarrow "w" \in R
\]

verbatim text (no escape sequences)
21. $r \in R \rightarrow \sim r \in R$

The *upto* operator matches the *shortest* string ending with $r$. predefined character classes

22. \[
\begin{align*}
\text{[:alnum:]} & \quad \text{[:alpha:]} & \quad \text{[:blank:]} \\
\text{[:cntrl:]} & \quad \text{[:digit:]} & \quad \text{[:graph:]} \\
\text{[:lower:]} & \quad \text{[:print:]} & \quad \text{[:punct:]} \\
\text{[:space:]} & \quad \text{[:upper:]} & \quad \text{[:xdigit:]} \\
\end{align*}
\]
Regular Expressions in Practice (precedences)

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

Example:
\[ a∗b | c+d e \equiv ( (a∗) b ) | ( (c+) d ) e \]

Rule of thumb: ∗, +, ? bind the smallest possible RE. Use () if in doubt!
Assume definition NAME \ DEF
  In later REs. \{NAME\} is expanded to (DEF)

Example:

DIGIT    [0–9]
INTEGER  {DIGIT}+
PAIR     \((\{INTEGER\},\{INTEGER\}\))
Example Code (definition section) (revisited)

```%
%%option noyywrap

DIGIT   [0-9]

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

```
Exercise FLRE

- Assume we work over $\Sigma_{\text{ascii}}$
- How would you express the following practical REs using only the simple REs we have used so far?
  - [a–z]
  - [^0–9]
  - (r)+
  - (r){3}
  - (r){3,7}
  - (r)?
This is the core of the scanner!

Rules have the form `PATTERN ACTION`

Patterns are regular expressions
  - Typically use previous definitions

THERE IS 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)

```c
{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 */
 }
```
- 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)
```c
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 Miscellany

- Flex online:
  - http://flex.sourceforge.net/
  - REs:
    http://flex.sourceforge.net/manual/Patterns.html

- make knows flex
  - Make will automatically generate file.o from file.l
  - Be sure to set LEX=flex to enable flex extensions
  - Makefile example:
    LEX=flex
    all: scan_numbers
    numbers.o: numbers.l
    
    scan_numbers: numbers.o
    gcc numbers.o -o scan_numbers
A security audit firm needs a tool that scans documents for the following:

- **Email addresses**
  - Format: 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 \([A-Za-z0-9_.~-]\)
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
Erstellen Sie ein solches Programm

Würde Sie so ein Auftrag nachdenklich machen? Warum oder warum nicht?

Beispieldateien zum Testen gibt es auf der Kurswebseite
Review of Goals

- ERASMUS+
- Refresher & Homework
- Real-world scanner
  - Compiler structure
  - Flex
  - Regular expressions - theory and practice
Feedback round

➤ What was the best part of today's lecture?
➤ What part of today's lecture has the most potential for improvement?
  ➤ Optional: how would you improve it?
Goals for Today

► Refresher & Homework
► Formal Grammars
  ► Definition
  ► The Chomsky Hierarchy
  ► Regular languages and right-linear grammars
  ► Context-free grammars
    ► Normal forms
Refresher

- Structure of programming languages and compilers
  - Regular vocabulary/Scanner
  - Context-free program structure/Parser
  - Context-sensitive constraints/Special hacks ;-
  - ... and then code generation

- Flex overview
  - Input: Patterns (REs)+Actions(C code)
  - Output: Scanner in C
  - Flex workflow

- Example: Scanning and adding numbers

- Practical regular expressions
  - Ranges [...], +, repetitions, ...
A security audit firm needs a tool that scans documents for the following:

- Email addresses
- Web addresses
- Bank account numbers

Develop the required program

Would such a request make you think? Why or why not?
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

\[ S \to aA, \quad A \to bB, \quad B \to \varepsilon \]
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

Noam Chomsky defined a grammar as a quadruple

\[ G = \langle V_N, V_T, P, S \rangle \]  \hspace{2cm} (114)

with

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

\[ \alpha \rightarrow \beta \]  \hspace{2cm} (115)

with \( \alpha \in V^* V_N V^* \), \( \beta \in V^* \), \( V = V_N \cup V_T \)

4. the distinguished **start symbol** \( S \in V_N \).
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 Study (Princeton), UC Berkeley, . . .
- 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
For the sake of simplicity, we will be using the short form

\[ \alpha \rightarrow \beta_1 \mid \cdots \mid \beta_n \] replacing \( \alpha \rightarrow \beta_1 \) \hspace{1cm} \text{(116)}

\[ \vdots \]

\[ \alpha \rightarrow \beta_n \]
Example: C identifiers

We want to define a grammar

\[ G = \langle V_N, V_T, P, S \rangle \]  \hspace{1cm} (117)

to describe identifiers of the C programming language:

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

\[ V_N = \{ I, R, L, D \} \text{ (identifier, rest, letter, digit)}, \]
\[ V_T = \{ a, \cdots, z, A, \cdots, Z, 0, \cdots, 9, \_ \}, \]
\[ P = \{ \]
\[ I \rightarrow LR|_R|L|_ \]
\[ R \rightarrow LR|DR|_R|L|D|_ \]
\[ L \rightarrow a|\cdots|z|A|\cdots|Z \]
\[ D \rightarrow 0|\cdots|9 \}

\[ S = I. \]
Formal grammars: derivation

Derivation: description of operation of grammars
Given the grammar

\[ G = \langle V_N, V_T, P, S \rangle, \quad (118) \]

we define the relation

\[ x \Rightarrow_G y \quad (119) \]

iff \( \exists u, v, p, q \in V^* : (x = upv) \land (p \rightarrow q \in P) \land (y = uqv) \) (120)

pronounced as “\( G \) derives \( y \) from \( x \) in one step”.

We also define the relation

\[ x \Rightarrow^*_G y \text{ iff } \exists w_0, \ldots, w_n \quad (121) \]

with \( w_0 = x, w_n = y, w_{i-1} \Rightarrow_G w_i \) for \( i \in \{1, \ldots, n\} \)

pronounced as “\( G \) derives \( y \) from \( x \) (in zero or more steps)”.
Formal grammars: derivation example I

\[ G = \langle V_N, V_T, P, S \rangle \]  \hspace{1cm} (122)

with

1. \( V_N = \{ S \} \),
2. \( V_T = \{ 0 \} \),
3. \( P = \{ S \to 0S, \quad S \to 0 \} \),
4. \( S = S \).

Derivations of \( G \) have the general form

\[ S \Rightarrow 0S \Rightarrow 00S \Rightarrow \cdots \Rightarrow 0^{n-1}S \Rightarrow 0^n \]  \hspace{1cm} (123)

Apparently, the language produced by \( G \) (or the language of \( G \)) is

\[ L(G) = \{ 0^n | n \in \mathbb{N}; n > 0 \} . \]  \hspace{1cm} (124)
Formal grammars: derivation example II

\[ G = \langle V_N, V_T, P, S \rangle \]  

(125)

with

1. \( V_N = \{ S \} \),
2. \( V_T = \{ 0, 1 \} \),
3. \( P = \{ S \rightarrow 0S1, \quad S \rightarrow 01 \} \),
4. \( S = S \).

Derivations of \( G \) have the general form

\[ S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow \cdots \Rightarrow 0^{n-1}S1^{n-1} \Rightarrow 0^n1^n. \]  

(126)

The language of \( G \) is

\[ L(G) = \{ 0^n1^n | n \in \mathbb{N}; n > 0 \} \].  

(127)

Reminder: \( L(G) \) is not regular!
Formal grammars: derivation example III

\[ G = \langle V_N, V_T, P, S \rangle \]  \hspace{1cm} (128)

with

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

\[
\begin{align*}
S & \rightarrow 0 SBC \hspace{1cm} 1 \\
S & \rightarrow 0 BC \hspace{1cm} 2 \\
CB & \rightarrow BC \hspace{1cm} 3 \\
0 B & \rightarrow 01 \hspace{1cm} 4 \\
1 B & \rightarrow 11 \hspace{1cm} 5 \\
1 C & \rightarrow 12 \hspace{1cm} 6 \\
2 C & \rightarrow 22 \hspace{1cm} 7
\end{align*}
\]
4. \( S = S \).
Formal grammars: derivation example III (cont.)

Derivations of $G$ have the general form

$$S \Rightarrow_1 0SBC \Rightarrow_1 00SBCBC \Rightarrow_1 \cdots \Rightarrow_1 0^{n-1}S(BC)^{n-1} \Rightarrow_2 0^n(BC)^n$$

$$\Rightarrow_3^* 0^nB^nC^n \Rightarrow_{4,5}^* 0^n 1^n C^n \Rightarrow_{6,7}^* 0^n 1^n 2^n$$

The language of $G$ is

$$L(G) = \{0^n 1^n 2^n | n \in \mathbb{N}; n > 0\}.$$ (130)

- 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).
The Chomsky hierarchy (0)

Given the grammar

\[ G = \langle V_N, V_T, P, S \rangle, \]  

we define the following grammar/language classes

- \( G \) is of Type 0 or unrestricted

All grammars are Type 0!
The Chomsky hierarchy (1)

\[ G = \langle V_N, V_T, P, S \rangle, \]  

(132)

- \( G \) is **Type 1 or context-sensitive** if all productions are of the form

\[ \alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2 \text{ with } A \in V_N; \alpha_1, \alpha_2 \in V^*, \beta \in VV^* \]  

(133)

Exception:

\[ S \rightarrow \varepsilon \in P \text{ is allowed if} \]

\[ \alpha_1, \alpha_2 \in (V \setminus \{S\})^* \text{ and } \beta \in (V \setminus \{S\})(V \setminus \{S\})^* \]  

(134)

- If \( S \rightarrow \varepsilon \in P \), then \( S \) is not allowed in any right hand side
- Consequence: Rules (almost) never derive shorter words
The Chomsky hierarchy (2)

\[ G = \langle V_N, V_T, P, S \rangle \]  

- \( G \) is of **Type 2 or context-free** if all productions are of the form

\[ A \rightarrow \beta \text{ with } A \in V_N; \beta \in VV^* \]  

Exception:

\[ S \rightarrow \epsilon \in P \text{ is allowed, if } \beta \in (V\backslash\{S\})(V\backslash\{S\})^* \]

- Only single non-terminals are replaced
- If \( S \rightarrow \epsilon \in P \), then \( S \) is not allowed in any right hand side
The Chomsky hierarchy (3)

\[ G = \langle V_N, V_T, P, S \rangle \quad (138) \]

- \( G \) is of Type 3 or *right-linear* (*regular*) if all productions are of the form
  \[ A \rightarrow aB \text{ or } A \rightarrow a \]
  \( A, B \in V_N; a \in V_T \)

Exception:
  \[ S \rightarrow \varepsilon \in P \text{ is allowed, if } B \in V_N \setminus \{S\} \quad (140) \]
For each grammar/language type, there is a corresponding type of machine model:

<table>
<thead>
<tr>
<th>grammar</th>
<th>language</th>
<th>machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 0</td>
<td>unrestricted</td>
<td>Turing machine</td>
</tr>
<tr>
<td>Type 1</td>
<td>context-sensitive</td>
<td>linear-bounded non-deterministic Turing machine</td>
</tr>
<tr>
<td>Type 2</td>
<td>context-free</td>
<td>non-deterministic pushdown automaton</td>
</tr>
<tr>
<td>Type 3</td>
<td>regular</td>
<td>finite automaton</td>
</tr>
</tbody>
</table>
The Chomsky hierarchy: examples

Returning to our example with identifiers of the C programming language:

\[
P : \quad I \rightarrow LR|_R|L|
\]
\[
R \rightarrow LR|DR|_R|L|D|
\]
\[
L \rightarrow a|\cdots|z|A|\cdots|Z
\]
\[
D \rightarrow 0|\cdots|9
\]

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

\[
P : \quad I \rightarrow A|\cdots|Z|a|\cdots|z|_|
\]
\[
AR|\cdots|ZR|aR|\cdots|zR|_R
\]
\[
R \rightarrow A|\cdots|Z|a|\cdots|z|_0|\cdots|9|
\]
\[
AR|\cdots|ZR|aR|\cdots|zR|_R|0R|\cdots|9R
\]
Returning to the three derivation examples:

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

II
- The grammar with $P = \{ \langle S \rightarrow 0 S 1 \rangle, \langle S \rightarrow 0 1 \rangle \}$ is context-free.
- So is the produced language $L = \{0^n 1^n | n \in \mathbb{N}; n > 0 \}$.
III

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

$$CB \rightarrow CX$$

$$CX \rightarrow BX$$

$$BX \rightarrow BC$$

without changing the grammar’s behavior.
– The resulting grammar is context-sensitive.
– So is the language $L = \{0^n1^n2^n | n \in \mathbb{N}; n > 0\}$. 
The Chomsky hierarchy: exercises

\[ G = \langle V_N, V_T, P, S \rangle \]  

with

1. \( V_N = \{ S, A, B \} \),
2. \( V_T = \{ 0 \} \),
3. \( P : \)
   \[
   \begin{align*}
   S & \rightarrow \varepsilon & 1 \\
   S & \rightarrow ABA & 2 \\
   AB & \rightarrow 00 & 3 \\
   0A & \rightarrow 000A & 4 \\
   A & \rightarrow 0 & 5
   \end{align*}
   \]
4. \( S = S \).

a) What is \( G \)'s highest type?
b) Show how \( G \) derives the word \( 00000 \).
c) Formally describe the language \( L(G) \).
d) Define a regular grammar \( G' \) equivalent to \( G \).
An octal constant is a finite sequence of digits starting with 0 followed by at least one digit ranging from 0 to 7. Define a regular grammar encoding exactly the set of possible octal constants.
The Chomsky hierarchy: exercises (cont.)

\[ G = \langle V_N, V_T, P, S \rangle \]  

with

1. \( V_N = \{ S, N, E \} \),
2. \( V_T = \{ 0, 1, t \} \),
3. \( P : \)
   
   \[
   \begin{align*}
   S & \rightarrow 0NS & 1 \\
   S & \rightarrow 1ES & 2 \\
   S & \rightarrow t & 3 \\
   N_0 & \rightarrow 0N & 6 \\
   N_1 & \rightarrow 1N & 7 \\
   E_0 & \rightarrow 0E & 8 \\
   E_1 & \rightarrow 1E & 9 \\
   N_0 & \rightarrow t0 & 4 \\
   E_1 & \rightarrow t1 & 5 \\
   \end{align*}
   \]
4. \( S = S \).

a) What is \( G \)'s highest type?
b) Formally describe the language \( L(G) \).
Equivalence of regular languages and regular grammars

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

▸ Idea for proof?
  ▸ Convert DFA to regular grammar
  ▸ Convert regular grammar to NFA
Algorithm for transforming a DFA $A = (Q, \Sigma, \delta, q_0, F)$ into a grammar $V_N, V_T, P, S$:

- $V_N = Q$
- $V_T = \Sigma$
- $S = q_0$
- $P = \{ p \rightarrow aq \mid (p, a, q) \in \delta \} \cup \{ p \rightarrow a \mid (p, a, q) \in \delta \text{ for any } q \in F \}$
Consider the following DFA $A$:

![DFA Diagram]

- Give a formal definition of $A$
- Generate a regular grammar $G$ with $L(G) = L(A)$
Algorithm for transforming a grammar $V_N, V_T, P, S$ into an NFA $A = (Q, \Sigma, \delta, q_0, F)$:

- $Q = V_N \cup \{q_f\}$ \hspace{0.5cm} ($q_f \notin V_N$)
- $\Sigma = V_T$
- $q_0 = S$
- $F = \{q_f\}$
- $\delta = \{(A, c, B) \mid A \rightarrow cB \in P\} \cup \{(A, c, q_f) \mid A \rightarrow c \in P\}$
Context-free grammars

- Reminder: $G = \langle V_N, V_T, P, S \rangle$ is context-free, if all $l \rightarrow r \in P$ are of the form $A \rightarrow \beta$ with
  - $A \in V_N$ and $\beta \in VV^*$
  - (special case: $S \rightarrow \epsilon \in P$, then $S$ is not allowed in any $\beta$)
- Context-free languages/grammars are highly relevant
  - Core of most programming languages
  - Algebraic expressions
  - XML
  - Many aspects of human language
As for automata / regular expressions, two context-free grammars are called (weakly) equivalent if they generate the same language.

We will now compute grammars that are equivalent to some given $G$ but have “nicer” properties

- Reduced grammars contain no unproductive symbols
- Grammars in Chomsky normal form support efficient (and very efficient) solution of the word problem
For a context-free grammar $G$, $G_r$ is the equivalent reduced grammar, which contains only reachable and terminating symbols.

The reachable symbols can be computed as follows:

- $R := \{ S \}$
- for every $N \in R$, add all symbols $M$ for which there is a rule $N \rightarrow V^*MV^*$
- when no further symbols can be added, $R$ contains exactly the reachable symbols
The terminating symbols can be computed as follows:

- $T := \{ N \in V_T \mid \exists w \in V_T^* : N \rightarrow w \in P \}$
- add all symbols $M$ to $T$ for which there exists a rule $M \rightarrow D$ and all non-terminal symbols in $D$ are also contained in $T$
- when no further symbols can be added, $R$ contains exactly the reachable symbols

Removal of (a) non-terminating and (b) unreachable symbols (and the corresponding production rules) generates the reduced grammar $G_r$:

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

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

\[ G = \langle V_N, V_T, P, S \rangle \] (144)

with

1. \( V_N = \{ S, A, B, C \} \),
2. \( V_T = \{ a, b \} \),
3. \( P : \)

\[
\begin{align*}
S & \rightarrow A | aS | C \\
A & \rightarrow a \\
B & \rightarrow aa \\
C & \rightarrow aCb
\end{align*}
\]

4. \( S = S. \)
   - \( B \) is not reachable.
   - \( C \) does not terminate.
Compute the reduced grammar $G = (V_N, V_T, P, S)$ for the following grammar $G' = (V'_N, V'_T, P', S')$:

1. $V'_N = \{S, A, B, C, D\}$,
2. $V'_T = \{a, b\}$,
3. $P'$:
   
   - $S \rightarrow A | aS | B$
   - $A \rightarrow a$
   - $A \rightarrow AS$
   - $A \rightarrow Ba$
   - $B \rightarrow Ba$
   - $C \rightarrow Da$
   - $D \rightarrow Cb$
   - $D \rightarrow a$
4. $S' = S$. 
Rules of the kind $A \rightarrow B$ can be eliminated:

- for every $A \in V_N$, compute the set $N(A) = \{ B \in V_N \mid A \Rightarrow^* G B \}$
- add production rules
  \[ \{ A \rightarrow w \mid w \notin V_N \text{ and } B \rightarrow w \in P \text{ and } B \in N(A) \}. \]
- Remove $A \rightarrow B$

Example

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

is equivalent to

$A \rightarrow a|bb|ccc; \quad B \rightarrow bb|ccc; \quad C \rightarrow ccc$
Chomsky normal form

A context free grammar \( G \) is in Chomsky normal form if all production rules are of the kind

- \( A \rightarrow a \) with \( a \in V_T \) or
- \( A \rightarrow BC \) with \( \{B, C\} \subseteq V_N \).

Theorem: Transformation into Chomsky normal form

Every context free grammar (that does not contain \( S \rightarrow \varepsilon \)) can be transformed into an equivalent grammar in Chomsky normal form.
Algorithm for computing Chomsky normal form

1. remove $A \rightarrow B$ rules
2. compute reduced grammar (remove non-terminating and unreachable symbols)
3. in all rules $A \rightarrow w$ with $w \notin V_T$, replace all occurrences of $a$ with $X_a$ for all $a \in V_T$
4. add rules $X_a \rightarrow a$
5. replace rules $A \rightarrow B_1 B_2 \ldots B_n$ for $n > 2$ with

\[
\begin{align*}
A & \rightarrow B_1 C_1 \\
C_1 & \rightarrow B_2 C_2 \\
\vdots \\
C_{n-2} & \rightarrow B_{n-1} B_n
\end{align*}
\]
Chomsky normal form: exercise

Compute the Chomsky normal form of the following grammar:

\[ G = (V_N, V_T, P, S) \]

1. \( V_N = \{S, A, B, C\} \),
2. \( V_T = \{a, b\} \),
3. \( P : \)
   \[
   \begin{align*}
   S & \rightarrow AB | SB | B \\
   A & \rightarrow Aa \\
   B & \rightarrow bB \\
   C & \rightarrow SB \\
   B & \rightarrow BaB \\
   B & \rightarrow ab
   \end{align*}
   \]
4. \( S = S \).
Chomsky NF: purpose

Why transform $G$ into Chomsky NF?

- in a context-free grammar, derivations can have arbitrary length $C \rightarrow B; B \rightarrow C$

- **word problem** is difficult to decide

- if $G$ is in Chomsky NF, 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)**
Consider the following DFA

- Compute the corresponding regular grammar $G$
- Compute the reduced grammar $G_r$
- Convert $G_r$ into Chomsky normal form
Review of Goals

- Refresher & Homework
- Formal Grammars
  - Definition
  - The Chomsky Hierarchy
  - Regular languages and right-linear grammars
  - Context-free grammars
    - Normal forms
Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?
Goals for Today

- Refresher
- Chomsky Normal Form (again)
- Cocke-Younger-Kasami (CYK) parsing
- Pushdown automata and context-free grammars
Refresher: Grammars

▶ Grammar: \( G = \langle V_N, V_T, P, S \rangle \)
  ▶ \( ulv \Rightarrow_G urv \) if \( l \rightarrow r \in P \)
  ▶ \( L(G) = \{ w \in V_T^* | S \Rightarrow^*_G w \} \)

▶ Chomsky-Hierarchy
  ▶ Type 0: No restrictions
  ▶ Type 1: \( \alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2, A \in V_N, \alpha_1, \alpha_2 \in V^*, \beta \in VV^* \)
  ▶ Type 2: \( A \rightarrow VV^*, A \in V_N \)
  ▶ Type 3: \( A \rightarrow ab \) or \( A \rightarrow a, A, B \in V_N, a \in V_t \)
  ▶ Exception for types 1-3: \( S \rightarrow \epsilon \) is allowed if \( S \) does not occur in any right-hand-side

▶ \( L \) is type \( X \), if there is a type \( X \) grammar \( G \) with \( L(G) = L \)
  ▶ Every type 3 language is a type 2 language
  ▶ Every type 2 language is a type 1 language
  ▶ Every type 1 language is a type 0 language
Type 3 languages are exactly regular languages!

Proof:
- Generate NFA from grammar
- Generate grammar from DFA
- Idea:
  - The non-terminal (at most one) in a derived word corresponds to the state of the FA
  - The letter in the transition corresponds to the terminal (at most one!) in a rule
Chomsky Normal Form

- \( G = \langle V_N, V_T, P, S \rangle \) is in CNF, if for all \( p \in P \):
  - \( p \) is of the form \( A \rightarrow a \) with \( A \in V_N, a \in V_T \) or
  - \( p \) is of the form \( A \rightarrow BC \) with \( A, B, C \in V_N \)

- Algorithm:
  1. Remove \( A \rightarrow B \)
     - for every \( A \in V_N \), compute the set \( N(A) = \{ B \in V_N \mid A \Rightarrow^*_G B \} \)
     - add production rules
       \( \{ A \rightarrow w \mid w \notin V_N \text{ and } B \rightarrow w \in P \text{ and } B \in N(A) \} \).
     - Remove \( A \rightarrow B \)
  2. Find terminating symbols, remove non-terminating ones
  3. Find reachable symbols, remove non-reachable
  4. Add \( X_a \rightarrow a \) if necessary, replace terminals in compound RHSs
  5. Expand \( A \rightarrow B_1 B_2 \ldots B_n \) to \( A \rightarrow B_1 C_1, C_1 \rightarrow B_2 \ldots B_n \), repeat until no RHS has more than 2 non-terminals
Compute the Chomsky normal form of the following grammar:

\[ G = \langle V_N, V_T, P, S \rangle \]

1. \( V_N = \{ S, A, B, C \} \),
2. \( V_T = \{ a, b \} \),
3. \( P: \)
   
   \[
   \begin{align*}
   S & \rightarrow AB \mid SB \mid B \\
   A & \rightarrow Aa \\
   B & \rightarrow bB \\
   C & \rightarrow SB \\
   B & \rightarrow BaB \\
   B & \rightarrow ab
   \end{align*}
   \]
4. \( S = S \).
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 $A \rightarrow a$).
- iteratively find all NTS that can generate the required sequence of NTS (using production rules $A \rightarrow BC$).
- if $S$ can produce the required sequence, $w \in L(G)$ holds.

**Mechanism:**

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

Example of **dynamic programming**!
**CYK algorithm: example**

\[ S \rightarrow a \]
\[ B \rightarrow b \]
\[ B \rightarrow c \]
\[ S \rightarrow SA \]
\[ A \rightarrow BS \]
\[ B \rightarrow BB \]
\[ B \rightarrow BS \]

\[ w = abacba \]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>∅</td>
<td>S</td>
<td>∅</td>
<td>∅</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>A, B</td>
<td>B</td>
<td>B</td>
<td>A, B</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S</td>
<td>∅</td>
<td>∅</td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>B</td>
<td>A, B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>A, B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ w = \text{abacba} \]
CYK: formal algorithm

for $i := 1$ to $n$ do
    $N_{ii} := \{ A \mid A \rightarrow a_i \in P \}$
for $d := 1$ to $n - 1$ do
    for $i := 1$ to $n - d$ do
        $j := i + d$
        $N_{ij} := \emptyset$
        for $k := i$ to $j - 1$ do
            $N_{ij} := N_{ij} \cup \{ A \mid A \rightarrow BC \in P; B \in N_{ik}; C \in N_{(k+1)j} \}$
Consider the grammar
\[ G = \langle V_N, V_T, P, S \rangle \] from the previous exercise CNF

- \[ V_N = \{ S, A, B, C \} \]
- \[ \{ S, A, B, C_1, X_a, X_b \} \]
- \[ \{ S, A, B, D, X, Y \} \]

- \[ V_T = \{ a, b \} \]

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

a) \( w_1 = babaab \)

b) \( w_2 = abba \)
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

- Input is read left-to-right
- Stack is processed LIFO
- Transition is triggered by:
  - State
  - Current input symbol
  - Current top-of-stack
- Transition
  - can consume input character (ε transition are possible)
  - must consume top-of-stack
  - can write several characters onto stack
- Acceptance
  - PDA is in accepting state after processing word w
Pushdown automata: definition

Pushdown automaton
A pushdown automaton (PDA) is a 7-tuple \((Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)\) where

- \(Q, \Sigma, q_0, F\) 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

Intuitively, a transition \((p, a, R, STU, q)\) means:

- if the PDA is in state \(p\),
- the next input character is \(a\),
- and the top stack symbol is \(R\),
- then write \(STU\) on top of the stack
- and switch to state \(q\)
- We can write this as \(paR \rightarrow STUq\)
PDAs: important properties

▸ PDAs defined above are non-deterministic
▸ deterministic PDAs are weaker
▸ \( \varepsilon \) transitions are possible

Configuration

A configuration is a tuple \((q, w, \gamma)\) where

▸ \( q \) is the current state
▸ \( w \) is the input yet unread
▸ \( \gamma \) is the current stack content

\( A \) accepts a word \( w \) if after reading \( w \), \( a \) is in a final state.
PDA: example (1)

The PDA \( a = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F) \) accepts \( a^n b^n \):

\( Q = \{q_0, q_1, q_f\}; \)
\( \Sigma = \{a, b\}; \)
\( \Gamma = \{Z, Z_0\}; \)
\( F = \{q_f\}; \)
\( \delta : \)

\[
\begin{array}{lll}
q_0 & \varepsilon & Z_0 \rightarrow \varepsilon & q_f & \text{accept empty word} \\
q_0 & a & Z_0 \rightarrow ZZ_0 & q_0 & \text{read first } a, \text{ store } Z \\
q_0 & a & Z \rightarrow ZZ & q_0 & \text{read further } a, \text{ store } Z \\
q_0 & b & Z \rightarrow \varepsilon & q_1 & \text{read first } b, \text{ delete } Z \\
q_1 & b & Z \rightarrow \varepsilon & q_1 & \text{read further } b, \text{ delete } Z \\
q_1 & \varepsilon & Z_0 \rightarrow \varepsilon & q_f & \text{go to final state if input has been read} \\
& & & & \text{and all } Zs \text{ have been deleted}
\end{array}
\]
PDA: example (2)

Process $aabb$:
1. $(q_0, aabb, Z_0)$
2. $(q_0, abb, ZZ_0)$
3. $(q_0, bb, ZZZ_0)$
4. $(q_1, b, ZZ_0)$
5. $(q_1, \varepsilon, Z_0)$
6. $(q_f, \varepsilon, \varepsilon)$

Process $abb$
1. $(q_0, abb, Z_0)$
2. $(q_0, bb, ZZ_0)$
3. $(q_1, b, Z_0)$
4. No rule applicable
Define a PDA detecting all palindromes, i.e. all words \( \{ w \cdot \overline{w} \mid w \in \{a, b\} \} \) where \( \overline{w} = a_n \ldots a_1 \) if \( w = a_1 \ldots a_n \).

Can you define a deterministic automaton?
Equivalency of PDAs and Context-Free Grammars

Theorem: The languages accepted by a PDA are exactly the languages produced by any context-free grammar!

- If $L$ is context-free, there is a context-free grammar $G$ with $L(G) = L$
- If $L$ is context-free, there is a push-down automaton $A$ with $L(A) = L$

Proof (idea):
- Generate $G$ from $A$
- Generate $A$ from $G$
From context-free grammars to PDAs

For a grammar $G = (V_N, V_T, P, S)$, an equivalent PDA $A = ([q_0, q, q_f], \Sigma, \Sigma \cup V_N \cup \{Z_0\}, \delta, q_0, Z_0, \{q_f\})$ can be produced as follows:

$$\delta = \{(q_0, \varepsilon, Z_0, SZ_0, q)\} \cup$$
$$\{(q, \varepsilon, A, \gamma, q) \mid A \rightarrow \gamma \in P\} \cup$$
$$\{(q, a, a, \varepsilon, q) \mid a \in \Sigma\} \cup$$
$$\{(q, \varepsilon, Z_0, \varepsilon, q_f)\}$$

This PDA simulates the productions of $G$ in the following way:

- start by pushing $S$ onto the stack
- 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 input
- if only $Z_0$ is on the stack and the entire input is read, accept.
For the grammar $G = (\{S\}, \{a, b\}, P, S)$ with

$$
P = \{ S \rightarrow aSa \\
S \rightarrow bSb \\
S \rightarrow aa \\
S \rightarrow bb \}
$$

create an equivalent PDA $A$ and show how $A$ processes the input $abba$. 
From PDAs to context-free grammars

Transforming a PDA $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ into a grammar $G = (V_N, V_T, P, S)$ is more involved:

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

Problem:
- NTSs represent stack symbols
- when there are not NTSs left, the word belongs to $L(G)$
- when the stack is empty before a final state is reached, the word does not belong to $L(A)$

Solution: transform $A$ into a PDA
$A' = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta', X_0, p_0, F \cup \{p_f\})$ where the stack is empty iff $A$ reaches a final state.
- empty the stack if a final state is reached
  - add transitions $(q_f, \varepsilon, \gamma, \varepsilon, p_f)$ for all $q_f \in F, \gamma \in \Gamma$
- add a new initial stack symbol that can only be deleted in $p_f$
  - add transition $(p_0, \varepsilon, X_0, Z_0X_0, q_0)$
Step 2: Convert transitions to production rules

Transform $A' = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta', X_0, p_0, F \cup \{p_f\})$ into
$G = (V_N, \Sigma, P, [p_0X_0p_f])$

- $V_N = \{[p, Z, q] | \{p, q\} \subseteq Q, Z \in \Gamma\}$
- 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\} \subseteq \Gamma, P$ contains rules
  - $[p, Z, q_n] \rightarrow a[qY_1q_1][q_1 Y_2q_1] \ldots [q_{n-1} Y_nq_n]$
  - for all sets of states $\{q_1, q_2, \ldots q_n\} \subseteq Q$
- the start symbol $[p_0X_0p_f]$ means that the automaton has to go from the initial to the final state deleting the initial stack symbol
Step 3: Remove $\varepsilon$ rules

Problem: a transition $(p, \varepsilon, Y, \varepsilon, q)$ leads to a rule $[pYq] \rightarrow \varepsilon$

Solution: replace $\varepsilon$ rules with non-shortening rules

- iteratively find all symbols that can become $\varepsilon$
  - $E_0 = \{ Y \in V_N \mid Y \rightarrow \varepsilon \in P \}$
  - $E_i = E_{i-1} \cup \{ Y \in V_N \mid Y \rightarrow Y_1 \ldots Y_n \text{ and } \{Y_1, \ldots, Y_n\} \subseteq E_{i-1} \}$
- for each rule with $Y \in E_{\text{max}}$ on the right side, add a corresponding rule without $Y$
  - e.g. for $A \rightarrow YaX$, add $A \rightarrow aX$
- delete $Y \rightarrow \varepsilon$ rules
Closure properties

Closure under $\cup, \cdot,*$

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

For context-free grammars $G_1 = (V_{N_1}, V_T, P_1, S_1)$ and $G_2 = (V_{N_2}, V_T, P_2, S_2)$ with $V_{N_1} \cap V_{N_2} = \emptyset$ (rename NTSs if needed), let $S$ be a new start symbol.

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

A pumping lemma similar to the one regular languages also holds for context-free languages. It can be used to show that a language is not context-free.

Idea:
- If a grammar produces words of arbitrary length, there must be a repeated NTS.
- This NTS produces itself (and other symbols).
- This cycle can be repeated arbitrarily often.

Difference: instead of pumping one part of the word, two are pumped in parallel: \( uv^hwx^hy \in L(G) \).

Can not be applied to \( \{a^n b^n\} \), but to \( \{a^n b^n c^n\} \).

\( \{a^n b^n c^n\} \) is not context-free, but context-sensitive, as we have seen before.
Closure under $\cap$
Context-free languages are not closed under intersection.

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

- $\{a^n b^n c^m\}$ is context-free
- $\{a^m b^n c^n\}$ is context-free
- $\{a^n b^n c^n\} = \{a^n b^n c^m\} \cap \{a^m b^n c^n\}$
1. Define context-free grammars for $\{a^n b^n c^m \mid n, m \geq 0\}$ and $\{a^m b^n c^n \mid n, m \geq 0\}$

2. Use the known closure properties to show that context-free languages are not closed under complement.
The word problem for cf. languages
For a word $w$ and a context-free grammar $G$, it is **decidable** whether $w \in L(G)$ holds.

The CYK algorithm decides the word problem.
Decision problems: emptiness problem

The emptiness problem for cf. languages
For a context-free grammar $G$, it is decidable if $L(G) = \emptyset$ holds.

- The pumping lemma gives us a maximum length: if a grammar produces any words, then it also produces one of maximum length $n$ (depending on the properties of the grammar).
- Since there is only a finite number of words up to length $n$ and the word problem is decidable, emptiness is also decidable.
The 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.
Homework

- Read through the open material on
  - Context-free grammars
  - Push-down automata
  - Closure properties of context-free languages
- Bonus: Understand the Pumping Lemma for CF languages (e.g. in Hoffmann)
Review of Goals

- Refresher
- Chomsky Normal Form (again)
- Cocke-Younger-Kasami (CYK) parsing
- Pushdown automata and context-free grammars
Feedback round

- What was the best part of today’s lecture?
- What part of today’s lecture has the most potential for improvement?
  - Optional: how would you improve it?
Goals for Today

- Refresher
- Practical Parsing with YACC/Bison
  - Background and Principles
  - Workflow
  - Desk calculator example
- Turing Machines
  - Basics
  - A working example
  - Skimming over some topics of computability
Chomsky Normal Form for context-free grammars

Parsing with Cocke-Younger-Kasami (CYK)

Pushdown-automata (PDA)

- Unlimited stack
- “Instructions”: $qcZ \rightarrow Wq'$
- Transitions consume top-symbol, can write back many symbols
- Transition can use and consume current letter of word
- Non-determinism!
- Success: PDA is in accepting state after consuming word

PDAs are equivalent to context-free grammars

- PDA can “execute” grammar
- Grammar can “simulate” PDA (with some thinking)

Context-free language properties:

- Pumping possible with $uvwxy$-lemma
- CF languages are closed under $\cup, \cdot,^*$ (construct grammar)
- CF languages are not closed under $\cap (a^nb^nc^m \cap a^mb^nc^n)$
- The word problem for CF-languages is decidable (CYK)
- Emptiness is decidable, equivalency is not decidable
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)
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

Code generation (several optimisation passes)

e.g. assembler code

Variable Type
<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
</tr>
<tr>
<td>b</td>
<td>int</td>
</tr>
</tbody>
</table>

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

Flex

Bison
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: Definitions, grammar rules with actions, support code
  - Definitions: Token names, associated values, includes, declarations
  - 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

Bison Input File
<file>.y
Definitions file
<file>.tab.h
Parser Source
<file>.tab.c
Flex Input file
<file>.l
Lexer Source
<file>.c
Lexer object
<file>.o
Parser object
<file>.tab.o
Final executable
parser
Some input
to process
Some output
produced
Bison
Flex
gcc
gcc
linker (gcc)
#include

Execution time

Development time

Flex Input file
<file>.l
Definitions file
<file>
Parser Source
<file>.c
Lexer Source
<file>.c
Lexer object
<file>.o
 GCC
Parser object
<file>.tab.o
Final executable
parser
Some input
to process
Some output
produced
Bison
Flex
gcc
gcc
linker (gcc)
#include
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)

\[ G_{DC} = \langle V_N, V_T, P, S \rangle \]

\[ V_T = \{ \text{PLUS, MULT, ASSIGN, OPENPAR, CLOSEPAR, REGISTER, FLOAT, \ldots } \} \]

- Some terminals are single characters (+, =, \ldots)
- Others are complex: R10, 1.3e7
- Terminals (“tokens”) are generated by the lexer

\[ V_N = \{ \text{stmt, assign, expr, term, factor, \ldots } \} \]

\[ P : \]

\[ \text{stmt} \rightarrow \text{assign} \]
\[ \quad | \quad \text{expr} \]
\[ \text{assign} \rightarrow \text{REGISTER ASSIGN expr} \]
\[ \text{expr} \rightarrow \text{expr PLUS term} \]
\[ \quad | \quad \text{term} \]
\[ \text{term} \rightarrow \text{term MULT factor} \]
\[ \quad | \quad \text{factor} \]
\[ \text{factor} \rightarrow \text{REGISTER} \]
\[ \quad | \quad \text{FLOAT} \]
\[ \quad | \quad \text{OPENPAR expr CLOSEPAR} \]

\[ S = *\text{handwave}* \]

- For a single statement, \( S = \text{stmt} \)
- In practice, we need to handle sequences of statements and empty input lines (not reflected in the grammar)
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)
- How state was reached (right column)

Parsing:

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>R A O F P F M F C</td>
<td></td>
<td>Start</td>
</tr>
<tr>
<td>A O F P F M F C</td>
<td>R</td>
<td>Shift R to stack</td>
</tr>
<tr>
<td>O F P F M F C</td>
<td>R A</td>
<td>Shift A to stack</td>
</tr>
<tr>
<td>F P F M F C</td>
<td>R A O</td>
<td>Shift O to stack</td>
</tr>
<tr>
<td>P F M F C</td>
<td>R A O F</td>
<td>Shift F to stack</td>
</tr>
<tr>
<td>P F M F C</td>
<td>R A O factor</td>
<td>Reduce F</td>
</tr>
<tr>
<td>P F M F C</td>
<td>R A O term</td>
<td>Reduce factor</td>
</tr>
</tbody>
</table>

...
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 a minimal "scientific" calculator.

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

This code is released under the GNU General Public License
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}?

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

%%%
Lexer for desk calculator (3)

```c
{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;}

%%
```
Desk calculator has simple state
  100 floating point registers
  R0-R99

Represented in C as array of doubles:

```c
#define MAXREGS 100

double regfile[MAXREGS];
```

Needs to be initialized in support code!
Bison code for desk calculator: Header

{%
#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 `yy1val` 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
Bison code for desk calculator: Grammar (2)

```
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();
}
Reminder: Workflow and dependencies

Bison Input File
<file>.y

Definitions file
<file>.tab.h

Parser Source
<file>.tab.c

Flex Input file
<file>.l

Lexer Source
<file>.c

Lexer object
<file>.o

Parser object
<file>.tab.o

Final executable
parser
Some input
to process
Some output
produced

Bison

Flex

#include

gcc

linker (gcc)
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
Bison exercise


➤ **Download** `scicalcpparse.y` and `scicalclex.l`

➤ **Build the calculator**

➤ **Run and test the calculator**

➤ **Add a command** `clear` that clears (sets to 0) all registers, rebuild, retest!
Extend the desk calculator example as follows:

- Add support for division and subtraction `/`, `−`
- Add support for unary minus (the negation operator), using `~` as the negation sign
  - Bonus exercise: Use plain `-` as both unary and binary operators!
- Add support for the trigonometric function `sin(x), cos(x)`, where `x` can be any valid expression
- Add support for `log(b, x)`, computing `log_b(x)`

Hints:

- You may need to `#include<math.h>` and link with `-lm`
- `log_b(x) = \frac{\log_{10}(x)}{\log_{10}(b)}`. `man log10` should be helpful
Four classes of languages described by grammars and equivalent machine models:

1. regular languages \(\sim\) finite automata
2. context-free languages \(\sim\) pushdown automata
3. context-sensitive languages \(\sim\) ?
4. Type-0-languages \(\sim\) ?

We need a machine model that is more powerful than PDAs: Turing machines
Turing machine (0)

Proposed in 1936 by Alan Turing
- Model of a universal computer
- Paper: *On computable numbers, with an application to the Entscheidungsproblem*

Properties:
- Storage: unlimited tape (in both directions)
- Read/write head can move arbitrarily on unlimited storage
- Finite control unit (FA)
- No separation between input medium (holds $\Sigma$) and working medium ($\Gamma$)
- Transition relation only reads one character from the tape, but contains moving instructions ($l$, $n$, $r$)
Turing machine (1)

\[ M = (Q, \Sigma, \Gamma, \Delta, q_0, F) \]

- **\( Q \)** = \{ \( q_0, q_1, q_2, \ldots, q_n \) \} \quad \text{states}
- **\( \Sigma \)** = \{ \( a_0, a_1, a_2, \ldots, a_m \) \} \quad \text{l/O alphabet}
- **\( \Gamma \)** = \( \Sigma \cup \{ \varepsilon \} \) \quad \text{tape alphabet}
- **\( q_0 \)** \( \in \) \( Q \) \quad \text{initial state}
- **\( F \)** \( \subseteq \) \( Q \) \quad \text{final states}
- **\( \Delta \)** \( \subseteq \) \( Q \times \Gamma \times \Gamma \times \{l, n, r\} \times Q \) \quad \text{transition relation}

Turing machines are often deterministic:

\[ \Delta : (Q \times \Gamma) \rightarrow (\Gamma \times \{l, n, r\} \times Q) \]
The Turing machine (2)

Also written as $q_4, a_2 \rightarrow a_5, r, q_2$
A configuration $c$ 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 $\varepsilon$ sequences)
- the tape content $\beta$ starting with the position of the write head (except unlimited $\varepsilon$ sequences)
- written as $\alpha q \beta$, e.g. $a_1 a_5 q_2 a_3$
The computation of a TM $M$ on a word $w$ is a sequence of configurations (according to the transition function) of configurations, starting from $q_0 w$.

- $c = \alpha q \beta$ is accepting if $q \in F$.
- $c$ is a stop configuration if there are no transitions from $c$.
- A Turing machine accepts $w$ if the computation of $T$ on $w$ results in accepting stop configuration.
Consider $\Sigma = \{a, b\}$ and $L = \{w \in \Sigma^* \mid |w|_a \text{ is even}\}$

- Give a TM $M$ that accepts (only) words from $L$
- Give the computation of $M$ on the words $abbab$ and $bbab$
Example: TM for \(a^n b^n c^n\)

\[ M = \langle Q, \Sigma, \Gamma, \Delta, q_0, F \rangle \]

- \(Q = \{\text{start, findb, findc, check, back, end, f} \}\)
- \(\Sigma = \{a, b, c\}\) and \(\Gamma = \Sigma \cup \{\varepsilon, x, y, z\}\)
- \(\Delta\) per tables below
- \(q_0 = \text{start and } F = \{f\}\)

<table>
<thead>
<tr>
<th>state</th>
<th>read</th>
<th>write</th>
<th>move</th>
<th>state</th>
<th>read</th>
<th>write</th>
<th>move</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>(\varepsilon)</td>
<td>(\varepsilon)</td>
<td>n</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>start</td>
<td>a</td>
<td>x</td>
<td>r</td>
<td>findb</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>findb</td>
<td>a</td>
<td>a</td>
<td>r</td>
<td>findb</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>findb</td>
<td>y</td>
<td>y</td>
<td>r</td>
<td>findb</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>findb</td>
<td>b</td>
<td>y</td>
<td>r</td>
<td>findc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>findc</td>
<td>b</td>
<td>b</td>
<td>r</td>
<td>findc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>findc</td>
<td>z</td>
<td>z</td>
<td>r</td>
<td>findc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>findc</td>
<td>c</td>
<td>z</td>
<td>r</td>
<td>check</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>check</td>
<td>c</td>
<td>c</td>
<td>l</td>
<td>back</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>check</td>
<td>(\varepsilon)</td>
<td>(\varepsilon)</td>
<td>l</td>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state</th>
<th>read</th>
<th>write</th>
<th>move</th>
<th>state</th>
<th>read</th>
<th>write</th>
<th>move</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>back</td>
<td>z</td>
<td>z</td>
<td>l</td>
<td>back</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>back</td>
<td>b</td>
<td>b</td>
<td>l</td>
<td>back</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>back</td>
<td>y</td>
<td>y</td>
<td>l</td>
<td>back</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>back</td>
<td>a</td>
<td>a</td>
<td>l</td>
<td>back</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>back</td>
<td>x</td>
<td>x</td>
<td>r</td>
<td>start</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>z</td>
<td>z</td>
<td>l</td>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>y</td>
<td>y</td>
<td>l</td>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>x</td>
<td>x</td>
<td>l</td>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>(\varepsilon)</td>
<td>(\varepsilon)</td>
<td>n</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise: Turing machines (2)

a) Simulate the computations of $M$ on $aabbcc$ and $aabc$.

b) Develop a Turing machine accepting all words 
\[ \{ wcw \mid w \in \{a, b\}^* \} \].

c) How do you have to modify the TM from b) 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 $\Gamma^k \times \{X, \epsilon\}^k$
  - the first $k$ language elements encode the tape content, the remaining ones the positions of the heads.
Nondeterminism

- just like FA and PDA, 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.

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

- use a 3-tape TM:
  - tape 1 stores the input $w$
  - tape 2 records which non-deterministic choices are made (for all non-deterministic transitions)
  - tape 3 encodes the computation on $w$ with choices stored on tape 2.
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 $\alpha \rightarrow \beta$, 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 input word $w \in \Sigma^*$ with blanks left and right
2. simulate the computation of $\mathcal{A}$ on $w$
   
   $$(p, a, b, r, q) \rightsquigarrow pa \rightarrow bq$$
   $$(p, a, b, l, q) \rightsquigarrow cpa \rightarrow qcb (\forall c \in \Gamma)$$
   $$(p, a, b, n, q) \rightsquigarrow pa \rightarrow qb$$

3. recreate $w$
   - requires a more complicated alphabet
Closure properties

The class of languages described by Type-0-grammars or Turing machines is:

- closed under $\cup$, $\cdot$, $^*$
  - more complicated than for cf. grammars because context can influence rule applicability
  - rename NTSs (as for cf. grammars)
  - only allow NTSs as context
  - only productions of the kind $N_1 N_2 \ldots N_k \rightarrow M_1 M_2 \ldots M_j$ or $N \rightarrow a$

- closed under $\cap$
  - use a 2-tape-TM
  - simulate computation of $A_1$ on tape 1, $A_2$ on tape 2
  - accept if both $A_1$ and $A_2$ accept

- not closed under complement
Linear bounded automata and context-sensitive grammars

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

```
... > input < ...
```
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-shortening, words longer than $w$ need not be considered

Transformation of LBA $A$ into cs. grammar:

- similar to construction for TM:
  - generate $w$ without blanks
  - simulate operation of $A$ on $w$
    - rules are not shortening ✓
    - $PA \rightarrow BQ$ is not cs. . .
    - . . .but $PA \rightarrow XA \rightarrow XY \rightarrow BY \rightarrow BQ$ is cs. (and equivalent) ✓
The class of languages described by context-sensitive grammars / LBAs is:

- closed under $\cup, \cdot, *, \cap$
  - as for Type-0-grammars / TMs
- closed under complement
  - shown in 1988
  - many scientists believed opposite to be true
Context-sensitive grammars: decision problems

Word problem for cs. languages
The word problem for cs. languages is decidable.

▶ $\Gamma, \Sigma$ and $P$ are finite
▶ rules are not shortening
▶ for a word of length $n$ only a finite number of derivations up to length $n$ has to be considered.

Emptiness problem for cs. languages
The emptiness problem for cs. languages is undecidable.
Also follows from undecidability of Post’s correspondence problem.

Equivalence problem for cs. languages
The equivalence problem for cs. languages is undecidable.
If this problem was decidable for cs. languages, ist would also be decidable for cf. languages (since every cf. language is also cs.).
The universal Turing machine $U$

$U$ is a Turing machine emulator

Input:
- encoding $c(A)$ of a TM $A$
- word $w$

emulates computation of $A$ on $w$
- encodes current configuration of $A$
- stops if $A$ stops
- accepts if $A$ accepts $w$
- runs forever if $A$ runs forever with input $w$

Every solvable problem can be solved in software.
The halting problem

Does the TM $A$ halt with input $w$?

Wanted: TMs $H_1$ and $H_2$, such that with input $c(A)$ and $w$

1. $H_1$ accepts iff $A$ halts on $w$ and
2. $H_2$ accepts iff $A$ does not halt on $w$.

decision procedure for HP: let $H_1$ and $H_2$ run in parallel

1. $U$ (almost) does what $H_1$ needs to do. ✔

2. Difficult: $H_2$ needs to detect that that $A$ does not terminate. ❓

- infinite tape $\sim$ infinite number possible configurations
- recognising repeated configurations not sufficient.
Assumption: there is a TM $\mathcal{H}2$ which, given $c(\mathcal{A})$ and $w$ 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 $S$ accepting exactly those encodings of TMs that do not accept their own encoding

1. input: TM encoding $c(\mathcal{A})$
2. $S$ replaces $c(\mathcal{A})$ with $c(\mathcal{A})c(\mathcal{A})$
3. afterwards $S$ operates like $\mathcal{H}2$
Computation of $S$ with input $c(S)$

Reminder

$S$ accepts $c(A)$ iff $A$ does not accept $c(A)$.

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

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

This implies:

1. There is no such TM $S$.
2. There is no TM $H2$.
3. The halting problem is undecidable. (Turing 1936)
Decision problems for Turing machines
The word problem, the emptiness problem, and the equivalence problem are undecidable.

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

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

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

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

<table>
<thead>
<tr>
<th>TM</th>
<th>input</th>
<th>c(A)</th>
<th>c(B)</th>
<th>c(C)</th>
<th>c(D)</th>
<th>c(E)</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Further diagonalisation arguments

Cantor diagonalisation (1891)
The set of real numbers is uncountable.

Epimenides paradox (6th century BC)
Epimenides [the Cretan] says: “[All] Cretans are always liars.”

Russell’s paradox (1903)
\[ R := \{ T \mid T \notin T \} \] Does \( R \in R \)? hold?

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 (1939)

*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)

*What is the impact on practice?*
It does not only affect halting

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

Rice’s theorem (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

Example (Property \( E \): TM accepts the set of prime numbers \( P \))
If \( E \) is decidable, then so is the halting problem for \( A \) and an input \( w_A \).

Approach: Turing machine \( E \), input \( w_E \)

1. simulate computation of \( A \) auf \( w_A \)
2. decide if \( w_E \in P \)

Check if \( E \) accepts the set of prime numbers:

yes \( \rightarrow \) \( A \) halts with input \( w_A \) no \( \rightarrow \) \( A \) does not halt on input \( w_A \)
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
- Type-0 grammars,
- *while* programs,
- von Neumann architecture,
- Java/C++/Lisp/Prolog programs,
- future machines and languages

*No interesting property for any powerful programming language is decidable!*
<table>
<thead>
<tr>
<th>Field</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>software</td>
<td>Does the program match the specification?</td>
</tr>
<tr>
<td>development</td>
<td>Does the program have a memory leak?</td>
</tr>
<tr>
<td>debugging</td>
<td>Does the program harm the system?</td>
</tr>
<tr>
<td>malware</td>
<td>Does the student’s TM compute the same function as the teacher’s TM?</td>
</tr>
<tr>
<td>education</td>
<td>Do two cf. grammars generate the same language?</td>
</tr>
<tr>
<td>formal languages</td>
<td>Hilbert’s tenth problem: find integer solutions for a polynomial with several variables</td>
</tr>
<tr>
<td>languages</td>
<td>Satisfiability of formulas in first-order predicate logic</td>
</tr>
<tr>
<td>mathematics</td>
<td>Yes, it does matter!</td>
</tr>
<tr>
<td>logic</td>
<td>Yes, it does matter!</td>
</tr>
</tbody>
</table>
Many people with programming experience do not know this...
It is possible to translate a program $P$ from one language into an equivalent one in another language.

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 $U$. 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 $U$
- TM is not too 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 $A$ 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

<table>
<thead>
<tr>
<th>property</th>
<th>regular (Type 3)</th>
<th>context-free (Type 2)</th>
<th>context-sens. (Type 1)</th>
<th>unrestricted (Type 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>closure</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>( \cup, \cdot, * )</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>( \cap )</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>decidability</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>word</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>emptiness</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>equiv.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>deterministic</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>equivalent to</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
<td>✓</td>
</tr>
<tr>
<td>non-det.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Review of Goals

- Refresher
- Practical Parsing with YACC/Bison
  - Background and Principles
  - Workflow
  - Desk calculator example
- Turing Machines
  - Basics
  - A working example
  - Skimming over some topics of computability
Feedback round

► What was the best part of today’s lecture?
► What part of today’s lecture has the most potential for improvement?
  ▶ Optional: how would you improve it?
Goals for Today

▶ General remarks
▶ Refresher
▶ Open questions
▶ Examn exercises
Zur Klausur

- Abschlussklausur *formale Sprachen und Automaten*
  - Ort: Rotebühlplatz 41, Raum 0.10 (Erdgeschoss!)
  - Zeit: Dienstag, 25.11.2014, 11:00 (Pünktlich!)
  - Dauer: 120 Minuten
  - Hilfsmittel:
    - Skript (geheftet oder im Ordner)
    - Eigene Notizen (geheftet oder im Ordner)
    - Keine Loseblattsammlungen!
    - Keine (!) Computer (auch nicht, wenn Sie Tablett oder Mobiltelefon heißen)

- Zur Übungsklausur (heute)
  - Umfang etwas größer
  - Aufgaben tendenziell etwas schwerer
  - Echte Klausur wird *ähnlich, nicht identisch!*
    - Generell: Andere Aufgaben!
    - Z.t. andere Stoffgebiete
General remark
Refresher

- Bison
  - Grammar with actions executed on reduce
  - Workflow and integration with flex
  - Example: Desk calculator
  - Homework (extend desk calculator)

- Turing machines
  - Simple model of instruction-executing machine
  - Infinite tape with read/write head
  - Finite control
  - "Turing-complete" - can do every computation any known computing paradigm can do

- Examples and meta-results
Exercise: Use the known closure properties to show that context-free languages are not closed under complement.

We know:

- The class of context-free languages is closed under union (Combine grammars with $S_0 \rightarrow S_1$, $S_0 \rightarrow S_2$).
- The class of context-free languages is not closed under intersection ($a^n b^n c^m \cap a^m b^n c^n$ is not CF).

Now assume the class of CF languages were closed under complement. Let $L_1$, $L_2$ be arbitrary CF languages.

- Then: $\overline{L_1}$, $\overline{L_2}$ are CF.
- Then $\overline{L_1 \cup L_2}$ is CF.
- Then $\overline{L_1 \cup L_2} = \overline{L_1} \cap \overline{L_2}$ is CF.
- Hence the CF languages would be closed under intersection, which is wrong. Hence the assumption is wrong and CF languages are not closed under complement.
Übungsklausur
Goals for Today

- General remarks
- Refresher
- Open questions
- Examn exercises
This is the End…