Regular Expressions

TOC

Learning Target

- Understand fundamental concepts of regular expressions and possible applications
- Learn about languagues and their formalisms

STUDIUM PLUS DUALES STUDIUM
THM TECHNISCHE HOCHSCHULE MITTELHESSEN

Motivation

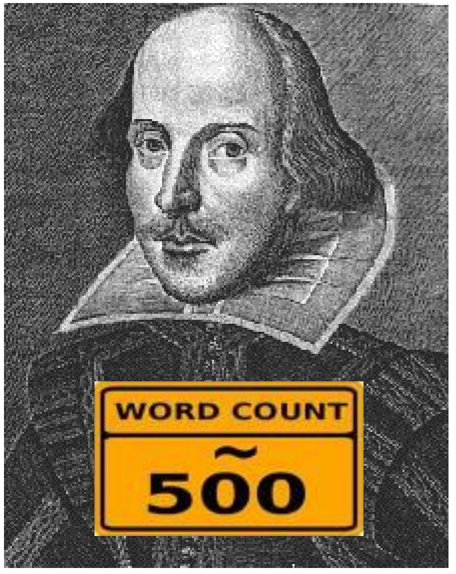
Opening Example: Shakespeare

The faculty for English literature runs a well funded project and you have the privilege to take part in it.

The project wants to measure William Shakespeare‘s language.
They claim that literature is just a matter of facts and figures.

Your job will be to provide the figures.

[http://shakespeare.mit.edu/]



Counting words

Count the words that begin with the letter **a** and end with the letter **e**
and divide that number by the total number of words in the text.

antique are alone

These words certainly have a common structure.
How could you write a Java program that detects all occurences of words
of this particular structure?

String Search – Pattern Matching

The task of scanning text for certain structures is called pattern matching.

Regular patterns follow rules that allow a very general treatment of such problems.

We will discuss techniques for textual pattern matching.

Handcrafted Solution

Pros and Cons?

```
private int countAE(String str) {
  int count=0;
  str=str.toLowerCase();
  String[] token=str.split(" ");
  for (String s : token)
  if ((s.charAt(0)=='a')&&(s.charAt(s.length()-1)=='e'))
      count++;
  return count;
}
```

Regular Expressions

What is a language?

Informally a language is a set of sentences that follow some common structure: they follow rules we call a grammar.

This structure is usually called syntax.

These concepts apply to natural languages as well as to computer languages.

We will ignore questions about semantics (meaning) in this discussion!

So, what actually is a language?

Let Σ be a set of symbols. A word over Σ is a finite sequence (string) of elements of Σ. Finally Σ^* is the set of all words over Σ.

A language L over Σ is a subset of Σ^* where all elements satisfy a set of specific rules.

A formal grammar $G=(N,\Sigma,S,P)$ consists of :

- a finite set N of **nonterminal** symbols
- a finite set Σ of **terminal** symbols ($N \cap \Sigma = \emptyset$)
- a **starting symbol** $s \in N$
- a finite set P of **production rules**

Sentence construction kit!

The production rules define how we can build words (i.e. sentences) in the language that is defined by G. We call it L(G).

Example (Synthesis)

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

$N=\{X,Y\}$

$\Sigma=\{1,2,3\}$

$S=X$

$P=\{X\to 1YX3,\ X\to 123,\ Y1\to 1Y, Y2 \to 22\}$

Synthesis (starting with X):

$X \xrightarrow[1]{} 1YX3 \xrightarrow[1]{} 1Y1YX33 \xrightarrow[2]{} 1Y1Y12333 \xrightarrow[3]{} 1Y11Y2333 \xrightarrow[3]{} 11Y1Y2333$

$\xrightarrow[3]{} 111YY2333 \xrightarrow[4]{} 111Y22333 \xrightarrow[4]{} 111222333$

$L(G)=L=\{1^n2^n3^n : n>0\}$

$\in L$

Example (Analysis)

1111222223333
↓
1111Y2223333
↓
111Y12223333
↓
111Y1Y223333
↓
111YYY123333
↓
111YYYX333
↓
11Y1YYX333
↓
11YY1YX333
↓
11YYX33
↓
1Y1YX33
↓
1YX3
↓
X

This method allows us to check whether a string belongs to L or not.

Whenever we can reduce a string to our starting symbol by applying our production rules in reverse the answer is yes.

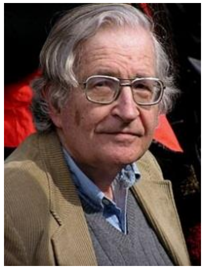
This process of parsing is basically what a compiler does.

Tools like ANTLR help you define your own languages and generate parsers.

Chomsky's Hierarchy

During his search for a universal grammar Chomksy classified languages according to their production rules.

The most general are type-0 languages more or less without any restrictions at all.
Most programming languages are context-free.
The most restrictive grammars are of type-3.



Noam Chomsky, *1928

Type	Languages	Rules
0	recursively enumerable	$\alpha \to \beta \quad (\alpha \in (N \cup \Sigma)^+)$
1	context-sensitive	$\alpha A \beta \to \alpha \gamma \beta$
2	context-free	$A \to \gamma$
3	Regular	$A \to a$ $A \to aB$

where: $\alpha, \beta, \gamma \in (N \cup \Sigma)^*, \quad a \in \Sigma \quad A, B \in N$

Regular Languages

According to Chomsky's hierarchy the language L of our example is context-sensitive (type-1).

Regular languages are simpler because their production rules stay fairly simple (somehow linear).

But they are the perfect tool for pattern matching anyway!

Try this grammar! Is it of type-3?

$G=\{\{S, E\},\{a,\ldots,z\},S,P\}$

$P=\{S\rightarrow aE, E\rightarrow aE, E\rightarrow bE,\ldots,E\rightarrow zE, E\rightarrow e\}$

Regular Expressions

Regular expressions can be formally defined. We will not do that now but come back to that concept later.
Then we will give a rather practical definition and start working with them right away.

However we just cite the following theorem now:

Regular languages and regular expressions are equivalent concepts.

[without proof]

Finite State Machines

Finite State Machines

A finite state machine (FSM) is a formal instrument that consists of a finite set of states and rules that define how the machine changes its state according to a given input.

A DFA (deterministic finite automaton) M is a tuple $M=(Q,\Sigma,\delta,q_0,F)$ with

- a finite set of **states** Q
- a finite set of **input** symbols Σ
- a **state transition** function $\delta : Q\times\Sigma\to Q$
- a **starting symbol** $q_0\in Q$
- a set of **final** (accepting) states $F\subseteq Q$

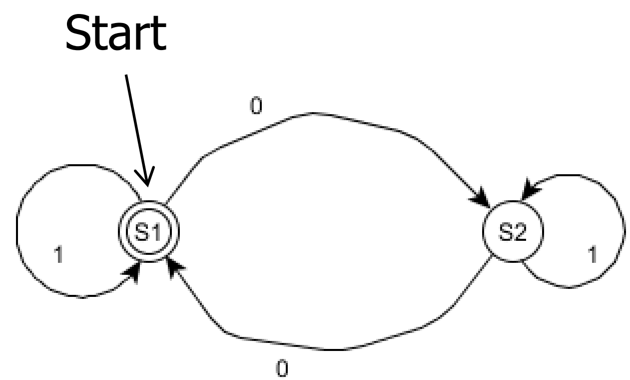
I know that you have seen them before, anyway….

A sequence of inputs that reaches an element in F is said to be accepted by M. The set of all such sequences is called the language accepted by M. In short: L(M).

Example

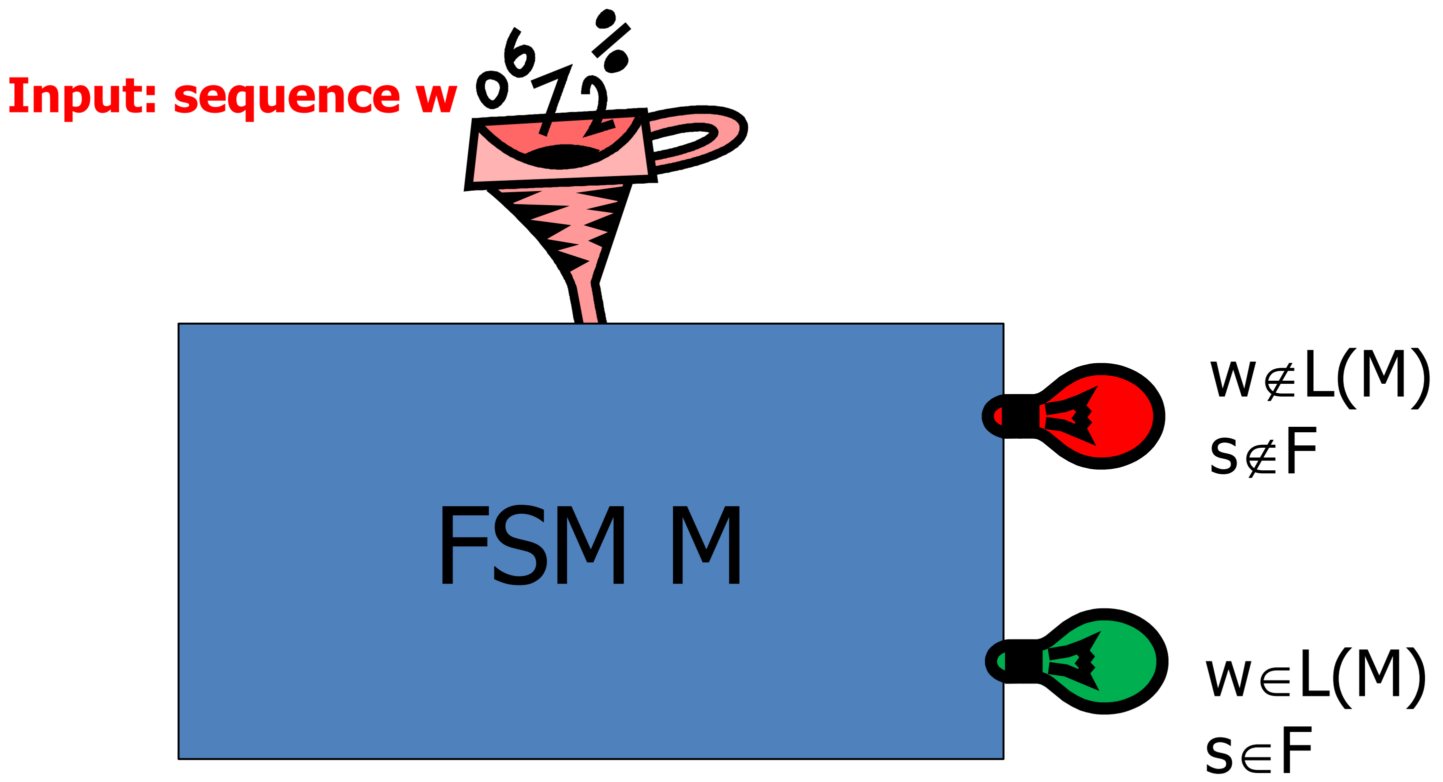
$Q=\{S_1,S_2\}$
$\Sigma=\{0,1\}$
$q_0=S_1$
$F=\{S_1\}$

$\delta(S_1,0)=S_2$
$\delta(S_2,0)=S_1$
$\delta(S_1,1)=S_1$
$\delta(S_2,1)=S_2$



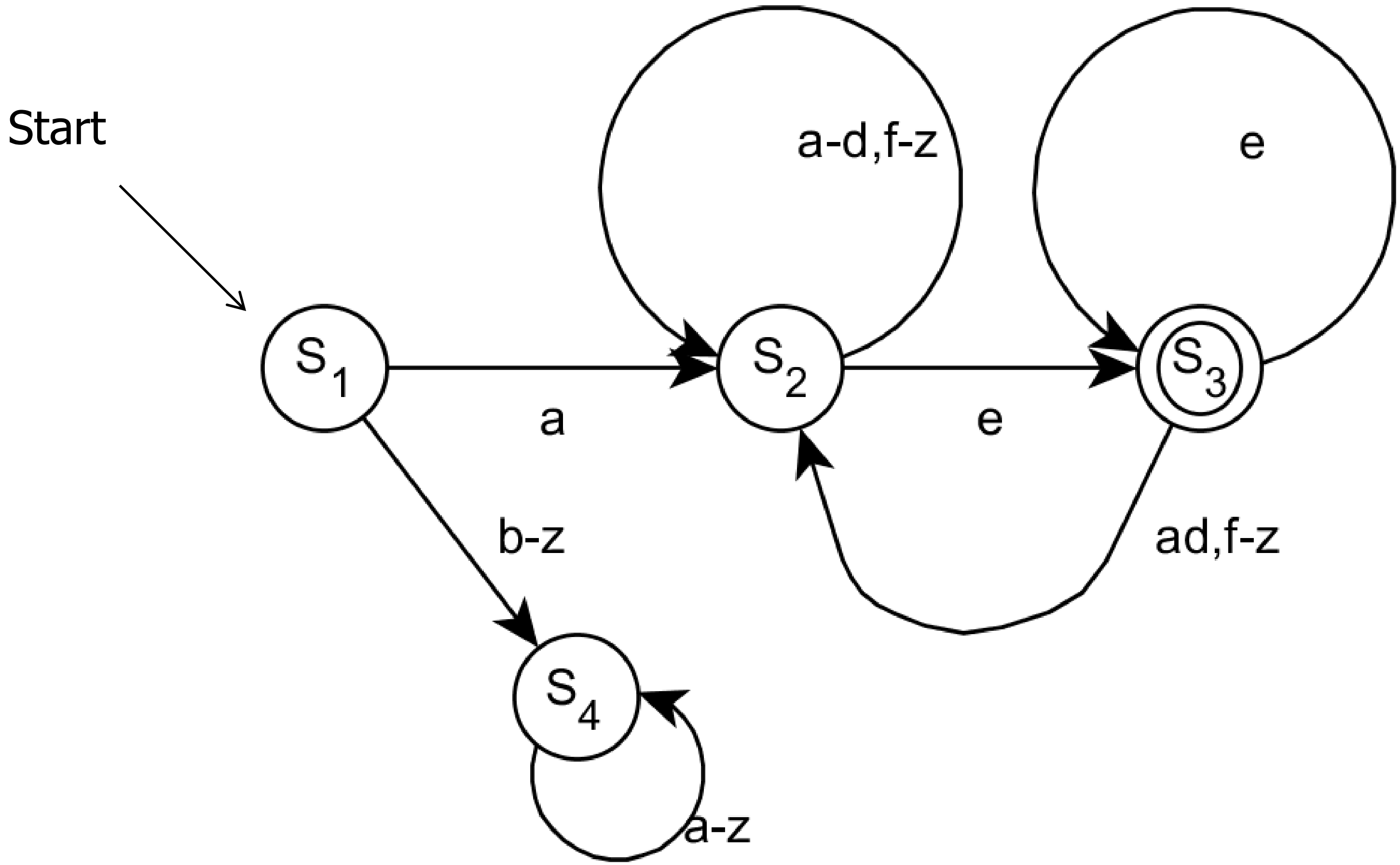
Whenever we are in S1 the input sequence so far is accepted.

[http://en.wikipedia.org/wiki/Deterministic_finite-state_machine]



Input: sequence w
0672%
FSM M
w∉L(M)
s∉F
w∈L(M)
s∈F

FSM recognizing languages



Theorem

The set of all languages accepted by DFAs is equivalent to the concept of regular languages.

[without proof]

We will continue with that and look at other types of languages and other automaton soon!

Practical Regular Expressions

Regular Expressions

UNIX tools like grep, sed and awk make extensive use of regular expressions.

Together with the piping mechanism of the UNIX operating systems these commands are powerful tools for editing datasets and scripting.

There is a convenient way for the formulation of regular expressions we can use in practical applications.

Capabilities of Java: http://download.oracle.com/javase/6/docs/api/

Regular expressions

Regular expressions are a very powerful pattern matching device.

Search patterns are defined via search strings that contain certain special characters that let you define the pattern.

Datasets with extension .txt:

```
.*\.txt
```

Email addresses:

```
\b[a-zA-Z0-9._%-]+@[a-zA-Z0-9._%-]+\.[a-zA-Z]{2,4}\b
```

regular_expression and regular_expressions, regex, regexp, regexes:

```
reg(ular_expressions?|ex(p|es)?)
```

[http://www.regular-expressions.info/tutorial.html]

Simple patterns

The most simple patterns are just single symbols or sequences of symbols (words).

<u>Pattern:</u>

`a`

<u>Example:</u>

`Jack is a boy`

<u>Pattern:</u>

`witch`

<u>Example:</u>

`Schneewitchen`

Metacharacters

The symbols **[ ] . * ( ) ? ^ \ + |** are metacharacters. They all have a special meaning.
To be used in a plain pattern they must be „switched off‟ with a leading ****.

Pattern:

```
1\+1
```

Example:

```
1+1=2
```

Pattern :

```
3\*4
```

Example :

```
3*4=12
```

Escaping

The symbol \ switches of metacharacters and transforms them into ordinary symbols.
But some ordinary symbols become meta symbols!

Examples:

\b	word boundary
\w	word character
\s	whitespace character (blank)
\d	0,1,…,9 (decimal character)
\n	line feed
\u20A0	Unicode for €

Pattern:

```
\bam\b
```

Example:

```
am Damm
```

Pattern:

```
3\d4
```

Example:

```
354
```

Character Classes

STUDIUM PLUS DUALES STUDIUM | THM TECHNISCHE HOCHSCHULE MITTELHESSEN

Character classes let you define patterns where a symbol must match one out of a set of symbols.

Examples:

gr[ae]y	matches grey and gray
A[0-9]B	matches A0B,…,A9B
\d	an element of {0,1,…,9} (predefined, there are more)
1[+*]1	metacharacters within brackets are treated as ordinary symbols
.	placeholder for any sign

Pattern:

`AD[AF]C`

Example:

`ADFC`

Pattern:

`AD[AF]C`

Example:

`ADAC`

Pattern:

`AD[+*]C`

Example:

`AD*C`

Repetition

The following metacharacters let you define how often a pattern must/may occur.

?	pattern is optional
+	1 to n
*****	0 to n
{min,max}	*min* to *max*

Pattern:

```
<[a-zA-Z][a-zA-Z0-9]*>
```

Example:

<H1><H99><zA555>

Pattern:

```
35?4
```

Example:

354 34

Greedy vs Lazy

The metacharacters **+** and ***** are treated greedily, i.e. the matched sequence is as large as possible. An additional **?** makes them lazy, so that matches occur on smaller parts.
Matches start as far left as possible. Greedy matches take as much as they can get.

`*?`	lazy star
`+?`	lazy plus

Pattern:

`a.*e oder a.+e`	**`a.*?e`**	**`a.+?e`**

Example:

`are alone ape ae`

Greedy: 1 match	Lazy: 4 matches	Lazy: 3 matches

Alternatives and Negation

STUDIUM PLUS DUALES STUDIUM | THM TECHNISCHE HOCHSCHULE MITTELHESSEN

`|`	separates alternatives
`^`	when used in brackets: the complement of the defined character class

Pattern:

```
\b(cat|dog)\b
```

Example:

```
cat catfish bird
```

Pattern:

```
a[^0-9]b
```

Example:

```
aab    a3b
```

Grouping

Simple brackets let you define groups that are again treated as patterns.

fall(s|acy)?	matches *fall*, *falls* and *fallacy*
(abc){1,2}	matches *abc* and *abcabc*

Pattern:

`M(ue|ü)he`

Example:

`Mühe Muehe`

Pattern:

`O(m|p){2}a`

Example:

`Omma Oppa Opa`

Anchors

Anchors define specific positions:

^	beginning of string/line
$	end of string/line

Pattern:

```
^\d+$
```

Example:

```
123    a234dff
```

Backreferences

Backrefences let you refer to patterns already matched in the current process. \n refers to the n-th match.

\1 refers to the first match

\w=word character

Pattern:

```
a(\w)\1\1b
```

Example:

```
acb axxxb a8b
```

Pattern:

```
123(4|5)\1(321)
```

Example:

```
12344321 12345321 12355321 12354321
```

More Examples

Example

(bc|BC)?d

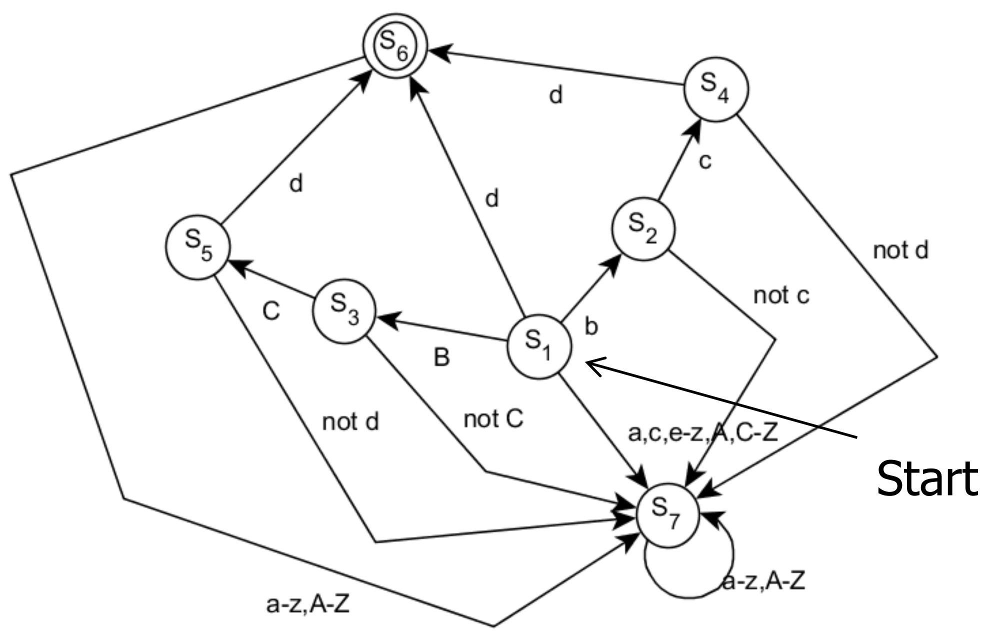
d

BCd

bcd

~~bcbcd~~

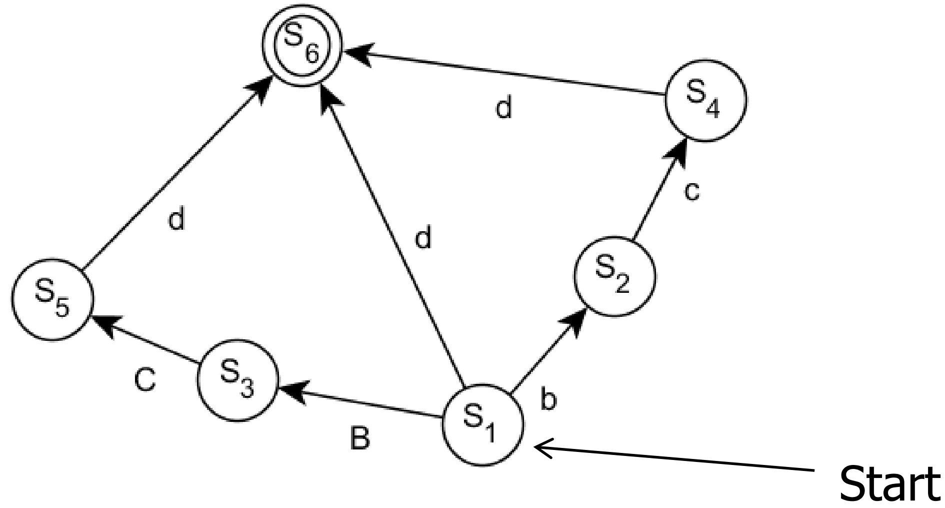
~~BCBCd~~



Example

State S_7 is a fail or error state. It is common practice not to draw that state and implicitly interpret every transition not in the picture as going to that now invisible fail state.

Our automaton then reduces to:

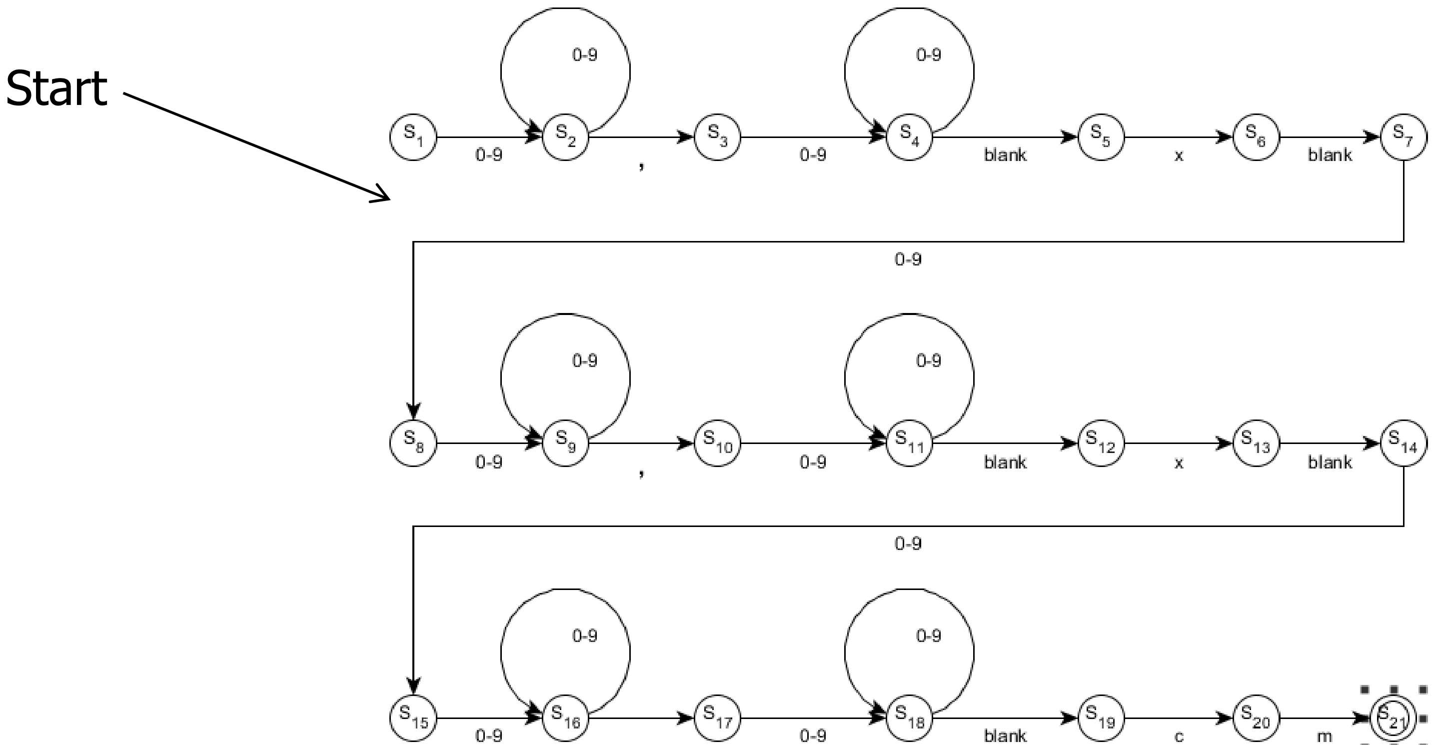


Example

What about: decimal point is comma?

Match string like: 24,4 x 16,4 x 1,8 cm

`[0-9]+,[0-9]+ x [0-9]+,[0-9]+ x [0-9]+,[0-9]+ cm`



A bit more complicated

1 2 3 4 5 6 7 8 9 10

\b[a-zA-Z0-9._%-]+@[a-zA-Z0-9._%-]+\.[a-zA-Z]{2,4}\b

1 **\b wordboundary marks the start**

2 **[a-zA-Z0-9._%-] set of possible symbols**

3 **+ at least one possibly more out of set 2**

4 **@ an at**

5 **[a-zA-Z0-9._%-] set of possible symbols**

6 **+ at least one possibly more out of set 5**

7 **\. a dot**

8 **[a-zA-Z] set of possible symbols**

9 **{2,4} between 2 and 4 out of set 8**

10 **\b wordboundary marks the end**

Regular Expressions and FSM in Java

Java and Regular Expressions

The packages *java.util.regex* contains all you need to use regular expressions in your Java programs. *Pattern* defines search patterns while The *Matcher* object does the active parts of the work.

The symbols \ und ^ must be duplicated when used as command line parameters.

Pattern Matching

STUDIUM PLUS DUALES STUDIUM

THM TECHNISCHE HOCHSCHULE MITTELHESSEN

```
import java.util.regex.*;
public class RegexTest {
       public  static  void main(String[] args) {
               Pattern p = Pattern.compile("a*e");
               String seq="aaaab";
               Matcher m = p.matcher(seq);
               boolean b = m.matches();
               System.out.println(b);
       }
}
```

Iteration over Matches

```
import java.util.regex.*;

...

  public  int countAE(String pattern,String str) {
      int count=0;
      Pattern p = Pattern.compile(pattern);
      Matcher m = p.matcher(str);
      m.reset();
      while (m.find())
         count++;
      return count;
  }
```

Invoke like: `countAE("a\\w*e",some_string);`

Real World Application

```
import java.util.regex.*;

public class CheckMailSyntax {

 public  boolean check(String arg) {
  Pattern p = Pattern.compile("\\b[A-Z0-9._%-]+@[A-Z0-9._%-
]+\\.[A-Z]{2,4}\\b",Pattern.CASE_INSENSITIVE);
  Matcher m = p.matcher(arg);
  if (m.matches())
      return true;
  else
      return false;
  }
}
```

Java requires \\ for a single \ in your expression!

FSM in Java

```
public class FiniteStateMachine {

        public enum States {START,S1,END,FAIL};

        public static boolean sm(String s) {

                States state = States.START;

                int i=0;

                while (i<s.length() &&
                        state!=States.FAIL) {

                  char c = s.charAt(i++);
```

FSM matches the whole string to the pattern.
If you want to accept partial matches add:

```
&& state!=States.END
```

Cont'd

```
            switch (state){
            case START:
              if (c=='a')
                 state=States.S1;
             else
                 state=States.FAIL;
              break;
            case S1:
              ...
              break;
              ...
            }
          }   //end of while
        return (state==States.END);
    } // end of method
}
```

Alternatively you can move to START
if you want to accept partial matches.
FAIL stops the search.
Resetting to START continues the search
until the end of the string is referenced by s.

Literature and Tools

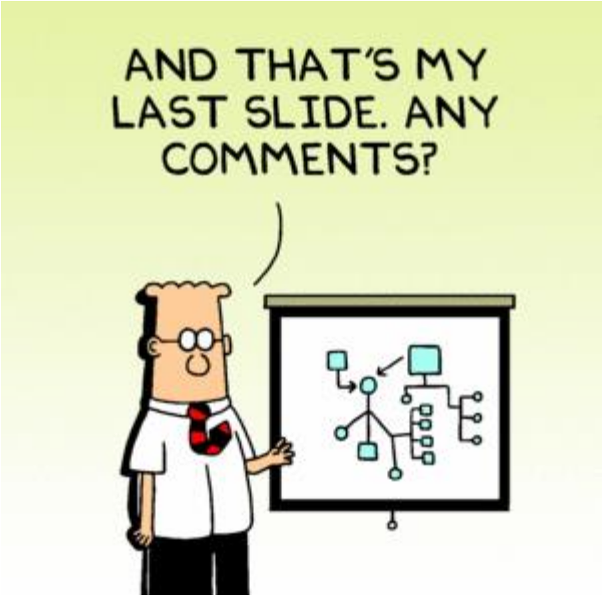
Hopcroft, Motwani, Ullmann:
Introduction to Automata Theory, Languages, and Computation
Second Edition, Amsterdam, 2001

Friedl:
Mastering Regular Expressions
Second Edition, Sebastopol 2002

Regexp-Tool:
http://erik.eae.net/playground/regexp/regexp.html
http://myregexp.com/
Look for: The RegexCoach

Goal achieved?

- Understand fundamental concepts of regular expressions and possible applications
- Learn about languagues and their formalisms


AND THAT'S MY LAST SLIDE. ANY COMMENTS?