Aalto University
School of Science

CS-C2160 Theory of Computation

Lecture 4: Regular Expressions

Pekka Orponen
Aalto University
Department of Computer Science

Spring 2021

Topics

- Syntax and semantics of regular expressions
- Regular expressions and finite automata
- * Excursion: Regular expressions in programming languages

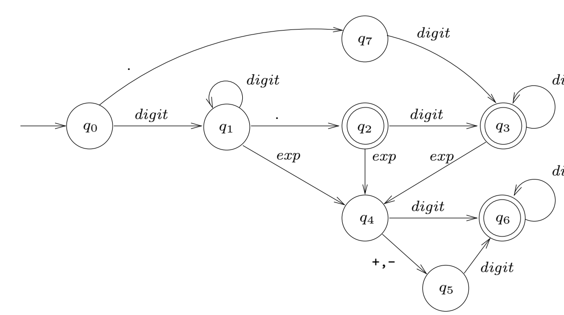
Material:

- in Finnish: Sections 2.6–2.7 in Finnish lecture notes
- in English: Section 1.3 in the Sipser book

Regular Expressions

Finite automata vs. regular expressions

$M :=$



$\downarrow$ recognises $\mathcal{L}(M)$

$$\{.256, 1., 3.14, 2.3E{-}10, \ldots\}$$

$\uparrow$ describes $\mathcal{L}(r)$

$$r := (dd^*.d^* \cup .dd^*)(e(+ \cup - \cup \varepsilon)dd^* \cup \varepsilon) \cup (dd^*e(+ \cup - \cup \varepsilon)dd^*)$$

Applications

Regular expressions (and their extensions) are used in many places:

- searching and modifying text files
- lexical analysis in compilers (recognising keywords etc.)
- property specification languages such as IEEE PSL
- etc.

Example: Hiding student numbers with Python

```
from sys import stdin, stdout
import re
idPattern = re.compile(r'(\d{5}[A-Z])|(\d{6})|(K\d{5})')
for line in stdin:
    stdout.write(idPattern.sub('xxxxxx', line))
```

Input: $\longrightarrow$ Output:

```
Oiva :Opiskelija:12345X:5:2:4:9        Oiva :Opiskelija:xxxxxx:5:2:4:9
F.I. :Nance     :K12345:5:9:4:2        F.I. :Nance     :xxxxxx:5:9:4:2
Raimo:Raketti   :123456:7:7:7:9        Raimo:Raketti   :xxxxxx:7:7:7:9
```


4.1 Syntax and semantics of regular expressions

We first define some elementary operations on languages.
Let A and B be languages over an alphabet Σ.

- *Union:* $A \cup B = \{x \in \Sigma^* \mid x \in A$ or $x \in B\}$
- *Concatenation:* $AB = \{xy \in \Sigma^* \mid x \in A$ and $y \in B\}$
- *Powers:*

$$\begin{cases} A^0 &= \{\varepsilon\}, \\ A^k &= AA^{k-1} = \{x_1 \dots x_k \mid x_i \in A \quad \forall i = 1, \dots, k\}, \quad \text{for } k \geq 1 \end{cases}$$

- *Kleene closure (or "Kleene star"):*

$$\begin{aligned} A^* &= \bigcup_{k \geq 0} A^k \\ &= \{x_1 \dots x_k \mid k \geq 0,\ x_i \in A \quad \forall i = 1, \dots, k\} \end{aligned}$$

Definition 4.1 (Syntax of regular expressions)

Regular expressions over an alphabet Σ are defined inductively by the following rules:

1. $\emptyset$ and ε are regular expressions over Σ.
2. a is a regular expression over Σ when $a \in \Sigma$.
3. If r and s are regular expressions over Σ, then also $(r \cup s)$, (rs), and r^* are regular expressions over Σ.
4. There are no other regular expressions over Σ.

Note

All the rules are purely syntactic. Thus, for instance '$\emptyset$' and '$\cup$' are here just symbols, without any meaning (yet).

The first two rules are the "base cases" while the third one is the inductive (recursive) case.

The last case is usually implicitly assumed and thus omitted.

Definition 4.2 (Semantics of regular expressions)

A regular expression r over Σ *describes* the language $\mathcal{L}(r)$ defined inductively as follows:

- $\mathcal{L}(\emptyset) = \emptyset$
- $\mathcal{L}(\varepsilon) = \{\varepsilon\}$
- $\mathcal{L}(a) = \{a\}$ when $a \in \Sigma$
- $\mathcal{L}((r \cup s)) = \mathcal{L}(r) \cup \mathcal{L}(s)$
- $\mathcal{L}((rs)) = \mathcal{L}(r)\mathcal{L}(s)$
- $\mathcal{L}(r^*) = (\mathcal{L}(r))^*$

Example

Some regular expressions over the alphabet $\{a,b\}$:

$$r_1 = ((ab)b), \quad r_2 = (ab)^*, \quad r_3 = (ab^*), \quad r_4 = (a(b \cup (bb)))^*$$

The languages described by the expressions are:

$$\begin{aligned}
\mathcal{L}(r_1) &= (\{a\}\{b\})\{b\} = \{ab\}\{b\} = \{abb\} \\
\mathcal{L}(r_2) &= \{ab\}^* = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^i \mid i \geq 0\} \\
\mathcal{L}(r_3) &= \{a\}(\{b\})^* = \{a, ab, abb, abbb, \ldots\} = \{ab^i \mid i \geq 0\} \\
\mathcal{L}(r_4) &= (\{a\}\{b, bb\})^* = \{ab, abb\}^* \\
&= \{\varepsilon, ab, abb, abab, ababb, \ldots\} \\
&= \{x \in \{a,b\}^* \mid \text{if } x \neq \varepsilon \text{ then it begins with an } a \text{ and} \\
&\qquad \text{each } a \text{ in } x \text{ is followed by 1 or 2 } b\text{s.}\}
\end{aligned}$$

Reducing the number of parentheses:

- Precedence between operators:

$$* \quad \succ \quad \cdot \quad \succ \quad \cup$$

  Thus, instead of $(a(b \cup (bb)))^*$, we can write $(a(b \cup bb))^*$. But $(ab \cup bb)^*$ would correspond to the different expression $((ab) \cup (bb))^*$.

- Associativity of union and concatenation operators:

$$\begin{aligned}
\mathcal{L}(((r \cup s) \cup t)) &= \mathcal{L}((r \cup (s \cup t))) \\
\mathcal{L}(((rs)t)) &= \mathcal{L}((r(st)))
\end{aligned}$$

  $\Rightarrow$ no parentheses needed for consecutive unions/concatenations

Example

The expressions of the previous Example in a simpler form:

$$r_1 = abb, \quad r_2 = (ab)^*, \quad r_3 = ab^*, \quad r_4 = (a(b \cup bb))^*.$$

Example:

Unsigned floating point numerals in C:

$$\mathit{number} = (dd^*.d^* \cup .dd^*)(e(+ \cup - \cup \varepsilon)dd^* \cup \varepsilon) \cup (dd^*e(+ \cup - \cup \varepsilon)dd^*),$$

where d is an abbreviation for

$$d = (0 \cup 1 \cup 2 \cup 3 \cup 4 \cup 5 \cup 6 \cup 7 \cup 8 \cup 9)$$

and e abbreviates

$$e = (\mathtt{E} \cup \mathtt{e}).$$

One often also uses r^+ as an abbreviation for rr^*.

Example:

$$(d^+.d^* \cup .d^+)(e(+ \cup - \cup \varepsilon)d^+ \cup \varepsilon) \cup (d^+e(+ \cup - \cup \varepsilon)d^+)$$

Definition 4.3 (Regular languages)

A language is *regular* if it can be described with a regular expression.

Simplification rules for regular expressions

- A regular language can usually be described with many different regular expressions, e.g.,

$$\begin{aligned}\Sigma^* &= \mathcal{L}((a \cup b)^*)\\ &= \mathcal{L}((a^*b^*)^*)\end{aligned}$$

- Two regular expressions, r and s, are *equivalent*, denoted by $r = s$, if $\mathcal{L}(r) = \mathcal{L}(s)$.
- Simplification of an expression $\approx$ finding the "simplest" equivalent expression.
- Testing whether two regular expressions are equivalent is a nontrivial (but mechanically solvable) problem.

Some simplification rules:

$$\begin{aligned}r \cup (s \cup t) &= (r \cup s) \cup t\\ r(st) &= (rs)t\\ r \cup s &= s \cup r\\ r(s \cup t) &= rs \cup rt\\ (r \cup s)t &= rt \cup st\\ r \cup r &= r\end{aligned}
\qquad
\begin{aligned}r \cup \emptyset &= r\\ \varepsilon r &= r\\ \emptyset r &= \emptyset\\ r^* &= \varepsilon \cup r^* r\\ r^* &= (\varepsilon \cup r)^*\end{aligned}$$

In fact, any valid equivalence between regular expressions can be derived from these equations and the rule:

if $\varepsilon \notin \mathcal{L}(s)$ and $r = rs \cup t$, then $r = ts^*$

[A. Salomaa 1966]

If one wants to verify that two regular expressions are equivalent, it is usually simplest to show that the languages described by them are included in each other:

Let us write $r \subseteq s$ for $\mathcal{L}(r) \subseteq \mathcal{L}(s)$.
Now $r = s$ if and only if $r \subseteq s$ and $s \subseteq r$.

Example:

Let us verify that $(a^*b^*)^* = (a \cup b)^*$.

1. Clearly $(a^*b^*)^* \subseteq (a \cup b)^*$, because $(a \cup b)^*$ describes *all* the strings over $\{a, b\}$.
2. As $(a \cup b) \subseteq a^*b^*$, then $(a \cup b)^* \subseteq (a^*b^*)^*$ holds as well.

4.2 Regular expressions and finite automata

Theorem 4.1

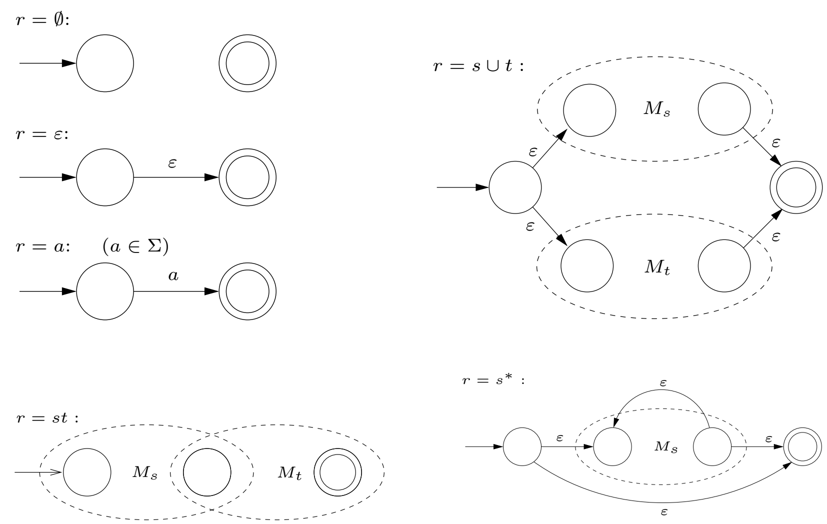
If a language can be described with a regular expression, then it can be recognised by a finite automaton.

Proof

By using the inductive construction on the next slide, we can design, for each regular expression r, a nondeterministic automaton M_r with ε-transitions such that $\mathcal{L}(M_r) = \mathcal{L}(r)$. The resulting automaton can then be determinised if needed (cf. Lecture 3). In the construction, the intermediate component automata always have a standard form with unique and distinct initial and final states, and no transitions either

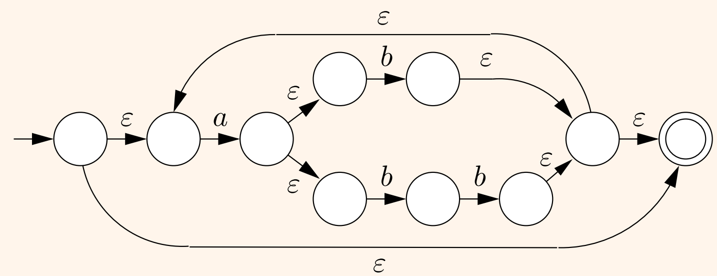
1. entering the initial state, or
2. exiting the final state.

This is important when putting the component automata together. $\square$

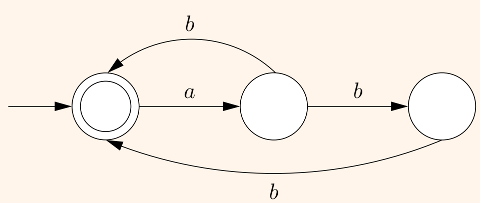


Example:

By applying the construction of Lemma 4.2 to the expression $r = (a(b \cup bb))^*$, we get the following nondeterministic automaton:



The automaton on the previous slide contains quite a lot of redundancy. If one masters the semantics of regular expressions, it is sometimes easier to design corresponding automata directly.
For instance, for the expression $r = (a(b \cup bb))^*$ it is quite straightforward to design a simple nondeterministic automaton:



The same automaton can be obtained by removing the ε-transitions from the systematically constructed automaton on the previous slide.
If desired, this automaton can then be further determinised and minimised using the methods from Lecture 3.

Theorem 4.2

If a language can be recognised by a finite automaton, then it can be described with a regular expression.

Proof

We need one more extension of finite automata, the *generalised nondeterministic finite automata* (abbreviated GNFA), which allow transitions that are labelled with regular expressions.
Formalisation: Let RE_Σ be the set of regular expressions over Σ. A GNFA is a tuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where the transition function δ is a *finite* mapping

$$\delta : Q \times \mathrm{RE}_\Sigma \to \mathcal{P}(Q)$$

(that is, $\delta(q,r) \neq \emptyset$ holds only for finitely many pairs $(q,r) \in Q \times \mathrm{RE}_\Sigma$).
Note: This definition is different from that in Sipser's book (Definition 1.64) but serves the same purpose.

The "leads directly", or "leads in one step" relation is now defined

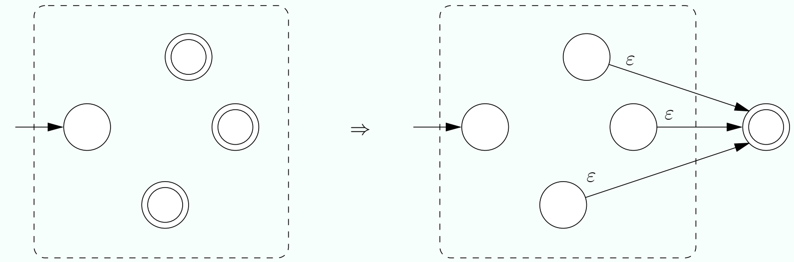
$$(q,w) \underset{M}{\vdash} (q',w')$$

if $q' \in \delta(q,r)$ for some $r \in \mathrm{RE}_\Sigma$ such that $w = zw'$ and $z \in \mathcal{L}(r)$. Other definitions are as earlier.

Let us now prove: every language recognised by a GNFA can be described with a regular expression.

Let M be a GNFA. A regular expression that describes the language recognised by M can be constructed in three phases:

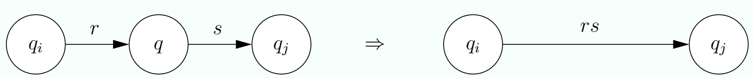
Phase I: If M has multiple final states, merge them as follows:



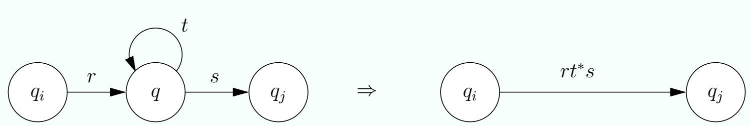
Phase II: Reduce M to an equivalent GNFA with at most two states, by removing all non-initial and non-final states one by one, using the following transformations:

Let q be any state in M that is not the initial or the final state. Consider all the "transition paths" in the state diagram of M that go through q. Let q_i and q_j be the immediate predecessor and successor states of q on some such path. Remove q from the path $q_i \to q_j$ by rule (i) below if q has no transition to itself, and by rule (ii) if it has:

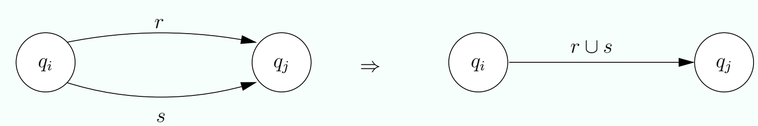
(i):



(ii):

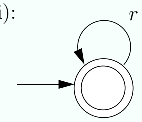


At the same time, merge parallel transitions with the rule:



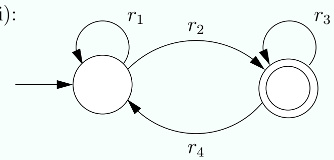
Phase III: At the end of the reduction process in Phase II, the automaton has at most two states. The corresponding regular expression is then constructed as follows:

(i):



$\Rightarrow \quad r^*$

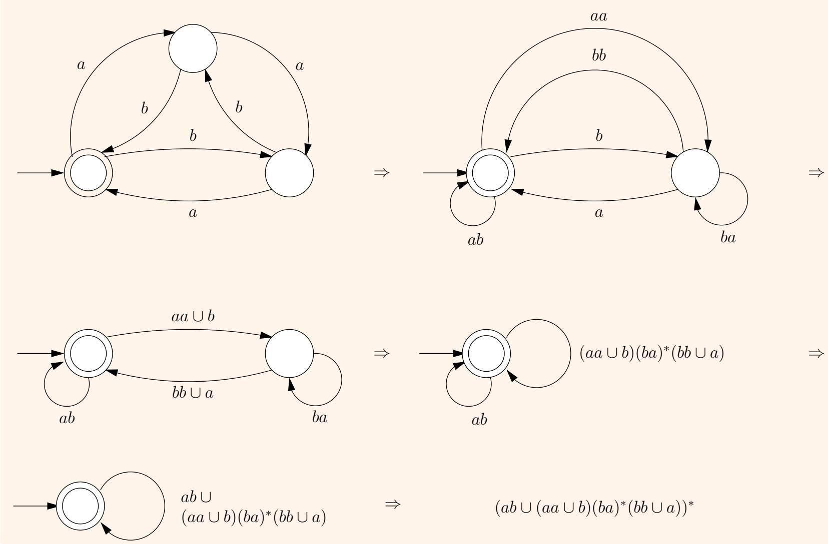
(ii):



$\Rightarrow \quad r_1^* r_2 (r_3 \cup r_4 r_1^* r_2)^*$

□

Example:



* Excursion: Regular Expressions in Programming Languages

* Regular expressions in programming languages

- Manipulating text strings is important in many applications (validating input in web forms, finding patterns in text and replacing them with others in text editors, and so on).
- Thus most (all recent?) programming languages include support for regular expressions (or for their extensions):
  - `re` library in Python (a starting point: HOWTO)
  - regex library in the latest C++ version
  - `scala.util.matching.Regex` class in Scala
  - `java.util.regex` package in Java
  - JavaScript (aka ECMAScript), see e.g. W3Schools
  - and many others!
- Also "sequential expressions" (SEREs) in the IEEE Property Specification Language PSL (IEEE standard 1850).
- A book on the topic: Jeffrey Friedl, Mastering Regular Expressions

Example: Reading "Nodes:" lines in Perl

A snippet of a perl program:

```
sub getNodes {
 my($f) = $_[0];
 open(FILE, $f);
 while($line = <FILE>) {
  if($line =~ /^Nodes:\W*([0-9]+)/){
   $nodes = $1;
  }
 }
 close(FILE);
 $nodes;
}
```

a data file:

```
Canrep updates:  1
Generators:      2
Max level:       3
|Aut|:           8
Nodes:           109294
Leaf nodes:      3
Bad nodes:       107682
Total time:      2.60 seconds
```

Here s =~$/r/$ tries to match a string s to regular expression r, symbol ^ matches the beginning of a string, `\W` means "whitespace", `[0-9]` denotes any symbol in the set $\{0,1,...,9\}$, and the substring that matches the expression inside the parentheses is assigned to variable `$1`.

* Extensions to regular expressions: Case Python

Many languages support various extensions to "pure" regular expressions. As an example, let us consider the `re` library of Python.

- Fixed powers, such as $(ab)^5$, can be expressed with the repetition operator `{m,n}`, where `m` is the minimum and `n` the maximum number of repetitions. For instance, `(a|b){20,200}` describes (matches to) a string with 20–200 `a` and `b` symbols. This could also be expressed with a (quite large) "pure" regular expression.
- By default, matching is *greedy*, meaning that only the longest matching substrings are considered:

```
>>> import re
>>> for m in re.finditer(r'(a|b)+', 'abbacbaabaa'): m.span()
...
(0, 4)
(5, 11)
```


Example: A Scala program for finding Finnish social security numbers

```
val hetu = """(?:^|\D)(\d\d)(\d\d)(\d\d)(\+|-|A)(\d{3})([0-9A-Y])(?:$
    |[^0-9A-Y])""".r
val centuryMap = Map("+"->1800, "-"->1900, "A"->2000)
val checksumToLetter = "0123456789ABCDEFHKLMNPQRSTUVWXY".zipWithIndex.
    map(_.swap).toMap
for(line <- io.Source.stdin.getLines) {
  hetu.findAllMatchIn(line).foreach {
    case hetu(day,month,year,century,id,check) => {
      val enBirthDate = s"$month/$day/${centuryMap(century)+year.toInt
          }"
      val checksum = (day+month+year+id).toLong % 31
      val isValid = checksumToLetter(checksum.toInt) == check(0)
      println(s"""A${if(!isValid) "n in" else " "}valid hetu for
          someone born on $enBirthDate""")
    }
  }
}
```

☞ Scala Regex and Java Pattern

- With some extensions, it is possible to express also non-regular properties.
- As an example, with "back-references" one can recognise the language $\{wcw \mid w \in \{a,b\}^*\}$ as follows:

```
>>> import re
>>> pattern = re.compile(r'([ab]*)c\1$')
>>> for s in ['abacaba','c','abbcaba']:
...   if pattern.match(s): print(s+" is in the language")
...
abacaba is in the language
c is in the language
```

  The "variable" `\1` on line 2 matches only the same substring that matched with the expression in the parentheses earlier in the expression.
- This language is not regular! (We'll see later why this is the case.)

Some voluntary brain teasers for winter evenings

Regular expression crosswords (the syntax is the fairly standard one used in many programming languages, see e.g. this link)