Introduction
to FSA and Regular Expressions

Carlo Strapparava
FBK-irst
strappa@fbk.eu

Introduction

- Regular Languages and Finite Automata are among the oldest topics in formal language theory (early '40)
- Formal language theory uses algebra and set theory to define formal languages as a sequence of symbols
- RL and FA have a wide range of applications:
  - Lexical analysis in programming language compilation
  - Circuit design, text editing, pattern matching, …
  - More recently: parallel processing, image generation and compression, type theory for OO languages, DNA computing, …

Naïve definitions

- Basically, a regular expression is a pattern describing a certain amount of text
- A regular expression is a string that is used to describe or match a set of strings, according to certain syntax rules
- A regular expression, often called a *pattern*, is an expression that describes a set of strings. They are usually used to give a concise description of a set, without having to list all elements
- For example, the three strings Handel - Händel - Haendel could be described by the pattern H(a|ä|ae)ndel

Representations for languages

- A formal language is a language that is defined by precise mathematical or machine processable formulas.
- Formal languages generally have two aspects:
  - the syntax of a language is what the language looks like (i.e. the set of possible expressions that are valid utterances in the language)
  - the semantics of a language are what the utterances of the language mean (which is formalized in various ways, depending on the type of language in question)

Representations for languages

- The branch of mathematics and computer science which studies exclusively the theory of language syntax is known as formal language theory
- In formal language theory, a language is nothing more than its syntax
- Questions of semantics are not addressed

Formal languages and computability

- Strong connection with the computability theory, i.e. the branch of the theory of computation that studies which problems are computationally solvable using different models of computation
- The study of abstract machines and problems they are able to solve
- Typical questions asked about such formalisms include:
  - *What is their expressive power?* (Can formalism X describe every language that formalism Y can describe? Can it describe other languages?)
  - *What is their recognizability?* (How difficult is it to decide whether a given word belongs to a language described by formalism X?)
  - *What is their comparability?* (How difficult is it to decide whether two languages, one described in formalism X and one in formalism Y, or in X again, are actually the same language?).

Representations for languages

- We will discuss the two principal methods for defining languages: the generator and the recognizer
- In particular we will focus on a particular class of generators (grammars) and of recognizers (automata)
- There are many types of formal languages, some of them are very "simple", others are more "complex"
- It is possible to put them in a hierarchy
- Regular languages are the *simplest* formal languages:
  - Their generators are the regular expressions
  - Their recognizers are the finite state automata

Automata theory: formal languages and formal grammars

Chomsky hierarchy	***Grammars***	***Languages***	***Minimal automaton***
Type-0	Unrestricted	Recursively enumerable	Turing machine
n/a	(no common name)	Recursive	Decider
Type-1	Context-sensitive	Context-sensitive	Linear-bounded
n/a	Indexed	Indexed	Nested stack
n/a	Tree-adjoining	Mildly context-sensitive	Thread
Type-2	Context-free	Context-free	Nondeterministic pushdown
n/a	Deterministic context-free	Deterministic context-free	Deterministic pushdown
Type-3	Regular	Regular	Finite state

Each category of languages or grammars is a proper subset of the category directly above it.

Strings and Languages

- An alphabet is defined as any set of symbols
  - Two examples:
    - the set of 26 upper and 26 lower case Roman letters (*the Roman alphabet*)
    - the set {0,1} -> *the binary alphabet*
- Strings over an alphabet Σ are defined as
  - ε (i.e. the *empty string*) is a string of Σ
  - if *x* is a string of Σ and *a* is in Σ, then *xa* is in Σ (concatenation)
- A language over Σ is a set of string over Σ

Operations on strings and languages

- Concatenations (or product):
  if *x* and *y* are strings over an alphabet Σ, then *xy* is called the concatenation of x
  Ex: if *x = ab* and *y = cd* then xy = abcd
- Reversal:
  x^R is the string *x* written in the reverse order
  Ex: *x = abcd* then x^R = dcba
- Closure:
  $a^0 = \varepsilon$
  $a^n = aa^{n-1}$ for $n \geq 1$
  $a^* = \cup_{n\geq0} a^n$
- Positive Closure:
  $a^+ = aa^* = \cup_{n\geq1} a^n$

Motivations

- How to represent a language *L* ? (e.g. when *L* is infinite, that is contains an arbitrary number of strings)
- Two principal methods:
  - Use a generative system, called grammar -> a set of rules that tell us which are the well-formed sentences in the language
  - Use a device (an automaton) that for a given input string will halt and answer "yes" if the string belongs to the language

Regular Sets

- Regular sets are a class of languages central to much of the language theory
- We will see several methods for specifying these languages
  - Regular expressions
  - Right-linear grammars
  - Deterministic finite-state automata
  - Non deterministic finite-state automata

⇒ All this formalisms are in fact *equivalent*

Regular sets - definition

- Let Σ be a finite alphabet. A regular set over Σ is defined recursively as follows:
- the empty language Ø is a regular language.
- the empty string language { ε } is a regular language.
- For each $a \in \Sigma$, the singleton language { a } is a regular language.
- If A and B are regular languages, then $A \cup B$ (union), AB (concatenation), and A^* (Kleene star) are regular languages.
- No other languages over Σ are regular.

A simple example of a language that is not regular is $\{a^n b^n \mid n \geq 0\}$

Regular expressions

- Regular expressions over Σ and the regular sets they denote are defined recursively as follows:
  - Ø is a regular expression denoting the empty set
  - ε is a regexpr denoting the regular set { ε }
  - a in Σ is a regexp denoting { a }
  - If p and q are regexp denoting P and Q, then
    - (p|q) is a regexp denoting P ∪ Q
    - (pq) is a regexp denoting PQ
    - (p)* is regexp denoting P*
  - Nothing else is a regular expression

- Sometimes the symbols ∪, +, or ∨ are used for alternation instead of the vertical bar |.
- To avoid brackets it is assumed that the Kleene star has the highest priority.

Examples

- The finite languages, i.e. those containing only a finite number of words
  - ⇨ These are obviously regular as one can create a regular expression that is the union of every word in the language, and thus are regular
- 01 denoting {01}
- 0* denoting {0}*
- (0|1)* denoting {0, 1}*
- (0|1)*011 denoting all strings of 0's and 1's ending in 011

Examples (cont.)

- Given the alphabet Σ = {a, b}:

→ ba* - all the strings that begin with a b followed only by a's

→ a*ba*ba* - strings that contain exactly two b's

→ (a | b)* - all the strings on Σ

→ (a | b)* (aa | bb) (a | b)* - all the string on Σ that contain either two consecutive a's or two consecutive b's

→ [aa | bb | (ab | ba)(aa | bb)*(ab | ba)]* - strings that contain an even number of a's and an even number of b's

→ (b | abb)* - strings on Σ in which an a is followed immediately by at least two b's

Basic algebraic properties

- Let α, β, and γ regular expressions
  - $\alpha \mid \beta=\beta \mid \alpha$
  - $\alpha|(\beta|\gamma)=(\alpha|\beta)|\gamma \quad \alpha(\beta\gamma)=(\alpha\beta)\gamma$
  - $\emptyset^* = \varepsilon$
  - $\alpha(\beta \mid \gamma) = \alpha\beta \mid \alpha\gamma \quad (\alpha|\beta)\gamma=\alpha\gamma|\beta\gamma$
  - $\alpha\varepsilon=\varepsilon\alpha=\alpha$
  - $\alpha^*=\alpha|\alpha^* \quad (\alpha^*)^*=\alpha^*$
  - $\alpha|\alpha=\alpha \quad \alpha|\emptyset=\alpha$
- All these properties are demonstrable by reasoning on the respective denoted sets

Finite State Automata

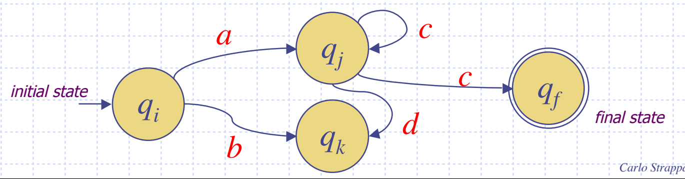
- We have seen some ways to define the class of the regular sets:
- The regular sets are those sets defined by *regular expressions*
- The regular sets are the languages generated by *right-linear grammar*
- We will see another way: regular sets defined by Finite Automata

Finite State Automata

- A finite-state automaton consists only of an input tape and a *finite control*
- A finite control means that the device that can be in one among a finite number of *states*
- In certain conditions, it can switch to another state => this is called a *transition*
- Allowable *input symbols*
- *Initial* and *final* states
- If the automaton is in a final state when it stops working, it is said to *accept* its input

FSA - transitions

- A state transition function that, given the "current" state and the "current" input symbol, returns all possible next states
- In principle, this device is *non-deterministic*: the device goes in all its next states, such as it replicates itself
- The device accepts the inputs if any of its parallel existences reaches an accepting state



FSA - definitions

- A *non-deterministic finite state automaton* is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, such that
  1. Q is a finite state of *states*
  2. Σ is a finite set of allowable *input symbols*
  3. δ is a *state transition function*, i.e. a mapping from $Q \times \Sigma$ to $\mathcal{P}(Q)$ that defines the finite state control
  4. q_0 in Q is the *initial state*
  5. $F \subseteq Q$ is the set of *final states*

FSA - definitions

- To determine the future behavior of a FSA, all we need to know is its *configuration*
  - The *current state* of the finite control
  - The *string symbol on the input tape* (= the symbol under the input head, followed by all symbols on the right)
- A *move* is represented as

$$(q, aw) \rightarrow (q', w)$$

means:
  - The automaton is in the current state q
  - The input head is scanning the symbol a
  - The automaton may change its state to q' and shift the input head on the right

FSA - example

- Let $M = (\{p, q, r\}, \{0, 1\}, \delta, p, \{r\})$ a FSA where δ is defined as:

		Input	
	δ	*0*	*1*
State	*p*	{q}	{p}
	q	{r}	{p}
	r	{r}	{r}

p: Two consecutive 0' have not appeared yet

q: Two consecutive 0' have not appeared , but the previous symbol was a 0

r: Two consecutive 0' have appeared

- M accepts string of 0's and 1's that contains two consecutive 0's

On input 01001, we have:
(p, 01001) → (q, 1001) → (p, 001) → (q, 01) → (r, 1) → (r, ε)

FSA - non-deterministic case

- Design a non-deterministic FSA to accept the strings
  - in the alphabet {1, 2, 3},
  - and such that the last symbol in the input string also appear previously in the string
  - e.g. 121 is accepted, 31312 not
- We will need some state, an initial state q_0 (nothing has been recognized), q_1 q_2 q_3 some guessing has been made, and a final q_f

FSA - non-deterministic case (2)

- More formally:
  $M = (\{q_0, q_1, q_2, q_3, q_f\}, \{1, 2, 3\}, \delta, q_0, \{q_f\})$

	δ	Input 1	2	3
State	q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_3\}$
	q_1	$\{q_1, q_f\}$	$\{q_1\}$	$\{q_1\}$
	q_2	$\{q_2\}$	$\{q_2, q_f\}$	$\{q_2\}$
	q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3, q_f\}$
	q_f	$\emptyset$	$\emptyset$	$\emptyset$

FSA - non-deterministic case (3)

- On input *12321*, the configurations will be

$(q_0, 12321) \rightarrow (q_0, 2321)$ –

$(q_1, 2321)$ –

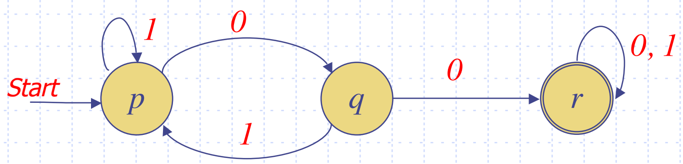
Since $(q_0, 12321) \xrightarrow{*} (q_f, e)$, the string *12321 is in L(M)*

FSA - transition graph

- It is often convenient to have a graph representation of finite automata
- E.g.: *$M = (\{p, q, r\}, \{0, 1\}, \delta, p, \{r\})$ with*

	Input		
	δ	*0*	*1*
State	*p*	$\{q\}$	$\{p\}$
	q	$\{r\}$	$\{p\}$
	r	$\{r\}$	$\{r\}$

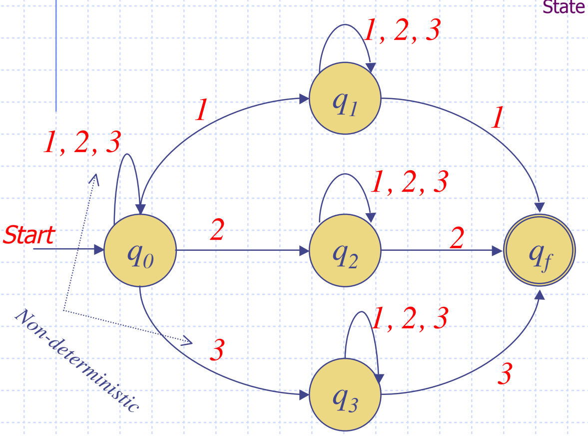
can be represented as



FSA - transition graph

- $M = (\{q_0, q_1, q_2, q_3, q_f\}, \{1, 2, 3\}, \delta, q_0, \{q_f\})$ with

	Input			
	δ	*1*	*2*	*3*
State	q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_3\}$
	q_1	$\{q_1, q_f\}$	$\{q_1\}$	$\{q_1\}$
	q_2	$\{q_2\}$	$\{q_2, q_f\}$	$\{q_2\}$
	q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3, q_f\}$
	q_f	$\emptyset$	$\emptyset$	$\emptyset$



FSA and non deterministic FSA

- There is an equivalence to deterministic and non-deterministic FSA:
- *Theorem:*
  If L= L(M) for some non-deterministic FSA M, then there is a M′ such that L = L(M′)

⇨ In the case of finite state automata, determinism and non-determinism have the same expressive power

Non-deterministic → deterministic transformation

- *Theorem:*
  If L= L(M) for some non-deterministic FSA M, then there is a M′ such that L = L(M′)
  - *$M = (Q, \Sigma, \delta, q_0, F)$.*
    We construct $M' = (Q', \Sigma, \delta', q'_0, F')$, such that

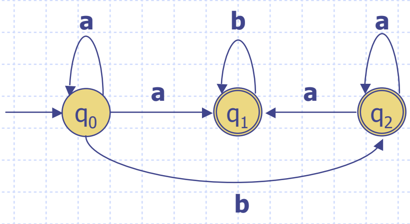
1) $Q' = \mathcal{P}(Q)$, i.e. the powersets (sets of states) of M
2) $q'_0 = \{q_0\}$
3) F′ consists of all subsets S of Q s.t. $S \cap F \neq \emptyset$
4) For all $S \subseteq Q$, $\delta'(S,a) = S'$, where
   $S' = \{p \mid \delta(q,a)$ contains p for some q in S$\}$

N-FSA to D-FSA in practice

- Given an N-FSA, we can construct an equivalent D-FSA
- States in the D-FSA correspond to the powersets of states in the N-FSA
- Straightforward way of computing D-FSA:
  - Create a list of all powersets of states in N-FSA
  - Add transitions according to those in the original N-FSA
  - Remove any states which cannot be reached

N-FSA to D-FSA in practice

Example:



We recall that $|\mathcal{P}(X)| = 2^{|X|}$

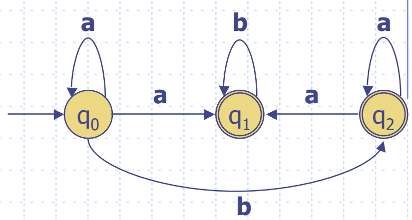
Powersets are

$\emptyset, q_0, q_1, q_2, \{q_0,q_1\}, \{q_0,q_2\}, \{q_1,q_2\}, \{q_0,q_1,q_2\}$

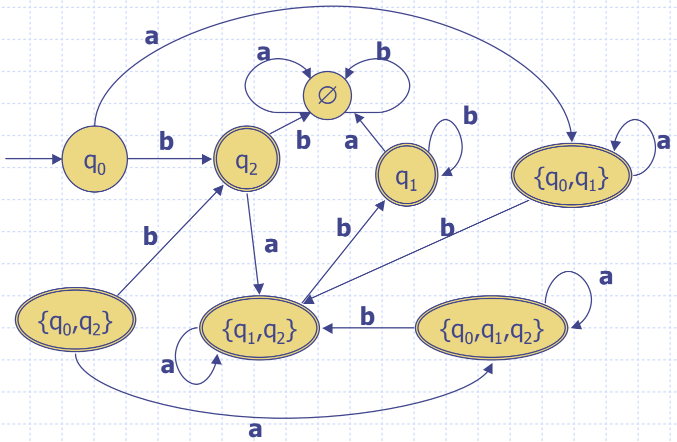
N-FSA to D-FSA

Example (continued)

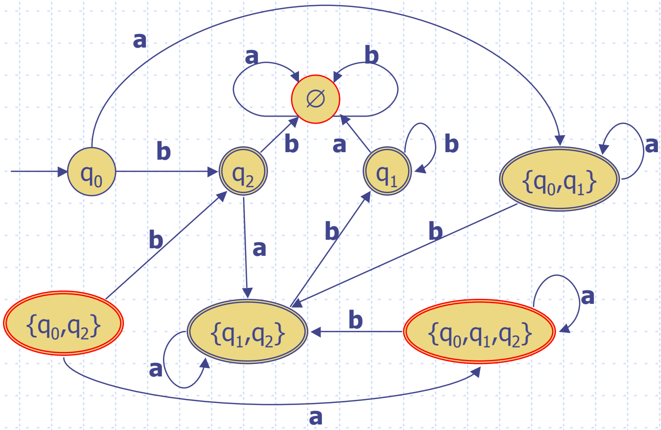
	a	b
$\emptyset$	$\emptyset$	$\emptyset$
q_0	$\{q_0,q_1\}$	q_2
q_1	$\emptyset$	q_1
q_2	$\{q_1,q_2\}$	$\emptyset$
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_1,q_2\}$
$\{q_0,q_2\}$	$\{q_0,q_1,q_2\}$	q_2
$\{q_1,q_2\}$	$\{q_1,q_2\}$	q_1
$\{q_0,q_1,q_2\}$	$\{q_0,q_1,q_2\}$	$\{q_1,q_2\}$



N-FSA to D-FSA

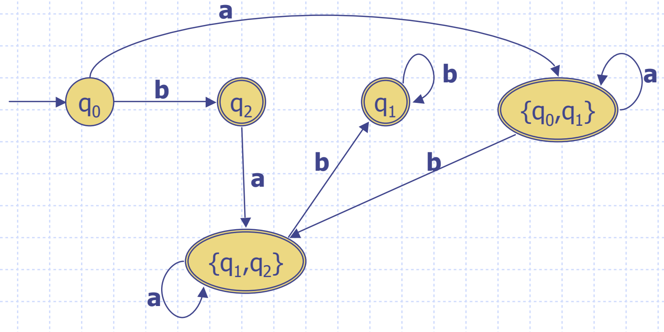


N-FSA to D-FSA



Highlighted states can't be reached (there are no transitions to them) or they are sink (lead to no acceptance states). So we can eliminate them.

N-FSA to D-FSA



We now have a D-FSA

N-FSA to D-FSA

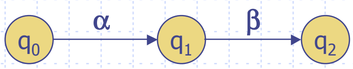
- Considering *all* powersets can lead to states in the D-FSA which cannot be reached and they have to be removed
- The number of powersets immediately becomes very large (an N-FSA with 20 states would have 2^{20} = 1.048.576 states!)
- We don't really need to consider all powersets: only those to which there are transitions in the original N-FSA have to be considered

Transformation Regexp <-> FSA

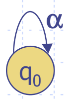
- *Theorem (Kleene)*:
  To each regular expression there corresponds a FSA and to each FSA there corresponds a regular expression
- We will give an algorithm to switch from these two objects

Transformation Regexp <-> FSA

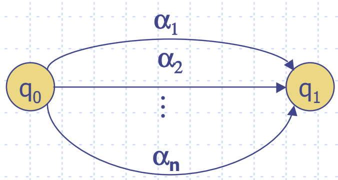
- We can observe that (α, β, α_i are regular expressions):



$\Leftrightarrow$ $\alpha\beta$



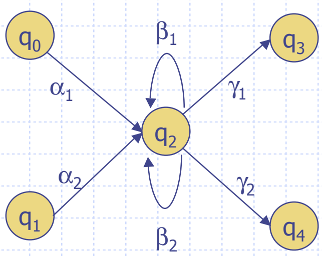
$\Leftrightarrow$ α^*



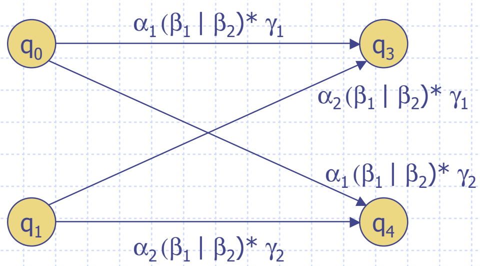
$\Leftrightarrow$ $\alpha_1 | \alpha_2 | \ldots | \alpha_n$

Node elimination

- Suppose we want to eliminate the node q_2 from the graph:

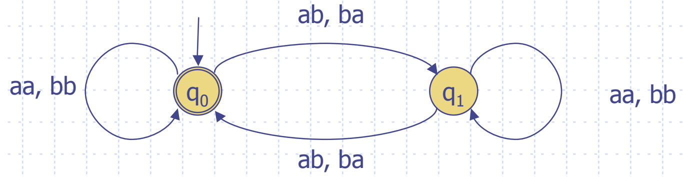


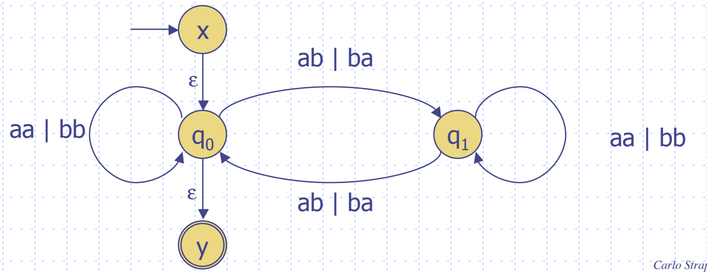
$\Rightarrow$



FSA -> Regexp

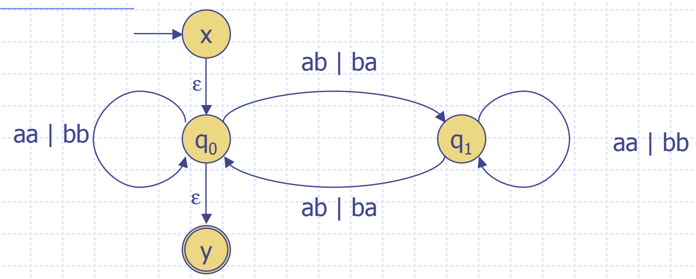
- An example to transform a FSA into a regexp

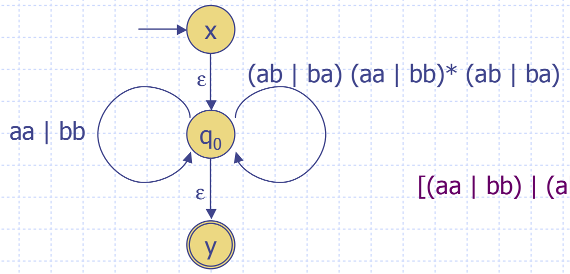


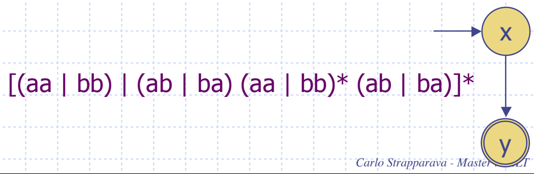


Carlo Strapparava - Master in HLT

FSA -> Regexp



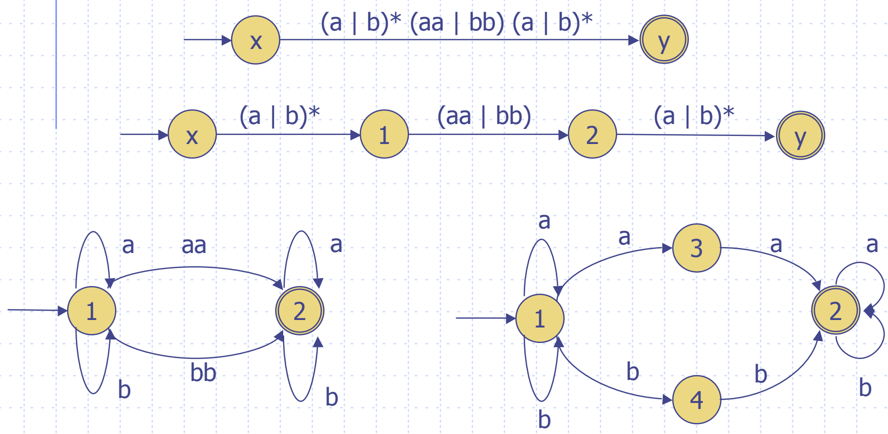




Carlo Strapparava - Master in HLT

Regexp -> FSA

- Let us consider the regexp
  (a | b)* (aa | bb) (a | b)*

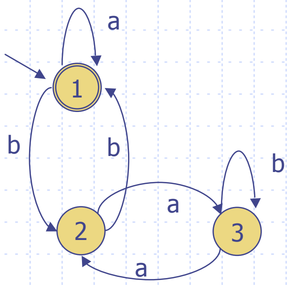


Equivalence of FSA’s

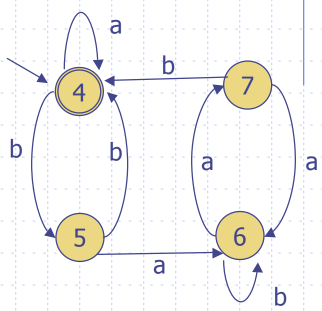
- *Theorem (Moore)*:
  There exists an algorithm, to determine if two FSA’s on an alphabet Σ are equivalent
- An algorithm:
  - A and A’ two FSA’s on Σ = {0,1}.
  - We rename the nodes, to have different labels in A and A’
  - We build a table of comparisons, with three columns, in this way:

Equivalence of FSA's (cont.)

A =



A′ =



(v, v')	(v_a, v'_a)	(v_b, v'_b)
(1,4)	(1,4)	(2,5)
(2,5)	(3,6)	(1,4)
(3,6)	(2,7)	(3,6)
(2,7)	(3,6)	(1,4)

Equivalence of FSA's (cont.)

- If in the table, we get to a pair *(v, v′)*, where *v* is an acceptance state and *v′* not, => A and A′ are not equivalent
- If we get to an end, i.e. there is no pair in the columns 2 and 3 that is not present in column 1, => the A and A′ are equivalent

Automata theory: formal languages and formal grammars

Chomsky hierarchy	***Grammars***	***Languages***	***Minimal automaton***
Type-0	Unrestricted	Recursively enumerable	Turing machine
n/a	(no common name)	Recursive	Decider
Type-1	Context-sensitive	Context-sensitive	Linear-bounded
n/a	Indexed	Indexed	Nested stack
n/a	Tree-adjoining	Mildly context-sensitive	Thread
Type-2	Context-free	Context-free	Nondeterministic pushdown
n/a	Deterministic context-free	Deterministic context-free	Deterministic pushdown
Type-3	Regular	Regular	Finite state

Each category of languages or grammars is a proper subset of the category directly above it.

Tokenization

- Wordforms, inflected words as it appears in the corpus
  - e.g. *cat* and *cats* are treated as two separated words
- Lemma
  - We might want to treat *cat* and *cats* as instances of a single lemma "*cat*"
- Types: distinct words in a corpus, i.e. the size of the vocabulary
- Tokens: the total number of running words
- The Brown corpus contains 1 million wordform tokens, that is 61,803 wordform types, that is 37,851 lemma types

Tokenization

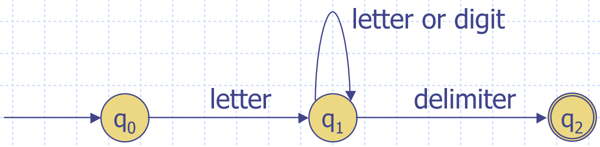
- Types and tokens
- The following sentence taken from the Brown corpus:

  "*They picnicked by the pool, then lay back on the grass and looked at the stars*"

- has 16 word tokens and 14 word types (not counting punctuation)

Tokenization

- A simple automaton for the recognition of the tokens



A delimiter can be any character that is not a letter or a digit

Regexp in the “real world”

- It is worth noting that many real-world "regular expression" engines implement features that cannot be expressed in the regular expression algebra
- Some examples:
- `grep`, *Unix command line*
- `AWK`, *Unix command line, progr. language*
- `Emacs`, *a powerful editor*
- `Perl`, *a programming language*
- `Pregexp` package, in Scheme

Grep - a Unix command

- `grep`, `egrep`, `fgrep` - print lines matching a pattern `[egrep = grep -e]`
- SYNOPSIS
  - `grep` [options] PATTERN [FILE...]
  - `grep` [options] [-e PATTERN | -f FILE] [FILE...]
- `grep` searches the named input FILEs (or standard input if no files are named, or the file name - is given) for lines containing a match to the given PATTERN. By default, `grep` prints the matching lines
- `egrep` is used when the pattern is a regular expression

Carlo Strapparava - Master in HLT

Grep - a Unix command

- **`grep fish fortunes`**
  - A woman without a man is like a fish without a bicycle.
  - No one can feel as helpless as the owner of a sick goldfish.
  - Time is about the stream I go a-fishing in.
- **`fgrep inst /etc/passwd`**
  - glenn:*:301:300:Glenn Stafford-instructor:/u/glenn:/bin/ksh
  - institution:*:301:300:Database Acct:/u/db:/bin/ksh

grep - other examples

- **`grep -i apple fruitlist.txt`**

⇨ returns all lines with the words 'apple', 'Apple', 'apPLE', or any other mixing of capital and lower case

- **`grep -r 'hello' /home/gigi`**

⇨ searches for '`hello`' in all files under the directory '`/home/gigi`'

Grep - regular expressions

- A regular expression may be followed by one of several repetition operators:
  - **.** The period . matches any single character.
  - **?** The preceding item is optional and will be matched at most once.
  - * The preceding item will be matched zero or more times.
  - [^] Match any one character **except** those enclosed in [ ], as in [^0-9].
  - \+ The preceding item will be matched one or more times.
  - {*n*} The preceding item is matched exactly *n* times.
  - {*n*,} The preceding item is matched n or more times.
  - {*n*,*m*} The preceding item is matched at least *n* times, but not more than *m* times.
- Two regular expressions may be concatenated;
- Two regular expressions may be joined by the infix operator | ; the resulting regular expression matches any sub-expression

Carlo Strapparava - Master in HLT

grep - examples

- An example is

  (hurrah){2,3}

  which matches

  hurrah hurrah

  as well as

  hurrah hurrah hurrah
- A more complex example combines alternation and grouping with a quantifier:

  (hurrah |yahoo){2,3}

  That gives twelve possible combinations, including for example

  hurrah yahoo

  and

  yahoo hurrah yahoo

Carlo Strapparava - Master in HLT

grep - examples

```
egrep '((the|a) (big( red)?|small( yellow)?) (car|bike))' car.txt
```

```
the big red car
a small bike
the small yellow car
a big red bike
```


Anchors

- Using `^` and `$`, you can force a regexp to match only at the beginning `^` or at the end `$` of a line
  - E.g. `^cat` matches only those lines that start with cat, and `cat$` matches only those lines that end with *cat*
- `\<` and `\>` are *start-of-word*, *end-of-word* anchors
  - E.g. `\<cat\>` looks for only the word cat

Anchors

```
grep 'cat' cats.txt
 cat
 cattle
 catalog
 scrawny cat
 vacation
 wildcat
```

```
grep '\<cat' cats.txt
 cat
 cattle
 catalog
 scrawny cat
```

```
grep '\<cat\>' cats.txt
 cat
 scrawny cat
```


Anchors

- These word boundaries are not supported in all regexp engines implementations
- Some implementations (inluding *perl*) offer *is-a-word-boundary* and *not-a-word-boundary*
- \b and \B respectively

```
grep '\bcat\b' cats.txt
 cat
 scrawny cat
```

Character classes

- The `[…]` construct indicates the presence of one of the enclosed characters
- E.g. c[ao]ke matches *cake* and *coke*
- `[0123456789abcdefABCDEF]` is also written as `[0-9a-fA-F]`
- `[^…]` means a 'negated' character set
- E.g. `[^0-9]` means any character *except* digits

Dot

- The dot `.` is a special character and matches any character
- E.g. `th.s` matches `this`, `thus`, `thgs`, `th@s`, …
- When you have to match a dot, you need to 'escaped' it => \.
- E.g. to match the IP address `74.6.7.121` all three dots need to be escaped `74\.6\.7\.121`

Quantifiers

- Using quantifiers, it is possible to specify how often a pattern may or must be repeated
- The general form is `{min,max}`
- Examples:
- `bo{1,2}k` matches both `book` and `bok`
- `[aeiou]{3,5}` matches any sequence of three to five vowels
- `finds{0,1}` matches `find` and `finds`
- `finds{0,1}` = `finds?`
- `^-{80,80}$`
  - matches lines of exactly eighty dash

Alternation and grouping

- The meta character `|` means *or*
- `^(From|Subject|Date):`
  - filters e-mail headers
- `(…)` has the function of grouping for quantifiers
  - `(hurrah ){2,3}` matches `hurrah hurrah hurrah`
  - `(hurrah | yahoo ){2,3}` matches `hurrah yahoo` or `yahoo hurrah yahoo` etc.

Backreferencing

- Grouping has a very useful side-effect
- Certain regexp implementations *remember* the matched text in a grouping
- E.g. searching for double words in a text, like … `when when` …
  - `([a-zA-Z]+)\1`
    the `\1` is called a backreference to the first group, in this case `([a-zA-Z]+)`
  - maybe better `([a-zA-Z]+) \1\>`

- The max number of backreferences is limited to nine in most regexp implementations

grep - regular expressions

- How to express palindromes in a regular expression?
- It can be done by using the back references, for example a palindrome of 5 characters can be written in
- **`grep -e '\(.\)\(.\).\2\1' file`**
- It matches the word "radar" or "civic".

```
\(.\)\(.\).\2\1
  r    a   d a r
```

Emacs and regexp

- Emacs is a powerful text editor
- Let us give a look at its regexp facilities
- An interactive command “replace-regexp”

- Transform every line in a file (e.g. `/etc/passwd`) that matches
  - `^\([^:]*\):[^:]*:\([0-9]*\):[0-9]*:\([^:]*\):.*$`
- into
  - `Login {\1} Full Name {\3} UID {\2}`

- Ex. It matches the line
  - `mysql:*:74:74:MySQL Server:/var/empty:/usr/bin/false`
  - `^\([^:]*\):[^:]*:\([0-9]*\):[0-9]*:\([^:]*\):.*$`

Exercise

- ALPHABET: a b c
- Write a regular expression for the language of all strings over the alphabet {a,b,c} that start with character a

Solution: a(a|b|c)*

Exercise

- ALPHABET: a b c
- Write a regular expression for the language of all strings over the alphabet {a,b,c} that start and end with the character a

SOLUTION: a(a|b|c)*a|a

Exercise

- ALPHABET: a b c
- Write a regular expression for the language of all strings over the alphabet {a,b,c} that start with character a, but do not end with character a

SOLUTION: a(a|b|c)*(b|c)

Exercise

- ALPHABET: a b c
- Give a regular expression over {a, b, c} where a must appear in blocks of even length

SOLUTION: (aa|b|c)*

Carlo Strapparava - Master in HLT

Exercise

- ALPHABET: 0 1 x
- Write a regular expression for the language of all strings over the alphabet {0,1,x} that contain at least one x

SOLUTION: (0|1)*x(0|1|x)*

Carlo Strapparava - Master in HLT

Different syntax in the real engines

- The practical regexp engines use different syntax for writing the regular expressions
  - Simple matching
  - POSIX basic
  - POSIX extended
  - Emacs
  - Grep
  - GNU regex
  - Java
  - Perl
  - Ruby
  - …
- Mainly small differences, but before using a tool you have to read the manual