Finite Automata and Formal Languages

TMV026/DIT321– LP4 2011

Ana Bove

Lecture 6

April 4th 2011

Overview of today's lecture:

- **NFA with ϵ-Transitions**
- **Regular Expressions**

Recall: ϵ-Closures

Definition: Formally, we define the *ϵ-closure* of a set of states with the following 2 rules:

$$\frac{q \in S}{q \in \mathsf{ECLOSE}(S)} \qquad\qquad \frac{q \in \mathsf{ECLOSE}(S) \qquad p \in \delta(q, \epsilon)}{p \in \mathsf{ECLOSE}(S)}$$

Intuitively, $p \in \mathsf{ECLOSE}(S)$ iff there exists $q \in S$ and a sequence of ϵ-transitions such that

$$q_1 \in \delta(q, \epsilon) \quad q_2 \in \delta(q_1, \epsilon) \quad \cdots \quad p \in \delta(q_n, \epsilon)$$

Definition: We say that S is *ϵ-closed* iff $S = \mathsf{ECLOSE}(S)$.

S is ϵ-closed iff $q \in S$ and $p \in \delta(q, \epsilon)$ implies $p \in S$.

Extending the Transition Function to Strings

Definition: Given an ϵ-NFA $E = (Q, \Sigma, \delta, q_0, F)$ we define

$$\hat{\delta} : Q \times \Sigma^* \to [Q]$$
$$\hat{\delta}(q, \epsilon) = \mathsf{ECLOSE}(\{q\})$$
$$\hat{\delta}(q, ax) = \bigcup_{p \in \Delta(\mathsf{ECLOSE}(\{q\}), a)} \hat{\delta}(p, x)$$
$$\text{where } \Delta(S, a) = \cup_{p \in S} \delta(p, a)$$

Remark: By definition we have that
$\hat{\delta}(q, a) = \mathsf{ECLOSE}(\Delta(\mathsf{ECLOSE}(\{q\}), a))$.

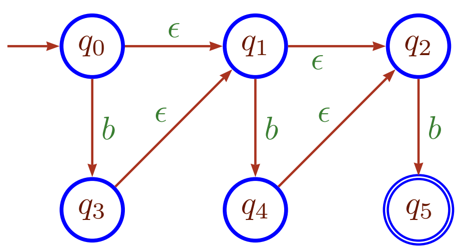
Remark: We can prove by induction on x that all sets $\hat{\delta}(q, x)$ are ϵ-closed.

This result uses that the union of ϵ-closed sets is also a ϵ-closed set.

Language Accepted by a ϵ-NFA

Definition: The *language* accepted by the ϵ-NFA $(Q, \Sigma, \delta, q_0, F)$ is the set $\mathcal{L} = \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) \cap F \neq \emptyset\}$.

Example: Let $\Sigma = \{b\}$.



The automaton accepts the language $\{b, bb, bbb\}$.

Note: Yet again, we could write a program that simulates a ϵ-NFA and let the program tell us whether a certain string is accepted or not.

Functional Representation of an ϵ-NFA

Let us implement the ϵ-NFA that recognises numbers (slide 21 lecture 5).

```
data Q = Q0 | Q1 | Q2 | Q3 | Q4  deriving (Eq,Show)

final :: Q -> Bool
final Q4 = True
final _ = False

e_jump :: Q -> [Q]
e_jump Q0 = [Q1]
e_jump Q2 = [Q4]
e_jump _ = []

isSub :: [Q] -> [Q] -> Bool

closure :: [Q] -> [Q]
```


Functional Representation of an ϵ-NFA (cont.)

```
delta :: Char -> Q -> [Q]
delta a Q0 | elem a "+-" = [Q1]
delta a Q1 | elem a "0123456789" = [Q2]
delta a Q2 | elem a "0123456789" = [Q2]
delta '.' Q2 = [Q3]
delta a Q3 | elem a "0123456789" = [Q4]
delta a Q4 | elem a "0123456789" = [Q4]
delta _ _ = []

run :: String -> Q -> [Q]
run [] q = closure [q]
run (a:xs) q = closure [q] >>= delta a >>= run xs

accepts :: String -> Bool
accepts xs = or (map final (run xs Q0))
```

Eliminating ϵ-Transitions

Definition: Given an ϵ-NFA $E = (Q_E, \Sigma, \delta_E, q_E, F_E)$ we define a DFA $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$ as follows:

- $Q_D = \{\mathsf{ECLOSE}(S) \mid S \in \mathcal{P}ow(Q_E)\}$
- $\delta_D(S, a) = \mathsf{ECLOSE}(\Delta(S, a))$ with $\Delta(S, a) = \cup_{p \in S}\delta(p, a)$
- $q_D = \mathsf{ECLOSE}(\{q_E\})$
- $F_D = \{S \in Q_D \mid S \cap F_E \neq \emptyset\}$

Note: This construction is similar to the subset construction but now we need to ϵ-close after each step.

Eliminating ϵ-Transitions

Let E be an ϵ-NFA and D the corresponding DFA.

Lemma: $\forall x \in \Sigma^*.\ \hat{\delta}_E(q_E, x) = \hat{\delta}_D(q_D, x)$.

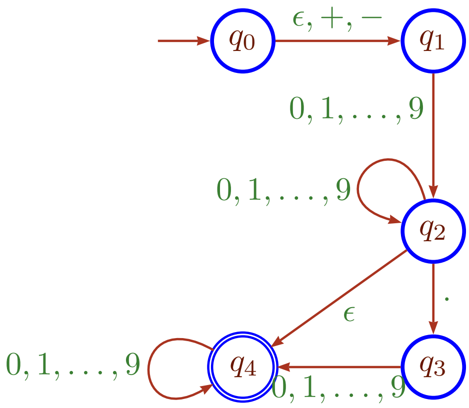
Proof: By induction on x.

Proposition: $\mathcal{L}(E) = \mathcal{L}(D)$.

Proof: $x \in \mathcal{L}(E)$ iff $\hat{\delta}_E(q_E, x) \cap F_E \neq \emptyset$ iff $\hat{\delta}_E(q_E, x) \in F_D$ iff (by previous lemma) $\hat{\delta}_D(q_D, x) \in F_D$ iff $x \in \mathcal{L}(D)$.

Example: Eliminating ϵ-Transitions

Let us eliminate the ϵ-transitions in the following ϵ-NFA.

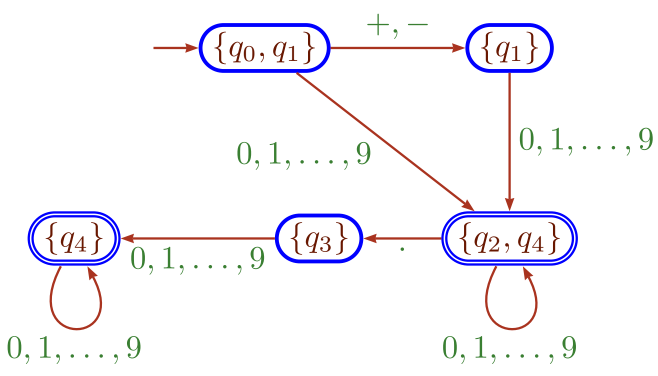


	+,-	.	0,1,...,9	ϵ
$\to q_0$	$\{q_1\}$	$\emptyset$	$\emptyset$	$\{q_1\}$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\{q_3\}$	$\{q_2\}$	$\{q_4\}$
q_3	$\emptyset$	$\emptyset$	$\{q_4\}$	$\emptyset$
$*q_4$	$\emptyset$	$\emptyset$	$\{q_4\}$	$\emptyset$

ϵ-NFA – Regular Expressions

Example: Eliminating ϵ-Transitions

We obtain the following DFA:



Functional Representation of Eliminating ϵ-Transitions

```
pDelta :: Char -> [Q] -> [Q]
pDelta a qs = closure (qs >>= delta a)

pRun :: [Char] -> [Q] -> [Q]
pRun [] qs = qs
pRun (a:x) qs = pRun x (pDelta a qs)

run  :: String -> Q -> [Q]
run xs q = pRun xs (closure [q])

accepts :: String -> Bool
accepts xs = or (map final (run xs Q0))
```


Finite Automata and Regular Languages

We have shown that DFA, NFA and ϵ-NFA are equivalent in the sense that we can transform one to the other.

Hence, a language is *regular* iff there exists a finite automaton (DFA, NFA or ϵ-NFA) that accepts the language.

Regular Expressions

Regular expressions (RE) are an "algebraic" way to denote languages.

Given a RE R, it defines the language $\mathcal{L}(R)$.

We will show that RE are as expressive as DFA and hence, they define all and only the *regular languages*.

RE can also be seen as a declarative way to express the strings we want to accept and serve as input language for certain systems.

Example: `grep` command in UNIX (K. Thompson).

(**Note:** UNIX regular expressions are not exactly as the RE we will study in the course.)

Inductive Definition of Regular Expressions

Definition: Given an alphabet Σ, we can inductively define the *regular expressions* over Σ as:

Basis cases:
- The constants $\emptyset$ and ϵ are RE
- If $a \in \Sigma$ then a is a RE

Inductive steps: Given the RE R and S, we define the following RE:
- $R + S$ and RS are RE
- R^* is RE

The precedence of the operands is the following:

- The closure operator * has the highest precedence
- Next comes concatenation
- Finally, comes the operator $+$
- We use parentheses (,) to change the precedences

Another Way to Define the Regular Expressions

A nicer way to define the regular expressions is by giving the following BNF (Backus-Naur Form), for $a \in \Sigma$:

$$R ::= \emptyset \mid \epsilon \mid a \mid R + R \mid RR \mid R^*$$

alternatively

$$R, S ::= \emptyset \mid \epsilon \mid a \mid R + S \mid RS \mid R^*$$

Question: Can you guess their meaning?

Note: BNF is a way to declare the syntax of a language.

It is very useful when describing *context-free grammars* and in particular the syntax of most programming languages.

Functional Representation of Regular Expressions

```
data RExp a = Empty | Epsilon | Atom a |
              Plus (RExp a) (RExp a) | Concat (RExp a) (RExp a) |
              Star (RExp a)
```

For example the expression $b + (bc)^*$ is given as

```
Plus (Atom "b") (Star (Concat (Atom "b") (Atom "c")))
```

Recall: Some Operations on Languages (Lecture 3)

Definition: Given $\mathcal{L}$, $\mathcal{L}_1$ and $\mathcal{L}_2$ languages then we define the following languages:

Union: $\mathcal{L}_1 \cup \mathcal{L}_2 = \{x \mid x \in \mathcal{L}_1 \text{ or } x \in \mathcal{L}_2\}$

Intersection: $\mathcal{L}_1 \cap \mathcal{L}_2 = \{x \mid x \in \mathcal{L}_1 \text{ and } x \in \mathcal{L}_2\}$

Concatenation: $\mathcal{L}_1\mathcal{L}_2 = \{x_1x_2 \mid x_1 \in \mathcal{L}_1,\ x_2 \in \mathcal{L}_2\}$

Closure: $\mathcal{L}^* = \bigcup_{n\in\mathbb{N}} \mathcal{L}^n$
where $\mathcal{L}^0 = \{\epsilon\}$, $\mathcal{L}^{n+1} = \mathcal{L}^n\mathcal{L}$.

Note: We have then that $\emptyset^* = \{\epsilon\}$ and
$\mathcal{L}^* = \mathcal{L}^0 \cup \mathcal{L}^1 \cup \mathcal{L}^2 \cup \ldots = \{\epsilon\} \cup \{x_1 \ldots x_n \mid n > 0, x_i \in \mathcal{L}\}$

Notation: $\mathcal{L}^+ = \mathcal{L}^1 \cup \mathcal{L}^2 \cup \mathcal{L}^3 \cup \ldots$ and $\mathcal{L}? = \mathcal{L} \cup \{\epsilon\}$.

Language Defined by the Regular Expressions

Definition: The *language* defined by a regular expression is defined by recursion on the expression:

Basis cases:
- $\mathcal{L}(\emptyset) = \emptyset$
- $\mathcal{L}(\epsilon) = \{\epsilon\}$
- Given $a \in \Sigma$, $\mathcal{L}(a) = \{a\}$

Recursive cases:
- $\mathcal{L}(R + S) = \mathcal{L}(R) \cup \mathcal{L}(S)$
- $\mathcal{L}(RS) = \mathcal{L}(R)\mathcal{L}(S)$
- $\mathcal{L}(R^*) = \mathcal{L}(R)^*$

Note: $x \in \mathcal{L}(R)$ iff x is generated/accepted by R.

Notation: We write $x \in R$ or $x \in \mathcal{L}(R)$ indistinctly.

Example of Regular Expressions

Let $\Sigma = \{0, 1\}$.

- $(01)^*$
- $0^* + 1^*$
- $(0 + 1)^*$
- $(000)^*$
- $01^* + 1$
- $((0(1^*)) + 1)$
- $(01)^* + 1$
- $(\epsilon + 1)(01)^*(\epsilon + 0)$
- $(01)^* + 1(01)^* + (01)^*0 + 1(01)^*0$

What do they mean? Are there expressions that are equivalent?

Algebraic Laws for Regular Expressions

The following equalities hold for any RE R, S and T:

- Associativity: $R + (S + T) = (R + S) + T$ and $R(ST) = (RS)T$
- Commutativity: $R + S = S + R$
- In general, $RS \neq SR$
- Distributivity: $R(S + T) = RS + RT$ and $(S + T)R = SR + TR$
- Identity: $R + \emptyset = \emptyset + R = R$ and $R\epsilon = \epsilon R = R$
- Annihilator: $R\emptyset = \emptyset R = \emptyset$
- Idempotent: $R + R = R$
- $\emptyset^* = \epsilon^* = \epsilon$
- $R? = \epsilon + R$
- $R^+ = RR^* = R^*R$
- $R^* = (R^*)^* = R^*R^* = \epsilon + R^+$

Algebraic Laws for Regular Expressions

Other useful laws to simplify regular expressions are

- *Shifting rule:* $R(SR)^* = (RS)^*R$
- *Denesting rule:* $(R^*S)^*R^* = (R+S)^*$

  Note: By the shifting rule we also get $R^*(SR^*)^* = (R+S)^*$
- Variation of the denesting rule: $(R^*S)^* = \epsilon + (R+S)^*S$

Example: Proving Equalities Using the Algebraic Laws

Example: A proof that $a^*b(c+da^*b)^* = (a+bc^*d)^*bc^*$:

$$a^*b(c+da^*b)^* = a^*b(c^*da^*b)^*c^* \qquad \text{by denesting } (R = c, S = da^*b)$$
$$a^*b(c^*da^*b)^*c^* = (a^*bc^*d)^*a^*bc^* \qquad \text{by shifting } (R = a^*b, S = c^*d)$$
$$(a^*bc^*d)^*a^*bc^* = (a+bc^*d)^*bc^* \qquad \text{by denesting } (R = a, S = bc^*d)$$

Example: The set of all words with no substring of more than two adjacent 0's is $(1+01+001)^*(\epsilon+0+00)$. Now,

$$\begin{aligned}(1+01+001)^*(\epsilon+0+00) &= ((\epsilon+0)(\epsilon+0)1)^*(\epsilon+0)(\epsilon+0)\\ &= (\epsilon+0)(\epsilon+0)(1(\epsilon+0)(\epsilon+0))^* \qquad \text{by shifting}\\ &= (\epsilon+0+00)(1+10+100)^*\end{aligned}$$

Then $(1+01+001)^*(\epsilon+0+00) = (\epsilon+0+00)(1+10+100)^*$

Equality of Regular Expressions

Remember that RE are a way to denote languages.

Then, for RE R and S, $R = S$ actually means $\mathcal{L}(R) = \mathcal{L}(S)$.

Hence we can prove the equality of RE in the same way we can prove the equality of languages.

Example: Let us prove that $R^* = R^*R^*$. Let $\mathcal{L} = \mathcal{L}(R)$.

$\mathcal{L}^* \subseteq \mathcal{L}^*\mathcal{L}^*$ since $\epsilon \in \mathcal{L}^*$.

Conversely, if $\mathcal{L}^*\mathcal{L}^* \subseteq \mathcal{L}^*$ then $x = x_1x_2$ with $x_1 \in \mathcal{L}^*$ and $x_2 \in \mathcal{L}^*$.

If $x_1 = \epsilon$ or $x_2 = \epsilon$ then it is clear that $x \in \mathcal{L}^*$.

Otherwise $x_1 = u_1u_2 \ldots u_n$ with $u_i \in \mathcal{L}$ and $x_2 = v_1v_2 \ldots v_m$ with $v_j \in \mathcal{L}$.

Then $x = x_1x_2 = u_1u_2 \ldots u_nv_1v_2 \ldots v_m$ is in $\mathcal{L}^*$.

Proving Algebraic Laws for Regular Expressions

Given the RE R and S we can prove the law $R = S$ as follows:

1. Convert R and S into *concrete* regular expressions C and D, respectively, by replacing each variable in the RE R and S by (different) concrete symbols.

   Example: $R(SR)^* = (RS)^*R$ can be converted into $a(ba)^* = (ab)^*a$.

2. Prove or disprove whether $\mathcal{L}(C) = \mathcal{L}(D)$. If $\mathcal{L}(C) = \mathcal{L}(D)$ then $R = S$ is a true law, otherwise it is not.

Theorem: *The above procedure correctly identifies the true laws for RE.*

Proof: See theorems 3.14 and 3.13 in pages 121 and 120 respectively.

Example: Proving the shifting law was (somehow) one of the exercises in assignment 1: prove that for all n, $a(ba)^n = (ab)^na$.

Example: Proving the Denesting Rule

We can state $(R^*S)^*R^* = (R+S)^*$ by proving $\mathcal{L}((a^*b)^*a^*) = \mathcal{L}((a+b)^*)$:

$\subseteq$: Let $x \in (a^*b)^*a^*$, then $x = vw$ with $v \in (a^*b)^*$ and $w \in a^*$.

By induction on v.

If $v = \epsilon$ we are done. Otherwise $v = av'$ or $v = bv'$.

Observe that in both cases $v' \in (a^*b)^*$ hence by IH $v'w \in (a+b)^*$ and so is vw.

$\supseteq$: Let $x \in (a+b)^*$. By induction on x. If $x = \epsilon$ then done.

Otherwise $x = x'a$ or $x = x'b$ and $x' \in (a+b)^*$.

By IH $x' \in (a^*b)^*a^*$ and then $x' = vw$ with $v \in (a^*b)^*$ and $w \in a^*$.

If $x'a = v(wa) \in (a^*b)^*a^*$ since $v \in (a^*b)^*$ and $(wa) \in a^*$.

If $x'b = (v(wb))\epsilon \in (a^*b)^*a^*$ since $v(wb) \in (a^*b)^*$ and $\epsilon \in a^*$.

Regular Languages and Regular Expressions

Theorem: *If $\mathcal{L}$ is a regular language then there exists a regular expression R such that $\mathcal{L} = \mathcal{L}(R)$.*

Proof: Recall that each regular language has an automata that recognises it.

We shall construct a regular expression from such automata.

The book shows 2 ways of constructing a regular expression from an automata (sections 3.2.1 –computing $R_{ij}^{(k)}$– and 3.2.2. –eliminating states–).

From FA to RE: Computing $R_{ij}^{(k)}$ from an Automaton A

Let $Q_A = \{1, 2, \ldots, n\}$ with 1 being the initial state.

We construct a collection of RE that progressively describe the paths in the transition diagram of A:

> Let $R_{ij}^{(k)}$ be the RE whose language is the set of strings w which label a path from state i to state j in A without passing by an intermediate state bigger than k.
> Note that neither i nor j are intermediate states!

We define $R_{ij}^{(k)}$ by induction on k.

If $F_A = \{f_1, \ldots, f_r\}$ then our final regular expression is

$$R_{1f_1}^{(n)} + \ldots + R_{1f_r}^{(n)}$$

Base Case: $R_{ij}^{(0)}$

We have no intermediate states here! We have the following scenarios:

- Arcs from state i to j?:
  - $*$ If there are no arc then $R_{ij}^{(0)} = \emptyset$
  - $*$ If there is one arc labelled a then $R_{ij}^{(0)} = a$
  - $*$ If there are m arcs labelled $a_1, \ldots, a_m$ then $R_{ij}^{(0)} = a_1 + \ldots + a_m$

  Note: If $i = j$ then we must consider the loops from i to itself.

- We have a path of length 0 from i to itself.

  In a ϵ-NFA we can also have paths of length 0 between i and j.

  Such a path is represented as an ϵ-transition in the automaton and as the RE ϵ.

  Then we need to add ϵ to the corresponding case above, obtaining then $R_{ij}^{(0)} = \epsilon$, $\quad R_{ij}^{(0)} = \epsilon + a \quad$ or $\quad R_{ij}^{(0)} = \epsilon + a_1 + \ldots + a_m \quad$ respectively.

Inductive Step: from $R_{ij}^{(k)}$ to $R_{ij}^{(k+1)}$

Given a path from state i to state j without passing by an intermediate state bigger than $(k+1)$, we have 2 possible cases:

- The path does not actually pass by state $(k+1)$.
  Hence the label of the path is in the language of the RE $R_{ij}^{(k)}$.

- The path goes through $(k+1)$ at least once.
  We can break the path into pieces that do not pass through $k+1$: first from i to $(k+1)$, one or more from $(k+1)$ to $(k+1)$, last from $(k+1)$ to j.
  The label for this path is represented by the RE
  $R_{i(k+1)}^{(k)}(R_{(k+1)(k+1)}^{(k)})^* R_{(k+1)j}^{(k)}$.

The resulting RE is $R_{ij}^{(k+1)} = R_{ij}^{(k)} + R_{i(k+1)}^{(k)}(R_{(k+1)(k+1)}^{(k)})^* R_{(k+1)j}^{(k)}$

Remarks on the Method for Computing $R_{ij}^{(k)}$

- Works for any kind of FA (DFA, NFA and ϵ-NFA).

- The method is similar to Floyd-Warshall algorithm (graph analysis algorithm for finding shortest paths in a weighted, directed graph).
  See Wikipedia.

- It is expensive: we need to compute n^2 RE!
  It also produces very big and complicated expressions!
  The (intermediate) RE can usually be simplified. Still not trivial!

  Example: See example 3.5 in the book (pages 95-97).