Regular Expressions

5DV037 — Fundamentals of Computer Science

Umeå University
Department of Computing Science

Stephen J. Hegner
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

The Idea of Regular Expressions

- The *regular expressions* (or *RE*'s) are a way of defining languages in a recursive fashion, based upon simple primitives.
- The primitive regular expressions over Σ and the languages which they define:

Regular Expression e	Language $\mathcal{L}(e)$	Note
$\emptyset$	$\emptyset$	
λ	$\{\lambda\}$	
a	$\{a\}$	for each $a \in \Sigma$

- The recursively defined regular expressions over Σ and the languages which they define:

Regular Expression e	Language $\mathcal{L}(e)$
$(r_1 + r_2)$	$\mathcal{L}(r_1) \cup \mathcal{L}(r_2)$
$(r_1 \cdot r_2)$	$\mathcal{L}(r_1) \cdot \mathcal{L}(r_2)$
${r_1}^*$	$(\mathcal{L}(r_1))^*$
(r_1)	$\mathcal{L}(r_1)$

An Example of the Language of a Regular Expression

- Let $r = (((a \cdot b) + c) + a^*)^*$.
- To find $\mathcal{L}(r)$, simply apply the rules:

$$
\begin{aligned}
\mathcal{L}(r) &= \mathcal{L}((((a \cdot b) + c) + a^*)^*) \\
&= (\mathcal{L}((((a \cdot b) + c) + a^*)))^* \\
&= (\mathcal{L}(((a \cdot b) + c)) \cup \mathcal{L}(a^*)))^* \\
&= ((\mathcal{L}((a \cdot b)) \cup \mathcal{L}(c)) \cup \mathcal{L}(a^*)))^* \\
&= (((\mathcal{L}(a) \cdot \mathcal{L}(b)) \cup \mathcal{L}(c)) \cup \mathcal{L}(a^*)))^* \\
&= (((\mathcal{L}(a) \cdot \mathcal{L}(b)) \cup \mathcal{L}(c)) \cup (\mathcal{L}(a))^*)^* \\
&= (\{ab, c\} \cup \{\lambda, a, aa, aaa, aaaa, \ldots\})^* \\
&= (\{ab, c, a\})^*
\end{aligned}
$$

- The last step requires a little thought and does not follow automatically from the rules.
- Some useful simplifications can be developed, however.

Properties of Regular Expressions

- The REs r_1 and r_2 are *equivalent* if $\mathcal{L}(r_1) = \mathcal{L}(r_2)$.
  - Write $r_1 = r_2$.
- $+$ and $\cdot$ are associative:
$$\begin{aligned} ((r_1 + r_2) + r_3) &= (r_1 + (r_2 + r_3)) \\ ((r_1 \cdot r_2) \cdot r_3) &= (r_1 \cdot (r_2 \cdot r_3)) \end{aligned}$$
- $+$ is commutative: $(r_1 + r_2) = (r_2 + r_1)$
- $\cdot$ distributes over $+$:
$$\begin{aligned} (r_1 \cdot (r_2 + r_3)) &= ((r_1 \cdot r_2) + (r_1 \cdot r_3)) \\ ((r_1 + r_2) \cdot r_3) &= ((r_1 \cdot r_3) + (r_2 \cdot r_3)) \end{aligned}$$
- $\emptyset$ is an identity for $+$: $(r + \emptyset) = (\emptyset + r) = r$
- λ is an identity for $\cdot$: $(r \cdot \lambda) = (\lambda \cdot r) = r$
- Positivity: $(r_1 + r_2) = \emptyset$ implies $r_1 = \emptyset$ and $r_2 = \emptyset$
- Dual of positivity: $(r_1 \cdot r_2) = \emptyset$ implies $r_1 = \emptyset$ or $r_2 = \emptyset$
- Mathematicians call this a *positive semiring*.

Additional Conventions for and Properties of REs

- Just as with the the usual (semiring of) integers, parentheses may be dropped:

Examples:

$$
\begin{aligned}
r_1 + r_2 &= (r_1 + r_2) \\
r_1 \cdot r_2 &= (r_1 \cdot r_2) \\
r_1 + r_2 + r_3 &= ((r_1 + r_2) + r_3) = (r_1 + (r_2 + r_3)) \\
r_1 \cdot r_2 \cdot r_3 &= ((r_1 \cdot r_2) \cdot r_3) = (r_1 \cdot (r_2 \cdot r_3))
\end{aligned}
$$

- Multiplication has higher precedence than addition:

$$r_1 \cdot r_2 + r_3 = (r_1 \cdot r_2) + r_3$$

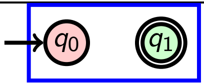
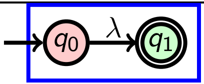
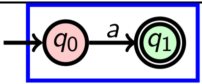
- Star has higher precedence than multiplication: $r_1{}^* \cdot r_2 = (r_1{}^*) \cdot r_2$
- Dot may be dropped: $a \cdot b = ab$
- Some additional properties of regular expressions:
  - $r^{**} = r^*$
  - $(\lambda + r)^* = r^*$
  - $(r_1{}^* \cdot r_2{}^*)^* = (r_1 + r_2)^*$
- Test your knowledge of REs by proving the last property ...
- ... or find the answer as a solution to an exercise in the book.

Some Examples of Constructing Regular Expressions

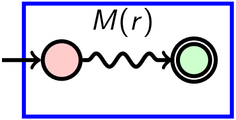
- The set of all strings over $\Sigma = \{a, b\}$ which contain *ab* as a substring: $(a + b)^* \cdot ab \cdot (a + b)^*$
- The set of all strings over $\Sigma = \{a, b\}$ which contain *ab* as a substring at least twice: $(a + b)^* \cdot ab \cdot (a + b)^* \cdot ab \cdot (a + b)^*$
- The set of all strings over $\Sigma = \{a, b\}$ which do not contain *ab* as a substring:
    - This is more difficult, since the REs do not have a negation construct: $b^* \cdot a^*$.
- The set of all strings over $\Sigma = \{a, b, c\}$ which do not contain *ab* as a substring:
    - This is even more difficult, and requires some thought: $(b + a^* c)^* \cdot a^*$.
- The set of all strings over $\Sigma = \{a, b\}$ which contain *ab* as a substring exactly twice: $(b + a^* c)^* \cdot ab \cdot (b + a^* c)^* \cdot ab \cdot (b + a^* c)^*$

Constructing an NFA from an RE

- For the primitive REs, a "building block" with exactly one accepting state is required.

$\emptyset$	λ	a
		

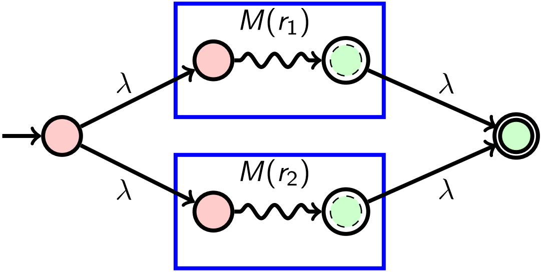
- For a complex RE r, assume that an NFA $M(r)$ with exactly one accepting state and with $\mathcal{L}(M(r)) = \mathcal{L}(r)$ is given for each constituent.



- These NFAs are then connected together to obtain the NFA accepting a more complex RE.

Constructing an NFA from an RE — the "+" Case

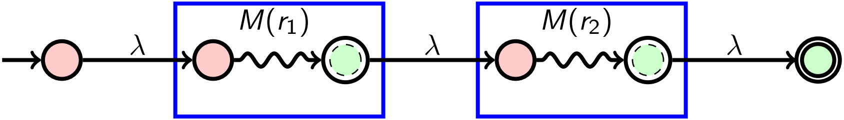
- To obtain an accepter for $r_1 + r_2$, use a "parallel" connection of the two accepters, as follows.



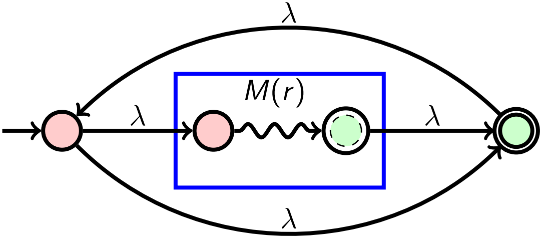
- Note the utility of λ transitions.
- The direct realization of a deterministic accepter for $r_1 + r_2$ is much more complex.

Constructing an NFA from an RE — "·" and "∗" Cases

- To obtain an accepter for $r_1 \cdot r_2$, use a "serial" connection of the two accepters, as follows.



- To obtain an accepter for r^*, use a "feedback/feedforward" connection of the two accepters, as follows.



- Note that these constructions all preserve the condition of a single accepting state, so they may be applied repeatedly.

The Result Stated Formally

Theorem: Given any regular expression r, there is an algorithm to construct an NFA M with $\mathcal{L}(M) = \mathcal{L}(r)$.

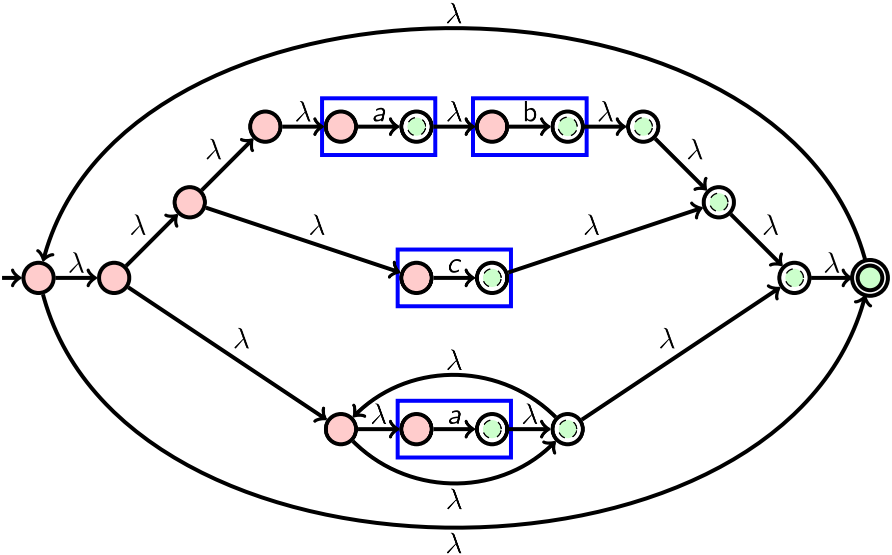
Proof: Just apply the constructions just illustrated repeatedly to the regular expression "bottom up". $\square$

Corollary: Given any regular expression r, there is an algorithm to construct a DFA M with $\mathcal{L}(M) = \mathcal{L}(r)$.

Proof: First construct the NFA using the above method, and then convert it to a DFA. $\square$

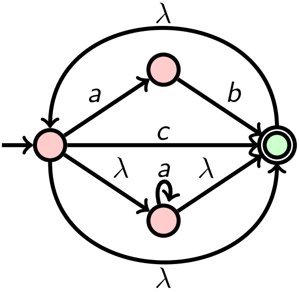
An Example of the RE-to-NFA Construction

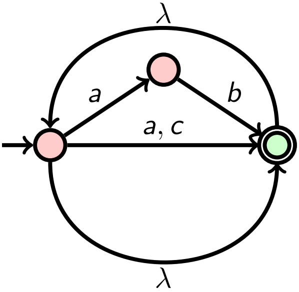
- Let $r = (((a \cdot b) + c) + a^*)^*$.



Simplification for a Particular Example

- The formal construction often results in an automaton which is more complex than necessary.
- Here are simpler solutions for $r = (((a \cdot b) + c) + a^*)^*$.

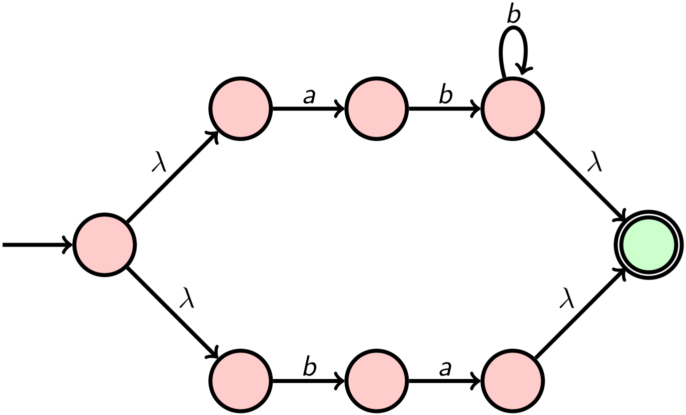




- The solution on the left is a direct simplification of the result of the algorithm.
- The solution on the right requires further analysis of the RE.

Another Example

- $r = abb^* + ba$.



Construction of an NFA from an RE

- Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA.
- Assume, without loss of generality, that the states of M are numbered, beginning with 0.
  - $Q = \{q_0, q_1, \ldots, q_n\}$.
- Define R_{ij}^k to be the set of all $\alpha \in \Sigma^*$ such that there is a computation

$$(q_i, \alpha) \vdash_M (q_{m_1}, \alpha_1) \ldots \vdash_M (q_{m_p}, \alpha_p) \vdash_M (q_j, \lambda)$$

  for which $\{q_{m_1}, \ldots, q_{m_p}\} \subseteq \{q_0, \ldots, q_k\}$.
- Thus, the computation is only allowed to go through intermediate states indexed by $0, 1, \ldots, k$.
- It is easy to see that $\mathcal{L}(M) = \bigcup_{q_j \in F} R_{0j}^n$.
- The idea of the construction is to build R_{ij}^n recursively and construct the RE from the pieces.

Recursive Construction of the RE of an NFA

- First, note that

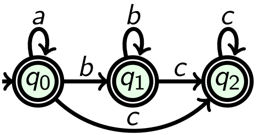
$$R_{ij}^{-1} = \begin{cases} \{x \in \Sigma \cup \{\lambda\} \mid q_j \in \delta(q_i, x)\} & \text{if } i \neq j \\ \{a \in \Sigma \mid q_j \in \delta(q_i, a)\} \cup \{\lambda\} & \text{if } i = j \end{cases}$$

- Now the inductive step:

$$\begin{array}{rlr} R_{ij}^{k+1} = & R_{ij}^{k} & \text{only } \{q_0, \ldots, q_k\}. \\ & \cup R_{i(k+1)}^{k} \cdot R_{(k+1)j}^{k} & \text{exactly one } q_{k+1} \\ & \cup R_{i(k+1)}^{k} \cdot R_{(k+1)(k+1)}^{k} \cdot R_{(k+1)j}^{k} & \text{exactly two } q_{k+1}\text{'s} \\ & \cup R_{i(k+1)}^{k} \cdot (R_{(k+1)(k+1)}^{k})^2 \cdot R_{(k+1)j}^{k} & \text{exactly three } q_{k+1}\text{'s} \\ & \vdots & \\ & \cup R_{i(k+1)}^{k} \cdot (R_{(k+1)(k+1)}^{k})^m \cdot R_{(k+1)j}^{k} & \text{exactly } m\ q_{k+1}\text{'s} \\ & \vdots & \\ = & \cup R_{i(k+1)}^{k} \cdot (R_{(k+1)(k+1)}^{k})^* \cdot R_{(k+1)j}^{k} & \text{any number of } q_{k+1}\text{'s} \end{array}$$

Recursive Construction of the RE of an NFA Continued

- The algorithm constructs an RE r_{ij}^k from R_{ij}^k and is best illustrated by example.



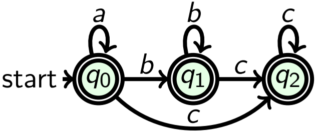
k	-1	0	1
r_{00}^k	$a+\lambda$	a^*	a^*
r_{01}^k	b	a^*b	a^*bb^*
r_{02}^k	c	a^*c	$a^*c + a^*bb^*c = a^*b^*c$
r_{10}^k	$\emptyset$	$\emptyset$	$\emptyset$
r_{11}^k	$b+\lambda$	$b+\lambda$	b^*
r_{12}^k	c	c	b^*c
r_{20}^k	$\emptyset$	$\emptyset$	$\emptyset$
r_{21}^k	$\emptyset$	$\emptyset$	$\emptyset$
r_{22}^k	$c+\lambda$	$c+\lambda$	$c+\lambda$

$$r_{00}^2 = r_{00}^1 + r_{02}^1 \cdot (r_{22}^1)^* \cdot r_{20}^1 = a^* + a^*b^*c \cdot (c+\lambda)^* \cdot \emptyset \qquad = a^*$$

$$r_{01}^2 = r_{01}^1 + r_{02}^1 \cdot (r_{22}^1)^* \cdot r_{21}^1 = a^*bb^* + a^*b^*c \cdot (c+\lambda)^* \cdot \emptyset \qquad = a^*bb^*$$

$$r_{02}^2 = r_{02}^1 + r_{02}^1 \cdot (r_{22})^* \cdot r_{22}^1 = a^*b^*c + a^*b^*c \cdot (c+\lambda)^* \cdot (c+\lambda) \qquad = a^*b^*cc^*$$

Recursive Construction of the RE of an NFA Continued



$$
\begin{aligned}
r_{00}^2 &= r_{00}^1 + r_{02}^1 \cdot (r_{22}^1)^* \cdot r_{20}^1 = a^* + a^*b^*c \cdot (c+\lambda)^* \cdot \emptyset && = a^* \\
r_{01}^2 &= r_{01}^1 + r_{02}^1 \cdot (r_{22}^1)^* \cdot r_{21}^1 = a^*bb^* + a^*b^*c \cdot (c+\lambda)^* \cdot \emptyset && = a^*bb^* \\
r_{02}^2 &= r_{02}^1 + r_{02}^1 \cdot (r_{22})^* \cdot r_{22}^1 = a^*b^*c + a^*b^*c \cdot (c+\lambda)^* \cdot (c+\lambda) && = a^*b^*cc^*
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{L}(M) &= \mathcal{L}(r_{00}^2 + r_{01}^2 + r_{02}^2) \\
&= \mathcal{L}(a^* + a^*bb^* + a^*b^*cc^*) \\
&= \mathcal{L}(a^*(\lambda + bb^* + b^*cc^*)) \\
&= \mathcal{L}(a^*(b^* + b^*cc^*)) \\
&= \mathcal{L}(a^*b^*(\lambda + cc^*)) \\
&= \mathcal{L}(a^*b^*c^*)
\end{aligned}
$$

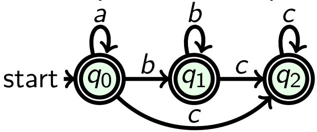
A Better Algorithm

- The algorithm to convert an RE to an NFA is very tedious to execute.

Question: Is there a better algorithm for humans to use?

Answer: Yes

- There is an algorithm which solves the equations algebraically, using *formal power series*.
- For the previous example, the equations are:



$$\begin{aligned} X_0 &= aX_0 + bX_1 + cX_2 + \lambda \\ X_1 &= bX_1 + cX_2 + \lambda \\ X_2 &= cX_2 + \lambda \end{aligned}$$

- It is similar in principle to solving linear equations in algebra.
- However, it requires the development of the theory of formal power series and so will not be presented here.

The Main Result So Far

Theorem: Let L be a language over the alphabet Σ. The following statements are equivalent.

- $L = \mathcal{L}(M)$ for some DFA M.
- $L = \mathcal{L}(M)$ for some NFA M.
- $L = \mathcal{L}(r)$ for some RE r.

Furthermore, there are algorithms for converting between the three representations. $\square$