Lecture #4: Regular Operations and Regular Expressions
Parsing Regular Expressions

Introduction

Recall that if Σ is an alphabet that does not include any of the special symbols

$$\Sigma, \lambda, \emptyset, \cup, \circ, ^\star, (\text{ or }),$$

then a ***regular expression*** over Σ is a string of symbols over a larger alphabet,

$$\Sigma_{\text{regexp}} = \Sigma \cup \{\Sigma, \lambda, \emptyset, (,), \cup, \circ, ^\star\}$$

that can be formed using a finite number of applications of the following rules.

(a) σ is a regular expression over Σ, for every symbol $\sigma \in \Sigma$

(b) "λ" (that is, the string in $\Sigma^\star_{\text{regexp}}$ with length one, including the symbol λ) is a regular expression over Σ.

(c) "$\emptyset$" (that is, the string in $\Sigma^\star_{\text{regexp}}$ with length one, including the symbol $\emptyset$) is a regular expression over Σ.

(d) "Σ" (that is, the string in $\Sigma^\star_{\text{regexp}}$ with length one, including the symbol Σ) is a regular expression over Σ.

(e) If R_1 and R_2 are regular expressions over Σ then the string

$$(R_1 \cup R_2)$$

(that is, the concatenation of the symbol "(", the regular expression R_1, the symbol "$\cup$", the regular expression R_2, and the symbol "(") is also a regular expression over Σ.

(f) If R_1 and R_2 are regular expressions over Σ then the string

$$(R_1 \circ R_2)$$

(that is, the concatenation of the symbol "(", the regular expression R_1, the symbol "$\circ$", the regular expression R_2, and the symbol "(") is also a regular expression over Σ.

(g) If R is a regular expression over Σ then the string

$$(R)^\star$$

(that is, the concatenation of the symbol "(", the regular expression R, the symbol ")", and the symbol "$\star$") is also a regular expression over Σ.

One way to show that a string, $\omega \in \Sigma^\star_{\text{regexp}}$, is a regular expression over Σ, is to list a sequence of applications of the above rules that can be used to form Σ. Unfortunately, this involves guesswork — and it does not provide a reliable way to prove that a string in $\Sigma^\star_{\text{regexp}}$ *is not* a regular expression over Σ.

This document presents a reliable method to decide whether a given string $\omega \in \Sigma^\star_{\text{regexp}}$ is a regular expression over $\Sigma^\star$. It also introduces a useful data structure — a ***parse tree*** — corresponding to any regular expression, and describes how this can be obtained and used.

Useful Technical Results

Let Σ and Σ_{regexp} be as above. The following results concern a string

$$\omega = \sigma_1 \sigma_2 \dots \sigma_n \tag{1}$$

where $\sigma_1, \sigma_2, \dots, \sigma_n \in \Sigma_{\text{regexp}}$ (so that $\omega \in \Sigma^\star_{\text{regexp}}$) and $n \geq 2$.

Lemma 1. *Let ω be as shown at line* (1)*, and let k be an integer such that $1 \leq k \leq n-1$. Suppose that ω is a regular expression over Σ.*

(a) If σ_n is not the symbol "$\star$", then the number of copies of "(" in the prefix $\sigma_1\sigma_2 \dots \sigma_k$ is strictly greater than the number of copies of ")" in this prefix.

(b) If σ_n is not the symbol "$\star$", then the number of copies of "(" in ω is equal to the number of copies of ")" in ω.

(c) If σ_n is the symbol "$\star$" and $k \leq n-2$ then the number of copies of "(" in the prefix $\sigma_1\sigma_2 \dots \sigma_k$ is strictly greater than the number of copies of ")" in this prefix.

(d) If σ_n is the symbol "$\star$" then the number of copies of "(" in the prefix $\sigma_1\sigma_2 \dots \sigma_{n-1}$ is equal to the number of copies of ")" in this prefix.

Exercise: Prove this result by induction on n. The strong form of mathematical induction should be used.

Note that if ω is as shown at line (1), above, ω is a regular expression over Σ, and $\sigma_n =$ "$\star$", then $n \geq 4$, the substring

$$\mu = \sigma_2\sigma_3 \ldots \sigma_{n-2}$$

is also a regular expression over Σ, and $\omega = (\nu)^\star$. The following result concerns the only other nontrivial case (concerning a string whose length is greater than one).

Lemma 2. *Let ω be a regular expression over Σ that is as shown at line* (1)*, and suppose that $\sigma_n \neq$ "$\star$". Then there exists a unique integer k, such that $2 \leq k \leq n-1$, which satisfies the following properties.*

(a) The number of copies of "$($" in the string

$$\nu = \sigma_2\sigma_3 \ldots \sigma_{k-1}$$

is equal to the number of copies of "$)$" in the above string ν.

(b) Either $\sigma_k =$ "$\cup$" or $\sigma_k =$ "$\circ$".

Furthermore, if k is the (unique) integer described above then both of the substrings

$$\nu = \sigma_2\sigma_3 \ldots \sigma_{k-1} \quad \text{and} \quad \varphi = \sigma_{k+1}\sigma_{k+2} \ldots \sigma_{n-1}$$

of ω are regular expressions over Σ, so that $\omega = (\nu \cup \varphi)$ if $\sigma_k =$ "$\cup$", and $\omega = (\nu \circ \varphi)$ if $\sigma_k =$ "$\circ$".

Lemma 2 can be proved using the recursive definition of a regular expression over Σ (which can be used to describe the form that ω can have), Lemma 1, and a consideration of the case that $\sigma_k =$ " $\star$", given above. Its proof is also left as an ***exercise***.

Parse Trees

Definition

A ***parse tree*** for a regular expression ω over Σ is a rooted tree that can be defined as follows, for a given regular expression $\omega \in \Sigma^\star$:

(a) If ω is a symbol $\sigma \in \Sigma$ then the parse tree for ω has a single node, with label σ.

(b) If ω is the symbol "λ", then the parse tree for ω has a single node, with label "λ".

(c) If ω is the symbol "$\emptyset$", then the parse tree for ω has a single node, with label "$\emptyset$".

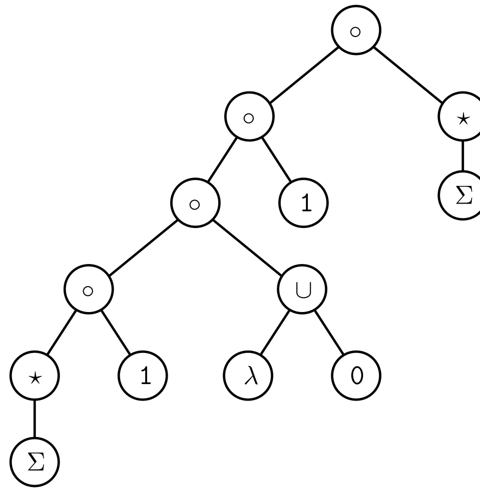


Figure 1: Parse Tree for the Regular Expression $(((((\Sigma^\star) \circ 1) \circ (\lambda \cup 0)) \circ 1) \circ (\Sigma)^\star)$

(d) If ω is the symbol "Σ", then the parse tree for ω has a single node, with label "Σ".

(e) If ω is a regular expression "$(\mu \cup \nu)$", for regular expressions μ and ν over Σ, then the parse tree for ω has a root with label '$\cup$" with two children. The first child is the root of a parse tree for μ, and the second child is the root of a parse tree for ν.

(f) If ω is a regular expression "$(\mu \circ \nu)$, for regular expressions μ and ν over Σ, then the parse tree for ω has a root with label "$\circ$" with two children. The first child is the root of a parse tree for μ, and the second child is the root of a parse tree for ν.

(g) If ω is a regular expression "$(\mu)^\star$, for a regular expression μ over Σ, then the parse tree for ω has a root with label "$\star$", with one child. The child is the root of a parse tree for μ.

For example, if $\Sigma = \{0, 1\}$, then the parse tree for the regular expression

$$(((((\Sigma^\star) \circ 1) \circ (\lambda \cup 0)) \circ 1) \circ (\Sigma)^\star)$$

is as shown in Figure 1.

Parsing

Consider, now, the following computational problem.

The "Parsing" Problem

Precondition: A string $\omega \in \Sigma^\star_{\text{regexp}}$ is given as input.

Postcondition: If ω is a regular expression over Σ then a parse tree for ω is returned as output. An empty tree is returned as output, otherwise.

It will be useful to have a solution for the following problem when developing a solution for the above one.

The "Splitting" Problem

Precondition: A string

$$\omega = \sigma_1\sigma_2 \dots \sigma_n \in \Sigma^\star_{\text{regexp}}$$

with length $n \geq 3$, such that $\sigma_1 =$ "$($" and $\sigma_n =$ "$)$" is given as input.

Postcondition: An integer k such that $1 \leq k \leq n$ is returned as output. If ω is a regular expression over Σ then $n \geq 5$ and the following properties are satisfied.

(a) $3 \leq k \leq n-2$ and either $\sigma_k =$ "$\cup$" or $\sigma_k =$ "$\circ$".

(b) The strings

$$\mu = \sigma_2\sigma_3 \dots \sigma_{k-1} \quad \text{and} \quad \nu = \sigma_{k+1}\sigma_{k+2} \dots \sigma_{n-1}$$

are regular expressions over Σ.

Lemma 2, above, can be used to develop and prove the correctness of an algorithm for the "Splitting" problem: It suffices to sweep over the input string, ω, in order to keep track of the difference between the number of copies of "$($" and "$)$" that have been seen, so far: If an integer k is found such that $1 \leq k \leq n$, σ_k is either "$\cup$" or "$\circ$", and there is exactly one more copy of "$($" than there is of "$)$", then this integer k should be returned as output. If no such integer k with these properties exists then ω cannot be a regular expression at all, and it suffices to return the value $k = 1$. Similarly — since ω begins with "$($" and ends with "$)$" — ω cannot be a regular expression (so that 1 can be returned) if $|\omega| \leq 4$.

Pseudocode for an algorithm that uses this strategy to solve the "Splitting" problem is given in Figure 2 on page 6.

Exercise — If you have completed a course in which proofs of correctness of algorithms with loops has been discussed, prove the correctness of the `splitting` algorithm.

```
splitting ( ω: Σ*_regexp ) {
 1.  integer n := |ω|
 2.  if (n ≥ 5) {
       // Suppose that ω = σ_1σ_2 ... σ_n
 3.    integer k := 0
 4     integer diff := 0
 5.    while (k < n)
 6.      k := k + 1
 7.      if ( σ_k == "(" ) {
 8.        diff := diff + 1
 9.      } else if ( σ_k == ")" ) {
10.        diff := diff − 1
         }
11.      if ((diff == 1) and ((σ_k == "∪") or (σ_k == "∘"))) {
12.        return k
         }
       }
13.    return 1
     } else {
14.    return 1
     }
}
```

Figure 2: An Algorithm for the "Splitting" Problem

Once an algorithm (like the `splitting` algorithm) that correctly solves the "Splitting" problem is available, an algorithm that recursively solves the "Parsing" problem is easy to describe. Pseudocode for one such algorithm is given in Figure 3 on page 7 and in Figure 4 on page 8.

The ***correctness*** of this recursive algorithm (as a solution for the "Parsing" problem) can be proved by induction on the length of the input string, ω, using the strong form of mathematical induction — and using Lemma 2, as needed, to argue that ω is *not* a regular expression over Σ (so that the output is correct) if the step at line 22 is reached and executed.

With a bit of work, one can also prove that this algorithm is also reasonably *efficient:* The number of executions of numbered steps, included in any execution of the `splitting` algorithm, is at most linear in the length of the input string. This can be used to prove (again, by induction on the length of the input string) that the number of executions of numbered steps, included in

```
parsing ( ω : Σ*_regexp ) {
 1.  integer n := |ω|
     // Suppose ω = σ1σ2 . . . σn
 2.  if (n == 0) {
 3.     Return an empty tree.
 4.  } else if ( n == 1 ) {
 5.     if ((σ1 ∈ Σ) or (σ1 ∈ {"λ", "∅", "Σ"})) {
 6.        Return a parse tree with size one, whose root has label σ1.
        } else {
 7.        Return an empty tree.
        }
 8.  } else if ( (n ≥ 4 ) and ( σ1 == "(" ) and ( σn−1 == ")" ) and ( σn == "*" ) ) {
 9.     Set μ to be the string σ1σ2 . . . σn−1, so that ω = ( μ )*
10.     Set T̂ to be the tree parsing(μ)
11.     if ( T̂ is not an empty tree ) {
12.        Return a parse tree whose root has label "⋆" and a single child — which is
           the root of the parse tree T̂.
        } else {
13.        Return an empty tree.
        }
```

Figure 3: Beginning of an Algorithm for the "Parsing" Problem

any execution of the `parsing` algorithm, is at most *quadratic* in the length of the input string. Thus, this is a "polynomial-time" algorithm.

Exercise: Suppose, again, that $\Sigma = \{0, 1\}$. Trace the execution of the `parsing` algorithm on the input string

$$(((((\Sigma^\star) \circ 1) \circ (\lambda \cup 0)) \circ 1) \circ (\Sigma)^\star)$$

in order to confirm that the parse tree shown in Figure 1, on page 4, would be returned as output.

14. } else if (($n \geq 5$) and (σ_1 == "(") and (σ_n == ")")) {
15. integer k := splitting(ω)
16. if ((σ_k == "$\cup$")) or (σ_k == "$\circ$")) {
17. Set μ to be the string $\sigma_2\sigma_3 \ldots \sigma_{k-1}$ and set ν to be the string $\sigma_{k+1}\sigma_{k+2} \ldots \sigma_{n-1}$, so that $\omega = (\ \mu\ \sigma_k\ \nu\ )$
18. Set T_L to be the parse tree parsing(μ)
19. Set T_R to be the parse tree parsing(ν)
20. if ((T_L is not an empty tree) and (T_R is not an empty tree)) {
21. Return a parse tree whose root has label σ_k, with two children: The left child is the root of the parse tree T_L, and the right child is the root of the parse tree T_R.
} else {
22. Return an empty tree.
}
} else {
23. Return an empty tree.
}
} else {
24. Return an empty tree.
}
}

Figure 4: Conclusion of an Algorithm for the "Parsing" Problem

Applications

Application: Construction of a Nondeterministic Finite Automaton with the Same Language

Consider, now, the following computational problem. Once again, this concerns alphabets Σ and Σ_{regexp} as described above.

Conversion to NFA

Precondition: A string $\omega \in \Sigma_{\text{regexp}}\star$ is given as input.

Postcondition: If ω is not a regular expression over Σ then an exception (which might be an `InvalidInputException`) is thrown. Otherwise a nondeterministic finite automaton

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

such that the language of ω is the same as the language of M, is returned as output.

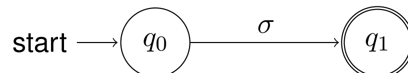
The algorithm for this problem, informally given below, will always return a nondeterministic finite automaton of the type described in Lemma 2 of the supplemental document “Proofs of Closure Properties”. That is, there will be no transitions into the NFA’s start state. The set of accepting states will be a set of size one — so that there will be exactly one “accepting state” — and there will be no transitions out of this accepting state.

In order to solve this problem, one can begin by applying the `parsing` algorithm, shown in Figures 3 and 4, to try to obtain a parse tree for the input string $\omega \in \widehat{\Sigma}^{\star}$. If an empty tree is returned then $\widehat{\omega}$ is not a regular expression over Σ at all, and the exception mentioned in the specification of this problem should be thrown.

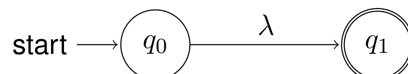
Let T_ω be the parse tree for ω that is obtained, otherwise.

The problem can be solved, for the case that ω *is* a regular expression over Σ, using a recursive algorithm that accepts a parse tree T for a given regular expression as input. The algorithm will proceed in different ways — all described in the supplemental document, “Proof of Equivalence Claim”. This, in turn, makes use of the details of various proofs of claims. The claims are stated in the lecture notes and the proofs are given in the supplemental document, “Proofs of Closure Properties”.

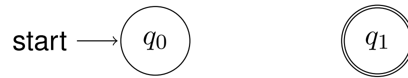
- If the root of T stores a symbol $\sigma \in \Sigma$ then ω is the symbol σ, the language of ω is the set $\{\sigma\}$, and a nondeterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ such that $Q = \{q_0, q_1\}$, $F = \{q_1\}$, and transitions are as follows, can be returned.



- If the root of T stores a symbol λ then ω is the symbol λ, the language of ω is the set $\{\lambda\}$, and a nondeterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ such that $Q = \{q_0, q_1\}$, $F = \{q_1\}$, and transitions are as follows, can be returned.



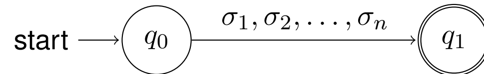
- If the root of T stores a symbol $\emptyset$ then ω is the symbol $\emptyset$, the language of ω is $\emptyset$, and a nondeterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ such that $Q = \{q_0, q_1\}$, $F = \{q_1\}$, and transitions are as follows, can be returned.



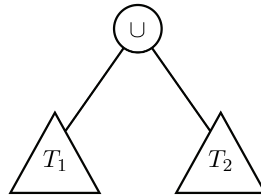
- If the root of T stores a symbol Σ then ω is the symbol Σ, the language of ω is Σ, and a nondeterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ such that $Q = \{q_0, q_1\})$, $F = \{q_1\}$, and — assuming that

$$\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$$

for a positive integer $n = |\Sigma|$ — transitions are as follows, can be returned.



- If the root of T stores a symbol $\cup$ then $\omega = (\omega_1 \cup \omega_2)$ for a pair of shorter regular expressions, ω_1 and ω_2, over Σ — and T is a shown below.



T_1 is a parse tree for the regular expression ω_1, and T_2 is a parse tree for the regular expression ω_2.

This algorithm can be recursively applied, with input ω_1, to compute a nondeterministic finite automaton $M_1 = (Q_1, \Sigma, \delta_1, q_{1,0}, F_1)$ such that $F_1 = \{q_{1,F}\}$ for a state $q_{1,F} \in Q_1$, that does not include any transitions into $q_{1,0}$ or out of $q_{1,F}$ — such that the language of M_1 is also the language of ω_1. The algorithm can also be recursively applied, with input ω_2, to compute a nondeterministic finite automaton $M_2 = (Q_2, \Sigma, \delta_2, q_{2,0}, F_2)$ such that $F_2 = \{q_{2,F}\}$ for a state $q_{2,F} \in Q_2$, that does not include any transitions into $q_{2,0}$ or out of $q_{2,F}$ — such that the language of M_2 is also the language of ω_2. Renaming states in Q_1 and Q_2 as needed it can be assumed that $q_0, q_F \notin Q_1$, $q_0, q_2 \notin Q_2$, and $Q_1 \cap Q_2 = \emptyset$.

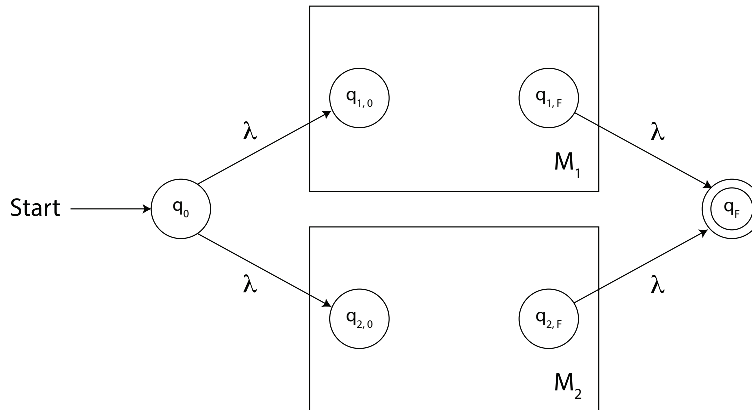


Figure 5: A Nondeterministic Finite Automaton for the Regular Expression $(\omega_1 \cup \omega_2)$

Now consider a nondeterministic finite automaton

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

that has M_1 and M_2 as components and whose structure is as shown in Figure 5. That is,

$$Q = \{q_0, q_F\} \cup Q_1 \cup Q_2,$$

the alphabet Σ is the same as for M_1 and M_2, the new state, q_0, is the start state,

$$F = \{q_F\},$$

and the transition function $\delta : Q \times \Sigma_\lambda \to \mathcal{P}(Q)$ is defined as follows.

- It is only possible to move from the new start state to one of the old start states, and no symbols are processed when doing this — so that

  $$\delta(q_0, \lambda) = \{q_{1,0}, q_{2,0}\}$$

  and

  $$\delta(q_0, \sigma) = \emptyset \quad \text{for every symbol } \sigma \in \Sigma.$$

- All transitions for states in Q_1, except for $q_{1,F}$, are the same in M as they were in M_1. That is,

  $$\delta(q, \sigma) = \delta_1(q, \sigma) \quad \text{for every state } q \in Q_1 \setminus \{q_{1,F}\} \text{ and for all } \sigma \in \Sigma_\lambda.$$

- The only transition out of $q_{1,F}$ is a λ-transition to the new accepting state. That is, $\delta(q_{1,F}, \lambda) = \{q_F\}$ and $\delta(q_{1,F}, \sigma) = \emptyset$ for all $\sigma \in \Sigma$.
- All transitions for states in Q_2, except for $q_{2,F}$, are the same in M as they were in M_2. That is,

$$\delta(q, \sigma) = \delta_2(q, \sigma) \quad \text{for every state } q \in Q_2 \setminus \{q_{2,F}\} \text{ and for all } \sigma \in \Sigma_\lambda.$$

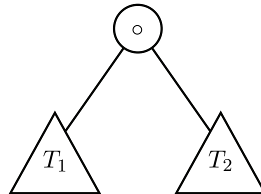
- The only transition out of $q_{2,F}$ is a λ-transition to the new accepting state. That is, $\delta(q_{2,F}, \lambda) = \{q_F\}$ and $\delta(q_{2,F}, \sigma) = \emptyset$ for all $\sigma \in \Sigma$.
- There are no transitions out of the new accepting state. That is, $\delta(q_F, \lambda) = \emptyset$ and $\delta(q_F, \sigma) = \emptyset$ for all $\sigma \in \Sigma$.

This nondeterministic finite automaton has a unique accepting state, with no transitions into the start state or out of the accepting state. It is similar to — but not, quite, the same as — the nondeterministic finite automaton described in the proof of Lemma #3 in the supplemental document “Proof of Closure Properties” — and it is a reasonably straightforward ***exercise*** to modify the proof of that lemma, in order to show that the language of the nondeterministic finite automaton in Figure 5 is

$$L(M_1) \cup L(M_2) = L(\omega_1) \cup L(\omega_2) = L((\omega_1 \cup \omega_2)) = L(\omega),$$

as desired.

• If the root of T stores a symbol $\circ$ then $\omega = (\omega_1 \circ \omega_2)$ for a pair of shorter regular expressions, ω_1 and ω_2, over Σ — and T is a shown below.



T_1 is a parse tree for the regular expression ω_1, and T_2 is a parse tree for the regular expression ω_2.

This algorithm can be recursively applied, with input ω_1, to compute a nondeterministic finite automaton $M_1 = (Q_1, \Sigma, \delta_1, q_{1,0}, F_1)$ such that $F_1 = \{q_{1,F}\}$ for a state $q_{1,F} \in Q_1$, that does not include any transitions into $q_{1,0}$ or out of $q_{1,F}$ — such that the language of M_1 is also the language of ω_1. The algorithm can also be recursively applied, with input ω_2, to compute a nondeterministic finite automaton $M_2 = (Q_2, \Sigma, \delta_2, q_{2,0}, F_2)$ such that $F_2 = \{q_{2,F}\}$ for a state $q_{2,F} \in Q_2$, that does not include any transitions into $q_{2,0}$

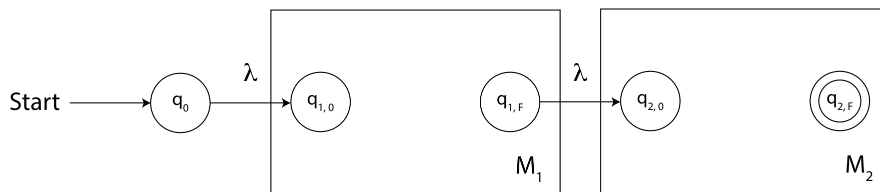


Figure 6: A Nondeterministic Finite Automaton with Language $\omega_1 \circ \omega_2$

or out of $q_{2,F}$ — such that the language of M_2 is also the language of ω_2. Renaming states in Q_1 and Q_2 as needed it can be assumed that $q_0, q_F \notin Q_1$, $q_0, q_2 \notin Q_2$, and $Q_1 \cap Q_2 = \emptyset$.

Now consider a nondeterministic finite automaton

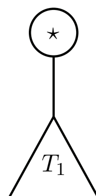
$$M = (Q, \Sigma, \delta, q_0, F)$$

that has M_1 and M_2 as components and whose structure is as shown in Figure 6, above — so that this is the nondeterministic finite automaton considered in the proof of Lemma #4 in the supplemental document "Proof of Closure Properties". This nondeterministic finite automaton has a unique accepting state, with no transitions into the start state or out of the accepting state. Since it is the nondeterministic finite automaton considered in the above proof, it suffices to review the details of this proof to confirm that its language is

$$L(M_1) \circ L(M_2) = L(\omega_1) \circ L(\omega_2) = L((\omega_1 \circ \omega_2)) = L(\omega),$$

as desired.

- In the only remaining case, the root of T stores a symbol $\star$ and $\omega = (\omega_1)^\star$ for a shorter regular expression, ω_1, over Σ — and T is a shown below.



T_1 is a parse tree for the regular expression ω_1.

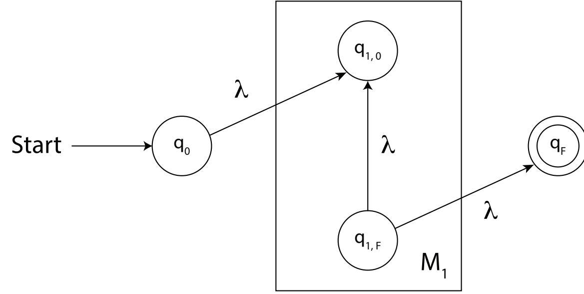


Figure 7: A Nondeterministic Finite Automaton with Language $L(\omega_1)^\star$

Once again, this algorithm can be recursively applied, with input ω_1, to compute a nondeterministic finite automaton $M_1 = (Q_1, \Sigma, \delta_1, q_{1,0}, F_1)$ such that $F_1 = \{q_{1,F}\}$ for a state $q_{1,F} \in Q_1$, that does not include any transitions into $q_{1,0}$ or out of $q_{1,F}$ — such that the language of M_1 is also the language of ω_1. Renaming states in Q_1 as needed it can be assumed that $q_0, q_F \notin Q_1$.

Now consider a nondeterministic finite automaton

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

that has M_1 as a component and whose structure is as shown in Figure 7, above. That is,

$$Q = \{q_0, q_F\} \cup Q_1,$$

the alphabet Σ is the same as for M_1, the new state, q_0, is the start state,

$$F = \{q_F\},$$

and the transition function $\delta : Q \times \Sigma_\lambda \to \mathcal{P}(Q)$ is defined as follows.

- It is only possible to move from q_0 to the start state, $q_{1,0}$, for M_1, and no symbols are processed when doing this — so that

  $$\delta(q_0, \lambda) = \{q_{1,0}\}$$

  and

  $$\delta(q_0, \sigma) = \emptyset \quad \text{for every symbol } \sigma \in \Sigma.$$

- Transitions out of all states in Q_1, except for $q_{1,F}$, are the same for M as they are for M_1. That is,

  $$\delta(q, \sigma) = \delta_1(q, \sigma) \quad \text{for all } q \in Q \setminus q_{1,F} \text{ and for all } \sigma \in \Sigma_\lambda.$$

– There are λ-transitions from $q_{1,F}$ back to M_1's start state and to M's accepting state — and no other transitions out of $q_{1,F}$ are defined. That is,

$$\delta(q_{1,F}, \lambda) = \{q_{1,0}, q_F\}$$

and

$$\delta(q_1, F), \sigma) = \emptyset \quad \text{for all } \sigma \in \Sigma.$$

– There are no transitions out of q_F. That is,

$$\delta(q_F, \sigma) = \emptyset \quad \text{for all } \sigma \in \Sigma_\lambda.$$

This nondeterministic finite automaton has a unique accepting state, with no transitions into the start state or out of the accepting state. It resembles the nondeterministic finite automaton considered in the proof of Lemma #5 in the supplemental document "Proof of Closure Properties" — and the modification of this proof, to establish that the language of M is

$$(L(M_1))^\star = (L(\omega_1))^\star = L((\omega_1)^\star) = L(\omega)$$

(as desired) is left as an exercise.

Exercises:

1. Use the above information to write pseudocode for an algorithm to solve the "Conversion to NFA" problem and sketch a proof that your algorithm is correct. You do not need to worry about the *encodings* of inputs, outputs, or intermediate values (so that these could be included in a computer program) when you solve this problem.

2. Let $\omega \in \Sigma^\star_{\text{regexp}}$ be a regular expression over Σ, such that ω is a string with length n. Prove that the parse tree for ω, produced using the algorithm shown in Figures 3 and 4, has at most $(n + 1)/2$ nodes. (That is, its "size" is at most $(n + 1)/2$.)

3. Let T be a parse tree, for a regular expression over Σ, with at most k nodes. Prove that the nondeterministic finite automaton (with the same language) that is generated, using the process described above, has at most $2k$ states.

4. Let $\omega \in \Sigma^\star_{\text{regexp}}$ be a regular expression over Σ that is also a string with length n. Use the above results to show that the nondeterministic finite automaton with the same language as ω, produced using the above construction, has at most $n + 1$ states.

Application: Deciding Membership of a String in the Language of a Regular Expression

Finally, consider the following computational problem — which also concerns alphabets Σ and $\widehat{\Sigma}$, as described above.

Membership in the Language of a Regular Expression

Precondition: A string $\omega \in \Sigma_{\text{regexp}}^\star$ and a string $\mu \in \Sigma^\star$ are given as input.

Postcondition: If ω is not a regular expression over Σ then an exception (which might be an `InvalidInputException`) is thrown. Otherwise, `true` is returned if μ is in the language of the regular expression ω, and `false` is returned otherwise.

Let $\omega \in \Sigma_{\text{regexp}}^\star$. An algorithm for the above problem, with input ω, can begin execution by using a solution for the above problem, "Conversion of an NFA", to throw the kind of exception that is needed, if ω *is not* a regular expression over Σ, and to produce a nondeterministic finite automaton

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

such that $L(M) = L(\omega)$, if ω *is* a regular expression over ω. As noted above M will have at most one accepting state, and $|Q|$ will be at most one more the length of ω, whenever ω is a regular expression over Σ.

The algorithm for "Membership in the Language of a Regular Expression" could then be completed (again, for the case that the input is a regular expression) by implementing an algorithm that decides whether the given string $\mu \in \Sigma^\star$ is in the language of the regular expression of the nondeterministic finite automaton, M, that has been generated. As noted below, an algorithm for this problem can be developed using information introduced in the lecture notes and supplemental material for Lecture #3:

- To begin (in a "pre-computing stage") one should compute the λ-closure, $\mathit{Cl}_\lambda(q)$, for every state $q \in Q$. The supplemental document, "Computation of λ-Closures", for Lecture #3, describes an algorithm for this computation.

- As mentioned in the lecture notes for Lecture #3,

  $$\delta^\star(q_0, \lambda) = \mathit{Cl}_\lambda(q_0) \tag{2}$$

  and

  $$\delta^\star(q_0, \nu \cdot \sigma) = \bigcup_{r \in \delta^\star(q_0, \nu)} \left(\bigcup_{s \in \delta(r, \sigma)} \mathit{Cl}_\lambda(s) \right) \tag{3}$$

for every string $\nu \in \Sigma^\star$ and every symbol $\sigma \in \Sigma$. These equations can be used to develop a simple (and efficient) algorithm to compute $\delta^\star(q_0, \mu)$, assuming that the λ-closures, above, and a way to evaluate the transition function δ is available.

- Once $\delta^\star(q_0, \mu)$ has been computed it remains only to check whether $q_F \in \delta^\star(q_0, \mu)$, where q_F is the unique accepting state in M: μ is in the language of the regular expression ω — and `true` should be returned — if $q_F \in \delta^\star(q_0, \mu)$, and μ is *not* in the language of the regular expression ω — and `false` should be returned — otherwise.

Results from the supplemental documents, described above, can be applied to show that an algorithm, based on the above outline, would correctly solve the "Membership in the Language of a Regular Expression". Furthermore, if reasonable decisions are made about how to *encode* symbols in an (arbitrarily large) alphabet Σ, parse trees, and nondeterministic finite automata, then this algorithm can be implemented to produce a polynomial-time algorithm for this problem (that would also be reasonably efficient, in practice).