Finite Automata

Plan. We will define a kind of Finite Automaton called a **finite recognizer**. (FR)

- Given a regular expression we can create an equivalent nondeterministic FR (Thompson's construction)
- Given a nondeterministic FR, we can create a deterministic FR (subset construction)
- Given a nondeterministic FR, we can create a regular expression

Together these results will show that regular expressions, deterministic finite recognizers, and nondeterministic finite recognizers are three formalisms that define the same set of language.
This set is called the **regular languages**

Nondeterministic Finite State Recognizers

We will define a kind of finite state automaton for defining languages.

Syntax

A **Nondeterministic Finite State Recognizer** (**NDFR**) is a quintuple $A = (S, Q, q_{\text{start}}, F, T)$ consisting of

- A finite alphabet set S
- A finite set of states Q
- An *initial state* $q_{\text{start}} \in Q$
- A set of *accepting states* $F \subseteq Q$
- A set of *labelled transitions* $T \subseteq Q \times \left(S^1 \cup \{\epsilon\}\right) \times Q$

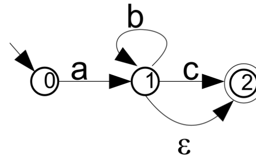
Example 1 *As a first example of an NDFR we have* $A = (S, Q, q_{\text{start}}, F, T)$ *where*

- $S = \{\text{'a'}, \text{'b'}, \text{'c'}\}$
- $Q = \{0, 1, 2\}$
- $q_{\text{start}} = 0$
- $F = \{2\}$
- $T = \{(0, \text{“a”}, 1), (1, \text{“b”}, 1), (1, \text{“c”}, 2), (1, \epsilon, 2)\}$

At the heart of an NDFR is an edge-labelled directed multigraph $(Q, T, \leftarrow, \rightarrow, \lambda)$ where

$$\overleftarrow{(q, a, r)} = q \qquad \overrightarrow{(q, a, r)} = r \qquad \lambda((q, a, r)) = a$$

We generally draw this graph to represent the NDFR with an extra arrow to indicate the start states and the members of F drawn as double circles.



An NDFR

Semantics

Each NDFR state $q \in Q$ in an NDFR $A = (S, Q, q_{\text{start}}, F, T)$ describes a language $L_A(q) \subseteq S^*$.
The language described by the automaton A is the language of its start state: $L(A) = L_A(q_{\text{start}})$.

Inductive definition approach

Two rules define the languages described by q:

- If $q \in F$ then $\epsilon \in L_A(q)$.
- If $(q, s, r) \in T$ and $t \in L_A(r)$ then $s\hat{\ }t \in L_A(q)$

Meta-rule: A string is in the language associated with a state iff it can be shown to be by a *finite* number of applications of these two rules.

Path through the graph approach

There is another, equivalent, way to define the language associated with each state. We can define that $w \in L_A(q)$ iff there is a path from q to some state in F such that w is the catenation of labels along the path.
Formally: $w \in L_A(q)$ exactly if for some $n \in \mathbb{N}$, there are

- n strings $w_0, w_1, \ldots, w_{n-1}$, such that
- $w_0\,\hat{}\,w_1\,\hat{}\,\ldots\,\hat{}\,w_{n-1} = w$, and there are
- $n + 1$ states $q_0, q_1, ..., q_n$, such that
- $q_0 = q$, $q_n \in F$ and
- for each $i \in \{0, ..n\}$, $(q_i, w_i, q_{i+1}) \in T$.

Example 2 *We can see that "abb" $\in L_A(0)$ for the NDFR above by observing the path*

$$(0, \text{"a"}, 1)\ (1, \text{"b"}, 1)\ (1, \text{"b"}, 1)\ (1, \epsilon, 2)$$

Extending the the L_A function to sets of states.

If $R = \{r_0, r_1, ...\}$ is a subset of Q, define $L_A(R)$ to be the union of $L_A(r_0) \cup L_A(r_1) \cup \cdots$. I.e.

$$L_A(R) = \bigcup r \in R \cdot L_A(r)$$

Systematic state renaming

The states don't really matter. We can systematically replace any set of states with almost any other.

Let A be an NDFR $A = (S, Q, q_{\text{start}}, F, T)$, $\dot{Q}$ be any set such that $\left|\dot{Q}\right| \geq |Q|$, and f be a one-one total function from Q to $\dot{Q}$. Then $L(A) = L(\dot{A})$, where $\dot{A} = (S, \dot{Q}, \dot{q}_{\text{start}}, \dot{F}, \dot{T})$ is derived from A as follows:

- $\dot{q}_{\text{start}} = f(q_{\text{start}})$
- $\dot{F} = \{q \in F \cdot f(q)\}$
- $\dot{T} = \{(q, a, r) \in T \cdot (f(q), a, f(r))\}$

For implementation purposes, we often rename states so that the set of states is $\{0, .. |Q|\}$.

All regular languages are described by NDFRs

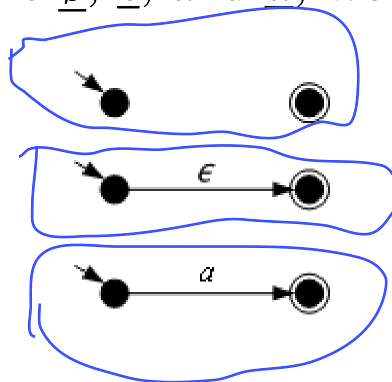
Next we will see that every language described by a regular expression can also be described by an NDFR.

To do this we will show how an arbitrary regular expression x can be translated into an NDFR $A(x)$ such that $L(x) = L(A(x))$.

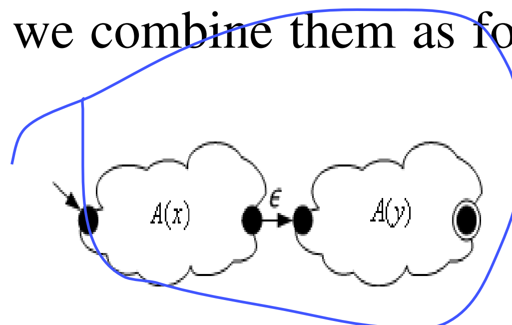
This translation is called *Thompson's construction.*

Each NDFR constructed by Thompson's construction will have a start state, that has no incoming transitions, and one accepting state, that has no outgoing transitions.

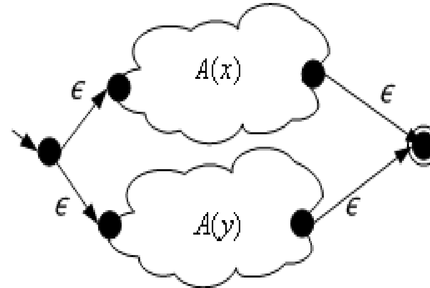
- For regular expressions $\underline{\emptyset}$, $\underline{\epsilon}$, and $\underline{a}$, we can construct automata:



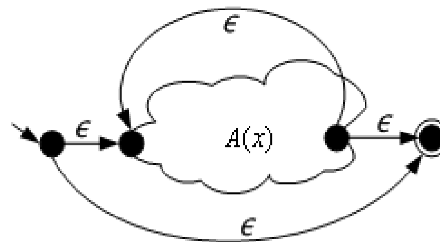
- For regular expression $\underline{(}x\underline{;}y\underline{)}$, we first construct $A(x)$ and $A(y)$, using Thompson's construction. If necessary, the states are renamed so that the two automata have disjoint state sets. Finally, we combine them as follows to get $A(\underline{(}x\underline{;}y\underline{)})$.



- For regular expression $\underline{(}x\,\underline{|}\,y\underline{)}$, we first construct $A(x)$ and $A(y)$, using Thompson's construction. If necessary, the states are renamed so that the two automata have disjoint state sets. Finally, we combine them and add two new states and four new transitions, as follows to get $A\left(\underline{(}x\,\underline{|}\,y\underline{)}\right)$.



- For regular expression $\underline{(}x\underline{^*)}$, we first construct $A(x)$, using Thompson's construction. Finally, we add two new states and two new transitions as follows to get $A\left(\underline{(}x\underline{^*)}\right)$.



Recognition algorithms

A two-finger algorithm

We want an algorithm for the following problem. Given a string w_0 and an NDFR $A = (S, Q, q_{\text{start}}, F, T)$ determine whether or not $w_0 \in L(A)$.

Suppose you put one finger of your right hand on a state in the NDFR and one finger of your left hand at a point in the string.

Initially your fingers are on q_{start} and at the start of the string.

- r — the state your right finger is on.
- w — the portion of the string to the right of your left finger.

At each step you can do one of the following

- Stop
- Follow an ϵ-labelled transition with your right finger
  - $r := q$ where q is some state such that $(r, \epsilon, q) \in T$
- Follow a symbol-labelled transition
  - Move your right finger along some transition labelled by the next symbol; move your left finger one place to the right.
  - $r, w := q, s$ where q, s, and a are such that
    $$w = [a]\hat{}s \text{ and } (r, [a], q) \in T$$

The algorithm ends after any number of moves

var $r := q_{\text{start}}$;
var $w := w_0$;
while true
  if $\mathfrak{true}$ then
    jump out of the loop
  $\square$ there is an ϵ-labelled transition out of r then
    $r := q$, where q is any state such that
    $(r, \epsilon, q) \in T$
  $\square$ $\|w\| > 0$ and there is an labelled transition out of r labelled with $[w(0)]$ then
    $r, w := q, \text{tail}(w)$, where q is any state such that
    $(r, [w(0)], q) \in T$;
$f := w = \epsilon \wedge r \in F$

You can think of this algorithm as the rules for a game of solitaire.

- You win the game if $f = \mathfrak{true}$ at the end of the game.
- You lose the game if $f = \mathfrak{false}$ at the end of the game

There are a number of places where you may have to make choices

(a) When to stop playing (if a move can be made)

(b) Which kind of move to make (if a both kinds are possible)

(c) Which ϵ-labelled transition to take (if more than one is available)

(d) Which $w(0)$-labelled transition to take (if more than one is available)

If you are lucky (or good) at making choices, you can always use this algorithm to show that a string is in the language described by the NDFR.

However, if luck is not with you, you may find the algorithm fails even if it could have succeeded. Furthermore, the algorithm could loop forever by following a cycle of epsilon transitions.

The best we can say for the algorithm is that

- if it succeeds ($f = \mathfrak{true}$), then the string is certainly in the language, but
- if it fails ($f = \mathfrak{false}$) or loops forever, we don't know, as we might have made a poor choice

Magical coins

Imagine, though, that you have a magical lucky coin. Whenever you have to make a choice, you flip the magic coin.

The magic coin has the following property: Whenever it is tossed to make a decision:

- If one option leads to certain failure and the other to possible success, the coin will pick the option that can lead to success.
- In any other circumstance, the result of the coin is arbitrary.

The two-finger algorithm used in conjunction with the magical lucky coin will succeed, if the string is in the language, and will either fail or loop forever if the string is not.

Let's take luck and magic out of the algorithm.

The nondeterministic recognition algorithm

First some definitions.

Let $A = (S, Q, q_{\text{start}}, F, T)$ be an NDFR, $r \in Q$, $R \subseteq Q$, and $a \in S$

- ϵ-closure(r), is the set of states reachable from r using 0 or more transitions labelled with ϵ.

  $r \in \epsilon$-closure(r) and

  if $p \in \epsilon$-closure(r) and $(p, \epsilon, q) \in T$ then $q \in \epsilon$-closure(r)

- ϵ-closure(R) is the set of states reachable from any $r \in R$ using only 0 or more transitions labelled with ϵ.

  $$\epsilon\text{-closure}(R) = \bigcup_{r \in R} \epsilon\text{-closure}(r)$$

  Note that $R \subseteq \epsilon$-closure(R).

- $\delta(r, a)$ is the set of states reachable from r by a single transition labelled a.

  $$\delta(r, a) = \{q \in Q \mid (r, [a], q) \in T\}$$

- $\delta(R, a)$ is the set of states reachable from any $r \in R$ using a single transition labelled a.

  $$\delta(R, a) = \bigcup_{r \in R} \delta(r, a)$$

Suppose we have a string "abc". If you follow the two-finger algorithm, what states could be the current state after reading this string, i.e. what states could your finger be on?

- You start with q_{start} as the current state but, even before reading the 'a', you can follow ϵ transitions, and so, after reading zero symbols, the current state can change to any of the states in ϵ-closure(q_{start}). Let R_0 be this set.
- After reading the 'a', the current state can become any state connected to a state in R_0 by a transition labelled 'a', thus

  $$\delta(R_0, \text{'a'})$$

  but you can then follow ϵ transitions to reach other states. Thus after reading the 'a' the current state could be any state in

  $$R_1 = \epsilon\text{-closure}(\delta(R_0, \text{'a'}))$$

- Similarly after reading "ab", the current state can be any of the states

  $$R_2 = \epsilon\text{-closure}(\delta(R_1, \text{'b'}))$$

- Finally after reading "abc", the current state can be any of the states in

  $$R_3 = \epsilon\text{-closure}(\delta(R_2, \text{'c'}))$$

  Thus "abc" is in the language iff any of these states are accepting, i.e., if

  $$R_3 \cap F \neq \emptyset$$

This is how the recognition algorithm works.

We can construct an algorithm for recognizing a string w_0 with an NDFR

$$\langle\, f' = (w_0 \in L(A)) \rangle$$

We use variables $w \in S^*$ and $R \subseteq Q$. The invariant is that[1]

$$w_0 \in L(A) \text{ if and only if } w \in L_A(R)$$

and that R is closed under ϵ transitions: $R = \epsilon\text{-closure}(R)$.

The nondeterministic recognition algorithm

input: a string w_0 and an NDFR $A = (S, Q, q_{\text{start}}, F, T)$

output: a boolean f

specification: $\langle f' = (w_0 \in L(A)) \rangle$

```
var w := w0·
var R := ε-closure(q_start)·
// inv: (w0 ∈ L(A)) = (w ∈ L_A(R))
// inv: R = ε-closure(R)
while w ≠ ε ∧ R ≠ ∅ do (
    let a, s | w = [a]^s ·
    R := ε-closure(δ(R, a)) ;
    w := s ) ;
f := (R ∩ F ≠ ∅)
```

This gives us a way of recognizing regular expressions: Convert the regular expression to an NDFR and then run the above algorithm.

The disadvantage is that the running time of each iteration is proportional to the size of R. The number of iterations is $\|w_0\|$. Since R can be as big as Q, the time taken can be roughly proportional to $|Q| \times \|w_0\|$.

[1] Recall that $L_A(R)$ is the union of all $L_A(r)$ for $r \in R$.

Correctness (optional)

The correctness of the recognition algorithm boils down to four simple theorems.

In all cases $A = (S, Q, q_{\text{start}}, F, T)$, $R \subseteq Q$, $a \in S$, and $s \in S^*$:

Theorem: $L(A) = L\left(\epsilon\text{-closure}(q_{start})\right)$

Theorem:

$s \in L(R)$ if and only if $s \in L_A(\epsilon\text{-closure}(R))$

Theorem: If $R = \epsilon\text{-closure}(R)$ then

$[a]\hat{}\, s \in L_A(R)$ if and only if $s \in L_A(\delta(R, a))$

Theorem: If $R = \epsilon\text{-closure}(R)$ then

$\epsilon \in L_A(R)$ if and only if $R \cap F \neq \emptyset$

Exercise: Prove these theorems.

Functional approach to recognition (optional)

We can succinctly write the recognition algorithm as a functional program.

Define a function $\rho : S^* \overset{\text{tot}}{\to} 2^Q$.

$$\begin{aligned} \text{recognize}(w) \;=\; & (\rho(w) \cap F \neq \emptyset) \text{ where} \\ & \rho \quad : \quad S^* \overset{\text{tot}}{\to} 2^Q \\ & \rho(\epsilon) = \epsilon\text{-closure}(q_{\text{start}}) \\ & \rho(s \hat{}\, [a]) = \epsilon\text{-closure}(\delta(\rho(s), a)) \end{aligned}$$

Theorem $s \hat{}\, t \in L(A)$ if and only if $t \in L(\rho(s))$.

Then $w \in L(A)$ if and only if $\rho(w) \cap F \neq \emptyset$.

Deterministic Finite State Recognizers (DFRs)

A *Deterministic Finite State Recognizer (DFR)* is an NDFR such that

- There are no transitions labeled by ϵ
  Thus ϵ-closure$(q) = \{q\}$ for all $q \in Q$
- For each state-symbol pair (q, a), there is exactly one r such that $(q, [a], r) \in T$
  I.e. $|\delta(q, a)| = 1$, for all $q \in Q$ and $a \in S$

Under these restrictions, the size of R in the nondeterministic recognition algorithm always 1.

If we have a DFR, we can define $\bar{\delta}(q, a)$ such that $\delta(q, a) = \left\{\bar{\delta}(q, a)\right\}$. I.e. $\bar{\delta}(q, a)$ is the sole member of $\delta(q, a)$

Consider the recognition algorithm running when we have a DFR. The size of set R will always be 1
We can represent variable $R \subseteq Q$ by variable $r \in Q$:

$$R = \{r\}$$

This is a data transformation.

The deterministic recognition algorithm

input: a string w_0 and a DFR $A = (S, Q, q_{\text{start}}, F, T)$

output: a boolean f

specification: $\langle f' = (w_0 \in L(A)) \rangle$

```
var w := w_0
var r := q_start
f := true
// inv: w_0 ∈ L(A) ⇔ w ∈ L_A(r)
while w ≠ ϵ do (
    let a, s | w = [a]^s ·
    r := δ̄(r, a) ;
    w := s )
f := r ∈ F
```

By using numbers for states and symbols, we can represent $\bar{\delta}$ as an matrix.

Then the running time for this algorithm is proportional to the length of of w_0.

This gives us another approach to recognizing regular expressions, which is potentially very efficient.

- Convert the RE to an NDFR using Thompson's construction, or some other construction that accomplishes the same thing.
- Convert the NDFR to an equivalent DFR.
- Match using the deterministic recognition algorithm.

Next, we look at how to accomplish the second step.

From NDFRs to DFRs

If you try to design a DFR for a complex language, you may find it is far more difficult than designing an NDFR for the same language. It seems that NDFRs may be a more powerful tool for describing languages than are DFRs. However, in a sense this is not true. It turns out that for **any** NDFR A there is an DFR $\dot{A}$ such that $L(A) = L(\dot{A})$. Here is one way to do it.

The naive subset construction algorithm
input: An NDFR $A = (S, Q, q_{\text{start}}, F, T)$
output: A DFR $\dot{A} = (S, \dot{Q}, \dot{q}_{\text{start}}, \dot{F}, \dot{T})$
specification: $\left\langle L(\dot{A}') = L(A)\right\rangle$

$\dot{Q} := \{R \mid R \subseteq Q\}$ // So $\dot{Q}$ is the power set of Q.
$\dot{q}_{\text{start}} := \epsilon\text{-closure}(q_{\text{start}})$
$\dot{F} := \{R \subseteq Q \mid F \cap R \neq \emptyset\}$
$\dot{T} := \{a \in S, R \subseteq Q \cdot (R, a, \epsilon\text{-closure}(\delta(R, a)))\}$

(NB In this algorithm, the ϵ-closure and δ functions are with respect to A.)

Proof sketch: The idea is to show that, for each $R \subseteq Q$,

$$L_{\dot{A}}(R) = \bigcup_{q \in R} L_A(q)$$

The problem with this algorithm is that it always generates a large number of states: $\left|\dot{Q}\right| = 2^{|Q|}$. Some of these states may not be reachable from the start state, and hence contribute nothing.

As an optimization, we could consider only ϵ-closed sets as states. I.e. $\dot{Q} := \{R \mid R \subseteq Q \wedge R = \epsilon\text{-closure}(R)\}$.

There is a better algorithm that only generates states that are reachable from the start state.

The subset construction algorithm

input: An NDFR $A = (S, Q, q_{\text{start}}, F, T)$

output: A DFR $\dot{A} = (S, \dot{Q}, \dot{q}_{\text{start}}, \dot{F}, \dot{T})$

specification: $\left\langle L(\dot{A}') = L(A) \right\rangle$

What we do is to compute the set of all sets R that might arise in the nondeterministic recognition algorithm.

W is a set of DFR states that have been discovered, but not explored.

$\dot{Q}$ is the set of states that have been discovered and explored.

When we explore a state we find all its out-going transitions and all the states they go to.

$\dot{q}_{\text{start}} := \epsilon$-**closure**$(q_{\text{start}})$;
var $W := \{\dot{q}_{\text{start}}\}\cdot$
$\dot{Q} := \emptyset$; $\dot{T} := \emptyset$; $\dot{F} := \emptyset$;
while $W \neq \emptyset$ **do** (
  let $\dot{q} \mid \dot{q} \in W\cdot$
  $W := W - \{\dot{q}\}$;
  $\dot{Q} := \dot{Q} \cup \{\dot{q}\}$;
  if $\dot{q} \cap F \neq \emptyset$ **then** $\dot{F} := \dot{F} \cup \{\dot{q}\}$ **else skip** ;
  for each $a \in S$ **do** (
    let $\dot{r} = \epsilon\text{-closure}(\delta(\dot{q}, a))\cdot$
    $\dot{T} := \dot{T} \cup \{(\dot{q}, [a], \dot{r})\}$;
    if $\dot{r} \notin \dot{Q}$ **then** $W := W \cup \{\dot{r}\}$ **else skip**))

(NB In this algorithm, the ϵ-closure and δ functions are with respect to A.)

Example: Find a DFR for $L(\underline{\text{‘x’; ‘y’}^* \mid \text{‘x’}^*\text{; ‘y’}})$.

After running the subset construction algorithm we can systematically rename the states in $\dot{Q}$ with small natural numbers so that $\dot{T}$ can be efficiently represented with an array.
Now we can 'efficiently' recognize regular expressions

- Translate the r.e. to an NDFR
- Translate the NDFR to a DFR
- Replace subsets with numbers
- Optionally minimize the number of states in the DFR
- Execute the deterministic recognition algorithm

Note that the NDFR will be about the same size as the regular expression, but the number of states in the DFR can be *exponential* in the number of states of the NDFR. This is why I put quotes around 'efficiently'.
An alternative approach is

- Translate the r.e. to an NDFR
- Execute the nondeterministic recognition algorithm

The first approach may be best if the regular expression is not too large and you intend to execute the recognition algorithm many times for the same r.e., or on a large complex text. The Unix program grep uses this approach.
The second approach may be best if the r.e. is so large that exponential blow-up is worrying or if speed of constructing the machine is more important than speed of executing it. The Unix program fgrep uses the second approach (for speed), as do many lexer generators (e.g. JavaCC).

Regular Languages

Let’s make the following definitions (some are temporary).

- A **regular language** is a language described by some regular expression
- An **NDFR language** is a language described by some NDFR
- A **DFR language** is a language described by some DFR

We have shown that any regular expression can be translated to an equivalent NDFR and any NDFR to an equivalent DFR and so we know

regular languages $\subseteq$ NDFR languages $\subseteq$ DFR languages

Furthermore any DFR is an NDFR and so every DFR language must also be an NDFR language:

regular languages $\subseteq$ NDFR languages $=$ DFR languages

In the next section, we will see that any NDFR can be translated into an equivalent regular expression and so

regular languages $=$ NDFR languages $=$ DFR languages

Once we have done that, we no longer need the terms “NDFR language” and “DFR language,” we just use the term “regular language.”

From NDFRs to Regular Expressions

Next we look at how to convert any NDFR into an RE.
To help us do this, we will define:
A *Regular Expression Finite Recognizer (REFR)* is just like an NDFR, except that the transitions are labeled with regular expressions over S.
The language described by each state is defined by two rules

- If $q \in F$ then $\epsilon \in L_A(q)$
- If $(q, x, r) \in T$ then $L(x)\hat{}L_A(r) \subseteq L_A(q)$

Meta-rule: A string w is in $L(q)$ only if it can be proved so by finite application of the above 2 rules.

Equivalently, we can define that $w \in L_A(q)$ iff there is a path from q to some state in F such that w is in the language of the catenation of labels along the path.
I.e., $w \in L_A(q)$ iff for some $n \geq 0$, there are

- $n + 1$ states $q_0, q_1, ..., q_n$,
- n regular expressions $x_0, x_1, ..., x_{n-1}$,
- such that, for each $i \in \{0, 1, ..., n-1\}$, $(q_i, x_i, q_{i+1}) \in T$,
- $q_0 = q$, $q_n \in F$, and $w \in L(\underline{(}x_0\underline{;}x_1\underline{;} \ldots \underline{;}x_{n-1}\underline{)})$

The language defined by the automaton is the language of its start state: $L(A) = L_A(q_{\text{start}})$.

Any NDFR can be trivially translated to an REFR.
We will look at an algorithm for translating an arbitrary NDFR into a regular expression.

The NDFR to RE algorithm

input: An NDFR $A_0 = (S_0, Q_0, q_{\text{start}_0}, F_0, T_0)$

output: An RE x

specification: $\langle L(x') = L(A_0)\rangle$

We start by making an REFR copy, $A = (S, Q, q_{\text{start}}, F, T)$, of A_0

$$A := \text{convertNDFRtoREFR}\,(A_0)$$

As the algorithm modifies A, we maintain, as an invariant, $L(A) = L(A_0)$.

The rest of the algorithm consists of four steps

- Step 0. Ensure that there is one accepting state and that there are no transitions into the initial state or out of the accepting state.
- Step 1. Make sure each pair of nodes has no more than one transition between them.
- Step 2. Eliminate all nodes that are not initial or accepting.
- Step 3. Output the remaining regular expression.

Step 0

In this step we ensure that the REFR has

- no edges into its initial state q_{start},
- exactly one accepting state $F = \{q_{\text{final}}\}$ with $q_{\text{final}} \neq q_{\text{start}}$
- no edges out of its accepting state.

This step is easily accomplished by adding (if needed) a new initial state, a new accepting state, and ϵ-labelled transitions.

Step 1

In this step we ensure that each pair of nodes has at most one transition between them

for all pairs of states q and r in Q
  coalesceTransitions(q, r)
where coalesceTransitions is defined by
procedure coalesceTransitions(q, r) **is**
  if there is more than one transition from q to r
    let $x_0, x_1, ..., x_{n-1}$ be the labels on those transitions
    remove all transitions from q to r from T
    add $(q, \underline{(}x_0\underline{|}x_1\underline{|}...\underline{|}x_{n-1}\underline{)}, r)$ to T

Step 2

Goal: reduce the number of states in the automaton to 2.

We eliminate states one at a time.

while there are more than two states **do** (
  let q be any state that is not initial nor accepting ·
  eliminate(q))

The loop invariant for this repetition is that the REFR has

- a single accepting state, q_{final}, with $q_{\text{final}} \neq q_{\text{start}}$
- no transition to its initial state nor from its accepting state
- at most one transition between any two states, and
- $L(A) = L(A_0)$

Eliminating state q is as follows:

procedure eliminate(q) **is**
  for all p and r in Q such that $p \neq q$ and $r \neq q$ and
  there are transitions from p to q and from q to r·
    let x be such that $(p, x, q) \in T$·
    let z be such that $(q, z, r) \in T$·
    if there is a transition from q to q **then** (
      let y be such that $(q, y, q) \in T$·
      add $(p, \underline{(}x\underline{;}\,\underline{(}y^{*}\underline{)};z\underline{)}, r)$ to T)
    else
      add $(p, \underline{(}x\underline{;}z\underline{)}, r)$ to T ;
    coalesceTransitions(p, r) ;
  remove from T all transitions either to or from q ;
  remove q from Q

Step 3

At this point we have two states, q_{start} and q_{final}, and one or zero transitions

- If there is a transition $(q_{\mathrm{start}}, y, q_{\mathrm{final}})$ then $x := y$
- Otherwise output $x := \underline{\emptyset}$

In summary

- Regular expressions, DFRs, NDFRs, and REFRs are formalisms that all can express the same set of languages: the regular languages.
- All have the same limitation: Fixed, finite memory. Thus this theory is very important for hardware designers as well as software designers.
- DFRs are efficient for recognition.
- For a given language, the smallest NDFR or regular expression can be far smaller than the smallest DFR.
- Thus converting an NDFR or RE or REFR to a DFR may entail an ‘explosion’ in the number of states. The DFR may be unacceptably big.
- Regular expressions, being textual, are easy to integrate into user-dialogs (e.g., search dialogs on websites, in Eclipse, UltraEdit, and vi) programming languages (e.g. Perl, JavaScript, sed, lex, JavaCC), and libraries (e.g., java.util.regexp, regex.h).
- Regular expressions are very convenient for expressing many languages, while NDFRs occasionally give elegant solutions to problems where regular expressions do not. REFRs give the best of both worlds. (For example, C/Java style comments.)
- Example: JavaCC’s lexical analyzer generator uses a kind of REFR for input and NDFRs for implementation.

Not all languages are regular

Here we will look at how to prove that not all languages are regular.

As an example we will prove that the language LP of balanced parentheses is not regular.

Or alphabet is $S = \{\text{‘(’}, \text{‘)’}\}$.

Some strings in the language: ϵ, “()”, “()()”, “((()())())”

Some strings not in the language: “)”, “(”, “((())”, “((()))))”

Aside.

Recall that in a DFA every state has one outgoing transition for each symbol.

And there are no ϵ transitions.

For any DFA $A = (S, Q, q_{\text{start}}, F, T)$ define a function

$$\begin{aligned} \sigma(q, \epsilon) &= q \\ \sigma(q, [a]\hat{\ }t) &= \sigma(r, t), \text{ where } \{r\} = \delta(q, a) \end{aligned}$$

So $\sigma(q, s)$ is the state the DFA is in if it starts in q and reads s.

Note that

- (0) $s\hat{\ }t \in L(A)$ if and only if $\sigma(\sigma(q_{\text{start}}, s), t) \in F$.
- (1) For any infinite sequence of states, some state must occur at least twice.
- (2) For any infinite sequence of strings s_0, s_1, s_2, ..., there must be two different numbers i and j such that $\sigma(q_{\text{start}}, s_i) = \sigma(q_{\text{start}}, s_j)$.

end of aside.

Now we prove that LP is not regular by contradiction.
Proof: Assume (falsely) that LP is regular.
Then there is a DFA $A = (S, Q, q_{\text{start}}, F, T)$ for LP.
Consider strings $s_0 = \epsilon$, $s_1 =$ "(", $s_2 =$ "((", $s_3 =$ "(((", ...
And strings $t_0 = \epsilon$, $t_1 =$ ")", $t_2 =$ "))", $t_3 =$ ")))", ...
Clearly $s_i \hat{} t_j \in LP$ if and only if $i = j$.
By (2
) there are numbers i and j such that $i \neq j$ and $\sigma(q_{\text{start}}, s_i) = \sigma(q_{\text{start}}, s_j)$.
Let $q = \sigma(q_{\text{start}}, s_i) = \sigma(q_{\text{start}}, s_j)$.
Since $s_i \hat{} t_i \in LP$, we have $\sigma(q, t_i) \in F$, from (0).
Therefore $\sigma(\sigma(q_{\text{start}}, s_j), t_i) \in F$.
And so $s_j \hat{} t_i \in LP$, by (0). But this is not true. Contradiction.
We can only conclude that there is no DFA for LP, and so LP is not regular.