NFA defined

Sipser pages 47 - 54

NFA

- A Non-deterministic Finite-state Automata (NFA) is a language recognizing system similar to a DFA.
- It supports a level of non-determinism. I.e. At some points in time it is possible for the machine to take on many next-states.
- Non-determinism makes it easier to express certain kinds of languages.

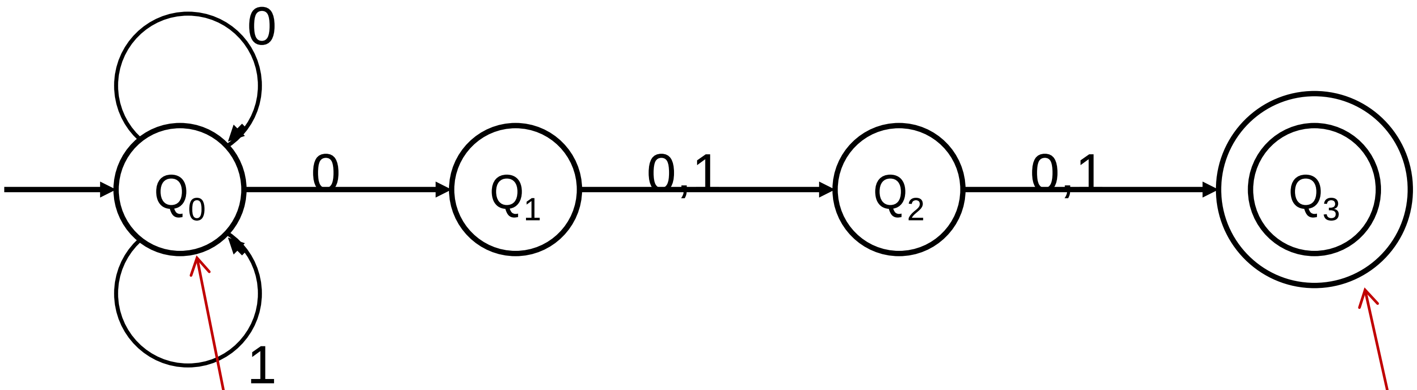
Nondeterministic Finite Automata (NFA)

- When an NFA receives an input symbol a, it can make a transition to zero, one, two, or even more states.
  - each state can have multiple edges labeled with the same symbol.

- An NFA accepts a string w iff there exists a path labeled w from the initial state to one of the final states.
  - In fact, because of the non-determinism, there may be many states labeled with w

Example N1

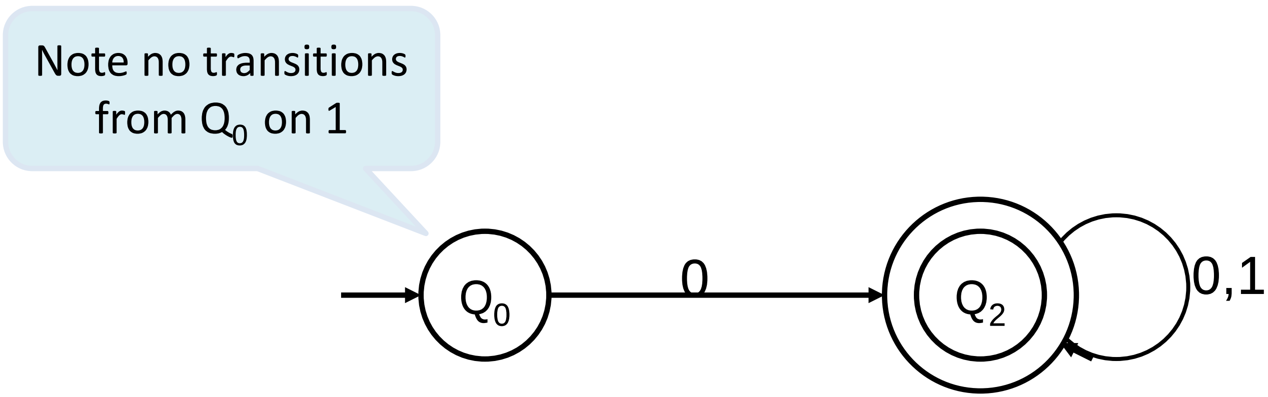
- The language of the following NFA consists of all strings over `{0,1}` whose 3rd symbol from the right is `0`.



- Note Q_0 has multiple transitions on 0 and Q_3 has no transitions on both 0 and 1

Example N2

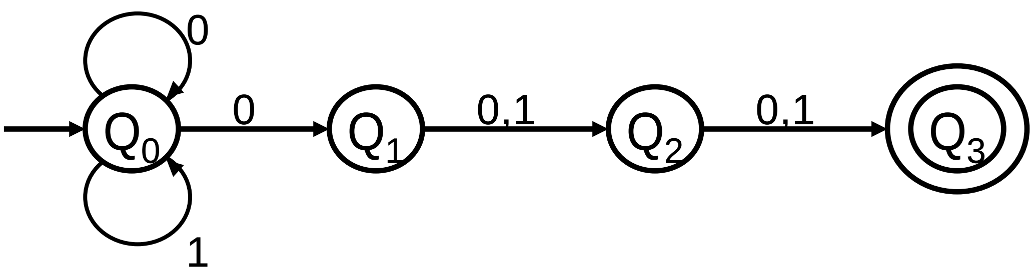
- The NFA N_2 accepts strings beginning with `0`.



- Note Q_0 has no transition on 1
  - It is acceptable for the transition function to be undefined on some input elements for some states.

NFA Processing

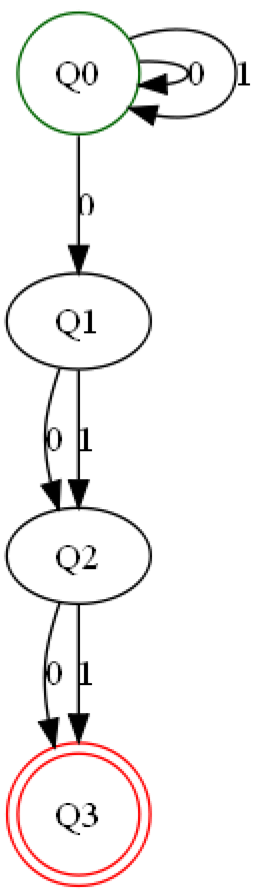
- Suppose N_1 receives the input string `0011`. There are three possible execution sequences:
- $q_0 \longrightarrow q_0 \longrightarrow q_0 \longrightarrow q_0 \longrightarrow q_0$
- $q_0 \longrightarrow q_0 \longrightarrow q_1 \longrightarrow q_2 \longrightarrow q_3$
- $q_0 \longrightarrow q_1 \longrightarrow q_2 \longrightarrow q_3$



- Only the second finishes in an accept state. The third even gets stuck (cannot even read the fourth symbol).
- As long is there is at least one path to an accepting state , then the string is accepted.

Input = 0011

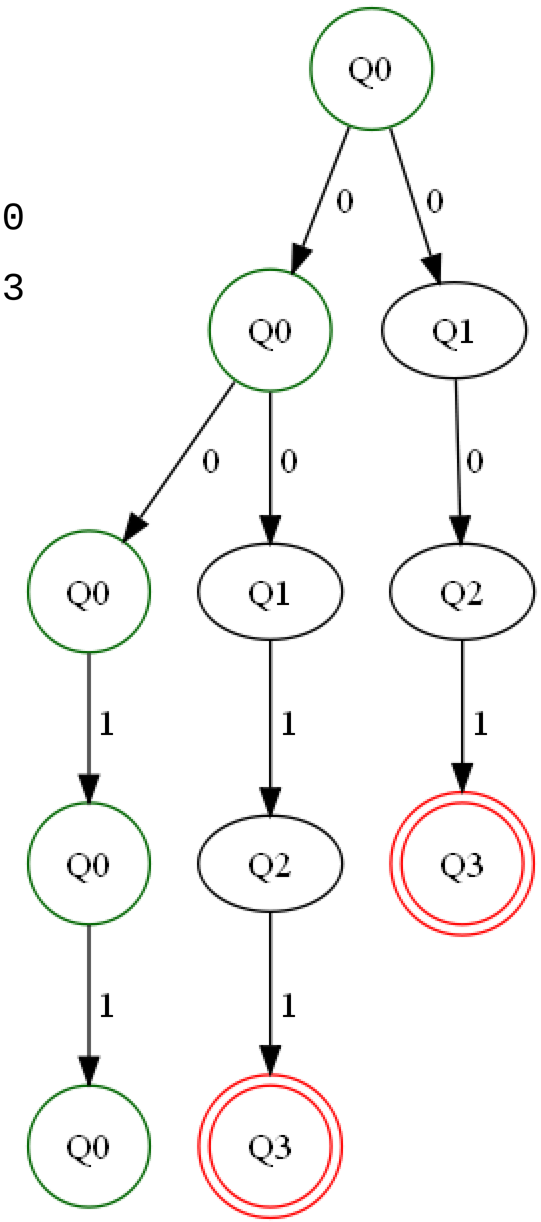
NFA



$$
\begin{array}{ccccccccc}
 & 0 & & 0 & & 1 & & 1 & \\
q_0 & \longrightarrow & q_0 & \longrightarrow & q_0 & \longrightarrow & q_0 & \longrightarrow & q_0 \\
q_0 & \longrightarrow & q_0 & \longrightarrow & q_1 & \longrightarrow & q_2 & \longrightarrow & q_3 \\
q_0 & \longrightarrow & q_1 & \longrightarrow & q_2 & \longrightarrow & q_3 & &
\end{array}
$$

Note, that this path is stuck at q3

Path Tree



A note about NFA’s

- In the Sipser text book (page 53) the definition for an NFA is slightly different from what we will see on the next page.
- The NFA that Sipser defines, we call an NFAe.
  – It allows transitions on edges labeled with ε (the empty string)
- We talk about this in a separate set of notes.

This is a simpler version of the definition on page 53-54 of Sipser. We disallow transitions on ε, and we changed the way we index the string.

Formal Definition

- An NFA is a quintuple A= (Q,Σ,δ,s,F) , where the first four components are as in a DFA, and the transition function produces values in $P(Q)$ (the power set of Q) instead of Q. Thus

$$\delta: Q \times \Sigma \longrightarrow P(Q)$$

note that δ returns a set of states!
It might return the emptyset!

- A NFA $A=(Q,\Sigma,\delta,s,F)$, *accepts* a string $\mathbf{w_1w_2...w_n}$ (an element of Σ^*) iff there exists a sequence of states $\mathbf{r_1r_2...r_nr_{n+1}}$ such that

1. $\mathbf{r_1 = s}$
2. $\mathbf{r_{i+1} \in \delta(r_i,w_i)}$
3. $\mathbf{r_{n+1} \in F}$

Compare with DFA

A DFA = $(\mathbf{Q},\Sigma,\delta,\mathbf{q_0},\mathbf{F})$, *accepts* a string $\mathbf{w}$ = "$w_1w_1...w_n$" iff

There exists a sequence of states $[r_0, r_1, \ldots r_n]$ with 3 conditions

1. $r_0 = q_0$
2. $\delta(r_i,w_{i+1}) = r_i+1$
3. $r_n \in F$

The extension of the transition function

- Let an NFA $A=(Q,\Sigma,\delta,s,F)$
- The extension $\underline{\delta} : Q \times \Sigma^* \longrightarrow P(Q)$ extends δ so that it is defined over a string of input symbols, rather than a single symbol. It is defined by
  - $\underline{\delta}(q,\varepsilon)=\{q\}$
  - $\underline{\delta}(q,x:xs) = \cup_{p\in\delta(q,x)} \underline{\delta}(p,xs)$

  Compute this by taking the union of the sets

  $\underline{\delta}(p,xs)$, where p varies over all states in the set

  $\delta(q,x)$

    - First compute $\delta(q,x)$, this is a set, call it S.
    - for each element, p in S, compute $\underline{\delta}(p,xs)$,
    - Union all these sets together.

Intuition

- At any point in the walk over a string, such as "000" the machine can be in a set of states.
- To take the next step, on a character 'c', we create a new set of states. All those reachable from any of the old sets on a single 'c'

$\underline{\delta}(q,\varepsilon)=\{q\}$
$\underline{\delta}(q,x:xs) = \bigcup_{p\in\delta(q,x)} \underline{\delta}(p,xs)$

Consider computing $\underline{\delta}(Q_0,001)$

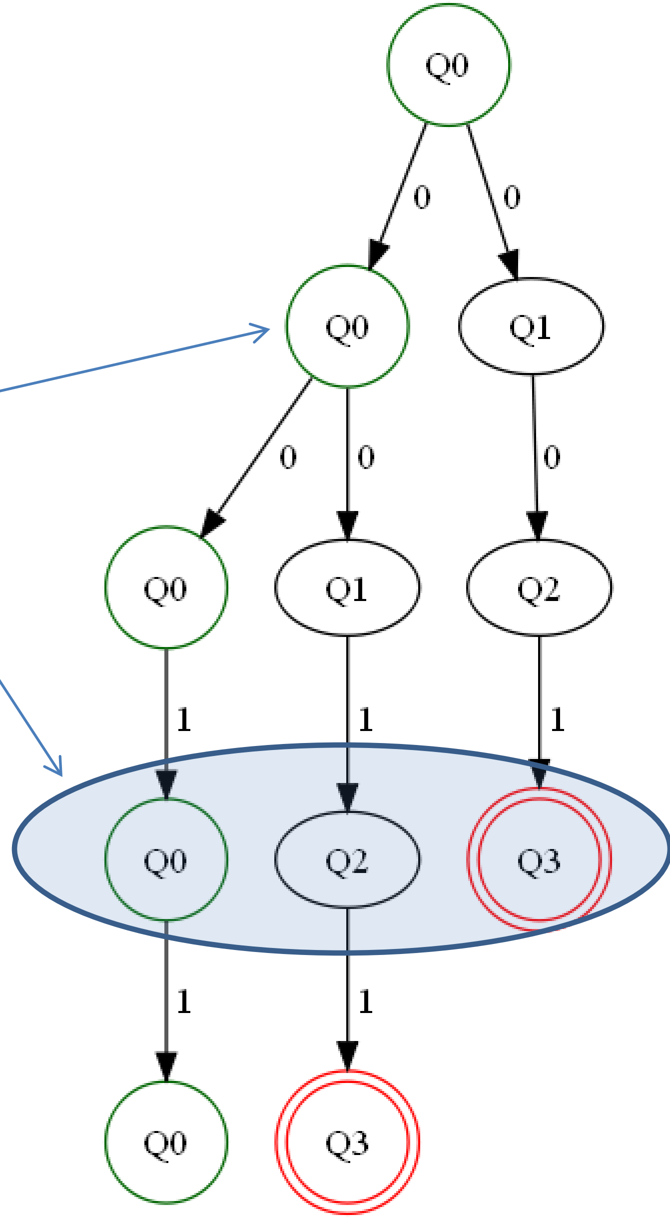
The answer will be $\{Q_0,Q_2,Q_3\}$

Start by one-step computing
$\delta(Q_0,0)=\{Q_0,Q_1\}$

So for each of Q_0,Q_1 recursively many-step compute

$$\underline{\delta}(Q_0,01) = \{Q_0,Q_2\}$$
$$\underline{\delta}(Q_1,01) = \{Q_3\}$$

Then union them together!



Another NFA Acceptance Definition

- An NFA accepts a string w iff $\underline{\delta}(s,w)$ contains a final state. The language of an NFA N is the set $L(N)$ of accepted strings:

- $L(N) = \{w \mid \underline{\delta}(s,w) \cap F \neq \varnothing\}$

- **Compare this with the 2 definitions of DFA acceptance in last weeks lecture.**

A DFA = $(\mathbf{Q},\Sigma,\delta,\mathbf{q_0},\mathbf{F})$, *accepts* a string $\mathbf{w} = \text{“}w_1 w_1 \ldots w_n\text{”}$ iff

There exists a sequence of states $[r_0, r_1, \ldots r_n]$ with 3 conditions

1. $r_0 = q_0$
2. $\delta(r_i, w_{i+1}) = r_i+1$
3. $r_n \in F$

A DFA $= (\mathbf{Q},\Sigma,\delta,\mathbf{q_0},\mathbf{F})$ *accepts* a string $\mathbf{w}$ iff $\underline{\delta}(\mathbf{q_0},w) \in \mathbf{F}$

More formally
$L(A)=\{w \mid \underline{\delta}(Start(A),w) \in Final(A)\}$

Implementation

- Implementation of NFAs has to be deterministic, using some form of backtracking to go through all possible executions .

- Any thoughts on how this might be accomplished?

In Haskell

```
data NFA q s =
  NFA [q]                      -- states
      [s]                      -- symbols
      (q -> s -> [q])          -- trans
      q                        -- start
      [q]                      -- accept states
```

Compare with DFA

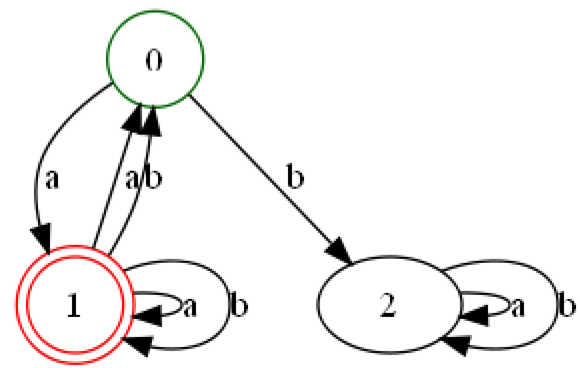
```
data DFA q s =
  DFA [q]                -- states
      [s]                -- symbols
      (q -> s -> q)      -- trans
      q                  -- start state
      [q]                -- accept states
```

Path acceptance

```haskell
allSeq xs 0 = []
allSeq xs 1 = [[x] | x <- xs ]
allSeq xs n = [ y:ys | ys <- allSeq xs (n-1), y <- xs]

cond1 nfa (r:rs) = r == (start nfa)
cond1 nfa [] = False

cond2 nfa [] [r] = True
cond2 nfa (w:ws) (r1:r2:rs) =
   (elem r2 (trans nfa r1 w)) && (cond2 nfa ws (r2:rs))
cond2 nfa _ _ = False

cond3 nfa [r] = isFinal nfa r
cond3 nfa (r:rs) = cond3 nfa rs
cond3 nfa _ = False

cond nfa ws path = cond1 nfa path &&
                   cond2 nfa ws path &&
                   cond3 nfa path

accept1 nfa ws = any (cond nfa ws) paths
  where paths = allSeq (states nfa) (1 + length ws)
```

w = “$w_1 w_1 \ldots w_n$” iff
There exists a sequence of states $[r_0, r_1, \ldots r_n]$ with 3 conditions

1. $r_0 = q_0$
2. $\delta(r_i, w_{i+1}) = r_i+1$
3. $r_n \in F$



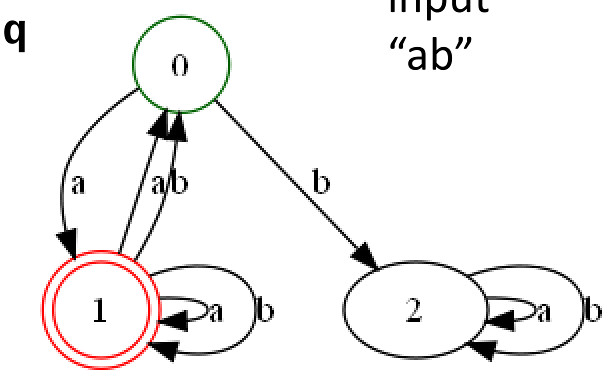
String = “ab”

Seq	C1	C2	C3
[0,0,0]=	T	F	F
[1,0,0]=	F	F	F
[2,0,0]=	F	F	F
[0,1,0]=	T	T	F
[1,1,0]=	F	T	F
[2,1,0]=	F	F	F
[0,2,0]=	T	F	F
[1,2,0]=	F	F	F
[2,2,0]=	F	F	F
[0,0,1]=	T	F	T
[1,0,1]=	F	F	T
[2,0,1]=	F	F	T
[0,1,1]=	T	T	T
[1,1,1]=	F	T	T
[2,1,1]=	F	F	T
[0,2,1]=	T	F	T
[1,2,1]=	F	F	T
[2,2,1]=	F	F	T
[0,0,2]=	T	F	F
[1,0,2]=	F	T	F
[2,0,2]=	F	F	F
[0,1,2]=	T	F	F
[1,1,2]=	F	F	F
[2,1,2]=	F	F	F
[0,2,2]=	T	F	F
[1,2,2]=	F	F	F
[2,2,2]=	F	T	F

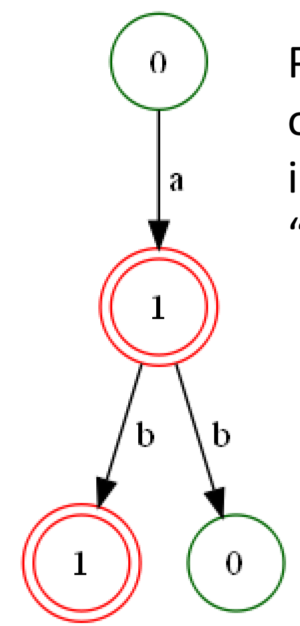
Transition extension acceptance

```
closure:: Ord q => NFA q s -> [q] -> s -> [q
closure nfa qs s =
   unionsL [trans nfa q s | q <- qs]

deltaBar nfa q [] = [q]
deltaBar nfa q (w:ws) =
   unionsL [ deltaBar nfa p ws
           | p <- closure nfa [q] w]

acceptNFA2 nfa ws =
   not(null(intersect last (accept nfa)))
  where last = deltaBar nfa (start nfa) ws
```

Trace input "ab"



Paths on input "ab"



deltaBar n2 (start n2) "ab " = [0,1]

Not(null(intersect [0,1] (accept n2))) = True