Finite Automata

Finite Automata

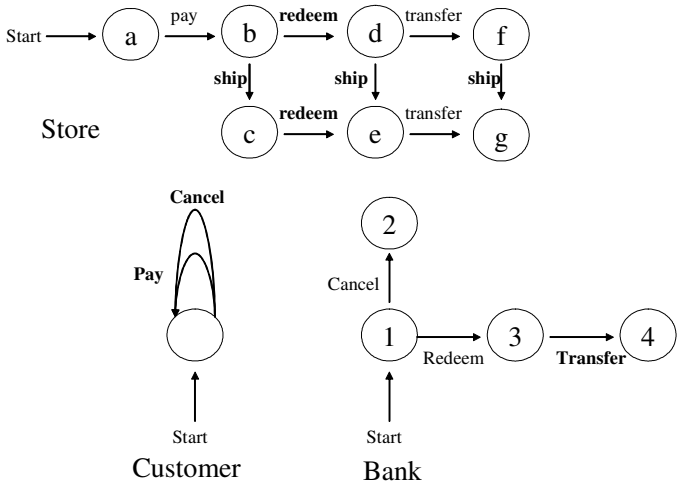
- Two types – both describe what are called **regular languages**
  - Deterministic (DFA) – There is a fixed number of states and we can only be in one state at a time
  - Nondeterministic (NFA) –There is a fixed number of states but we can be in multiple states at one time
- While NFA's are more expressive than DFA's, we will see that adding nondeterminism does not let us define any language that cannot be defined by a DFA.
- One way to think of this is we might write a program using a NFA, but then when it is "compiled" we turn the NFA into an equivalent DFA.

Informal Example

- Customer shopping at a store with an electronic transaction with the bank
  - The customer may *pay* the e-money or *cancel* the e-money at any time.
  - The store may *ship* goods and *redeem* the electronic money with the bank.
  - The bank may *transfer* any redeemed money to a different party, say the store.
- Can model this problem with three automata

Bank Automata

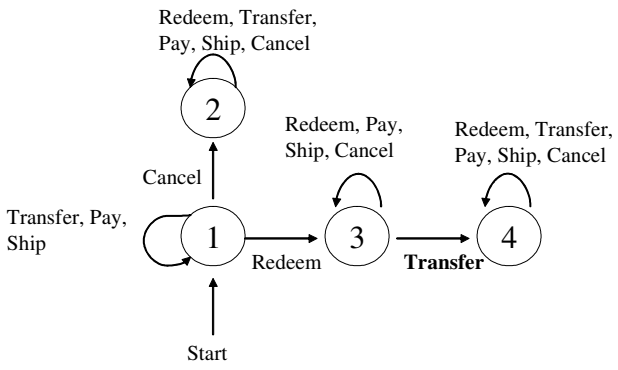
Actions in bold are initiated by the entity.
Otherwise, the actions are initiated by someone else and received by the specified automata



Ignoring Actions

- The automata only describes actions of interest
  - To be more precise, with a DFA (deterministic finite automaton) we should specify arcs for all possible inputs.
  - E.g., what should the customer automaton do if it receives a "redeem"?
  - What should the bank do if it is in state 2 and receives a "redeem"?
- The typical behavior if we receive an unspecified action is for the automaton to **die**.
  - The automaton enters no state at all, and further action by the automaton would be ignored.
  - The best method though is to specify a state for all behaviors, as indicated as follows for the bank automaton.

Complete Bank Automaton



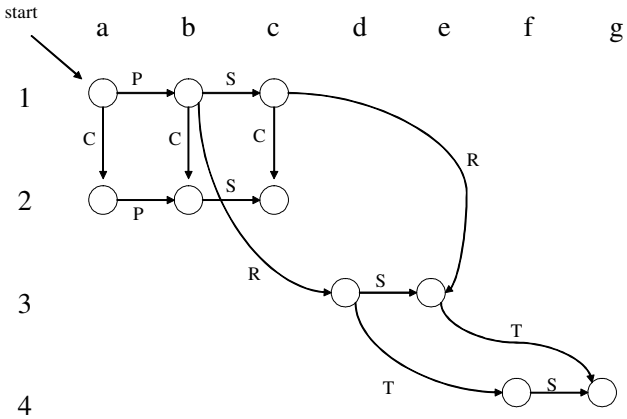
Bank

Ignores other actions that may be received

Entire System as Automaton

- When there are multiple automata for a system, it is useful to incorporate all of the automata into a single one so that we can better understand the interaction.
- Called the *product* automaton. The product automaton creates a new state for all possible states of each automaton.
- Since the customer automaton only has one state, we only need to consider the pair of states between the bank and the store.
  - For example, we start in state (a,1) where the store is in its start state, and the bank is in its start state. From there we can move to states (a,2) if the bank receives a cancel, or state (b,1) if the store receives a pay.
- To construct the product automaton, we run the bank and store automaton "in parallel" using all possible inputs and creating an edge on the product automaton to the corresponding set of states.

Product Automaton

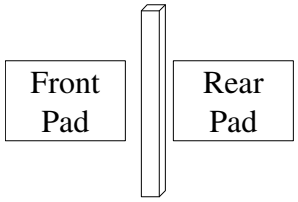


Product Automaton

- How is this useful? It can help validate our protocol.
- It tells us that not all states are reachable from the start state.
  - For example, we should never be in state (g,1) where we have shipped and transferred cash, but the bank is still waiting for a redeem.
- It allows us to see if potential errors can occur.
  - We can reach state (c, 2). This is problematic because it allows a product to be shipped but the money has not been transferred to the store.
  - In contrast, we can see that if we reach state (d, 3) or (e, 3) then the store should be okay – a transfer from the bank must occur
    - assuming the bank automaton doesn't "die" which is why it is useful to add arcs for all possible inputs to complete the automaton

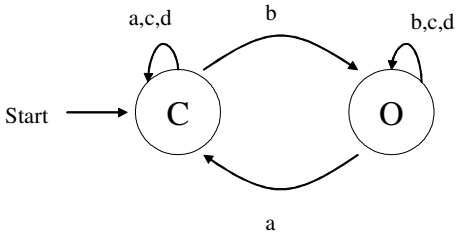
Simple Example – 1 way door

- As an example, consider a one-way automatic door. This door has two pads that can sense when someone is standing on them, a front and rear pad. We want people to walk through the front and toward the rear, but not allow someone to walk the other direction:



One Way Door

- Let's assign the following codes to our different input cases:
  - a - Nobody on either pad
  - b - Person on front pad
  - c - Person on rear pad
  - d - Person on front and rear pad
- We can design the following automaton so that the door doesn't open if someone is still on the rear pad and hit them:



Formal Definition of a Finite Automaton

1. Finite set of states, typically Q.
2. Alphabet of input symbols, typically Σ
3. One state is the start/initial state, typically q_0 // $q_0 \in Q$
4. Zero or more final/accepting states; the set is typically F. // $F \subseteq Q$
5. A transition function, typically δ. This function
   - Takes a state and input symbol as arguments.

One Way Door – Formal Notation

Using our formal notation, we have:

Q = {C, O} (usually we'll use q_0 and q_1 instead)
F = { } There is no final state
q_0 = C This is the start state
Σ = {a,b,c,d}
The transition function, δ, can be specified by the table:

	a	**b**	**c**	**c**
→ **C**	C	O	C	C
O	C	O	O	O

Write each δ(state,symbol)?

The start state is indicated with the →
If there are final accepting states, that is indicated with a * in the proper row.

Exercise

- Using Σ={0,1} a "clamping" circuit waits for a 1 input, and forever after makes a 1 output regardless of the input. However, to avoid clamping on spurious noise, design a DFA that waits for two 1's in a row, and "clamps" only then.
- Write the transition function in table format as well as graph format.

Formal Definition of Computation

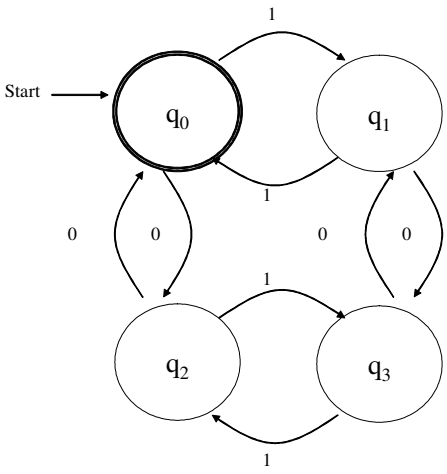
- Let M = $(Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \ldots w_n$ be a string where each w_i is a member of alphabet Σ.
- M **accepts** w if a sequence of states $r_0 r_1 \ldots r_n$ in Q exists with three conditions:
  1. $r_0 = q_0$
  2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for i=0, … , n-1
  3. $r_n \in F$

We say that M **recognizes** language A if A = {w | M accepts w }

In other words, the language is all of those strings that are accepted by the finite automata.

DFA Example

- Here is a DFA for the language that is the set of all strings of 0's and 1's whose numbers of 0's and 1's are both even:



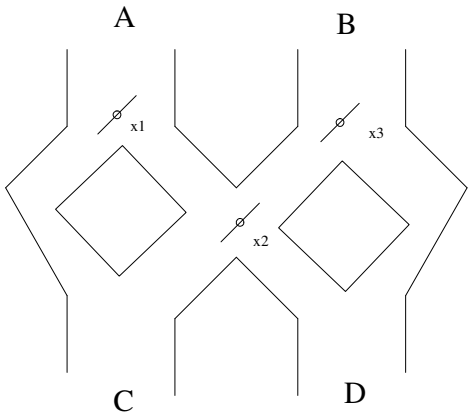
Aside: Type Errors

- A major source of confusion when dealing with automata (or mathematics in general) is making “type errors."
- Don't confuse A, a FA, i.e., a program, with L(A), which is of type “set of strings."
- The start state q_0 is of type “state," but the accepting states F is of type “set of states."
- *a* could be a symbol or *a* could be a string of length 1 depending on the context

DFA Exercise

- The following figure below is a marble-rolling toy. A marble is dropped at A or B. Levers x_1, x_2, and x_3 cause the marble to fall either to the left or to the right. Whenever a marble encounters a lever, it causes the lever to reverse after the marble passes, so the next marble will take the opposite branch.
- Model this game by a finite automaton. Let acceptance correspond to the marble exiting at D. Non-acceptance represents a marble exiting at C.

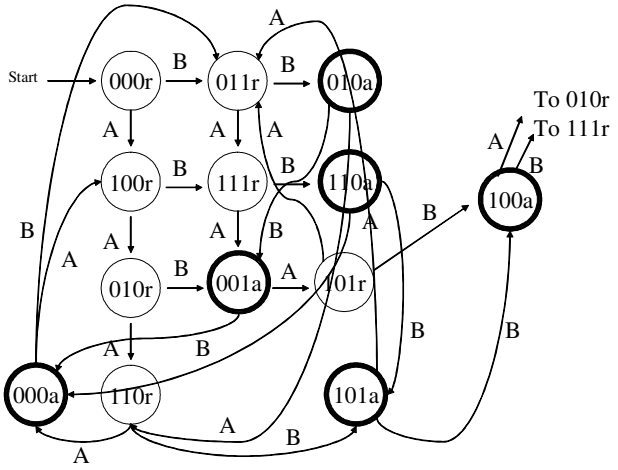
Marble Rolling Game



Marble Game Notation

- The inputs and outputs (A-D) become the alphabet of the automaton, while the levers indicate the possible states.
- If we define the initial status of each lever to be a 0, then if the levers change direction they are in state 1.
- Let's use the format $x_1x_2x_3$ to indicate a state. The initial state is 000. If we drop a marble down B, then the state becomes to 011 and the marble exits at C.
- Since we have three levers that can take on binary values, we have 8 possible states for levers, 000 to 111.
- Further identify the states by appending an "a" for acceptance, or "r" for rejection.
- This leads to a total of 16 possible states. All we need to do is start from the initial state and draw out the new states we are led to as we get inputs from A or B.

Messy Marble DFA



Marble DFA – Table Format

- Easier to see in table format. Note that not all states are accessible.

	A	B
->000r	100r	011r
*000a	100r	011r
*001a	101r	000a
010r	110r	001a
*010a	110r	001a
011r	111r	010a
100r	010r	111r
*100a	010r	111r
101r	011r	100a
*101a	011r	100a
110r	000a	101a
*110a	000a	101a
111r	001a	110a

Regular Operations

- Brief intro here – will cover more on regular expressions shortly
- In arithmetic, we have arithmetic operations
  - + * / etc.
- For finite automata, we have **regular operations**
  - Union
  - Concatenation
  - Star

Algebra for Languages

1. The union of two languages L and M is the set of strings that are in both L and M.
   - Example: if L = { 0, 1} and M = {111} then L ∪ M is {0, 1, 111}.
2. The concatenation of languages L and M is the set of strings that can be formed by taking any string in L and concatenating it with any string in M. Concatenation is denoted by LM although sometimes we'll use L•M (pronounced "dot").
   - Example, if L = {0, 1} and M = {ε, 010} then LM is { 0, 1, 0010, 1010}.

Algebra for Languages

3. The closure, star, or **Kleene star** of a language L is denoted L* and represents the set of strings that can be formed by taking any number of strings from L with repetition and concatenating them. It is a unary operator.

   More specifically, L^0 is the set we can make selecting zero strings from L. L^0 is always { ε }.

   L^1 is the language consisting of selecting one string from L.
   L^2 is the language consisting of concatenations selecting two strings from L.
   …
   L* is the union of L^0, L^1, L^2, … L^∞

   For example, if L = {0 , 10} then
   L^0 = { ε }.
   L^1 = {0, 10 }
   L^2 = {00, 010, 100, 1010}
   L^3 = {000, 0010, 0100, 01010, 10010, 1000, 10100, 101010}
   … and L* is the union of all these sets, up to infinity.

Closure Properties of Regular Languages

- **Closure** refers to some operation on a language, resulting in a new language that is of the same "type" as those originally operated on
  - i.e., regular in our case
- We won't be using the closure properties extensively here; consequently we will state the theorems and give some examples. See the book for proofs of the theorems.
- The regular languages are closed under union, concatenation, and *. I.e., if A_1 and A_2 are regular languages then
  - $A_1 \cup A_2$ is also regular
  - A_1A_2 is also regular
  - A_1^* is also regular

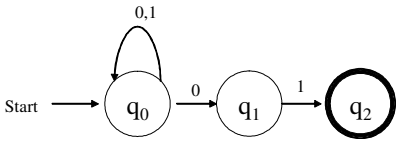
Later we'll see easy ways to prove the theorems

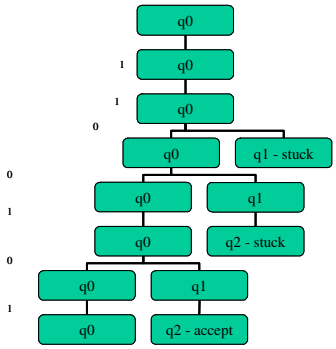
Nondeterministic Finite Automata

- A NFA (nondeterministic finite automata) is able to be in several states at once.
  - In a DFA, we can only take a transition to a single deterministic state
  - In a NFA we can accept multiple destination states for the same input.
  - You can think of this as the NFA "guesses" something about its input and will always follow the proper path if that can lead to an accepting state.
  - Another way to think of the NFA is that it travels all possible paths, and so it remains in many states at once. As long as at least one of the paths results in an accepting state, the NFA accepts the input.
- NFA is a useful tool
  - More expressive than a DFA.
  - BUT we will see that it is not more powerful! For any NFA we can construct a corresponding DFA
  - Another way to think of this is the DFA is how an NFA would actually be implemented (e.g. on a traditional computer)

NFA Example

- This NFA accepts only those strings that end in 01
- Running in "parallel threads" for string 1100101





Formal Definition of an NFA

- Similar to a DFA

1. Finite set of states, typically Q.
2. Alphabet of input symbols, typically Σ
3. One state is the start/initial state, typically q_0
4. Zero or more final/accepting states; the set is typically F.
5. A transition function, typically δ. This function:
   - Takes a state and input symbol as arguments.
   - Returns a **set of states** instead of a single state, as a DFA
6. A FA is represented as the five-tuple: A = ($Q, \Sigma, \delta, q_0, F$). Here, F is a set of accepting states.

Previous NFA in Formal Notation

The previous NFA could be specified formally as:

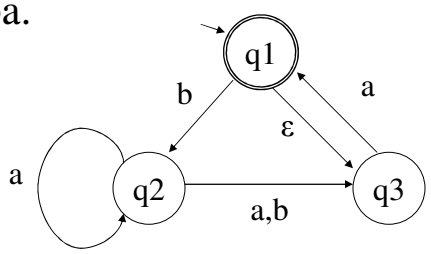
($\{q_0,q_1,q_2\}, \{0,1\}, \delta, q_0, \{q_2\}$)

The transition table is:

		0	**1**
→	**q0**	{q0,q1}	{q0}
	q1	**Ø**	{q2}
*	**q2**	**Ø**	**Ø**

NFA Example

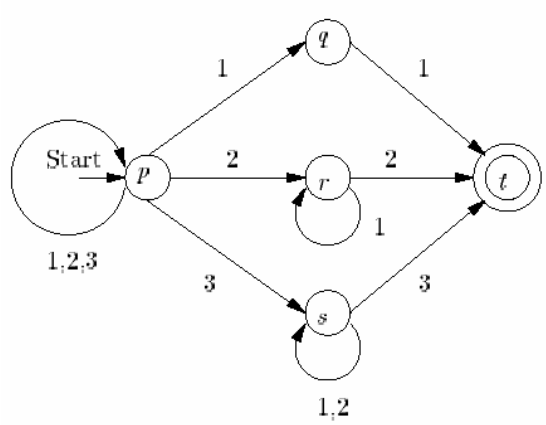
- Practice with the following NFA to satisfy yourself that it accepts ε, a, baba, baa, and aa, but that it doesn't accept b, bb, and babba.



NFA Exercise

- Construct an NFA that will accept strings over alphabet {1, 2, 3} such that the last symbol appears at least twice, but without any intervening higher symbol, in between:
  - e.g., 11, 2112, 123113, 3212113, etc.
- Trick: use start state to mean "I guess I haven't seen the symbol that matches the ending symbol yet." Use three other states to represent a guess that the matching symbol has been seen, and remembers what that symbol is.

NFA Exercise



You should be able to generate the transition table

Formal Definition of an NFA

- Same idea as the DFA
- Let N = (Q, ∑, δ,q_0, F) be an NFA and let w = $w_1w_2…w_n$ be a string where each w_i is a member of alphabet ∑.
- N **accepts** w if a sequence of states $r_0r_1…r_n$ in Q exists with three conditions:
  1. $r_0 = q_0$
  2. $r_{i+1} \in \delta(r_i, w_{i+1})$ for i=0, … , n-1
  3. $r_n \in F$

Observe that δ(ri, w_{i+1}) is the **set** of allowable next states

We say that N **recognizes** language A if A = {w | N accepts w }

Equivalence of DFA's and NFA's

- For most languages, NFA's are easier to construct than DFA's
- But it turns out we can build a corresponding DFA for any NFA
  - The downside is there may be up to 2^n states in turning a NFA into a DFA. However, for most problems the number of states is approximately equivalent.
- Theorem: A language L is accepted by some DFA if and only if L is accepted by some NFA; i.e. : L(DFA) = L(NFA) for an appropriately constructed DFA from an NFA.
  - Informal Proof: It is trivial to turn a DFA into an NFA (a DFA is already an NFA without nondeterminism). The following slides will show how to construct a DFA from an NFA.

NFA to DFA : Subset Construction

Let an NFA N be defined as $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$.
The equivalent DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ where:

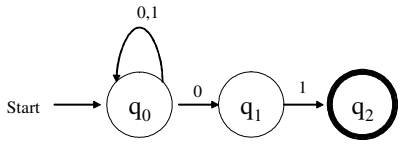
1. $Q_D = 2^{Qn}$; i.e. Q_D is the set of all subsets of Q_N; that is, it is the power set of Q_N. Often, not all of these states are accessible from the start state; these states may be "thrown away."
2. F_D is the set of subsets S of Q_N such that $S \cap F_N \neq \emptyset$. That is, F_D is all sets of N's states that include at least one accepting state of N.
3. For each set $S \subseteq Q_N$ and for each input symbol a in Σ:

$$\delta_D(S,a) = \bigcup_{p \in S} \delta_N(p,a)$$

That is, to compute $\delta_D(S, a)$ we look at all the states p in S, see what states N goes to starting from p on input a, and take the union of all those states.

Subset Construction Example (1)

- Consider the NFA:



The power set of these states is: { Ø, $\{q_0\}$, $\{q_1\}$, $\{q_2\}$, $\{q_0, q_1\}$, $\{q_0, q_2\}$,$\{q_1, q_2\}$, $\{q_0, q_1, q_2\}$ }

New transition function with all of these states and go to the set of possible inputs:

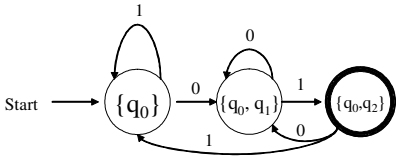
	0	**1**
Ø	Ø	Ø
→ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	Ø	$\{q_2\}$
*$\{q_2\}$	Ø	Ø
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
*$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
*$\{q_1, q_2\}$	Ø	$\{q_2\}$
*$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Subset Construction (2)

- Many states may be unreachable from our start state. A good way to construct the equivalent DFA from an NFA is to start with the start states and construct new states on the fly as we reach them.

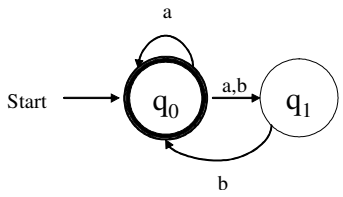
	0	**1**
Ø	Ø	Ø
→ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
*$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

Graphically:

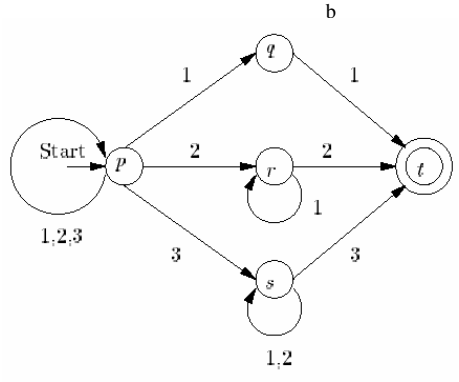


NFA to DFA Exercises

- Convert the following NFA's to DFA's



15 possible states on this second one (might be easier to represent in table format)



Corollary

- A language is regular if and only if some nondeterministic finite automaton recognizes it
- A language is regular if and only if some deterministic finite automaton recognizes it

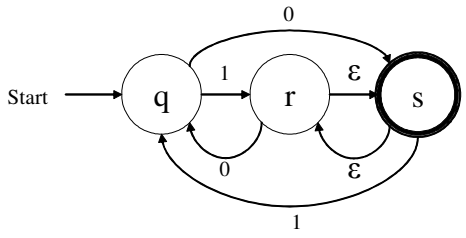
Epsilon Transitions

- Extension to NFA – a "feature" called epsilon transitions, denoted by ε, the empty string
- The ε transition lets us spontaneously take a transition, without receiving an input symbol
- Another mechanism that allows our NFA to be in multiple states at once.
  - Whenever we take an ε edge, we must fork off a new "thread" for the NFA starting in the destination state.
- While sometimes convenient, has no more power than a normal NFA
  - Just as a NFA has no more power than a DFA

Formal Notation – Epsilon Transition

- Transition function δ is now a function that takes as arguments:
  - A state in Q and
  - A member of $\sum \cup \{\varepsilon\}$; that is, an input symbol or the symbol ε. We require that ε not be a symbol of the alphabet $\sum$ to avoid any confusion.

Epsilon NFA Example

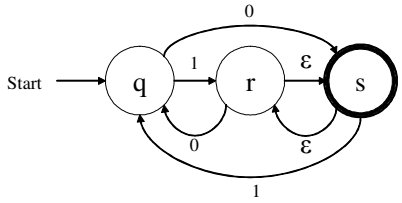


In this ε-NFA, the string "001" is accepted by the path qsrqrs, where the first qs matches 0, sr matches ε, rq matches 0, qr matches 1, and then rs matches ε. In other words, the accepted string is 0ε0ε1ε.

Epsilon Closure

- Epsilon closure of a state is simply the set of all states we can reach by following the transition function from the given state that are labeled ε.

Example:



- ε-closure(q) = { q }
- ε-closure(r) = { r, s}

Epsilon NFA Exercise

- Exercise: Design an ε-NFA for the language consisting of zero or more a's followed by zero or more b's followed by zero or more c's.

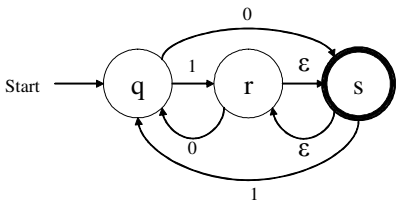
Eliminating Epsilon Transitions

To eliminate ε-transitions, use the following to convert to a DFA:

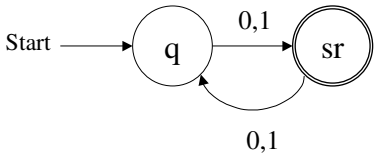
1. Compute ε-closure for the current state, resulting in a set of states S.
2. $\delta_D(S,a)$ is computed for all a in Σ by
   a. Let $S = \{p_1, p_2, \ldots p_k\}$
   b. Compute $\bigcup_{i=1}^{k} \delta(p_i, a)$ and call this set $\{r_1, r_2, r_3 \ldots r_m\}$ This set is achieved by following input a, not by following any ε-transitions
   c. Add the ε-transitions in by computing $\delta(S,a) = \bigcup_{i=1}^{m} \varepsilon - closure(r_i)$
3. Make a state an accepting state if it includes any final states in the ε-NFA.

In simple terms: Just like converting a regular NFA to a DFA except follow the epsilon transitions whenever possible after processing an input

Epsilon Elimination Example



Converts to:



Epsilon Elimination Exercise

- Exercise: Here is the ε-NFA for the language consisting of zero or more a's followed by zero or more b's followed by zero or more c's.
- Eliminate the epsilon transitions.