CS311—Computational Structures

Nondeterminism. Theorems about Finite State Automata

Lecture 4

Portland State University

Review: Formal Definition of DFA

- A (deterministic) finite automaton is a 5-tuple (Q, A, δ, q_0, F) where:
  1. Q is a finite set called the **states**,
  2. A is a finite set called the **alphabet**,
  3. $\delta: Q \times A \to Q$ is the **transition function**,
  4. q_0 is the **start state**, and
  5. $F \subseteq Q$ is the set of **final states**

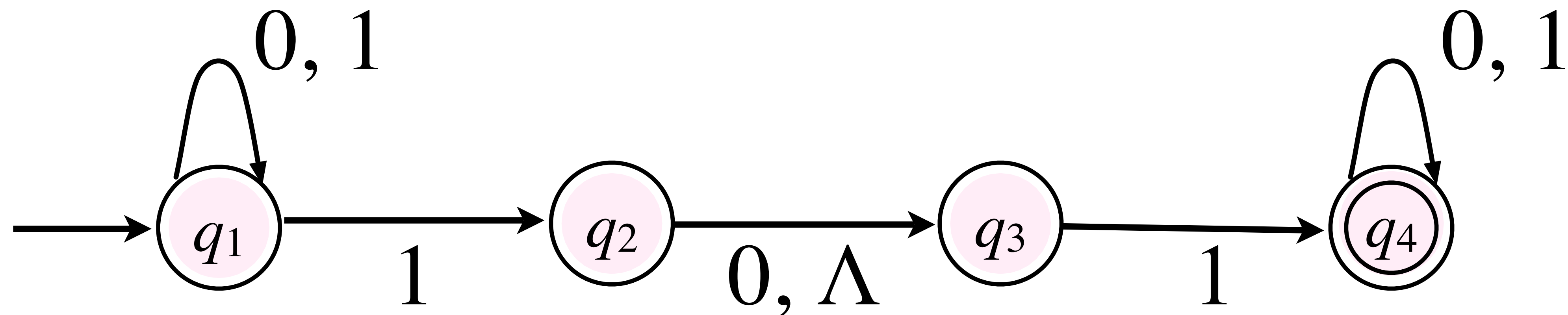
Let's build some DFAs

- Design DFAs that recognize each of these languages:
  - abb
  - $(x + y)^*$
  - $(x + y)^*abb$
  - $(x + y)^*xyy$

Nondeterminism

- In the FSA that we have seen so far, there is exactly one action to be taken on each input symbol.
  - that's what it means for δ to be a function!
- In a nondeterministic FSA, several choices may exist at each step.

Example of NFA



- How does it differ from a DFA?
  1. some states have multiple transitions on a given input
  2. some states have *no* transition on an input
  3. some transitions have label Λ

Formal Definition

- A nondeterministic finite automaton is a 5-tuple (Q, A, δ, q_0, F) where:

1. Q is a finite set called the **states**,
2. A is a finite set called the **alphabet**,
3. $\delta: Q \times (A \cup \{\Lambda\}) \rightarrow \wp(Q)$ is the **transition function**,
4. q_0 is the **start state**, and
5. $F \subseteq Q$ is the set of **final states**

Nondeterministic Computation

- What does it mean for an NFA to take a "step" when there are multiple possibilities at each step? What about Λ?
  - the NFA makes all possible transitions in parallel, or, equivalently,
  - the NFA clones itself and one clone explores each possibility.
- an NFA can reach a "dead end" (gets stuck)
- an NFA accepts its input if any of the clones reaches a final state.

Recap

- Two kinds of FSA:
  - Deterministic: for each input, there is exactly one “next state”.
  - Nondeterministic:
    - for each input there may be zero, one or many “next states”.
    - NFA can also make Λ-transitions at any time.
- Both are formally defined by a 5-tuple
  - the details of the transition function differ

- FSA can be used to define languages:
  - L(fsa) = { $w \in A^*$ | w is accepted by fsa }
- Unsubstantiated rumor:
  - For *any* RE, there is an NFA that accepts the language defined by that RE
  - The class of regular languages and the class of languages recognized by NFA are the same!

- Unsubstantiated Rumor: NFA and DFA are of equivalent power!
  - that is, for any NFA, there is an equivalent DFA, i.e., a DFA that recognizes the same language
  - and vice versa, but that's obvious
    - why is it obvious?

First Rumor: RE ⇒ NFA

- Theorem is by construction:
  - start with an RE
  - show how to build an NFA from it
  - show that the NFA accepts the same language as the RE
- How can we possibly do this when there are an infinite number of REs?

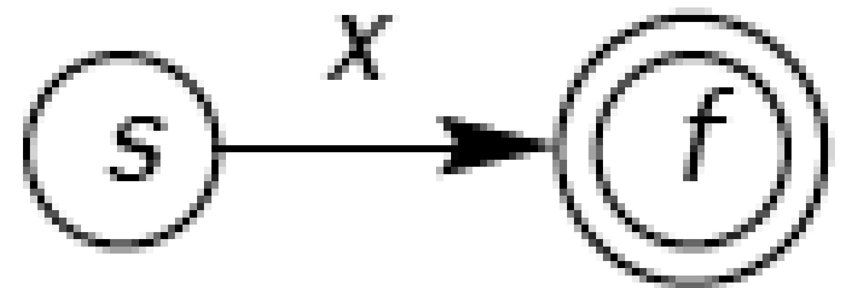
NFA construction

```
datatype 'a NFA = NFA of
    { allStates : State list,
      alphabet : 'a list,
      transitions : (State * 'a Label * State) list,
      startState : State,
      finalStates  : State list };

fun allStates (NFA {allStates, alphabet, transitions, startState,
finalStates}) = allStates;
fun startState (NFA {allStates, alphabet, transitions,  startState,
finalStates}) = startState;
fun alphabet (NFA {allStates, alphabet, transitions,  startState,
finalStates}) = alphabet;
fun transitions (NFA {allStates, alphabet, transitions,  startState,
finalStates}) = transitions;
fun finalStates (NFA {allStates, alphabet, transitions,  startState,
finalStates}) = finalStates;
```

Portland State UNIVERSITY

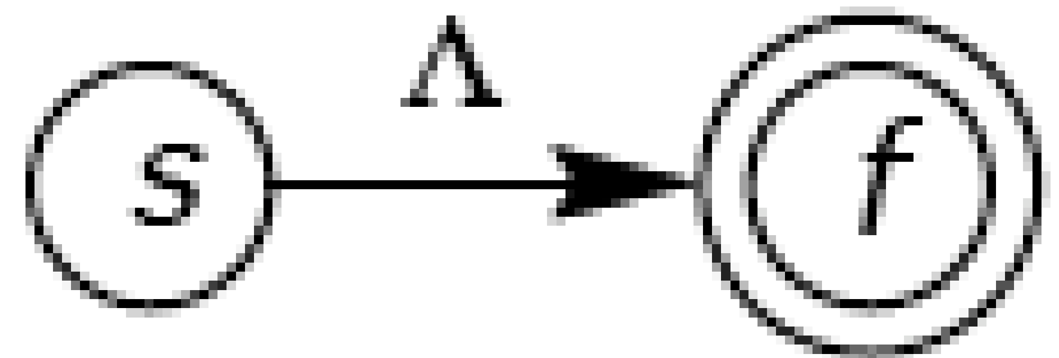
```
fun NFAfor (C x) =
        let   val q0 = newState () ;
              val q1 = newState ()
        in
           NFA { allStates = [q0, q1],
                 alphabet = [x],
                 transitions = [(q0, Label x, q1)],
                 startState = q0,
                 finalStates = [q1] }
        end
```



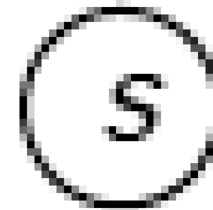
Hein construction (11.7)

Portland State UNIVERSITY

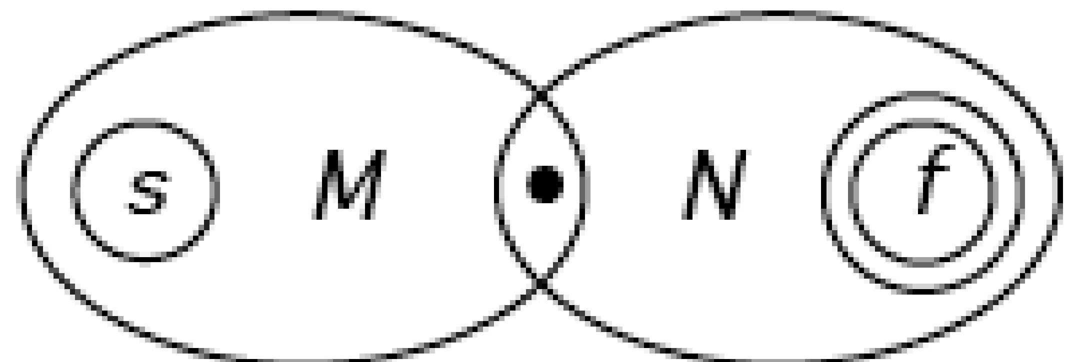
```
| NFAfor (Lambda) =
      let val q0 = newState () ;
          val q1 = newState ()
      in
         NFA { allStates = [q0, q1],
               alphabet = [],
               transitions = [(q0, EmptyLabel, q1)],
              startState = q0,
              finalStates = [q1] }
      end
```

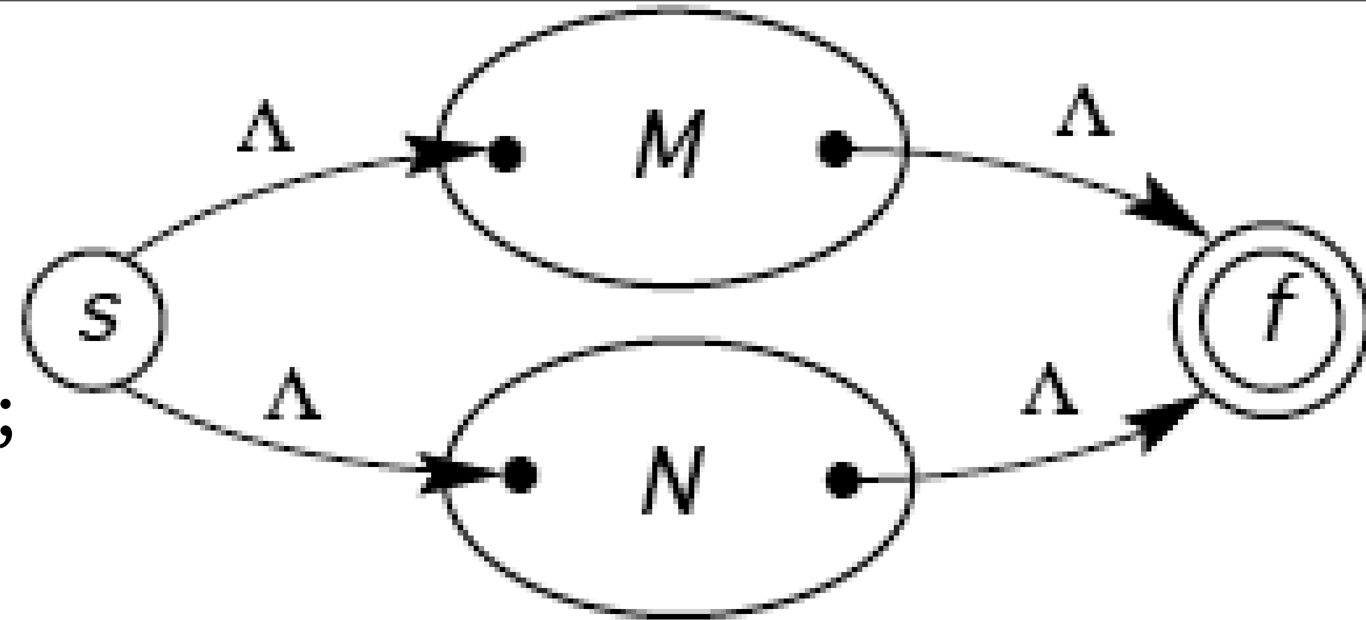


```
| NFAfor (Empty) =
    let val q0 = newState () ;
        val q1 = newState ()
    in
      NFA { allStates = [q0, q1],
            alphabet = [],
            transitions = [],
            startState = q0,
            finalStates = [q1] }
    end
```



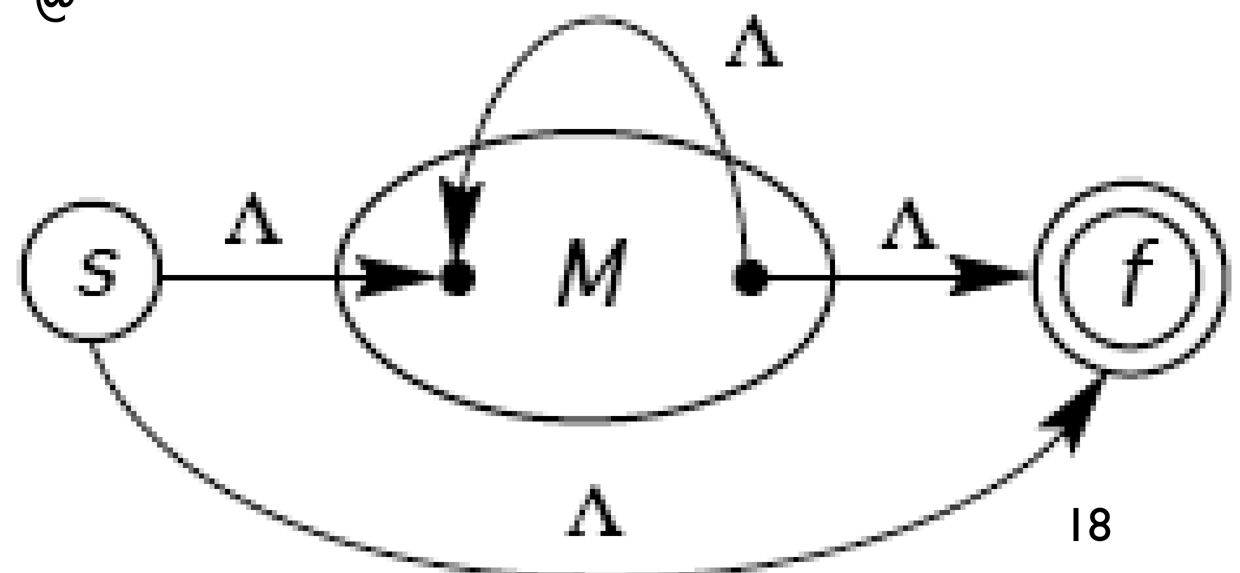
```
| NFAfor (Concat(re1, re2)) =
        let val m = NFAfor(re1);
            val n = NFAfor(re2);
        in
            NFA { allStates = allStates m @
                                         allStates n ,
                  alphabet = union (alphabet m
                                          alphabet n),
                  transitions = transitions m @
                                transitions n @
                               map (fn each =>
                 (each, EmptyLabel, startState n))
                                    (finalStates m),
                  startState = startState m ,
                  finalStates = finalStates n }
         end
```






```
| NFAfor (Union(re1, re2)) =
  let   val m = NFAfor(re1);
        val n = NFAfor(re2);
        val start = newState();
        val final = newState()
  in
        NFA { allStates = start :: final ::
                                   allStates m @ allStates n,
              startState = start,
              finalStates = [ final ],
              alphabet = union (alphabet m, alphabet n),
              transitions = (start, EmptyLabel,
                                                startState m) ::
                  (start, EmptyLabel, startState n) ::
                  transitions m @
                  transitions n @
                  map (fn source => (source, EmptyLabel,
                                     final)) (finalStates m) @
                  map (fn source => (source, EmptyLabel,
                                     final)) (finalStates n) }
  end
```

```
| NFAfor (Star re1) =
    let val m = NFAfor(re1);
        val start = newState();
        val final = newState();
    in
       NFA { allStates = start :: final :: allStates m,
       startState = start,
       alphabet = alphabet m,
       finalStates = [ final ],
       transitions = (start, EmptyLabel, final) ::
           (start, EmptyLabel, startState m) ::
           map (fn each => (each, EmptyLabel, final))
               (finalStates m) @
           map (fn each => (each, EmptyLabel, startState m))
               (finalStates m) @
           transitions m }
    end
```



Are we done yet, Sir?

- How do we know that this construction tells us what to do for *all* REs?
- Does the resulting FSM accept the same language as is described by the RE?

What about the ML program?

- The program doesn't prove the theorem
- But it does let us test:
  - that the construction works for a large number of examples, and
  - that in each case, strings in the language of the RE are accepted by the FSM.

Converting an NFA to a DFA

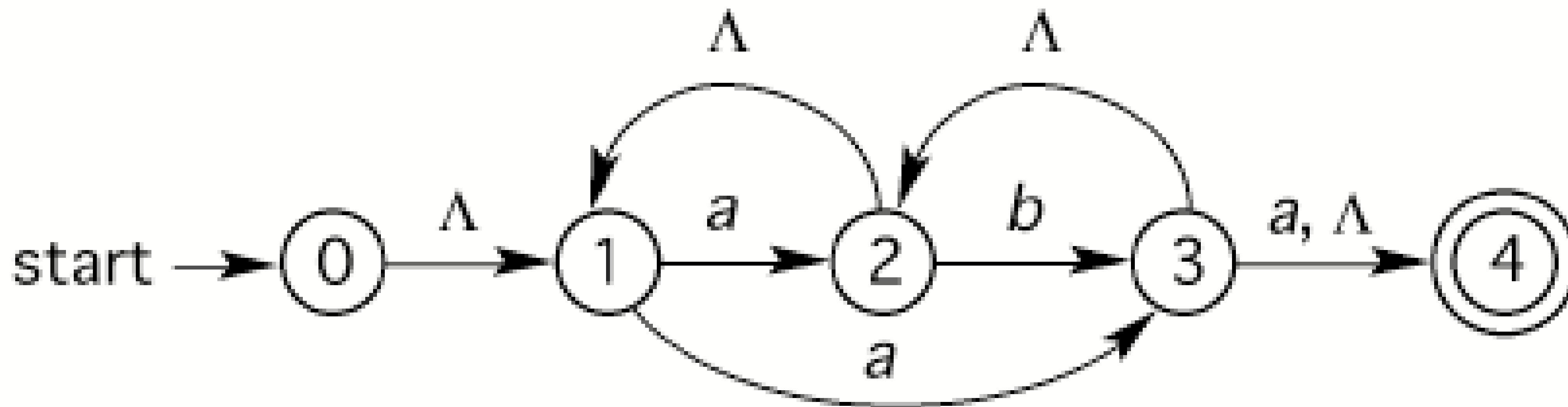
- When an NDFA computes, at any time it is in a *set* of states
  - or, equivalently, there are a set of machines each of which is in one state; the set grows and shrinks as the computation continues
- Key idea: build a DFA whose states represent *sets* of states in the NFA

Λ Closure

- a useful building-block in the construction of a DFA
- for all NFA states s, the Λ closure of s, written $\lambda(s)$, is defined by:

1. $s \in \lambda(s)$
2. if $p \in \lambda(s)$, and $\delta(p, \Lambda) = q$, then $q \in \lambda(s)$

Hein Example 11.16



	T_N	a	b	Λ
start	0	∅	∅	{1}
	1	{2, 3}	∅	∅
	2	∅	{3}	{1}
	3	{4}	∅	{2, 4}
final	4	∅	∅	∅

$\lambda(0) =$

$\lambda(1) =$

$\lambda(2) =$

$\lambda(3) =$

$\lambda(4) =$