Finite State Machines 2

15-121 Intro to Data Structures

Deterministic Finite-State Automata (review)

- A DFSA can be formally defined as

  $A = (Q, \Sigma, \partial, q_0, F)$:

  - Q, a finite set of states.
  - Σ, a finite alphabet of input symbols.
  - $q_0 \in Q$, an initial start state.
  - $F \subseteq Q$, a set of final states.
  - ∂ (delta): $Q \times \Sigma \rightarrow Q$, a transition function.

- We can define ∂ on words, ∂_w, by using a recursive definition:
  - $\partial_w : Q \times \Sigma^* \rightarrow Q$ — A function of (state, word) to a state.
  - $\partial_w(q,\varepsilon) = q$ — If in state q, output state q if word is ε.
  - $\partial_w(q,xa) = \partial(\partial_w(q,x),a)$ — Otherwise, use ∂ for one step and recurse.

- For an automaton A, we can define the language of A:
  - $L(A) = \{w \in \Sigma^* : \partial_w(q_0,w) \in F\}$
  - L(A) is a subset of all words w of finite length over Σ, such that the transition function $\partial_w(q_0,w)$ produces a state in the set of final states (F).
  - Intuitively, if we think of the automaton as a graph structure, then the words in L(A) represent the "paths" which end in a final state. If we concatenate the labels from the edges in each such path, we derive a string $w \in L(A)$.

Regular Languages

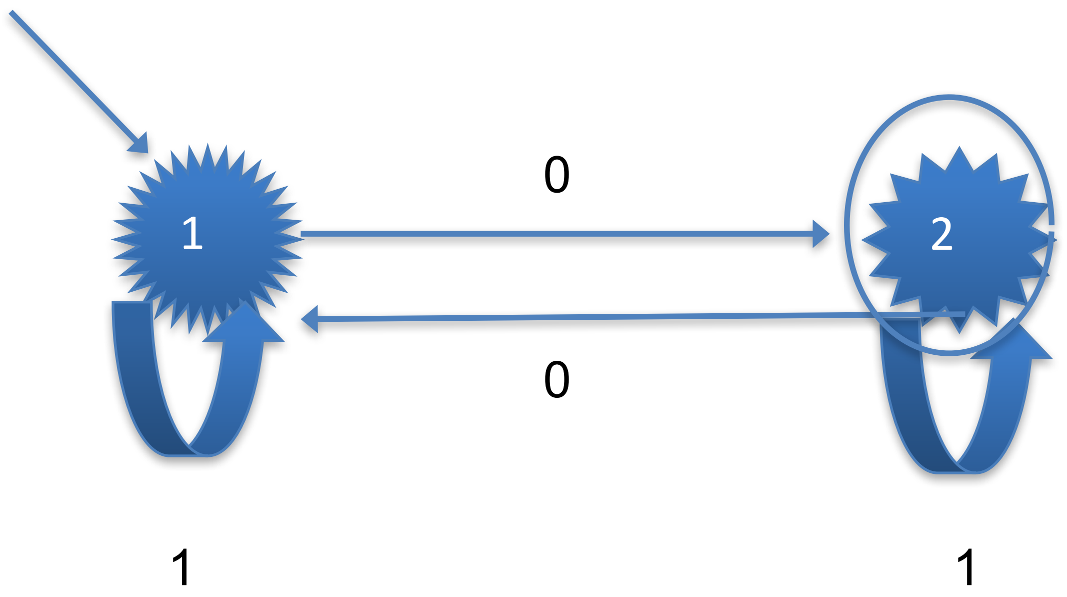
A language $L \subseteq \Sigma^*$ is called a *regular language* if there exists a finite-state automaton, A, such that $L = L(A)$.

Examples of regular languages:

$L = \Sigma^*$	(all finite words in Σ)
$L = \varnothing$	(the empty set)
$L = \{\varepsilon\}$	(set containing just the empty string)

(As a self-test, draw a DFSA, A, for each L, such that $L = L(A)$. Use $\Sigma = \{a,b,c\}$.)

Digression: Encoding a DFA



1. Number of states.
2. First on 0.
3. First on 1.
4. Second on 0.
5. Second on 1.
6. Accepting states.

0010010101001100

Quiz: Does every Java program have an encoding?
Is every possible Java program found in {0,1}* ?

Non-Deterministic Finite-State Automata (NDFSA)

- The DFSA we have studied so far are called *deterministic*, because for any given word w there is a single path through the automaton (or, more formally, $\partial_w(q_0,w) = q_n$; the automaton transitions to a single state on any given word). The result is "well-determined" because it can have only a single value.

NDFSA

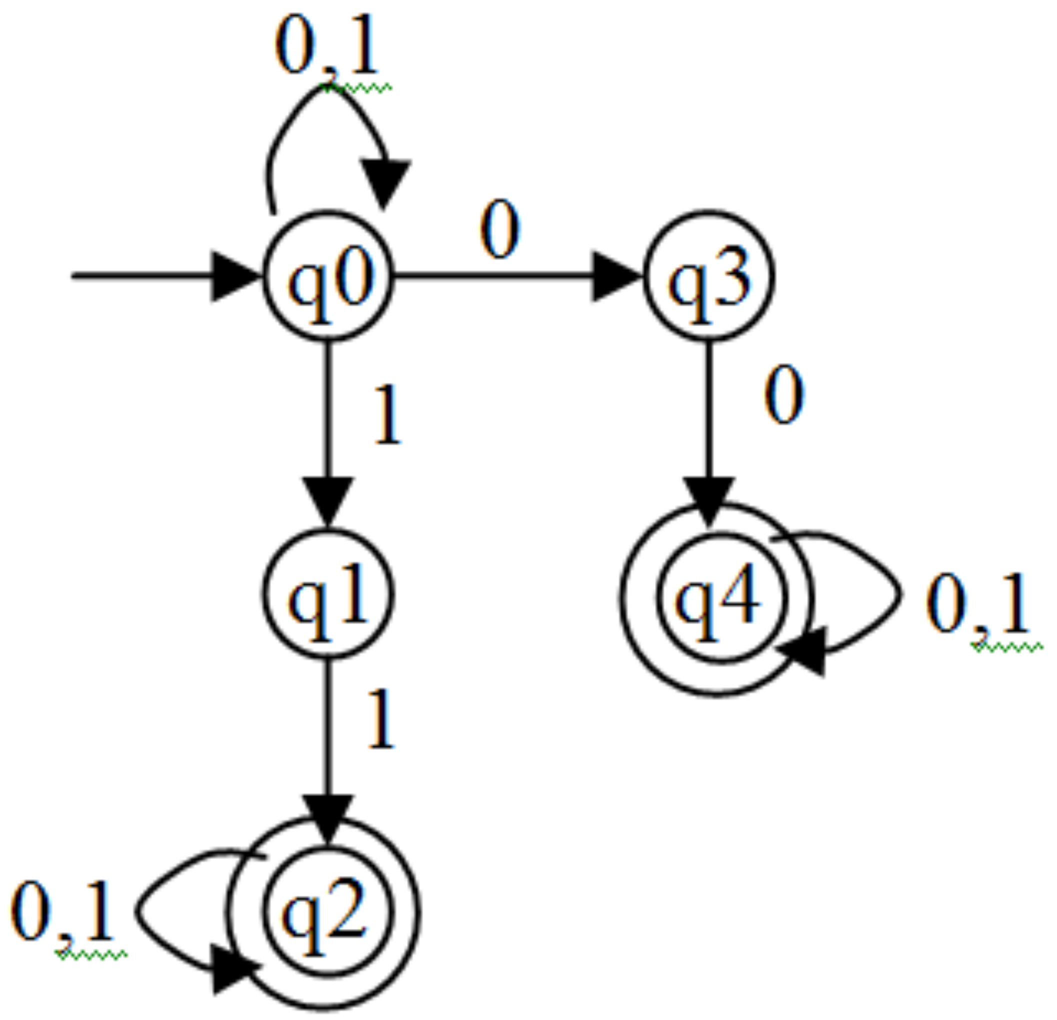
We can extend the definition of DFSA to be less restrictive, such that output of the transition function ∂ is a *set of states* rather than a single state:

$\partial : Q \times \Sigma \rightarrow 2^Q$

The non-deterministic ∂ can produce as output any subset of Q, so in the definition we use 2^Q to indicate the *power set* (set of all possible subsets) of Q. Will this change add power?

- Since there can be more than one "path" through an NDFSA for a given word w, we have to revise our notion of acceptance for NDFSA. An NDFSA accepts a word w if there exists *some* computation path that ends in a final state $q \in F$.

NDFSA Example



NDFSA

Note the following non-deterministic transitions:

$\partial(q_0,0) = \{q_0,q_3\}$

$\partial(q_0,1) = \{q_0,q_1\}$

(As a self-test, trace all of the paths through the example NDFSA for w = 0100. Indicate each path by writing down the sequence of states.)

- For NDFSA, we can expand the notion of ∂ on letters to ∂ on words, ∂_w, by using a recursive definition similar to the one we used for DFSA:
  - $\partial_w : Q \times \Sigma^* \rightarrow 2^Q$
  - $\partial_w(q,\varepsilon) = \{q\}$
  - $\partial_w(q,xa) = \{p \mid \text{for some state } r \in \partial_w(q,x), p \in \partial(r,a)\}$
- The third statement indicates that: starting in state q, and reading the string x, followed by the symbol a, we can be in state p, if and only if one possible state we can be in after reading x is r, and from r we may go to p upon reading a.

We can also define a transition function that accepts a *set* of states as one of its inputs:

$\partial_P : 2^Q \times \Sigma^* \to 2^Q$

So, if $P \subseteq Q$, then $\partial_P(P,w) = \cup_{q \text{ in } P}\, \partial_w(q,w)$.

Restated in English: If P is a subset of the states in Q, then the value of the transition function ∂_P on P and some word w will be the union of all of the sets of states produced by computing $\partial_w(q,w)$ for every q in P.

For example, in the automaton shown above, $\partial_P(\{q_1,q_3\},1) = \{q_2\}$.

The transition function on sets of states allows us to compute all possible next states after each symbol or word is encountered, giving the effect of trying all paths in parallel.

Equivalence of NDFSA and DFSA

- The set of languages L(A) for all NDFSA is also the set of regular languages.
- For every NDFSA A, we can construct an equivalent DFSA B such that L(A) = L(B).
- The equivalence is achieved by using a single state in the DFSA to represent a set of states in the NDFSA.
- The DFSA keeps track of all possible states that the NDFSA could be in after reading the same input. Formally, if the NDFSA has a set of states Q, then the equivalent DFSA has a set of states $Q' = 2^Q$ (all possible subsets of Q).

An element of Q' is denoted as $[q_1,q_2,\ldots,q_m]$. $q'_0 = [q_0]$. Then we can define

$$\partial'([q_1,q_2,\ldots,q_m],a) = [p_1,p_2,\ldots,p_n]$$

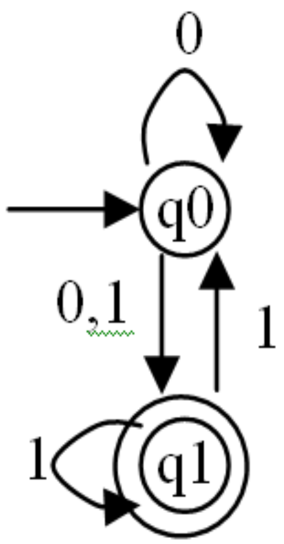
if and only if

$$\partial_P(\{q_1,q_2,\ldots,q_m\},a) = \{p_1,p_2,\ldots,p_n\}.$$

Example Conversion of NDFSA to DFSA

Let A = $(\{q_0,q_1\},\{0,1\},\partial,q_0,\{q_1\})$ be an NDFSA where:

$\partial(q_0,0) = \{q_0,q_1\}$
$\partial(q_0,1) = \{q_1\}$
$\partial(q_1,0) = \varnothing$
$\partial(q_1,1) = \{q_0,q_1\}$



We can construct an equivalent DFSA, A' = $\{Q,\{0,1\},\partial',[q_0],F)$ such that $L(A') = L(A)$, as follows.

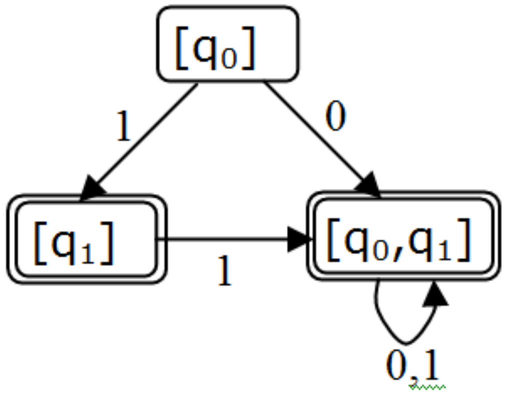
The elements of Q will be the power set of $\{q_0,q_1\}$, represented using the square bracket notation introduced in the previous section to indicate that each state in Q represents a set of states in the original NDFSA. Here is the set of states in Q:

$[q_0], [q_1], [q_0,q_1], \varnothing$

NDFSA to DFSA

Then we can define ∂' in terms of ∂, as follows:

$$
\begin{aligned}
\partial(q_0,0) &= \{q_0,q_1\} & &\Rightarrow \partial'([q_0],0) = [q_0,q_1]\\
\partial(q_0,1) &= \{q_1\} & &\Rightarrow \partial'([q_0],1) = [q_1]\\
\partial(q_1,0) &= \varnothing & &\Rightarrow \partial'([q_1],0) = \varnothing\\
\partial(q_1,1) &= \{q_0,q_1\} & &\Rightarrow \partial'([q_1],1) = [q_0,q_1]
\end{aligned}
$$



Then we must define the transitions from $[q_0,q_1]$:

$\partial'([q_0,q_1],0) = [q_0,q_1]$, since
$\partial(\{q_0,q_1\},0) = \partial(q_0,0) \cup \partial(q_1,0) = \{q_0,q_1\} \cup \varnothing = \{q_0,q_1\}$;

$\partial'([q_0,q_1],1) = [q_0,q_1]$, since
$\partial(\{q_0,q_1\},1) = \partial(q_0,1) \cup \partial(q_1,1) = \{q_1\} \cup \{q_0,q_1\} = \{q_0,q_1\}$.

The set F of final states is the set of all states in Q that contain the original final state, q_1; so $F = \{[q_1],[q_0,q_1]\}$.

- (As a self-test, trace the computation of some sample strings through both the NDFSA and the equivalent DFSA in order to convince yourself that they really are equivalent. Try the conversion yourself for another small NDFSA.)

Carnegie Mellon

Context-Free Grammars and Context-Free Languages

- A *context-free grammar* G is defined formally as:
  - V: a finite set of variables ("non-terminals"); e.g., A, B, C, …
  - T: a finite set of symbols ("terminals"), e.g., a, b, c, …
  - P: a set of production rules of the form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$
  - S: a start non-terminal; $S \in V$

- Production rules can also be thought of as *derivations*.

Assume

$G = (\{A\},\{a,b\},\{A \rightarrow aAb, A \rightarrow \varepsilon\}, A)$

Note that $L(G) = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$.

We can derive each string in L(G) by using the two production rules to rewrite the initial expression, which consists of just the start symbol. For example, the derivation of aabb:

A	
aAb	(apply first rule)
aaAbb	(apply first rule)
aabb	(apply second rule)

We use the symbol $\Rightarrow$ to indicate that a derivation exists from an expression to another expression for a given grammar;

e.g., for the grammar G defined above, A $\Rightarrow$ aabb.

Then we can define L(G) as follows:

$$L(G) = \{ w \in T^* \mid S \Rightarrow w \}$$

In plain English, the language of a grammar G is the set of all strings that can be derived from the start symbol S using the production rules in P.

A language L is a *context-free language* if there exists a grammar G such that L = L(G).

<u>Context-Free Language vs. Regular Languages</u>

- Consider the grammar we specified in a prior example. A closed-form expression for this grammar is:

  $L(G) = \{a^n b^n \mid n \geq 1\}$

- **Question**: Why is L(G) <u>not</u> a regular language? Recall the definition of a regular language. For every regular language L, there exists some DFSA A such that L(A) = L. Why isn't it possible to define a DFSA which accepts $L(G) = \{a^n b^n \mid n \geq 1\}$?

- **Answer**: Because the derivation of each $w \in L(G)$ adds exactly one a and one b to the word being constructed, each time the first production is fired. A DFSA can accept only one symbol at a time, and it cannot "remember" how many instances of a particular symbol it has seen. Any DFSA we define that accepts strings of the form $\{a^n b^n \mid n \geq 1\}$ would also accept other strings $\{a^m b^n \mid m \neq n\}$.
- By the pigeon hole principle, n+1 a's will require that an n state machine be in some same state more than once.

(Self-test: try to construct a DFSA that accepts precisely $\{a^n b^n \mid n \geq 1\}$, to convince yourself that this is the case.)

Carnegie Mellon

Pushdown Automata

- In the previous lecture, we explained how a DFSA is used to recognize (accept strings in) regular languages.
- Context-free languages also have a machine counterpart: the *pushdown automata* (PDA).
- To recognize context-free languages, we need to define a machine that solves the "memory problem" we noted above.
- The solution comes from adding a stack data structure to a finite-state machine.

- To understand how the stack is used in conjunction with a finite-state machine, let's visualize a pushdown automaton for our example context-free language,

$$L(G) = \{a^n b^n \mid n \geq 1\}.$$

- Let's define a machine with two states, as follows:
  - When the machine is in q_0 : If an a is read, push a marker on the stack and stay in q_0; if a b is read and there is a marker on the stack, pop the stack and go to q_1.
  - When the machine is in q_1: If a b is read and there is a marker on the stack, pop the stack and stay in q_1.
  - Assume that all other transitions are undefined and cause the machine to halt, rejecting the input.
  - The computation ends when both the input and the stack are empty.

- Why does this machine accept $L(G) = \{a^n b^n \mid n \geq 1\}$?
- In the start state q_0, the only possible moves are

a) read one or more a's, adding a marker to the stack for each a which is seen;

b) read exactly one b, popping the stack and moving to q_1.

  - Assuming we have read n a's and 1 b, then there will be n – 1 markers left on the stack.
  - In state q1, the only possible move is to read a b and pop the stack.
  - Since the machine will halt (and reject) if there is input remaining and the stack is empty, the only way to exhaust the input and end with an empty stack is to read exactly the same number of a's and b's.

Here's the formal definition of a pushdown automaton:

$$M = (Q,\Sigma,\Gamma,\partial,q_0,F)$$

Q: a set of states

Σ: the alphabet of input symbols

Γ: the alphabet of stack symbols

∂: $Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma$

q_0: the initial state

F: the set of final states

Intuitively, if $\partial(q,s,\beta) = (q',\gamma)$, then M, whenever it is in state q with β at the top of the stack, may read s from the input, replace β by γ on the top of the stack, and enter state q'.

Pushing, popping, and preserving the stack are possible:

$\partial(q,a,\varepsilon) = (q',A)$	push A on the stack without popping
$\partial(q,a,A) = (q',\varepsilon)$	pop A from the stack without pushing
$\partial(q,a,\varepsilon) = (q',\varepsilon)$	stack unchanged

Now we can define a PDA M, such that $L(M) = L(G) = \{a^n b^n \mid n \geq 1\}$:

$$M = (\{q_0,q_1\},\{a,b\},\{A\},\partial,q_0,\{q_1\})$$

$\partial(q_0,a,\varepsilon) = (q_0,A)$
$\partial(q_0,b,A) = (q_1,\varepsilon)$
$\partial(q_1,b,A) = (q_1,\varepsilon)$

(Self-test: Trace the operation of M on some strings in L(G), and some strings not in L(G). Assume computation is successful (accept) only if the input is empty, the stack is empty, and the machine is in final state $q_{1.)}$

(Self-test: Build a PDA that recognizes $\{0^i1^j2^k: k = i * j\}$. Note: You will not be able to succeed. The pumping lemma can be used to prove this result.)