CS 3100 – Models of Computation – Fall 2011 – Notes for L7

September 19, 2011

1 On the role of DFA and NFA, IMPORTANT TERMINOLOGY

- A DFA D is a five-tuple $(Q, \Sigma, \delta, q_0, F)$, and you can look up notes5.pdf to recap what these fields are. Pay attention to Σ for now.

  The language of a DFA D is

  $$\mathcal{L}(D) = \{w \mid \hat{\delta}(q_0, w) \in F\}.$$

  Here, $\hat{\delta}$ is the `run_dfa` function.

- Of course, the language of a DFA is defined by all the strings `s` for which the function `accepts` below returns true:

```
def accepts(D, q, s):
    """ Checks for DFA acceptance.
    """
    return run_dfa(D, q, s) in D["F"]
```

- A string is not accepted (or rejected) if the function `accepts` would return `False`.

- An NFA N is a structure $(Q, \Sigma, \delta, q_0, F)$, and we explain these components below. Again, pay attention to Σ for now.

- Given such a DFA D, only consider strings over Σ^* with respect to this DFA. Thus, if $\Sigma = \{a, b, c\}$, consider all possible strings over $\Sigma = \{a, b, c\}$ whenever doing any analysis with respect to this DFA. Never consider any other symbol such as 0 or 1 or $-$ or . or ¡ or ¿ or $<$ or $>$ or ! or whatever other symbol you might be tempted to consider with respect to this DFA. You can't even consider puffs of smoke signals as your "dot" and "dash" to communicate with this DFA.

- Likewise, if $\{?, 0, a, \rightarrow\}$ is your Σ, then simply consider strings over these symbols

  Example: $?a0? \rightarrow ?a$ may be a string considered.

  However, never ever consider a string of the form $1?a0?b \rightarrow ?a$ with respect to this DFA because such a string is not in the Σ^* for this Σ.

- For our "Python-built DFA" we don't allow $\rightarrow$ or ¡ or ¿ or other funny characters in the alphabet. We are concerned with only single character symbols for DFA experimented with in Python.

- Now, given a Σ, you must check that it is a non-empty set. Never never have a DFA with an empty Σ.

- Of course $\Sigma = \{0\}$ is allowed. Also $\Sigma = \{a\}$ is allowed. This is like a DFA connected to a keyboard that has a single key. You will never make a typing mistake!
- All of what we said about a DFA with respect to Σ are also true of an NFA. Later we will study regular expressions; even for them all this is true.
- The "job" of a DFA or an NFA is to **separate strings from Σ^* into two bins**
  - Those that are accepted
  - Those that are not accepted (or "rejected")

  Note that it is only strings over Σ^* that are so classified. The universe of strings is Σ^*.
- L is a regular language if L is the language of a DFA (or equivalently, that of an NFA).
- A string that is accepted takes the DFA (or NFA as the case may be) from its initial state to the final state (or one of its final states)
- A DFA/NFA rejecting a string is really accepting a string from the complement of L. Complementation is done with respect to Σ^*—see the complement function in `lang.py`

```
def lcomplem(L,alph,m):
    """Complement L relative to alphabset alph. alph is also given as a set of strings.
       We subtract from the "star up to m" of the alphabet alph, the language L.
    """
    return star(alph,m) - L
```

2 On NFA

In the last set of notes, we presented a few facts:

- Regular languages are those languages specified by a DFA. Each regular language is a language (a set of strings) L such that there exists a DFA who language L is.
- Some regular languages are easy to specify using a DFA whereas for some regular languages, their DFA become too hard to draw out. In many of those cases, NFA prove to far easier. We give two examples below.

We now formally introduce nondeterministic finite automata. Let Σ_ε stand for $(\Sigma \cup \{\varepsilon\})$. A *nondeterministic finite-state automaton* N is a structure $(Q, \Sigma, \delta, q_0, F)$, where:

- Q, a *finite non-empty* set of states;
- Σ, a *finite non-empty* alphabet;
- $\delta : Q \times \Sigma_\varepsilon \to 2^Q$, a *total* transition function.
- $q_0 \in Q$, an initial state; and
- $F \subseteq Q$, a *finite, possibly empty* set of final states.

Note, that the range of δ includes $\emptyset$; therefore, if for some state q, $\delta(q, x) = \emptyset$, the move from state q on input x is essentially to a "black hole". That is, the NFA is **no state at all**. Since the moves of any machine are always *from* a state *to* a state, and if the *from* states come from an empty set, naturally the *to* states also end up being the empty set. We will formally define the language of an NFA in § 6.

NFA acceptance: An NFA N accepts a string s if *any one* of its runs on a string s leaves the NFA in one of its final states F.

The language of an NFA: The language of an NFA N is all the strings it accepts.

Note on ε: Note that an NFA can take ε moves "whenever it likes" (**provided such a move is available for the NFA**).

These ε are incorporated into the run in an obvious way. For example, a run of the NFA of the form

$$\varepsilon 0101 \varepsilon 1 \varepsilon 0 \varepsilon$$

is the same as the run of the same NFA on

$$010110$$

That is, inserting ε in a string does not alter the string.

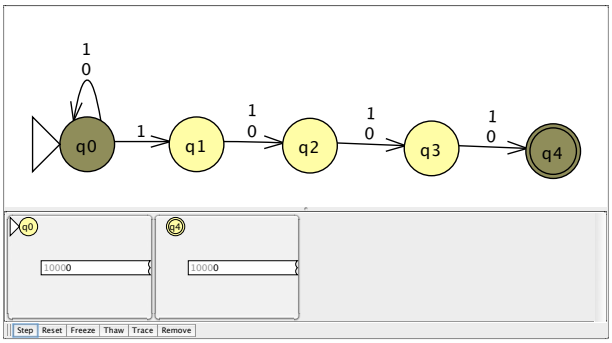
A *non-deterministic finite-state automaton* (NFA) is a preferred alternative to DFA for specifying regular languages. Using an NFA, we can be sure of two things:

- For many languages, the use of an NFA can make the automaton specification exponentially more succinct. That is, the number of states you draw and the number of transitions you draw will be vastly smaller in number than for a DFA.
- For no language is an NFA worse than (larger than) a DFA. In fact, a DFA is a special case of an NFA (one where there is no non-determinism). Thus no NFA needs to be bigger than an equivalent DFA.

Here are three languages whose definitions are made much easier by following the NFA route:

- An NFA for L_a where

$$L_a = \{x \mid \textit{the third last symbol of } x \textit{ is a } 1\}$$



- An NFA for $L_b = L_1 L_2$ (L_1 concatenated with L_2) where

  L_1 = strings ending in a 1.

  L_2 = the language of strings with equal numbers of 01 and 10 changes.

- An NFA for $L_c = L_1 \cup L_2$, the union of L_1 and L_2 which are defined above.

3 NFA simplifies construction of "Union and Concatenation languages"

More specifically, for languages defined through the Union or Concatenation operation, one can define NFAs for the individual languages and then compose them in the obvious way to obtain the desired language.

- For a "concatenation language",
  - Connect every final state of the first NFA using a ε labeled edge to the initial state of the second NFA.
  - Make every final state of the first NFA non-final.

In effect, the final states of the resulting NFA are just the final states of the second NFA.

- For a "union language," introduce a new state from which lead to the NFA of the individual languages using ε moves. In effect, the final states of the resulting NFA are the union of the final states of the individual NFA.

Sometimes it is dead-easy to see that some languages can be dramatically simplified. For instance, you can rewrite the above languages as follows:

- L_b as (do it first!)[1] Figure 1 shows this NFA.
- L_c as (do it first!)[2] Figure 2 shows this NFA.

Note that in the JFLAP-drawn NFA, the Greek symbol λ is used in lieu of ε. They mean the same thing. Read the JFLAP documentation for more information.

JFLAP is available on our CADE machines as `/home/cs3100/jflap/bin/jflap`. JFLAP is extremely easy to install on your own own machines, from `http://www.cs.duke.edu/csed/jflap/`

But often we cannot so simplify languages.

- Let L_{div3} = the set of all strings over $\{0,1\}^*$ that are evenly divisible by 3 (when fed MSB-first).
- Let $L_{ends1011}$ = the set of all strings over $\{0,1\}^*$ ending in 1011.

Now try to simplify this language in your mind:

- $L_{cat1} = L_{div3}\ L_{ends1011}$
- $L_{union1} = L_{div3}\ \cup\ L_{ends1011}$

4 NFA does not help simplify Intersection Languages

However, NFAs are not helpful in rendering languages formed through intersection. For example for L_e, there is no direct way to draw an NFA.

L_e = the set of all strings containing a 1011 **and** are evenly divisible by 3 (when fed MSB-first).

One has to painstakingly understand the language in question and draw an NFA directly. Another algorithm that we shall study later is to build DFA for the individual languages and perform a language intersection operation on the DFA. More on that later.

5 Illustration of NFA and DFA using JFLAP

I will demonstrate the use of JFLAP to study NFA and DFA. Assignment #4 is on the use of JFLAP.

[1] all strings over $\{0,1\}^*$ (all strings drawn from$< \{0,1\}^*$) that contain a 1.

[2] The complement of the language of strings that begin with a 1 and end with a 0. In other words, the complement of $1(0+1)^*0$—as we would expression in a regular-expression notation later.

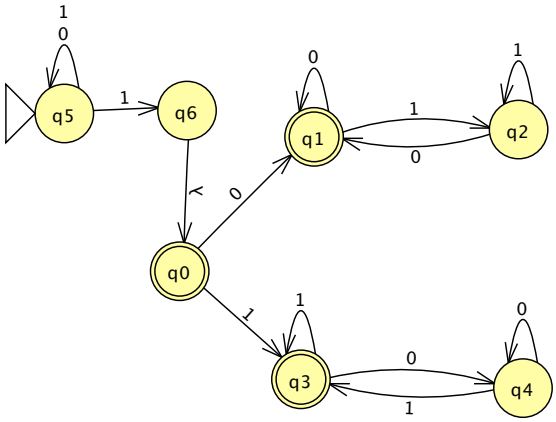


Figure 1: Language L_b has this NFA

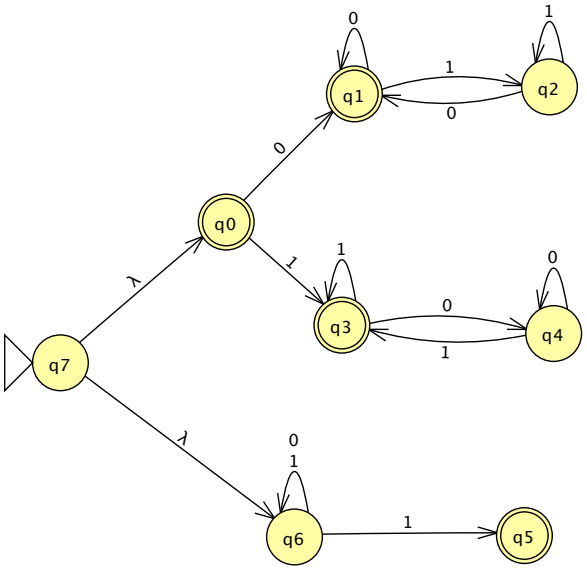


Figure 2: Language L_c has this NFA

6 The language of an NFA

We now present the concept of the language of an NFA through Python. You may not follow it rightaway, but give it a skim. Then come to the next section where the ideas are presented through math. Then when you understand something better, read the Python code, play again with JFLAP, etc.

6.1 Through Python

```
# First, import a few modules defined earlier.

from lang import *
from dfa import *
from praut import *

# Many higher-order functions are kept in functools. Import that now.

from functools import reduce

def fst(p):
    """ First of a pair."""
    return p[0]

def snd(p):
    """ Second of a pair."""
    return p[1]

def fn_dom(F):
    """ For functions represented as hash-maps (dicts), return their domain as a set.
    """
    return {k for k in F.keys()}

def fn_range(F):
    """ For functions represented as hash-maps (dicts), return their range as a set.
    """
    return {v for v in F.values()}

def mk_nfa(Q, Sigma, Delta, q0, F):
    """Make an NFA, doing the consistency checks needed.
    """
    assert(Sigma != {})
    assert("" not in Sigma)  # We don't allow epsilon in any alphabet (except for a GNFA)
    assert(q0 in Q)
    assert(set(F) <= Q)
    assert(fn_dom(Delta) <= product(Q, Sigma | {""}))
    # Delta maps state x input to sets of states...
    # One way to say this
    # assert(fn_range(Delta) <= pow(Q))
    # But this causes trouble as sets can't be members of sets..
    for x in list(Delta.values()):
        assert(set(x) <= Q)
    #
    return({"Q":Q, "Sigma":Sigma, "Delta":Delta, "q0":q0, "F":F})

# Here is how an NFA is made.

Q1 = {'S0','S1'}
```

```
Sigma1 = {'a','b'}

# Notice how smoothly we can define Delta1 - exactly as in math!

Delta1 = { ('S0', 'a'): {'S0', 'S1'},
           ('S1', 'a'): {'S0'},
           ('S1', 'b'): {},
           ('S0', '') : {'S1'} }

q01 = 'S0'

F1 = {'S1'}

NFA1 = mk_nfa(Q1, Sigma1, Delta1, q01, F1)

def mktot_nfa(N):
    """ Given a partially specified NFA,
    make it total by transitioning to the set of states {} wherever
    a move is undefined.
    """
    add_delta = { (q,c) : set({}) for q in N["Q"] for c in (N["Sigma"] | {""}) if (q,c) not in N["Delta"] }
    #
    add_delta.update(N["Delta"])
    #
    return {"Q": N["Q"], "Sigma": N["Sigma"], "q0": N["q0"], "F": N["F"], "Delta": add_delta}

def prnfa(N):
        """Prints the NFA neatly.
        """
        # Make the NFA total before printing
        Nt = mktot_nfa(N)
        print("")
        print("Q:", Nt["Q"])
        print("Sigma:", Nt["Sigma"])
        print("q0:", Nt["q0"])
        print("F:", Nt["F"])
        print("Delta:")
        print("\t".join(map(str, Nt["Q"])))
        print("-------------------------------------------------------------------------------------------")
        for c in (Nt["Sigma"] | {""}):
            nxt_qs = [Nt["Delta"][(q, c)] for q in Nt["Q"]]
            print("\t".join(map(str, nxt_qs)) + "\t\t" + c)
            print("")

def step_nfa(N, q, c):
    """Run NFA N from state q on character c or "". Return the next state.
    Step is without E-closure.
    """
    assert(c in (N["Sigma"] | {""}))
    assert(q in N["Q"])
    # We have to run it wrt the total version of the NFA. Expensive, so special case this check.
    if (q,c) in N["Delta"].keys():
        return N["Delta"][(q,c)]
    else:
        return set({})
```

```
#-- Eclosure version 1 ------------------------------------------------
# I leave it here for you to look at...
#
# def eps_fix(set_states, chain_len, N):
#     if (chain_len == 0):
#         return set_states
#     else:
#         all_state_sets_one_eps_away = list(map(lambda s: step_nfa(N, s, ""), set_states))
#         # basis case added to make reduction succeed
#         all_states_one_eps_away = reduce(lambda x, y: set(x) | set(y), all_state_sets_one_eps_away + [ set() ] )
#         return eps_fix( set(all_states_one_eps_away) | set(set_states), chain_len - 1, N)
#
# def eclosure(Q, N):
#     """Given nfa N and a SET OF states Q, close Q, return eclosure as a set of states.
#     """
#     return set() if Q == {} else eps_fix(Q, len(N["Delta"].items()), N)
#-----------------------------------------------------------------------

def ech(Allsofar, Previous, N):
    """Extend Allsofar until nothing new (that's not in Previous).
    """
    if (Allsofar == Previous):
        return Allsofar
    else:
        all_state_sets_one_eps_away = list(map(lambda q: step_nfa(N, q, ""), Allsofar))
         # basis case added to make reduction succeed
        all_states_one_eps_away = reduce(lambda x, y: set(x) | set(y), all_state_sets_one_eps_away + [ set({}) ] )
        return ech(set(all_states_one_eps_away) | set(Allsofar), Allsofar, N)

def eclosure(Q, N):
    """Given nfa N and a SET OF states Q, close Q, return eclosure as a set of states.
    """
    return ech(Q, set({}), N)

def ec_step_nfa(Q, c, N):
    """Return all states one "c" step away from Q (E-close before/after). i.e.,
    E-close Q, run N from all those states, Eclose those.
    If c is "", then se simply E-close q.
    len(NFA1["Delta"].items()) is the longest chain in the NFA.. we can iterate that much.
    """
    Eclosure = eclosure(Q, N)
    #
    all_state_sets_one_c_away = list(map(lambda s: step_nfa(N, s, c), Eclosure))
    #
    # basis case added to make reduction succeed
    all_states_one_c_away = reduce(lambda x, y: set(x) | set(y), all_state_sets_one_c_away + [ set({}) ] )
    #
    Eclosure_again = eclosure(all_states_one_c_away, N)
    #
    return Eclosure_again


def run_nfa(N, Q, s):
    """Run NFA from a set of states Q.
    """
    return eclosure(Q, N) if s=="" else run_nfa(N, ec_step_nfa(Q, s[0], N), s[1:])
```

```
def accepts_nfa(N, q, s):
    """NFA acceptance.
    """
    return (run_nfa(N, {q}, s) & N["F"]) != set({})

def accepts_nfav(N, q, s):
    """NFA acceptance - verbose version.
    """
    if (run_nfa(N, {q}, s) & N["F"]) != set({}):
        print("NFA accepts '" + s + "' by reaching " + str(run_nfa(N, {q}, s)))
        return True
    else:
        print("NFA rejects '" + s + "'")
        return False


ecs_S0_a = ec_step_nfa({'S0'}, 'a', NFA1)
ecs_S0_b = ec_step_nfa({'S0'}, 'b', NFA1)
#
ecs_S1_a = ec_step_nfa({'S1'}, 'a', NFA1)
ecs_S1_b = ec_step_nfa({'S1'}, 'b', NFA1)
#
ec_S0 = eclosure({'S0'}, NFA1)
ec_S1 = eclosure({'S1'}, NFA1)


Q2 = {'S0', 'S1', 'S2', 'S3', 'S4', 'S5', 'S6'}

Sigma2 = {'0','1'}

Delta2 = { ('S0', '')  : { 'S1' },
           ('S0', '0') : { 'S0' },
           ('S0', '1') : { 'S0', 'S5' },
           ('S1', '0') : { 'S2' },
           ('S2', '0') : { 'S3' },
           ('S2', '1') : { 'S3' },
           ('S3', '0') : { 'S4' },
           ('S3', '1') : { 'S4' },
           ('S5', '0') : { 'S6' },
           ('S5', '1') : { 'S6' } }

q02 = 'S0'

F2 = {'S4', 'S6'}

NFA2 = mk_nfa(Q2, Sigma2, Delta2, q02, F2)


ec_S0 = eclosure({'S0'}, NFA2)
ec_S1 = eclosure({'S1'}, NFA2)
ec_S4 = eclosure({'S4'}, NFA2)

ecs_S0_0 = ec_step_nfa({'S0'}, '0', NFA2)
ecs_S0_1 = ec_step_nfa({'S0'}, '1', NFA2)

ecs_S1_0 = ec_step_nfa({'S1'}, '0', NFA2)
ecs_S1_1 = ec_step_nfa({'S1'}, '1', NFA2)
```

```
ecs_S2_0 = ec_step_nfa({'S2'}, '0', NFA2)
ecs_S2_1 = ec_step_nfa({'S2'}, '1', NFA2)


assert(run_nfa(NFA2, {'S0'}, '000') == {'S3', 'S2', 'S1', 'S0', 'S4'})

assert(run_nfa(NFA2, {'S0'}, '00') == {'S3', 'S2', 'S1', 'S0'})

assert(run_nfa(NFA2, {'S0'}, '0') == {'S2', 'S1', 'S0'})

assert(run_nfa(NFA2, {'S0'}, '1') == {'S1', 'S0', 'S5'})

assert(run_nfa(NFA2, {'S0'}, '11') == {'S1', 'S0', 'S6', 'S5'})

assert(run_nfa(NFA2, {'S0'}, '101') == {'S3', 'S1', 'S0', 'S5'})


assert(accepts_nfa(NFA2, 'S0', '101') == False)

assert(accepts_nfa(NFA2, 'S0', '111') == True)

assert(accepts_nfa(NFA2, 'S0', '') == False)

assert(accepts_nfa(NFA2, 'S0', '0') == False)

assert(accepts_nfa(NFA2, 'S0', '00') == False)

assert(accepts_nfa(NFA2, 'S0', '000') == True)

assert(accepts_nfa(NFA2, 'S0', '001') == True)

assert(accepts_nfa(NFA2, 'S0', '011') == True)

assert(accepts_nfa(NFA2, 'S0', '100') == False)

assert(accepts_nfa(NFA2, 'S0', '10') == True)

assert(accepts_nfav(NFA2, 'S0', '10') == True) # prints also


assert(accepts_nfav(NFA2, 'S0', '010') == True) # prints also

assert(accepts_nfav(NFA2, 'S0', '010') == True) # prints also

assert(accepts_nfav(NFA2, 'S0', '0100') == False) # prints also

assert(accepts_nfav(NFA2, 'S0', '01000') == True) # prints NFA accepts '01000' by reaching {'S3', 'S2', 'S1', 'S0', 'S4'}
```

6.2 Through Math

We can define the language of an NFA using the idea of *Eclosure*. This notion, in effect, considers all ε-laden interpretations of strings in "one fell swoop." That is, when asked "does this NFA accept 010110", we must

consider strings of the form $\varepsilon 0101\varepsilon 1\varepsilon 0\varepsilon$ where a ε is snuck in wherever possible. This is precisely the idea of an NFA being allowed to take a ε wherever possible.

6.3 The language of an NFA

The language of an NFA consists of all those sequences of symbols that can be encountered while tracing a path from the start state to some final state. We eliminate all occurrences of ε from such sequences **unless the entire sequence consists of εs, in which case, we turn the sequence into a single**[3] **ε.**

6.4 *Eclosure* (also known as ε-closure)

$Eclosure(q)$ obtains, starting from a state q, the set of states reached by an NFA traversing zero or more ε labeled transitions. The best way to see which states are included in $Eclosure(q)$, for any q, is to imagine the following:

> Apply a *high-voltage* to state q; imagine that every ε edge is a diode that conducts in the direction of the arrow; now see which are the states that would be fatal to touch due to the high-voltage;[4] all those are in $Eclosure(q)$.

Example: Let us obtain the *Eclosure* of various states in Figure 3.

- The Eclosure of state IF is IF itself. The high-voltage spreads from IF to itself.
- What is *Eclosure* of FA? Applying high-voltage to this state, it spreads to state FA, to $B0$, to $A1$, and finally, to FB. For example, $Eclosure(FA) = \{FA, B0, A1, FB\}$.

Unfortunately, an intuitive definition in terms of voltages isn't rigorous enough! Hence, we set up an alternate definition of $Eclosure(q)$ as follows:

- First, we need a way to compute all states which can be reached from a given state by traversing ε edges.

  Let $\rightarrow \subseteq Q \times Q$ be an arbitrary relation over Q. Then, given $\rightarrow \subseteq Q \times Q$, we will define a postfix usage of this operator, namely $q \rightarrow$, to be the *image* of q under $\rightarrow$:

$$q \rightarrow = \{x \mid \langle q, x\rangle \in \rightarrow\}.$$

- Now define the relation $\overset{\varepsilon}{\rightarrow}$ which is a subset of $Q \times Q$:

$$\overset{\varepsilon}{\rightarrow} = \{\langle q_1, q_2\rangle \mid q_1, q_2 \in Q \wedge q_2 \in \delta(q_1, \varepsilon)\}.$$

- Next, define the reflexive and transitive closure of $\overset{\varepsilon}{\rightarrow}$ in the usual way. Call it $\overset{\varepsilon}{\rightarrow}^*$.

Given all this, we define

$$Eclosure(q) = q\overset{\varepsilon}{\rightarrow}^*.$$

Here, we use the $\overset{\varepsilon}{\rightarrow}^*$ as a postfix operator. The reflexive part above is very important to ensure that q gets included within $Eclosure(q)$. The overall effect of employing $\overset{\varepsilon}{\rightarrow}^*$ as a postfix operator is to force all ε-only paths to be considered.

[3] Please note that ε is *not* part of the alphabet of the NFA. Never, never, never! Same deal as with a DFA!

[4] An *ideal* non-leaky diode that does not break down.

Example: Redoing our example with respect to Figure 3,

$$Eclosure(FA) = FA\overset{\varepsilon}{\rightarrow}^*.$$

- This "grabs" state FA itself (the reflexive part of $\overset{\varepsilon}{\rightarrow}^*$ does this).
- Next, $B0$ enters this set, as it is one step away. In fact, $B0$ is in $FA\overset{\varepsilon}{\rightarrow}^1$.
- Next, $A1$ enters this set, as it is two steps away. In fact, $A1$ is in $FA\overset{\varepsilon}{\rightarrow}^2$.
- Finally, FB enters this set, as it is three steps away. In fact, FB is in $FA\overset{\varepsilon}{\rightarrow}^3$.
- No more states enter $Eclosure(FA)$.

As said earlier, *Eclosure* helps define the behavior of an NFA, as well as its language more directly. It also helps us define the NFA to DFA conversion algorithm, as we shall see very soon.

6.5 Language of an NFA

Having defined *Eclosure*, we can now formally define the language of an NFA. For a string $x \in \Sigma^*$, define $\hat{\delta}(q_0, x)$ of an NFA to be the set of states reached starting from $Eclosure(q_0)$ and traversing all the symbols in x, taking an *Eclosure* after every step. We are, in effect, taking the image of q_0 under string x, except that we are allowing an arbitrary number of εs to be arbitrarily inserted into x. Also, we will overload *Eclosure* to work over sets of states in the obvious manner, as follows:

$$Eclosure(S) = \{x \mid \exists s \in S : x \in Eclosure(s)\}.$$

This definition can also be written as

$$Eclosure(S) = \cup_{s \in S}\ Eclosure(s).$$

Likewise, we overload δ to work over sets of states:

$$\delta(S, a) = \{x \mid \exists s \in S : x \in \delta(s, a)\}.$$

Now we define $\hat{\delta}(q, x)$, the 'string transfer function,' for state q and string x inductively as follows:

$\hat{\delta}(q, \varepsilon) = Eclosure(q)$.

For $a \in \Sigma$ and $x \in \Sigma^*$,

$\hat{\delta}(q, ax) = \{y \mid \exists s \in Eclosure(\delta(Eclosure(q), a)) : y \in \hat{\delta}(s, x)\}$.

In other words, for every symbol a in ax, we *Eclose* state q, "run" a from each one of these states, and *Eclose* the resulting states. From each state s that results, we recursively run x. Notice that we apply *Eclosure* before as well as afterward. While this is strictly redundant, it leads to definitions that are simpler and more general, and hence easier to reason about.[5] Specifically, in the definition of $\hat{\delta}(q, ax)$, we need not assume that q is an

[5] Another way to set up the definitions would have been to start the NFA in the *Eclosure* of its start state, and at each stage perform a δ step followed by one *Eclosure* step.

already *Eclosed* state, even though in the current context of its usage, we will be inductively guaranteeing that to be the case because: (i) we begin with $\hat{\delta}(q, \varepsilon) = Eclosure(q)$, and (ii) in $\hat{\delta}(q, ax)$, we restore "Eclosedness." Finally, the language of an NFA N is

$$\mathcal{L}(N) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}.$$

In other words, after running the NFA from state q_0 with input w, we see whether any 'token' has reached a final state.

A good way to intuitively understand the above definitions pertaining to NFAs is through the following 'token game:'

- Place a token in state q_0. Spread one copy of the token to each state in $Eclosure(q_0)$.
- For each symbol a from Σ that is entered, advance each token to its set of a successors.
- *Eclose* the tokens and continue.
- If and when one of the tokens reaches some final state, the string seen so far is accepted.

Illustration 6.1 The NFA in Figure 3 has string 0001 in its language. First, $Eclosure(IF) = \{IF\}$. After the first 0, the NFA is in state FA, whose *Eclosure* is $\{FA, B0, A1, FB\}$. After the second 0, $A1$ goes to FA, and the *Eclosure* results in $\{FA, B0, A1, FB\}$. The same happens after the third 0. After the 1, FA and $A1$ go to $A1$, $B0$ goes to FB, and the token in FB goes to $\emptyset$, resulting in $\{A1, FB\}$, which is also its own *Eclosure*. This matches the definition

$$\hat{\delta}(q, ax) = \{y \mid \exists s \in Eclosure(\delta(Eclosure(q), a)) \ : \ y \in \hat{\delta}(s, x)\}$$

as argued below:

- $Eclosure(\delta(Eclosure(IF), 0))$ is $\{FA, B0, A1, FB\}$.
- We recursively process the remaining input 001 from these states to reach $\{A1, FB\}$.

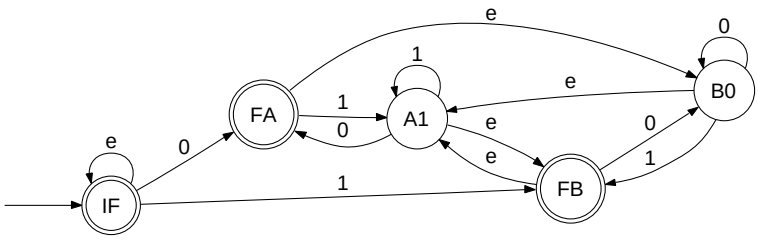


Figure 3: An example NFA