Introduction to the theory of computation
week 12 (Course Notes, chapter 7)

2nd August 2005

Regular expression	Language
$(0+1)^*$	
$((0+1)(0+1)^*)$	
$((0+1)(0+1))^*$	
$\varepsilon + 0 + 0(0+1)^*0$	

In general there is more than one way to denote the same language. Suppose $L = \{x \in \Sigma^* : x \text{ contains at least one } 0\}$. Then $L = L(R_1) = L(R_2) = L(R_3)$, for $R_1 = (0+1)^*0(0+1)^*$, $R_2 = 1^*0(0+1)^*$, $R_3 = (0+1)^*01^*$. To prove from scratch that two regular expressions are equivalent means proving that the languages denoted by them are equal as sets, usually by mutual inclusion:

Claim: $L(1^*0(0+1)^*) \subseteq L$.

Proof: Let x be an arbitrary string in $L(1^*0(0+1)^*)$. This means, by definition that $x = u0t$, where $u \in L(1^*)$, and $t \in L((0+1)^*)$ ($L(0)$ contains only the string 0). By inspection $u0t$ contains at least one 0, so $x \in L$. Since x was chosen arbitrarily, $L(1^*0(0+1)^*) \subseteq L$. QED.

Claim: $L \subseteq L(1^*0(0+1))$.

Proof: Let $x \in L$ be an arbitrary string in L. Since x has at least one zero, it certainly has a first zero, so x can be parsed as $x = u0v$, where u is the prefix (possibly empty) of x that precedes the first zero, and v is the suffix of x after the first zero. Thus $u \in L(1^*)$ (since it contains no zeros) and $v \in L((0+1)^*)$ (it is certainly a string over $\{0,1\}$), and $x = u0v \in L(1^*)L(0)L((0+1)^*) = L(1^*0(0+1)^*)$. Since x was chosen arbitrarily, $L \subseteq L(1^*0(0+1)^*)$. QED.

Equivalence: Since $L \subseteq L(1^*0(0+1)^*0$ and $L(1^*0(0+1)^*)$, the languages are equivalent.

The remaining equivalences are shown in an additional section to last week's lecture summary.

Here are some common equivalences of regular expressions. They either follow from the properties of sets or concatenation, or (for example, in the case of distributivity) they can be proved from scratch, using the mutual containment technique just illustrated. Suppose that R, S, and T are regular expressions.

Commutativity: $(R+S) \equiv (S+R)$

Associativity: $((R+S)+T) \equiv (R+(S+T))$. $((RS)T) \equiv (R(ST))$.

Left distributivity: $(R(S+T)) \equiv (RS+RT)$

Right distributivity: $((R+S)T) \equiv (RT+ST)$

IDENTITY (UNION): $(R + \emptyset) \equiv R \equiv (\emptyset + R)$

IDENTITY (CONCATENATION): $(R\varepsilon) \equiv R \equiv (\varepsilon R)$

ANNIHILATOR (CONCATENATION): $(\emptyset R) \equiv \emptyset \equiv (R\emptyset)$

IDEMPOTENCE: $(R^*)^* \equiv R^*$

This adds new tools to demonstrate equivalence between regular expressions. You can use the "from scratch" technique to prove that $((\varepsilon + R)R^*) \equiv R^*$, for any regular expression R, and call your result $(**)$. Now use $(**)$ plus the other equivalences to show that $(0110 + 01)(10)^* \equiv 01(10)^*$

$$\begin{array}{rrcl} \text{distributivity} & (0110 + 01)(10)^* & \equiv & (01(10 + \varepsilon))(10)^* \\ & \text{associativity :} & \equiv & 01((10 + \varepsilon)(10)^*) \\ & \text{commutativity :} & \equiv & 01((\varepsilon + 10)(10)^*) \\ & \text{by } (**) & \equiv & (01)(10)^* \end{array}$$

To show the opposite: that $R_1 \not\equiv R_2$, you need to find a string in $L(R_1)$ and prove that it could not be in (R_2) (or vice-versa). For example, if $R_1 = (0 + 1)^*(11 + 0)$ and $R_2 = 0(00)^*$, then $11 \in L(R_1)$, but $L(R_2)$ has no strings containing any 1s, so $R_1 \not\equiv R_2$.

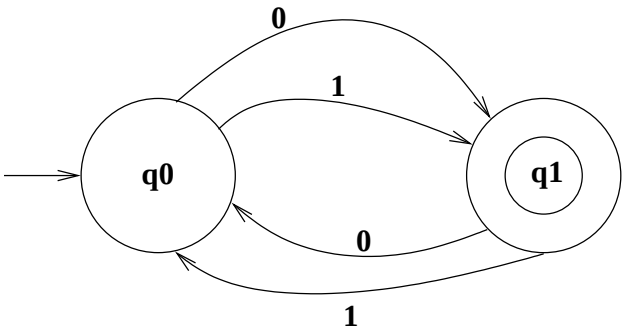
Often the translation from a verbal description of a language to a regular expression is so simple that we don't bother with a proof. If $L = \{x \in \{0,1\}^* : x \text{ contains no more than 2 } 0s\}$, then we are fairly confident that $L = L(1^* + 1^*01^* + 1^*01^*01^*)$. However, a rigorous proof is always possible, and sometimes catches glitches in our regular expression.

CAN REGULAR EXPRESSIONS EXPRESS EVERYTHING?

Since our regular expression language has built-in operations to express union, concatenation, and Kleene start, regular expression can certainly denote the union, intersection, and Kleene star of languages for which there are regular expressions. What about the intersection (How would you write a regular expression for $L(R) \cap L(S)$? Although it is not easy, it turns out that if R and S are regular expressions, then there is a regular expression for $L(R) \cap L(S)$ and for $\mathbf{Rev}(L(S))$ (the reverse of $L(S)$) and $\overline{L(S)}$.

However, some easily-described languages can not (in other words, even by the cleverest individual) be denoted by a regular expression. An example is the language over $\{0,1\}$ with an equal number of 0s and 1s. To see why, we take a detour and specify languages in a different way: we use a Finite State Automata (FSA) to accept exactly those strings in L and reject all other strings.

Here's a diagrammatic example that (I claim) accepts the language of strings over $\{0,1\}$ that have an odd number of characters. We start at the (unique) source state, process a string from left to right, and follow an edge labelled with our current character to the next state. If we are in the accepting state (there could be more than one) when we run out of characters, then the string is accepted. Otherwise it is rejected. Trace through where strings 0111 and 010 end up.



A Deterministic Finite State Machine (DFSA) is a quintuple $M = (Q, \Sigma, \delta, s, F)$, where

- Q is a finite set of states
- Σ is a finite alphabet
- δ $Q \times \Sigma \to Q$ is the transition function. In the diagrammatic representation, if $q, q' \in Q$, then $\delta(q, a) = q'$ means there is an edge labelled a from q to q'. By the way, $q = q'$ is certainly allowed.
- $s \in Q$ is the start state (indicated by an arrow without a predecessor state in my diagram)
- $F \subseteq Q$ is a set of accepting states.

It is natural to extend the transition function to all strings in Σ^*:

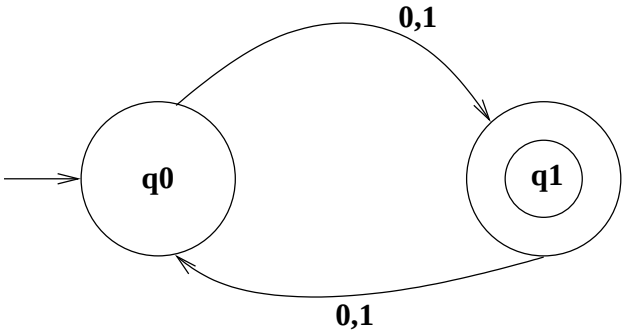
Basis: If $x = \varepsilon$, then $\delta^*(q, \varepsilon) = q$.

Induction step: If $x = x'a$ for $x' \in \Sigma^*$ and $a \in \Sigma$, then $\delta^*(q, x) = \delta(\delta^*(q, x'), a)$.

When $\delta^*(q, x) = q'$, we say the string x takes M from q to q'. This means that

Definition: String x is accepted (recognized) by M if and only if $\delta^*(s, x) \in F$, in other words, x takes M from its initial state to an accepting state. The language accepted (recognized) by M is denoted $L(M)$.

There are a couple of conventions for simplifying DFSAs without losing information about what language they accept. First, transitions can be combined, so that if there are several edges from q to q', each labelled with a single character, they can be replaced by a single edge labelled with a list containing all those characters. Second, if there are any states in the DFSA from which it is impossible to reach an accepting state (these are called dead states), then those states and all edges into or out of them may be removed. Here is my example with the first simplification:



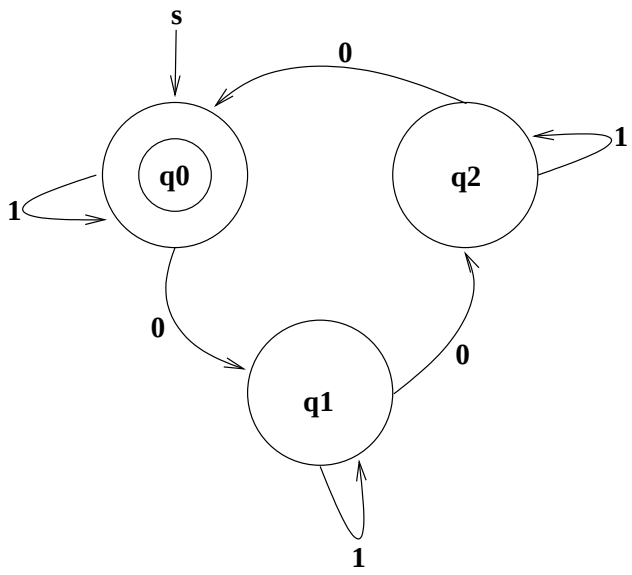
Can you come up with a regular expression that denotes $L(M)$ for my example?

Proving that a DFSA is correct

A useful tool for both designing a DFSA, and proving it correct is a state invariant. A DFSA M accepts language L if every string of L takes M to an accepting state, and every non-string of L takes M to some other (i.e. non-accepting state). Consider

$$L = \{x \in \{0, 1\}^* : x \text{ has a multiple of 3 } 0s\}$$

Each string in $\{0, 1\}^*$ has either a multiple of 3 zeros, or $3k + 1$ zeros (for some integer k) or $3k + 2$ zeros. If we design a DFSA so that each state corresponds to one of these possibilities, it seems to be correct:



If we write down why we think this FSA accepts L, we get a state invariant, that for any string $x \in \{0,1\}^*$:

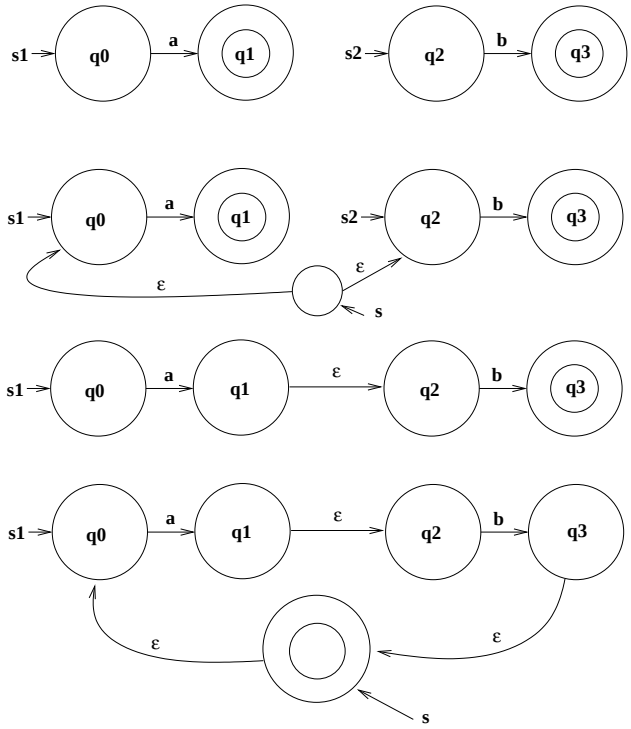
$$\delta^*(q_0, x) = \begin{cases} q_0, & \text{if } x \text{ has 0 mod 3 zeros} \\ q_1, & \text{if } x \text{ has 1 mod 3 zeros} \\ q_2, & \text{if } x \text{ has 2 mod 3 zeros} \end{cases}$$

This state invariant can be proved, by induction on $|x|$, for every string in $\{0,1\}^*$. Once the state invariant is proved, it is pretty easy to show that every string in L takes M to state q_0 (accepted) so $L \subseteq L(M)$, and every string not in L takes M to some other state (not accepted), so $L(M) \subseteq L$. This is a standard way to prove that a DFSA accepts a particular language (see an example in the Course Notes, pages 205–207).

DFSAs $\equiv$ NFSAs $\equiv$ Regular Expressions

Our next step is pretty ambitious. We'd like to show that for every regular expression R there is a DFSA M_R that accepts $L(R)$. The first step is not too hard: we can exhibit $DFSAs$ that accept $L(\emptyset)$, $L(\varepsilon)$, and $L(a)$, for any $a \in \Sigma$. The next step is to prove that we can always combine DFSAs in ways that correspond to union, concatenation, and Kleene star. In other words, if R and S are regular expressions, and M_R and M_S accept $L(R)$ and $L(S)$, we'd like to be able to say that we can build DFSAs that accept $L(R+S)$, $L(RS)$ and $L(R^*)$.

It turns out that although we can always do this in practice, we can't provide a standard recipe for directly transforming FSAs in this way. Suppose $R = a$, and $S = b$. We'd like DFSAs that would accept $L((a+b))$, $L(ab)$, and $L((ab)^*)$. The following diagrams suggest an approach, but the transition function, δ, is not deterministic (it may have two or more targets for the same input). We call such an FSA a Non-deterministic Finite State Automata (NFSA):



There are a few things to notice. Transitions for some characters are not indicated — by convention (in both DFSAs and NFSAs) they lead to dead states which, in turn, can never lead to an accepting state in a DFSA. You could probably design DFSAs for these three cases (without the ε transitions), but you couldn't give a master algorithm that would produce DFSAs for EVERY union, concatenation, or Kleene star of DFSAs.

It seems disturbing that the state a string ends up in depends on the roll of a dice: does it make the epsilon transition or not?

We need NFSAs

We need the flexibility of NFSAs in order to produce an FSA for every regular expression. The flexibility includes the possibility that the transition function $\delta(q, a)$ might have two or more targets, and that $\delta(q, \varepsilon)$ may be defined. We also need to preserve some meaning: we need to be able to say when an NFSA accepts (and doesn't accept) a string. Here's the definition

An NFSA is a quintuple $(Q, \Sigma, \delta, s, F)$, with the following meanings

- Q is a finite set of states
- Σ is a finite alphabet
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q)$ is the transition function, ($P(Q)$ is the power set, or the set of subsets, of Q). Assume that $q \in \delta(q, \varepsilon)$, then we call $\delta(q, \varepsilon)$ $\varepsilon(q)$ (the set of states reachable from q with ε-transitions).
- $s \in Q$ is the start state
- $F \subseteq Q$ are the accepting states

The difference from DFSAs is the transition function, δ, which now takes the FSA from a given state to several states (a subset), and can now make transitions that don't correspond to any symbol (ε transitions). Starting from the same state, q, the same string x takes an NFSA to several different states, depending

on which of several targets of δ each component of x takes the machine to. Thus, the extended transition function, $\delta^*(q, x)$ is defined to be the set of all possible states that x can take the NFSA to from q:

BASIS: $\delta^*(q, \varepsilon) = \varepsilon(q)$ (the set of states reachable from q with ε-transitions)

INDUCTION STEP: If $|x| > 0$, then $x = ya$, and

$$\delta^*(q, x) = \delta^*(q, ya) = \cup_{q' \in \delta^*(q,y)}(\cup_{q'' \in \delta(q',a)}\varepsilon(q''))$$

We say that NFSA M accepts string x if x *could* take M from s to an accepting state, in other words

NFSA M accepts x if and only if $\delta^*(s, x) \cap F \neq \emptyset$.

This allows us to specify which strings an NFSA accepts (and which it rejects). And the flexibility of NFSAs allow us to specify languages with simpler diagrams (fewer states) than DFSAs. But we have this nagging suspicion that the "magic" of non-determinism means that there's no connection between DFSAs and NFSAs.

NFSAs $\equiv$ DFSAs

It is easy to see that if M_D is a DFSA that accepts L, then there is an NFSA M_N that also accepts L: simply use M_D itself — it is an NFSA that never uses ε transitions, and $\delta(q, a)$ always has a single state (a subset containing one state).

The other direction takes more work. If M_N is an NFSA with states Q, then we can think of every subset of Q (there are lots — $2^{|Q|}$ of them) as a state. If $\hat{q}$ is a subset of Q, then $\delta(\hat{q}, a) = \hat{\hat{q}}$, where $\hat{\hat{q}}$ is the subset of states in Q that can be reached from any state in $\hat{q}$ by an a-transition plus some ε transitions (we assume there is always an epsilon transition from a state to itself). Here is the definition of the deterministic machine $\hat{M}_N$, in terms of the components of NFSA M_N:

- $\hat{Q} = P(Q)$
- $\Sigma = \Sigma$ (same alphabet labels edges)
- $\hat{\delta}(\hat{q}, a) = \cup_{q' \in \hat{q}}(\cup_{q'' \in \delta(q',a)}\varepsilon(q''))$ (deterministic!)
- $\hat{s} = \varepsilon(s)$ (the states reachable from s by an ε-transition)
- $\hat{F} = \{\hat{q} \in P(Q) : \hat{q} \cap F \neq \emptyset\}$ (all subsets with a non-empty intersection with the set of accepting states)

This is called the SUBSET CONSTRUCTION, and it can be fairly daunting (given the exponential size of $P(A)$). However, by inspecting $\hat{\delta}(\hat{q}, a)$ you should be able to convince yourself that a string x will move M_N from s to some subset of Q that has a non-empty intersection with F (definition of M_N accepting x) if and only if that string will move deterministic $\hat{M}_N$ to an element of $\hat{F}$ (the definition of $\hat{M}_N$ accepting x). This observation is made precise in Lemma 7.18 and Theorem 7.19 in the Course Notes.

The result is that we can talk about generic FSAs (Finite State Automata), and whenever it suits some purposes (flexibility) we can assume it is an NFSA, but whenever it suits other purposes (definite description) we can assume it is a DFSA.

EVERY REGULAR EXPRESSION HAS AN FSA

For every regular expression in the basis, we can easily exhibit an FSA (actually, you should be able to come up with a DFSA). Using the three constructions for $a + b$, ab, and $(ab)^*$ you can now create an NFSA for the induction step of regular expression construction. Repeating this, you can create an FSA for any regular expression. If you insist on it being a DFSA, you can use the subset construction to make it so.

This shows that if R denotes a language $L(R)$, then there is an FSA M_R that accepts $L(R)$. Any language you can denote with a regular expression can be accepted by some FSA, and (if you want) it can be a DFSA.

What about the other direction? Is there a regular expression for every FSA? The answer is yes, but (again) it takes a little work.