Introduction to the theory of computation
week 13 (Course Notes, chapter 7)

11th August 2005

- Midterm 2 is marked. I'll hand it back before lecture and during the break (?!!). The average was 65%, the term average is 71%
- Office hours: Tuesday after lecture, Wednesday 11–4:30, Thursday after lecture, and Monday 2:30–4:30, 5:30–8pm
- Re-mark requests, medical excuses, and special consideration: email me for my decision during exam week.

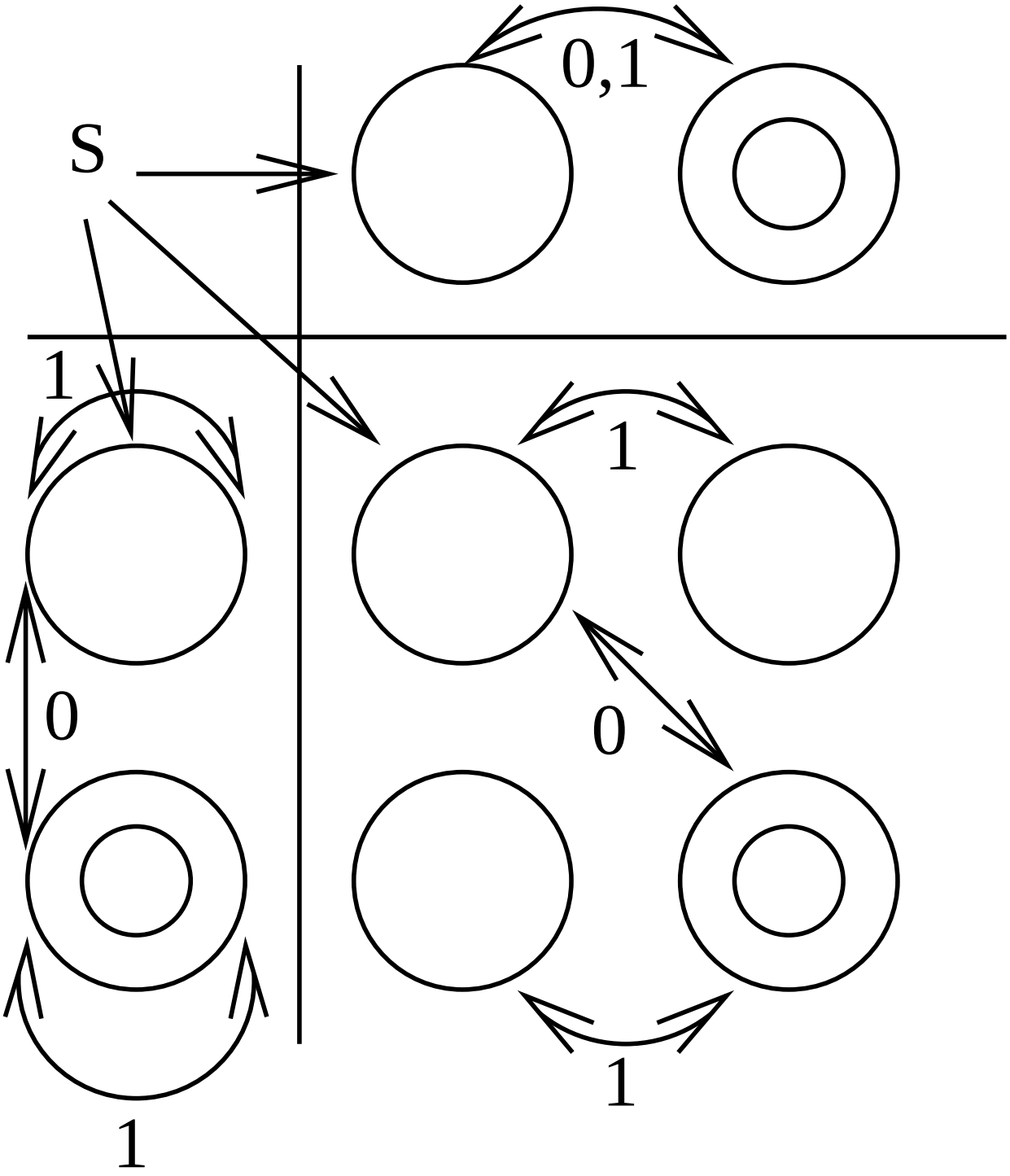
Every regex has an FSA

Last week we showed that if we used Non-deterministic Finite State Automata (NFSA) we could implement the union ($+$), concatenation (RS) and Kleene star ($*$) operations of regular expressions. We also showed that we could build DFSAs that accept $L(\emptyset)$, $L(\varepsilon)$, and $L(a)$ (for any $a \in \Sigma$). Thus every language that can be denoted by a regular expression can be accepted by some NFSA. In addition, the subset construction allows us to convert an NFSA into a DFSA, so if you can write a regular expression for a language, you can build a DFSA for it.

In addition, we can implement FSAs for some of the common operations on languages. If we hav e a DFSA that accepts language L, we can transform it into a DFSA that accepts $\overline{L}$ (the complement, or set of strings not in $\overline{L}$) by simply changing accepting states to non-accepting states and vice-versa. If we have an FSA that accepts L, we can transform it into an FSA that accepts $rev(L)$ (the language of reverses of strings in L) by making the start state the unique accepting state, reversing the direction of all transitions, and adding an ε-transition from a new start state to all the former accepting states. If we have DFSAs that accepts languages L_1 and L_2, then we can build a new DFSA that accepts $L_1 \cap L_2$ (the strings that are in both L_1 and L_2) as follows:

1. The new machine's states are the Cartesian product of the states of L_1 and L_2: $Q_{L_1 \cap L_2} = Q_{L_1} \times Q_{L_2}$.
2. The new machine's transitions function is the product of the old transition functions: $\delta_{L_1 \cap L_2}((q_{1i}, q_{2j}), a) = (\delta_{L_1}(q_{1i}, a), \delta_{L_2}(q_{2j}, a))$.
3. An accepting state is in the Cartesian product of accepting states, that is an accepting state is $(q_{1i}, q_{2j}) \in F_1 \times F_2$.

Here's an an example where M_1 accepts the set of binary strings with an odd number of characters and M_2 accepts the set of binary strings with an odd number of zeros:



For each transition in the machine that implements the intersection, M_1 tells you which column your state is in, and M_2 tells you which row your state is in (see Exercise 7.8 in the Course Notes).

Every FSA has a regex

If M is a DFSA, then we can certainly devise regular expressions for the simplest transitions, those that take us a single "hop" from one state to another. Suppose a string x takes M from state q to state q'. Then either $x = a$, where $\delta(q, a) = q'$, or $x = \varepsilon$ and $q = q'$.

Label all the states with positive integers $1, \ldots, n = |Q|$. Then the (very) small language that takes M directly from state i to state j without any intermediate states is denoted $L^0_{i,j}$ (meaning the set of all strings that take M from state i to state j using only intermediate states 0 or less). This means that $L^0_{i,j} = \{a : \delta(i, a) = j\}$ or $L^0_{i,j} = \{\varepsilon + a : \delta(i, a) = j, i = j\}$. This language is finite, since there are a finite set of characters that take M from i to j.

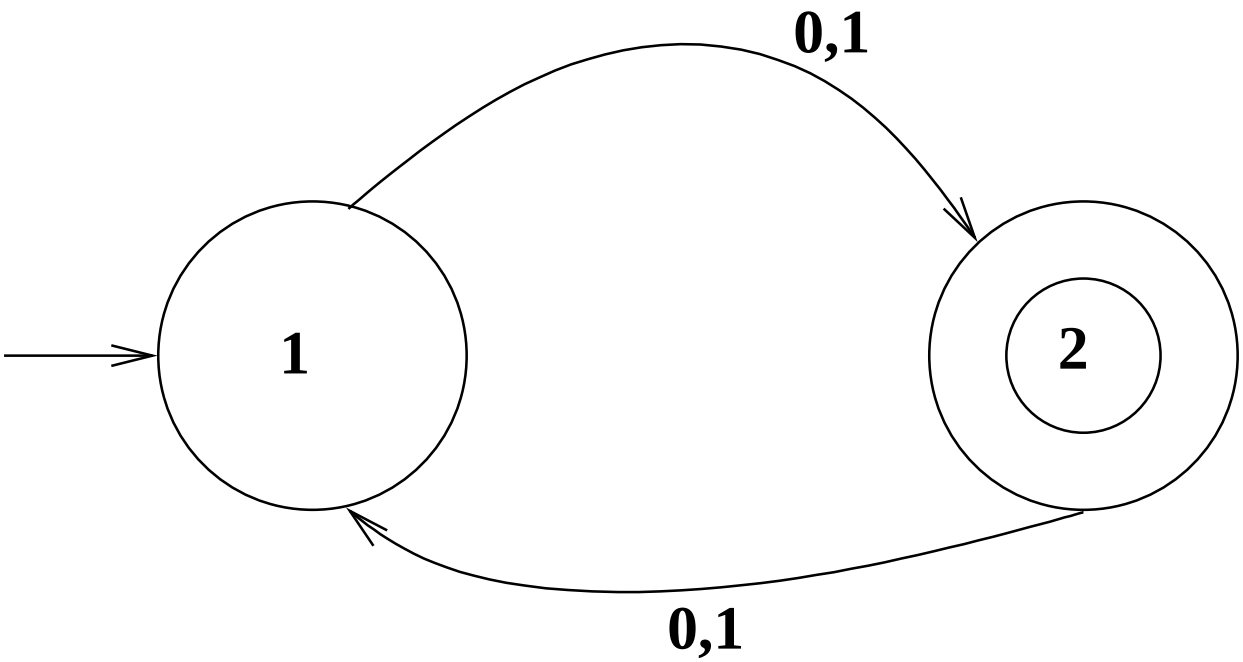
Now think inductively. If you already know $L^k_{i,j}$ for every possible pair of states $i, j \in Q$ and some particular k, then $L^{k+1}_{i,j}$ simply consists of $L^k_{i,j}$ (they worked without passing through $k+1$), plus the strings that take M from i to $k+1$ using only intermediate states up to k, then take M from $k+1$ to $k+1$ using only intermediate states up to k, and then take M from $k+1$ to j, using only intermediate states up to k:

$$L^{k+1}_{i,j} = L^k_{i,j} \cup L^k_{i,k+1} L^k_{k+1,k+1} L^k_{k+1,j}$$

If you've worked out the corresponding regular expressions $R^k_{i,j}$, $R^k_{i,k+1}$, and $R^k_{k+1,j}$, then you have:

$$R^{k+1}_{i,j} = R^k_{i,j} + R^k_{i,k+1} R^k_{k+1,k+1} R^k_{k+1,j}$$

This provides you with an algorithm for finding a regular expression for any DFSA. First find R^n_{s,f_t}, where s is the start state, and for any accepting state $f_t \in F$, and where n is the number of states, $|Q| = n$. This is not a nice algorithm with a paper and pencil, but it is a snap for a computer (it is an example of a polytime DYNAMIC PROGRAMMING algorithm). Consider the machine for accepting strings of odd length over $\{0, 1\}$:



Start/finish	0	1	2
R^k_{11}	ε	ε	
R^k_{12}	$(0+1)$	$(0+1)$	$(0+1)+(0+1)(\varepsilon+(0+1)(0+1))^*(\varepsilon+(0+1)(0+1))$
R^k_{21}	$(0+1)$	$(0+1)$	
R^k_{22}	ε	$\varepsilon+(0+1)(0+1)$	

New regular languages from old

We have shown that the set of languages denoted by regular expressions and the set of languages accepted by FSAs are identical — you can derive one from the other. These languages are called regular, and given regular languages L_1 and L_2, you can derive regular expressions for the following regular languages:

- $L_1 \cup L_2$
- $L_1 \cap L_2$
- $\overline{L_1}$
- $L_1 L_2$
- L_1^*
- $\text{rev}(L_1)$

Not every language is regular

Since we now have several tools for generating regular languages, it is tempting to think that all easily-described languages are regular. This is not true. Here is a property of all regular languages. If you can prove that some language you are interested in does NOT have this property, then it is not regular.

Theorem (pumping lemma): Suppose $L \subseteq \Sigma^*$ is a regular language. Then there is some $n \in \mathbf{N}$ (n depends on L) such that if $|x| \geq n$ and $x \in L$, then there are strings $u, v, w \in \Sigma^*$ such that $x = uvw$, $v \neq \varepsilon$, $|uv| \leq n$, and $uv^k w \in L$ for all $k \in \mathbf{N}$.

Proof: Let M_L be a DFSA that accepts L, and suppose M_L has n states. If $x \in L$ and $|x| \geq n$, then denote $x = a_0 a_1 \cdots a_n \cdots a_l$, where $a_0 = \varepsilon$ is prepended to make the bookkeeping easier. Now denote $q_m = \delta^*(s, a_0 \cdots a_m)$ (the state that $a_0 \cdots a_m$ takes the machine to), so $q_0 = \delta^*(s, \varepsilon)$ is the starting state). Since there are $n+1$ variables $q_0, \ldots, q_n$, there must be (pigeonhole principle) at least two variables that represent the same state, say $q_i = q_j$, for some $0 \leq i < j \leq n$. Let $u = a_1 \cdots a_i$ (u is empty if $i = 0$), $v = a_{i+1} \cdots a_j$, and $w = a_{j+1} \cdots a_l$. So $x = uvw$, $|uv| \leq n$, and $|v| \neq 0$ (since $i < j$). Then $\delta^*(s, u) = q_i$, $\delta^*(q_j, w) = q_l$ (an accepting state), and $\delta^*(q_i, v) = q_j = q_i$. By iterating the extended transition function, we get $\delta^*(q_i, v^k) = q_i$, so $\delta^*(s, uv^k w) = q_l$, an accepting state, and $uv^k w$ is accepted by M for all $k \in \mathbf{N}$. QED.

You can use the pumping lemma to prove that a given language is not regular. Here's an example.

Claim: Suppose L is the language of binary strings with the same number of 0s and 1s. Then L is not regular.

Proof: Suppose, for the sake of contradiction, that L is regular. Thus, by the pumping lemma, there is some n depending on L such that if $x \in L$ and $|x| \geq n$, then x can be expressed as uvw with $|uv| \leq n$, $v \neq \varepsilon$, and $uv^k w \in L$ for every natural number k. Let $x = 1^n 0^n$ (a string of n 1s concatenated with n 0s). Then $x \in L$, so (assuming the pumping lemma applies) uv is a prefix of 1^n, so $v = 1^j$, for some $1 \leq j \leq n$. The pumping lemma asserts that $uv^2 w = 1^{n+j} 0^n \in L$, which contradicts the definition of L as containing strings with the same number of 0s and 1s. Thus the assumption that L is regular is false.

Exam summary and tactics

The final exam will last three hours and consist of nine questions, worth 10 marks each. You are allowed to use pens, pencils, erasers, and ingenuity.

Although they have the same weight, you will probably find some questions easier than others. Roughly six questions will be similar to topics worked in assignments, lectures, or quizzes. Three questions may be somewhat less similar. You will have the option of leaving any question blank (or writing "I don't know how to answer this question") and getting 20% (or 2/10) for that question.

The exam is comprehensive, although Chapters 6 and 7 have greater weight. You should be familiar with the following topics:

- induction (simple, complete, structural, well-ordering)
- recursive definition of functions
- program correctness (both iterative and recursive)
- propositional and predicate logic
- regular expressions, FSAs, regular languages, pumping lemma

Exam preparation

I recommend that you review the following material in (roughly) this order of priority:

1. Assignments 1–4, plus their posted solutions
2. Lecture summaries
3. Midterm solutions
4. Quiz solutions
5. Course Notes

Office hours

My remaining office hours are Wednesday 11–4:30 in BA3222, Thursday 9pm–? in BA1180, and Monday 2–4:30 and 5:30–8pm in BA3222.

Tactics

The standard platitudes apply: try to be as well-rested as possible, have a source of caffeine (if you use it) handy. Also

- Read every exam question, since you may find some easier than others.
- Be sure you understand what you're being asked to do before you begin writing. Ask me or the other invigilator if you have questions, and we will try to prove a fair answer.
- Write the outline of a proof even if there are steps you cannot fill in. Indicate which steps you cannot fill in, rather than writing something you don't believe. Specify things you assume without proof.
- Use the space provided on the exam paper as a guide for length. The backs of the pages are provided for overflow.