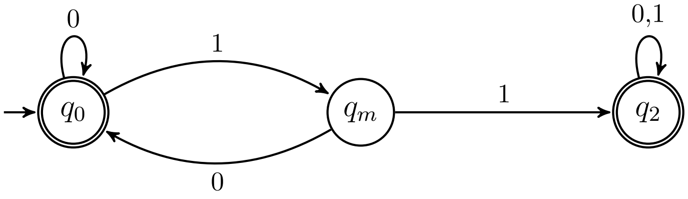
CS 334 Fall 2015 Exam 1 SAMPLE (100 points) NAME:

Here are 12 sample problems. Our exam will look like 7 or 8 of them, probably.

1. Consider the DFA Q_s:



Indicate the acceptance/rejection of each string by Q_s:

String	Accept	Reject
ε	✓	
00	✓	
1110	✓	
10101		✓
110001	✓	

Express in English the language recognized by Q_s.

This is easiest to express in English by what is not included, that is, anything that ends in the middle state q_m:
It is the set of all strings except those ending in "1" without "11" as a substring.

Given a regular expression representing the language recognized by Q_s.

There are two accepting states. The first, q_0, accepts 0^* and $(0^*10)^*$ while the second, q_2, accepts $\Sigma^*11\Sigma^*$.

So the union of these is: $0^* \cup (0^*10)^* \cup \Sigma^*11\Sigma^*$.[1]

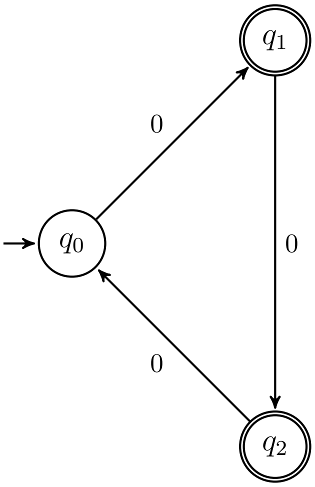
In English: All strings which are empty, end in "0", or contain "11" as a substring.

2. Give the formal definition of Q_s in terms of the 5-tuple $(Q, \Sigma, \delta, q_0, F)$.

$$\begin{array}{rcl} Q & = & \{q_0, q_m, q_2\} \\ \Sigma & = & \{0, 1\} \\ \delta & = & \\ q_0 & = & q_0 \\ F & = & \{q_0, q_2\} \end{array} \qquad \delta = \left| \begin{array}{ccc} & 0 & 1 \\ q_0 & q_0 & q_m \\ q_m & q_0 & q_2 \\ q_2 & q_2 & q_2 \end{array} \right.$$

[1] Tip o' the hat to NPatel & LSmith for pointing out that a piece was missing.

3. Give the state diagram for the DFA formally defined:
$(\{q_0, q_1, q_2\}, \{0\}, \{(q_0, 0) \to q_1; (q_1, 0) \to q_2; (q_2, 0) \to q_0\}, q_0, \{q_1, q_2\})$



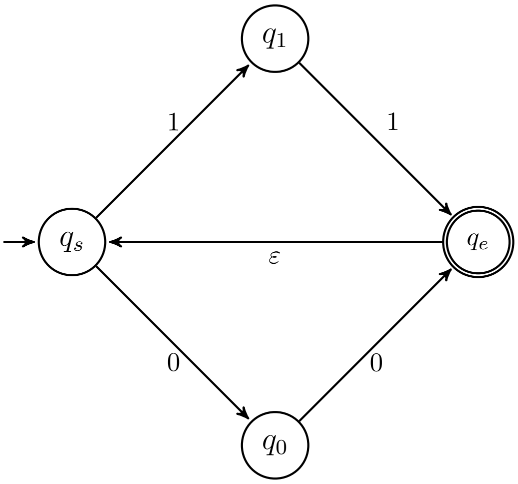
Express in English the language recognized by the DFA:

The set of all strings consisting of “0” which are not a multiple of 3 in length.

Give a regular expression representing the language recognized by the DFA.

$0(000)^* \cup 00(000)^*$

4. Consider the NFA Q_p:



Indicate the acceptance/rejection of each string by Q_p:

String	Accept	Reject
ε		$\surd$
00	$\surd$	
1110		$\surd$
10101		$\surd$
110011	$\surd$	

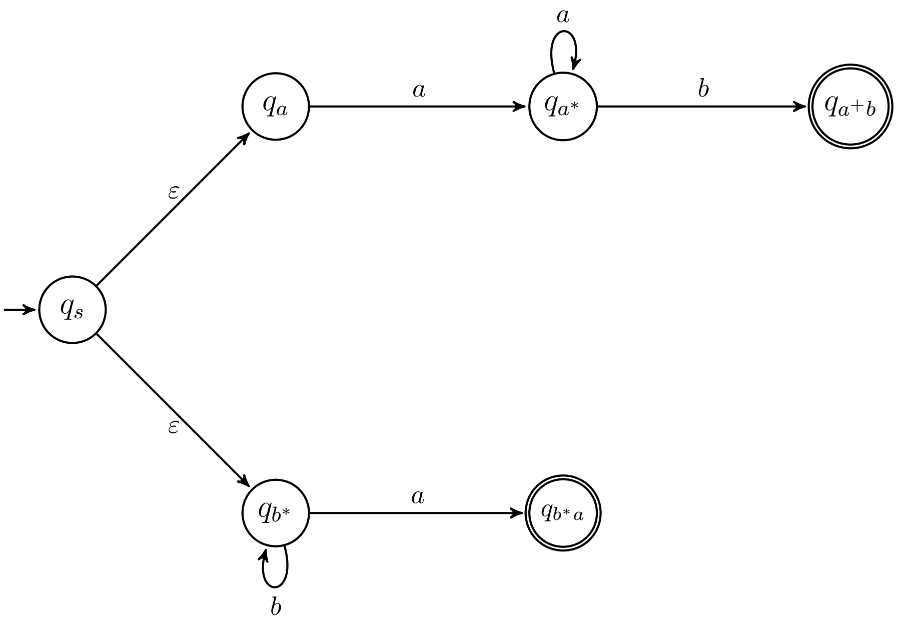
Express in English the language recognized by Q_p.

The language recognized by Q_p is the set of all strings consisting of alternating even-length runs of 0's and 1's, excluding ε.

Give a regular expression representing the language recognized by Q_p.

$(00 \cup 11)^+$

5. Construct an NFA recognizing the language represented by the following regular expression $a^+b \cup b^*a$.



6. Use the Pumping Lemma for regular languages to show that the language of palindromes with exactly one more “0” than “1” is not regular. (Think about what such palindromes must look like.)

Palindromes of this form look like $\{w0w^R \mid w \in \{0,1\}^*\}$. So we need to cleverly choose a string that, when pumped, creates string *not* of this form.

Some strings, such as $s = (01)^P0(10)^P$ have this form but *can* be pumped in such a way to create *good* strings in the language.

We cleverly choose $s = 0^P1^P01^P0^P$ where P is the pumping length of our language.

Now, assuming the language is regular, this string can be pumped in such a way as to create (infintitely many) new strings in the language:

1. Since $s = alz$ is long enough, $s_i = al^iz$ will be in out language for all $i \geq 0$.
2. $|l| > 0$, which means l has positive length, so when we pump the string, we must lengthen the string. (Except when $i = 0$ in which we are said to “pump down” or pop the bubble, and we then *shorten* s to get s_0.)
3. $|al| \leq P$, which means we are looking at a short pumping section at in the first P symbols of the string. (There may be longer ones, but we know a short one like this must exist, so we use it to our advantage.)

We’ll take $s = 0^P1^P01^P0^P$. Now, $|al| \leq P$, so we must pump only 0’s (because the first P symbols are all 0’s, and what we pump (l) is entirely included within the first P symbols). When we pump the string here, we get $s_2 = allz$ and that will look like $0^{P+|l|}1^P01^P0^P$, and that is no longer a palindrome. There are more 0’s in the initial block than in the final block.

So, if we assume that the language is regular, then any string that’s long enough can be pumped to generate more strings in the language. We found a string long enough to be pumped but which cannot be successfully pumped–all possible pumpings generate strings not in the language. Therefore, our assumption that this language is regular cannot be correct. The language must not be regular.

QED

7. Create a Context-Free Grammar that generates the language represented by the regular expression: 1(010)*1.

$$\begin{array}{lcl} S & \rightarrow & 1M1 \\ M & \rightarrow & 010M \mid \varepsilon \end{array}$$

Now, if you think generating three terminals in a string like that is cheating (it's not!), then you can try to create the "010" piece in pieces:

$$\begin{array}{lcl} S & \rightarrow & 1M1 \\ M & \rightarrow & TM \mid \varepsilon \\ T & \rightarrow & \text{WZW} \\ W & \rightarrow & 1 \\ Z & \rightarrow & 0 \end{array}$$

This is a nice little grammar to dissect and play with.
MOsorio suggests:

$$\begin{array}{lcl} P & \rightarrow & ABA \\ A & \rightarrow & 1 \\ B & \rightarrow & BB \mid 010 \mid \varepsilon \end{array}$$

I think that may be yet more elegant.

8. Consider the Context-Free Grammar:

$$\begin{array}{rcl} P & \rightarrow & ABA \\ A & \rightarrow & a \mid \varepsilon \\ B & \rightarrow & BAB \mid b \end{array}$$

What is the language generated by the grammar?

A few partial derivations get us:
ABA to $ABABA$ and then $ABABABA$ and we see that we can get any length of alternating A's and B's.

The B's generate only b for terminals while the A yield either a or vanish as the empty string.

So we get all strings of positive length with at least one "b" and no consecutive "a"s. (I believe I left out a piece of that in discussing it in the 9am class Tuesday!)

As a regular expression, there are FOUR possibilities:
The string begins in "a" but ends in "b": $a(ba \cup b)^*b$, or
The string begins in "b" but ends in "a": $b(ab \cup b)^*a$, or
The string begins and ends in "a": $a(ab \cup b)^+a$, or
The string begins and ends in "b": $b(ab \cup b)^*$.

This yields regular expression: $a(ba \cup b)^*b \cup b(ab \cup b)^*a \cup a(ab \cup b)^+a \cup b(ab \cup b)^*$.

This is sure to catch all possible good strings, but you can likely shorten it considerably. How would you go about doing that?

Show that the grammar is ambiguous.

To show the grammar is ambiguous, we must perform two different leftmost derivations that derive the same string:

$$\begin{array}{ccc} & ababaa & \\ ABA & & ABA \\ aBA & & aBA \\ aBABA & & aBABA \\ abABA & & aBABABA \\ abaBA & & abABABA \\ abaBABA & & abaBABA \\ ababABA & & ababABA \\ ababaBA & & ababaBA \\ ababab A & & ababab A \\ abababa & & abababa \end{array}$$

9. Convert the grammar above to Chomsky Noamal Form

$$\begin{array}{lcl} P & \rightarrow & ABA \\ A & \rightarrow & a \mid \varepsilon \\ B & \rightarrow & BAB \mid b \end{array}$$

1. Create a dedicated start variable:

$$\begin{array}{lcl} S_0 & \rightarrow & P \\ P & \rightarrow & ABA \\ A & \rightarrow & a \mid \varepsilon \\ B & \rightarrow & BAB \mid b \end{array}$$

2. Eliminate ε rules:

$$\begin{array}{lcl} S_0 & \rightarrow & P \\ P & \rightarrow & ABA \mid BA \mid AB \mid B \\ A & \rightarrow & a \\ B & \rightarrow & BAB \mid BB \mid b \end{array}$$

3. Eliminate unit rules (rules in which one variable merely substitutes for another):

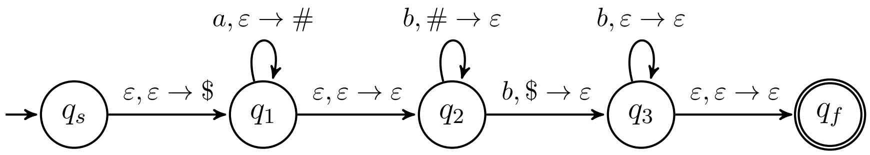
$$\begin{array}{lcl} S_0 & \rightarrow & ABA \mid BA \mid AB \mid BAB \mid BB \mid b \\ P & \rightarrow & ABA \mid BA \mid AB \mid BAB \mid BB \mid b \\ A & \rightarrow & a \\ B & \rightarrow & BAB \mid BB \mid b \end{array}$$

4. Cleanup any rules with bad form:

$$\begin{array}{lcl} S_0 & \rightarrow & XA \mid BA \mid AB \mid BX \mid BB \mid b \\ P & \rightarrow & XA \mid BA \mid AB \mid BX \mid BB \mid b \\ A & \rightarrow & a \\ B & \rightarrow & BX \mid BB \mid b \\ X & \rightarrow & AB \end{array}$$

Now every rule is of the form $V \rightarrow LR$ or $V \rightarrow t$, so we are done.

10. Construct a Pushdown Automaton that recognizes $M = \{a^i b^k \mid i < k, i \geq 0\}$


$a, \varepsilon \to \#$
$b, \# \to \varepsilon$
$b, \varepsilon \to \varepsilon$
$\varepsilon, \varepsilon \to \$$
$\varepsilon, \varepsilon \to \varepsilon$
$b, \$ \to \varepsilon$
$\varepsilon, \varepsilon \to \varepsilon$
q_s
q_1
q_2
q_3
q_f

We start in state q_s by priming the stack with \$ as we move to state q_1.

In state q_1 we push a # marker on the stack for every a we encounter. (Maybe none!)

From state q_1 we slide to state q_2 ignoring the stack.

In state q_2 we pop a marker off the stack for every b we see. If we run out of bs while there are still markers on the stack, we're stuck. The computation dies in q_2, as we wish it to.

On the other hand, if we run out of markers before we run out of bs, we spend a b as we pop the \$ off the stack and move to state q_3. This move guarantees that we have *more* bs than as as the language demands. (NOTE: if we have exactly the same number of as and bs, we run out of markers and bs at the same time. But we need a b to leave state q_2, so the computation dies, as it should.)

In state q_3, it's all over but the accepting. We burn off an extra bs (and we may have lots of them!) until the string is empty, and we can slide to the accepting state, q_f.

Does this machine accept ε? Should it?

Does this machine accept b? Should it?

Perfecto!

11. Use the Pumping Lemma for Context-Free Languages to show that the language of palindromes with exactly one more "0" than "1" is not context free. (Think about what such palindromes must look like.)

Palindromes of this form look like $\{w0w^R \mid w \in \{0,1\}^*\}$. So we need to cleverly choose a string that, when pumped, creates string *not* of this form.

Some strings, such as $s = (01)^P 0 (10)^P$ have this form but *can* be pumped in such a way to create *good* strings in the language.

We cleverly choose $s = 0^P 1^P 0 1^P 0^P$ where P is the pumping length of our language.

Now, assuming the language is context-free, this string can be pumped in such a way as to create (infintitely many) new strings in the CFL:

1. Since $s = uvxyz$ is long enough, $s_i = uv^i xy^i z$ will be in out language for all $i \geq 0$.
2. $|vy| > 0$, which means either v or y has positive length (probably both), so when we pump the string, we must lengthen the string. (Except when $i = 0$ in which we are said to "pump down" or pop the bubble, and we then *shorten* s to get s_0.)
3. $|vxy| \leq P$, which means we are looking at a short pumping section. (There may be longer ones, but we know a short one like this must exist, so we use it to our advantage.)

We'll take $s = 0^P 1^P 0 1^P 0^P$. Now, $|vxy| \leq P$, so we might pump only 0's or only 1's (in four different places). If we lengthen the string by adding extra symbols in one of those long stretches of P symbols, we will no longer have a palindrome. If we use the pumping lemma to reach across the "01" boundary and pump some 0's and some 1's on each side of it, we will again no longer have a palindrome. Ditto for pumping 1's and 0's across the boundary in the back half of our s. We *can* however have the v portion of s in the first block of 1's and the y portion of s in the second group of 1's. If we pump the string there, we get $s_2 = uvvxyyz$ and that will look like $0^P 1^{P+k} 0 1^{P+k} 0^P$, and that is still a palindrome. However, there are now $2P + 1$ 0's but $2P + 2k$ 1's. We insisted that the number of 0's be one greater than the number of 1's. That is no longer the case. No matter how we pump our string, we cannot get a good string as a result. (We didn't specifically mention that any pump that involves the middle 0 breaks the palindrome, too–but it does.)

So, if we assume that the language is context-free, then any string that's long enough can be pumped to generate more strings in the language. We found a string long enough to be pumped but which cannot be successfully pumped–all possible pumpings generate strings not in the language. Therefore, our assumption that this language is context-free cannot be correct. The language must not be context-free.

QED

12. Prove or disprove: The Context-Free Languages are closed under the union operation. That is, if A and B are CFLs, then $A \cup B$ is a CFL.
Hint: You may use either Context-Free Grammars or Pushdown Automata in your proof outline.

Solution: The Context-Free Languages ARE closed under the union operation. We present TWO proofs:

Proof 1 (Context-Free Grammars):

Given CFL A and CFL B, we show that $A \cup B$ is a CFL.

Because A is Context-Free, it has a CFG that generates it:

$$\begin{array}{ccl}
S_A & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals} \\
V_{A_1} & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals} \\
V_{A_2} & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals} \\
\vdots & \vdots & \qquad\qquad\qquad\qquad\quad \vdots \\
V_{A_n} & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals}
\end{array}$$

Similarly, B is Context-Free, so it has a CFG that generates it:

$$\begin{array}{ccl}
S_B & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals} \\
V_{B_1} & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals} \\
V_{B_2} & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals} \\
\vdots & \vdots & \qquad\qquad\qquad\qquad\quad \vdots \\
V_{B_m} & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals}
\end{array}$$

So, V has a CFG with $n+1$ variables, and B has a CFG with $m+1$ variables (including their start variables), and we write them so that these sets of variables are disjoint.

The language $A \cup B$ is then generated by:

$$S_{A \cup B} \quad \rightarrow \quad S_A \mid S_B$$

That is, to generate all the strings in $A \cup B$, either language A or B, we use the grammar for A or the grammar for B, as needed.

It's just that simple. We insisted the grammars have distinct variables to prohibit the commingling of rules.

Proof 2: We instead think of the CFLs as being recognized by PDAs, $M(A)$ recognizing A and $M(B)$ recognizing B. We construct $M(A \cup B)$ recognizing their union:

We create a new start state, q_u and connect it to the start states of $M(A)$ and $M(B)$ via $(\varepsilon, \varepsilon \rightarrow \varepsilon)$-edges, that is, edges that consume none of the string being computed and which also ignore the stack. We make the final states of our machine the final states of $M(A)$ and the final states of $M(B)$. Our PDA accepts precisely those strings accepted by $M(A)$ or $M(B)$, as desired.

$12'$. Prove or disprove: The Context-Free Languages are closed under the *concatenation* operation. That is, if A and B are CFLs, then $A \circ B$ is a CFL.
Hint: You may use either Context-Free Grammars or Pushdown Automata in your proof outline.

Solution: The Context-Free Languages ARE closed under the concatenation operation. We present TWO proofs:

Proof 1 (Context-Free Grammars):

Given CFL A and CFL B, we show that $A \circ B$ is a CFL.

Because A is Context-Free, it has a CFG that generates it:

$$\begin{array}{ccl}
S_A & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals} \\
V_{A_1} & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals} \\
V_{A_2} & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals} \\
\vdots & \vdots & \qquad\qquad\qquad\qquad\qquad\quad \vdots \\
V_{A_n} & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals}
\end{array}$$

Similarly, B is Context-Free, so it has a CFG that generates it:

$$\begin{array}{ccl}
S_B & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals} \\
V_{B_1} & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals} \\
V_{B_2} & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals} \\
\vdots & \vdots & \qquad\qquad\qquad\qquad\qquad\quad \vdots \\
V_{B_m} & \rightarrow & \text{variables \& terminals } |\ldots| \text{ more variables \& terminals}
\end{array}$$

So, V has a CFG with $n+1$ variables, and B has a CFG with $m+1$ variables (including their start variables), and we write them so that these sets of variables are disjoint.

The language $A \circ B$ is then generated by:

$$S_{A \circ B} \quad \rightarrow \quad S_A S_B$$

That is, to generate all the strings in $A \circ B$, language A concatenated with language B, we use the grammar for A on the left and the grammar for B on the right. Derivations with this grammar generate strings of the form $\alpha\beta$ where α is generated by S_A, and β is generated by S_B.

It's just that simple. We insisted the grammars have distinct variables to prohibit the commingling of rules.

Proof 2: We instead think of the CFLs as being recognized by PDAs, $M(A)$ recognizing A and $M(B)$ recognizing B. We construct $M(A \circ B)$ recognizing their concatenation:

We the start state of our machine the start state of $M(A)$. We connect the final states of $M(A)$ to the start state of $M(B)$ via $(\varepsilon, \varepsilon \rightarrow \varepsilon)$-edges, that is, edges that consume none of the string being computed and which also ignore the stack. We make the final states of our machine the final states of $M(B)$. Our PDA accepts precisely those strings of the form $\alpha\beta$ where α is accepted by $M(A)$ and β is accepted by $M(B)$, as desired.

There is one technicality in the concatenation construction we have overlooked. There is no guarantee that the stack will be empty when we transition from PDA $M(A)$ to PDA $M(B)$ as it would be in starting from scratch. We can overcome this in either of two ways: First, we can ignore it, assuming that the first step of $M(B)$ is to prime the stack with a "$" as we have thus far in class. Under most circumstances, this will be fine, but it might introduce a *very* subtle bug. A better solution would be to create a start state *before* PDA $M(A)$ and prime *it* with a special symbol used only in these constructions, say £. Then when we transition from $M(A)$ to $M(B)$ we pop off the stack any symbols that may have been left on–that *is* possible–and finally also pop off the £ as we transition to the start state of $M(B)$.

I did these two proofs in the 10am class Tuesday. I had intended to use the concatenation proof on the exam. Sigh. But somebody asked me about it *specifically.* Now I have to come up with a new tidy proof for the exam.

If only there were a third fundamental operation on languages besides union and concatenation. Oh, if only. . . .