Problem Set 1

Fall 23

Due: Monday, September 4th.

1. Yea or Nay

   State whether the following statements are true or false, and justify your answer.

   (a) Just-in-time (JIT) compilation typically yields slower execution than static compilation.

   (b) Jim creates a new language, Jaython, which changes Java's syntax to be Python-like (supporting elif, indentation-based blocks, etc.). To build a Jaython compiler, Jim only needs to modify the lexer/parser of a Java compiler.

Solution:

(a) True. JIT compilation happens at run time and needs extra time.

(b) True. The modified lexer/parser will handle the modified syntax and generate the same AST as a standard Java compiler does. Therefore the other components of the Java compiler can be reused.

2. Regular Expression

   For strings containing the letters a and b give a regular expression that captures all strings w such that

   (a) The same letter does not repeat in consecutive.

   (b) Starts and ends with the same letter.

   Example,

   (a) Expressions accepted: a, aba

   (b) Expressions not accepted: ab, aa

Solution:

The regular expression that captures the above language is:

$$a(ba)^*|b(ab)^*$$

.

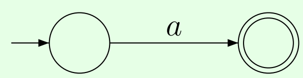
The first (or second) clause corresponds to the case that a (or b) appears at the start of string w. The the subexpression ab or ba represents any substring that does not any repeating a or b in consecutive; and the wrapping Kleene closure $(...)^*$ allows this pattern to repeat arbitrary number of times. Note that λ belongs to any Kleene closure, and also belongs to this language, as zero is also even.

3. NFA

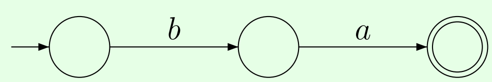
   Give a *non-deterministic* finite automaton that captures the regular expression from above. Show the automaton in graphical form.

Solution:

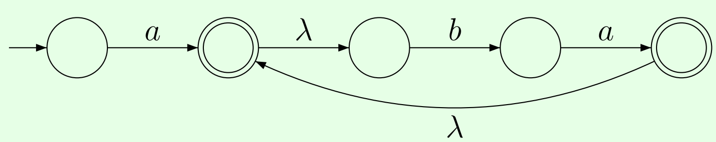
We will build this NFA up piece by piece. First, note that the NFA for a is:



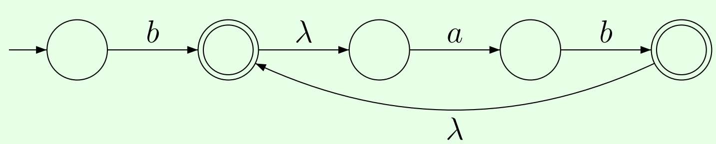
and the NFA for ba is:



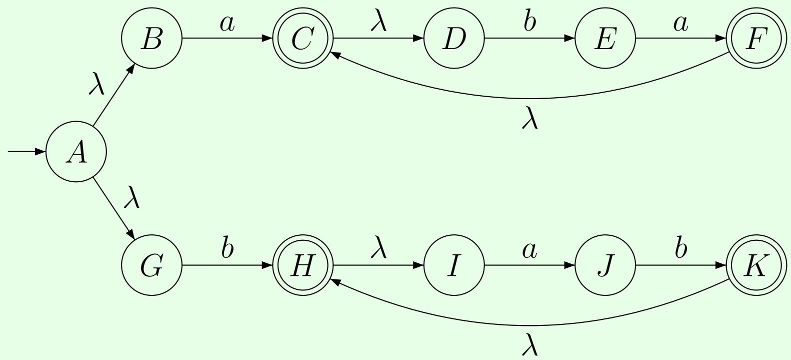
As per the construction rule for Kleene closure, we can then build an NFA for $a(ba)^*$ as follows:



Similarly, the NFA for $b(ab)^*$ is:



Now we can put the two parts together to build the overall NFA:



Note that we have assigned a label to each state for the convenience of the next step.

4. DFA

   Using the construction described in class, give a *deterministic* version of the automaton. You only need to show the transition table.

Solution:

Remember that the way to perform DFA construction is to simulate every possible "configuration" of states that can occur after seeing a particular letter (and taking all possible λ transitions). So when we start out, our "pointers" immediately take all possible lambda transitions, giving us an initial state of $\{A, B, G\}$. We can now build the table from here:

State	a	b	final?	new state
ABG	CD	HI	*no*	0
CD	ϕ	E	*yes*	1
HI	J	ϕ	*yes*	2
E	FCD	ϕ	*no*	3
J	ϕ	KHI	*no*	4
FCD	ϕ	E	*yes*	5
KHI	J	ϕ	*yes*	6

5. Minimization (for in-person courses only)

   Give a *minimized* version of the finite automaton, using the algorithm we used in class. You only need to show the state transition diagram.

Solution:

When each state gets assigned a number, the NFA looks like this:

State	a	b	final?
0	1	2	*no*
1	ϕ	3	*yes*
2	4	ϕ	*yes*
3	5	ϕ	*no*
4	ϕ	6	*no*
5	ϕ	3	*yes*
6	4	ϕ	*yes*

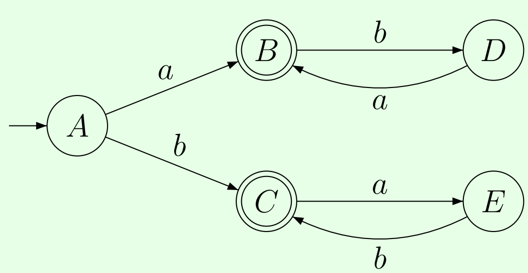
The minimization starts with two groups of states, depending on whether the state is accepting:

State	a	b	final?
034	?	?	*no*
1256	?	?	*yes*

Depending on the transitions on a and b the first state can be refined into the following groups:

State	a	b	final?
0	15	26	*no*
15	ϕ	3	*yes*
26	4	ϕ	*yes*
3	15	ϕ	*no*
4	ϕ	26	*no*

Now all of the four states are distinct and the automaton looks like this:



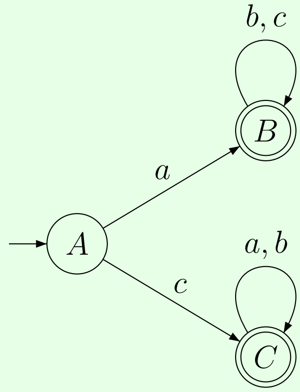
6. Regular Expression to DFA

   Build a *minimized, deterministic* automaton for the following regular expression:

   $ab^*(cb^*)^* \mid cb^*(ab^*)^*$

Solution:

The minimized, deterministic automaton has five states:


b, c
B
a
A
a, b
c
C