CSC173
FLAT 2014
ANSWERS AND FFQ
30 September 2014

Please write your name on the bluebook. You may use two sides of handwritten notes. Perfect score is 75 points out of 85 possible. Stay cool and please write neatly.

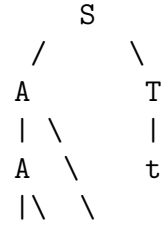
1 Parsing (25 pts, 5 each)

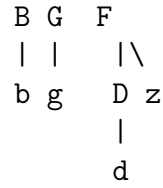
Here's a CSG Grammar G: uppercase means nonterminal symbol, lowercase is terminal.
$S \to AT$
$A \to AF$
$A \to BG$
$F \to Dx$
$F \to Dz$
$D \to d$
$B \to b$
$G \to g$
$T \to t$

1. Draw a parse tree for the string `bgdzt` using G.
2. Can you use G with an LL(1) parser? Why or why not?
3. If you think G is not an LL(1) grammar, rewrite it into a new grammar H that *is* LL(1). Call it H.
4. Draw a parse tree for `bgdzt` using H.
5. What are the FIRST(A) and FOLLOW(A) sets for G?

Answer:

1. from G:



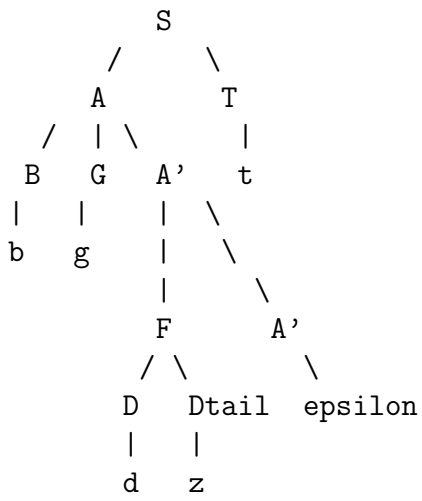


2. No, it's got Left recursion (affects 2nd and 3rd productions), and common prefixes (4th and 5th productions). Thus no tops-down parsing without some rewriting.

3. After applying our rules for removing left rec. and common prefixes:

   $S \rightarrow AT$
   $A \rightarrow BGA'$
   $A' \rightarrow FA' \mid \epsilon$
   $F \rightarrow DDtail$
   $Dtail \rightarrow x \mid z$
   $D \rightarrow d$
   $B \rightarrow b$
   $G \rightarrow g$
   $T \rightarrow t$

4. from H:



5. FIRST(A) = b, FOLLOW(A) = t,d.

2 Regular Expression to Minimized DFA (30 pts, 5 each)

1. In Thompson's method for converting NDFAs to DFAs, We've seen the "mini-machine" to produce (recognize) a set of strings σ drawn as

showing its start and end states but suppressing its contents. We've seen how to convert this mini-machine into a more complicated one that produces the Kleene * expression $\sigma*$. Show how to convert it into one that produces the Kleene + expression $\sigma+$, (one or more repetitions of σ.)

2. Write a regular expression that produces (recognizes) strings of 0 and 1 with the description "string contains exactly two 1s, or it ends in 1, or both". So 110, 0011, 010101, 101000 are OK but 100, 1110 aren't.

3. Consider the regular expression
   (0* 0 (a | b) 0)* 0*
   I was trying for "All strings from alphabet a,b,0 with each a or b preceded and followed by at least one 0." I think it's got a bug, though. Tell me what you think, and if you agree there's a problem do you see a fix? (Hint: are there legal strings it does not produce?)

4. Buggy or not, create an equivalent nondeterministic finite automaton for the above RE (0* 0 (a | b) 0)* 0*, by our "mini-machine" method (Thompson's method).

5. Use the subset construction to convert the NDFA to a DFA.

6. Use the equivalence class argument to minimize the resulting DFA or argue that it is minimal already. Also, does the DFA give you the same bug as the RE?

Answer:

1. Simply remove the forward ϵ arrow from the Kleene * construction.

2. (0*10*10*) | ((0 | 1)*1) is one RE that seems to work.

3. Trying to produce or recognize "All strings from alphabet a,b,0 with each a or b preceded and followed by at least one 0." I think (0* 0 (a | b) 0)* 0*
   is wrong and can't produce 0a0b0, for instance. It can do 0a00b0, but... (0+ (a | b))* 0+ works, though, as does (0*0(a | b)0*)00* and other variants.

4. (FFQ) If I see a huge NDFA with no state labels I know we're in trouble. Need the labels to do the state-set process.

   Following figure is My NDFA and equivalent minimized machine.

5. ,6
   (FFQ) Here, the idea is not that you invent some DFA out of your head that would work. The idea is you demonstrate the relevant algorithm. If I see no state sets, I worry!

Here's my DFA (e means ϵ, and sets-of-states from the NDFA form the labels of the DFA), which has the same bug. It is minimal since the two final states in the intial partition differ: one doesn't behave like the other for inputs (production of) a or b.

(FFQ) If your minimized DFA doesn't have the bug, why not? Does your NDFA-¿DFA algorithm automaqically have a bug removal feature? How does the algorithm know your RE is 'buggy'?

3 FLAT Short Answers (30 pts, 3 Each.)

Answer in a sentence or two of prose. No proofs needed. Recall a finite language is a finite list of sentences.

1. Alice and Bob are looking at a formal grammar whose last two productions are:
   ...
   $aCXXCb \rightarrow aXXb$
   $XX \rightarrow z$
   Alice: "Aha! Shrinking productions – so it's for a recursively enumerable language."
   Bob: "I'm thinking: could it be a finite language...?"
   Alice: "What!? It's clearly from a Turing Machine!"
   Who's right and why?

2. Why isn't $a^n b^n c^n$ a context-free language?

3. Is a Turing machine with two read-write tapes more powerful than a single-tape machine? Why or why not?

4. Is $0*$ a finite language? Why or why not?

5. Can a finite automaton recognize palindromes (strings that are the same forward and backward)? Why or why not?

6. Let our alphabet be $\Sigma = \{0, 1, x\}$, and let $S \in \{0, 1\}*$ be any string of 0's and 1's. Consider the language L, with sentences of the form SxS^R: x is a single symbol and S^R is the reverse of S. So sentences of L are palindromes. Now — can a deterministic pushdown automaton recognize L? How or why not?

7. With notation as above, can a deterministic pushdown automaton recognize palindromes SS^R? How or why not?

8. With notation as above, can a non-deterministic PDA recognize palindromes SS^R? How or why not?

9. How powerful would you say a two-stack PDA is? For instance, could we simulate a 2-stack PDA with a 1-stack PDA? Or an LBA or TM with a two-stack PDA?

10. Why do compiler-writers care about FIRST and FOLLOW sets?

Answers:

(FFQ): These questions seem approachable from language, grammar, or machine. Most of them are more naturally got through the machine. Important: there are an infinite number of grammars for any language, including CSGs, shrinking productions, whatever. So reasoning from grammar type either to machine type or language type is impossible! TM can do finite, context-free, or any others. Grammar could look complex, language could be simple (Question 1). So don't reason from the grammar. In fact question 1 is the only one that mentions grammars, and only to make the point you can be easily misled by the form... the issue is what it generates!

(FFQ) In the same vein, don't equate languages with states. Finite languages can be produced by a higher-order grammar or machine, for instance. Especially: finite state machine does NOT imply finite language or vice versa.

(FFQ) Need to understand which abstractions and mechanisms are relevant. Words like "polynomial" and "scanner" don't really belong. Assertions like " $a^n b^n c^n$ is a polynomial and so ..." is an example.

1. I'd say Bob's a little righter: Grammar could be
   $S \to aCXXCb$
   $aCXXCb \to aXXb$
   $XX \to z$
   So this language is just one string azb, is finite and so doesn't need a Turing Machine to generate it. But of course a finite language is also RE (though not conversely of course), so Alice isn't wrong there, but probably not right in the way it seems she thinks she is.

2. (FFQ) people remembered it takes a CSG to do this but I'd say it's not that easy to establish why (and noone did). Appealing to how the machine (say a PDA) would get in trouble or be able to cope is easier, more convincing.

   A DPDA (implements context-free languages) machine must first fill and then empty its stack matching a's and b's, so no memory of n when it gets to the c's.

3. Nope, nothing can compute more generally in the accepted sense of algorithms and computations than a TM (type 0 language is as high as the hierarchy goes). So e.g. non-determinism doesn't buy anything either. (FFQ) This question would have been clearer if it had said "powerful (accepts more languages, not simply faster)." Simulating non-determinism in a TM might need an exponential increase in the number of states and thus a big slowdown, and it seems reasonable that N tapes could run faster (fewer quintuples executed) – is there a simple example?

4. Nope, 0* has potentially more strings than any given list length. (FFQ) See above... don't confuse states w/ language: FSM can surely do 0* but it's not a finite language.

5. Nope, it has a given number of states, and if the forward half of a palindrome exceeds that, it would need more states than the machine posseses to remember it and thus check the second half.

6. Sure, just push the incoming stream until see the x, then transition into popping it and checking result against current input symbol.

7. Nope, darn it...with no x or other clue, the determininistic machine doesn't know when to stop pushing and start popping.

   (FFQ) This is the paradigmatic demonstration of the fact that NPDAs are more powerful than DPDAs (see below).

8. Yes: the non-deterministic machine can (and does by definition) correctly guess when to start popping.

   (FFQ) In terms of languages, NPDA > DPDA. Despite NTM = DTM and NFA= DFA. We didn't make a big deal out of this but even if not we're not entitled to believe nondeterminism can't make a difference. In terms of grammars (another demo that they may not tell you a lot), here's a lovely CFG for something DPDAs can't do: $S \rightarrow 0S0 \mid 1S1 \mid \epsilon$. Voila' SS^R.

9. Same as TM: one stack has "the tape to left of r/w head" and the other the tape to the right. Neat, eh?

10. They give you the cases that must be dealt with in a parser with one-character lookahead. Epsilon productions make the problem nontrivial, though it can be automated.