July 17, 2020

CS360 Quiz 2

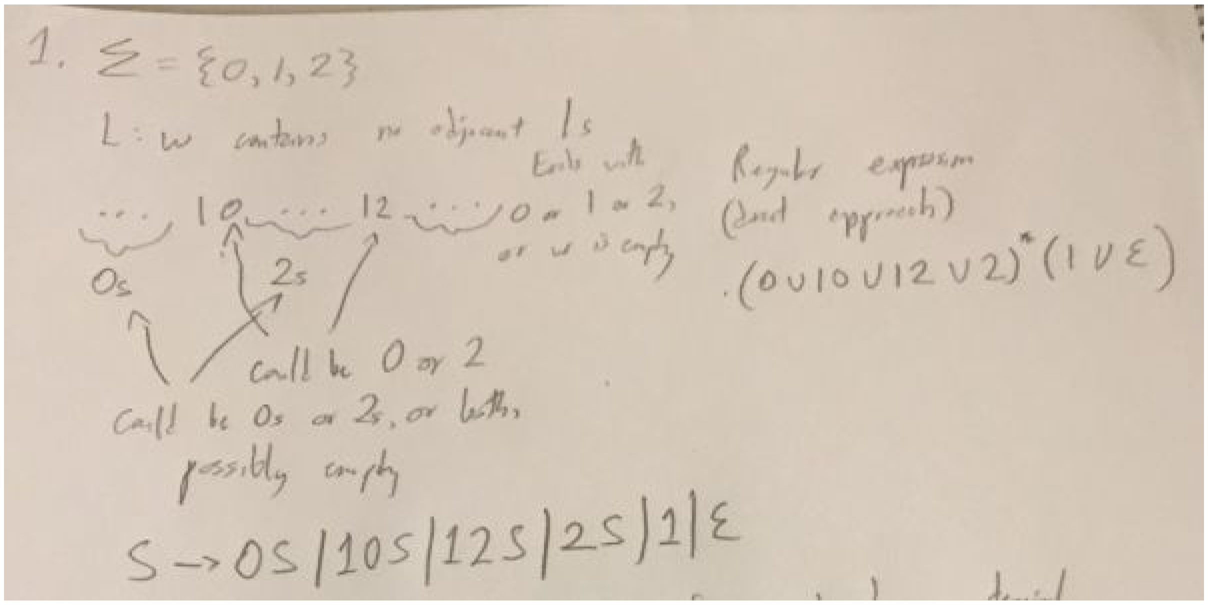
Group members: Shivanshi Nagar, Stephen Hansen, Dennis George

Start Time 12:35 pm
End Time 2:05 pm

Question 1

S -> 0S | 2S | 10S | 12S | 1 | ε

Justification for non ambiguous -> We are always parsing left to right so the branching will always go towards the right
No adjacent 1s -> There can be any number of 0s or 2s in a row, but every time we have a 1, it has to be followed by a 0 or a 2, or it should be the end of the string. This way, we've made sure that adjacent 1's are not possible.
As we parse left to right each possible accepted character combination is accounted for and no character combinations work for more than 1 rule (1 is only accepted at the end of the string, 10S and 12S are enforced by the 1 at front, 0S and 2S can only be valid rules if there is no 1 before them). Hence due to the left-to-right parsing nature and limiting each building block to one rule, we find that the grammar constructed is non-ambiguous.



1. $\Sigma = \{0, 1, 2\}$

L: w contains no adjacent 1s

$\underbrace{\cdots}_{0s}$ 10 $\underbrace{\cdots}_{2s}$ 12 $\underbrace{\cdots}$ Ends with 0 or 1 or 2, or w is empty

Could be 0 or 2

Could be 0s or 2s, or both, possibly empty

Regular expression (2nd approach)

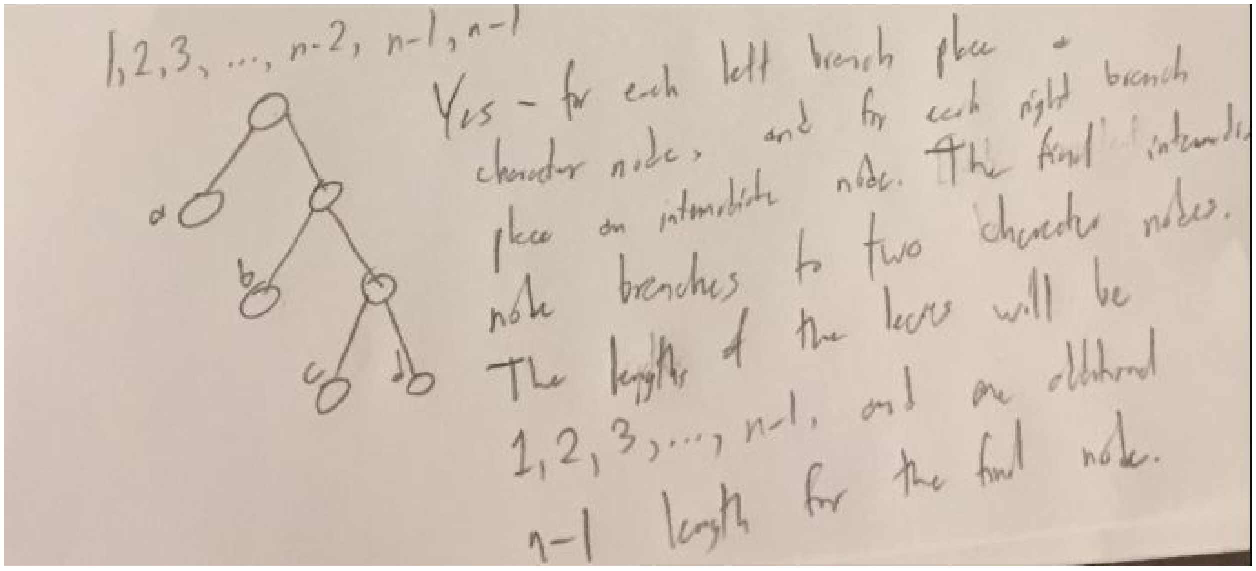
$(0 \cup 10 \cup 12 \cup 2)^*(1 \cup \varepsilon)$

$S \rightarrow 0S \mid 10S \mid 12S \mid 2S \mid 1 \mid \varepsilon$

Question 2

Maximal - Maximum depth from the root to a terminal character node.
Minimal - Minimum depth from the root to a terminal character node
Average in terms of the features of the tree - Taking average of all the code lengths of all the characters/features of the tree.

Can it happen that the lengths are equal to 1, 2, 3,…,n-2, n-1, n-1?



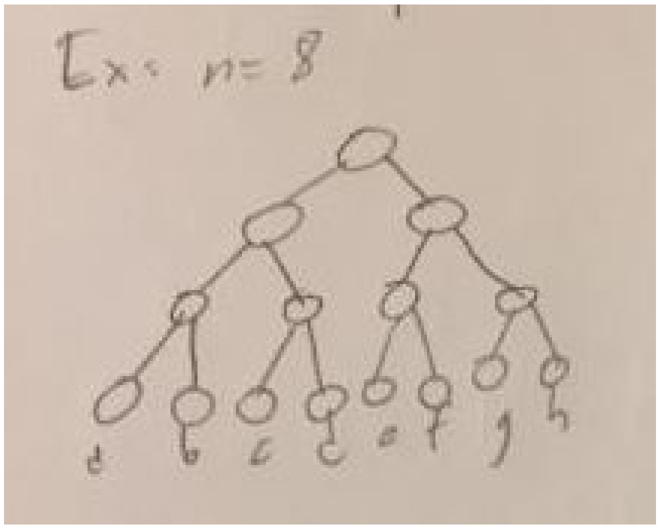
1,2,3, ..., n-2, n-1, n-1

Yes - for each left branch place a character node, and for each right branch place an intermediate node. The final intermedi[illegible] node branches to two character nodes. The lengths of the leaves will be 1, 2, 3, ..., n-1, and one additional n-1 length for the final node.

Yes, for each left branch place a character (leaf, terminal) node and for each right branch place an intermediate node. The final intermediate node branches to two character nodes (cannot have an intermediate node that branches to 0 or 1 nodes). The lengths of the leaves will be 1, 2, 3, …, n-1 and one additional n-1 length for the final node. In the example pictured if the top node has frequency 100, then a could have frequency 50, b could have frequency 25, and c and d could each have frequency 12.5. The intermediate nodes would sum correctly (c and d sum to 25, b sums with c and d's parent node to get 50, a sums with b's parent node to get 100 which is the root). The frequencies of each node would divide by two as you go down.

Can it happen that all code lengths are equal?
This can happen if there are two characters (n = 2). In a Huffman tree, there are only two possible branching directions. When n=2, it is possible to construct a tree where all code lengths are equal as shown in the example below. This holds for all values of n where n can be represented as a power of 2 (n = 2^x, x being a positive integer). In a perfectly balanced Huffman tree, all character nodes (leaf nodes) would be the same distance away from the root, being at a depth of log2(n). This is due to the property that each intermediate node of a Huffman tree must branch to two additional nodes - if each intermediate node branches to two intermediate nodes and at some level all intermediate nodes branch to terminal (character) nodes, then this is possible. A quite easy way to generate such a tree would be to give each

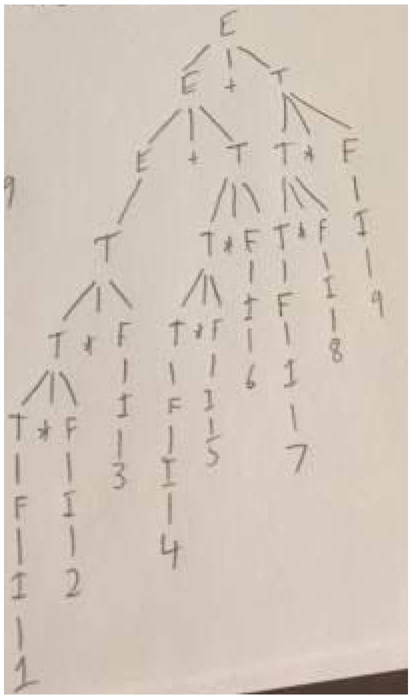
character equal frequency (for n=2, choose a and b both having frequency 50; for n=8 choose 8 characters with each having frequency 100/8, etc.).



Question 3

3 (justification on non-ambiguity for i-iv): We base our grammars off of the left-associative grammar shown in the weeks 1 to 2 notes which is non-ambiguous to start. Our grammars are non-ambiguous as for each operator +, *, we define a specific way for each to branch (either left associate or right associate). To generate a sequence of each operator, the branching will control the direction in which we generate multiple summations or multiplications. Eventually we have to parse out the addition first, then the multiplication, and then either reduce to integers or handle nested expressions. The order in which these operations are parsed is fixed and the order is non-ambiguous. There are no overlapping rules where two terminal symbols can be interpreted into multiple rules for each variable. As a result, our grammar is non-ambiguous for each part. For variable E, if there is a +, parse to E+T/T+E otherwise parse to T. For variable T, similarly parse to T*F/F*T otherwise parse to F. In the original grammar proposed by problem 3, the grammar was ambiguous because the order of parsing operations (+ and *) was not specified; introducing variables to create order and associativity renders this grammar non-ambiguous.

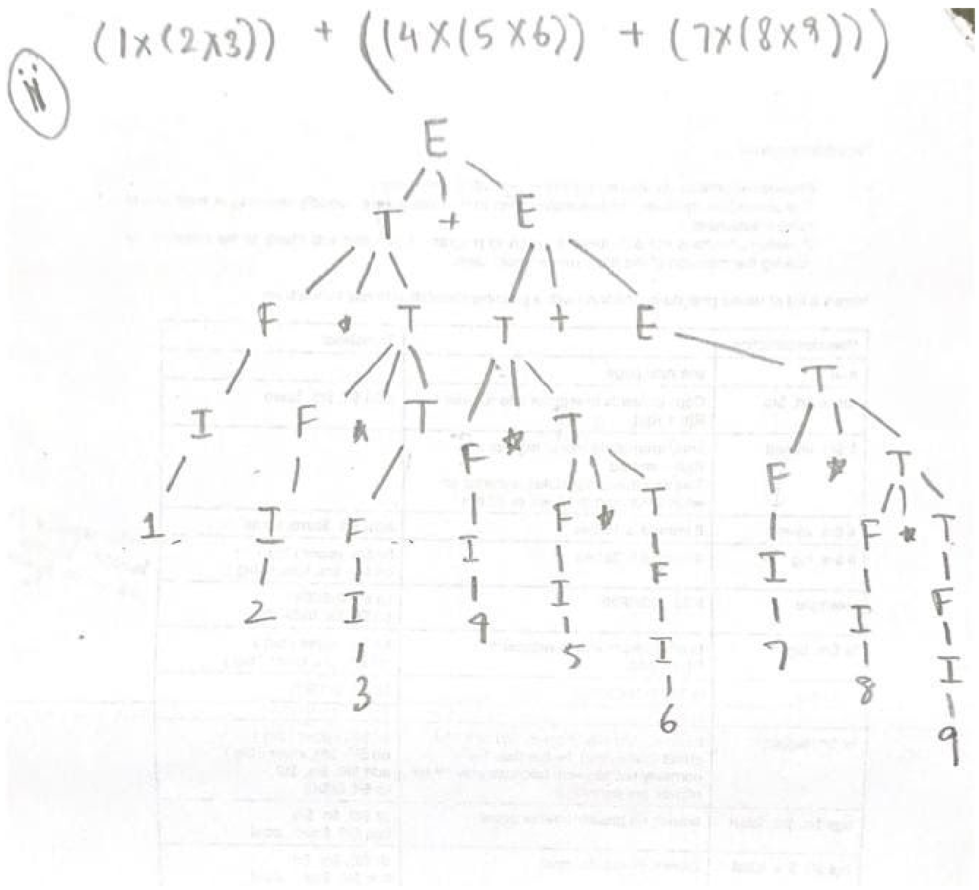
3(i)



Grammar:
Operation + is left associative and operation * is left associative

E->E+T|T
T->T+F|F
F->I|(E)
I->1|2|3|4|5|6|7|8|9

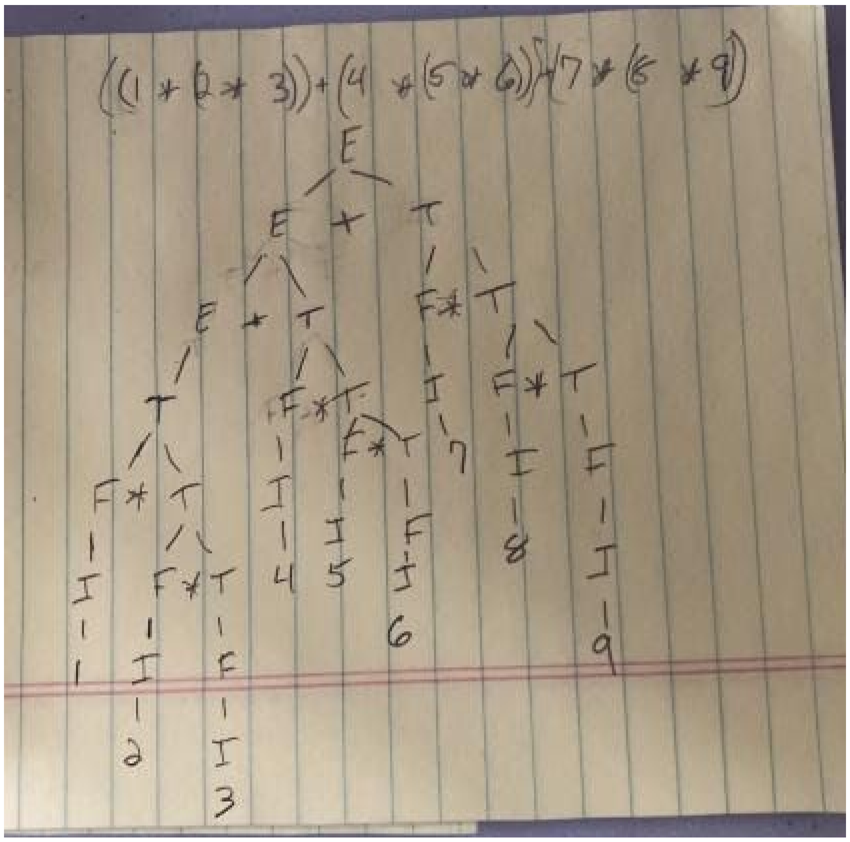
3 (ii)



Grammar:
Operation + is right associative and operation * is right associative,

E->T+E|T
T->F*T|F
F->I|(E)
I->1|2|3|4|5|6|7|8|9

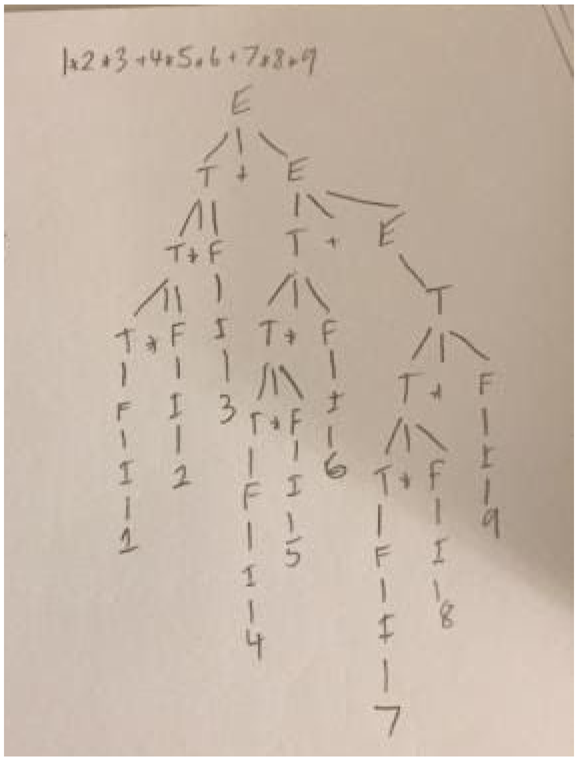
3 (iii)



Grammar:
Operation + is left associative and operation * is right associative

E->E+T|T
T->F*T|F
F->I|(E)
I->1|2|3|4|5|6|7|8|9

3 (iv)



Grammar:

Operation + is right associative and operation * is left associative

E->T+E|T

T->T*F|F

F->I|(E)

I->1|2|3|4|5|6|7|8|9

Question 4

Given a DFA it is perfectly reasonable to construct an equivalent non ambiguous context-free grammar since all regular languages also satisfy properties of context-free languages. To convert a DFA to a non-ambiguous grammar, do the following:

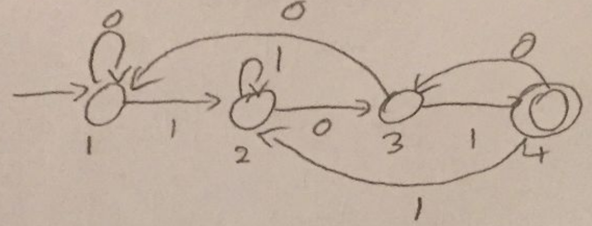
1. Create equivalent grammar symbols per each state in the DFA.
2. For each transition in the DFA, given a state a, an input x, and transition dictated by delta(a, x) = b (where b is the new state), create an equivalent rule a -> xb (using the corresponding variables for a and b).
3. For any states labelled as an accepting (final) state, add an additional rule for their corresponding variable such that V -> epsilon (indicating that no more input needs to be read to make the constructed string valid).
4. Set the start variable to whichever variable is the corresponding variable for the DFA start state.

Since each DFA state requires one transition per input symbol in the specified alphabet, it is impossible for the context-free grammar under this construction to result in generating two rules that cause the grammar to be unambiguous. Each rule (except for the epsilon rules) branch to the right and parse the input string from left-to-right. Per variable, there will be n rules where n is the size of the alphabet. The variables with epsilon rules are the only accepting variables as all other variables have rules that require a transition to another variable.

Here is an example of generating an equivalent non-ambiguous CFG for the DFA found in last week's quiz:

4. (Ends with 101)

$\Sigma = \{0,1\}$



$S_1 \to 0S_1 \mid 1S_2$

$S_2 \to 0S_3 \mid 1S_2$

$S_3 \to 0S_1 \mid 1S_4$

$S_4 \to 0S_3 \mid 1S_2 \mid \varepsilon$

Equivalent non-ambiguous CFG for the DFA matching strings ending in 101.

As mentioned each variable has a rule for each symbol in the alphabet, and only one rule per each. Since it always branches right and there is only one rule per symbol per variable, our generated grammar will always be non-ambiguous (due to the properties of DFAs enforcing one transition per symbol, for all states).