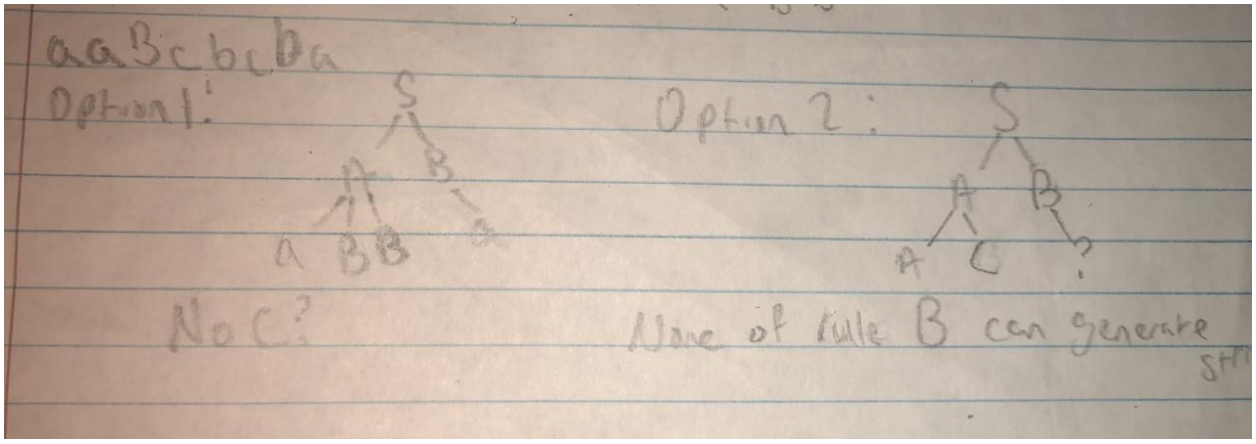
Homework 2
Akash Dansinghani
Professor Umoja
Due: 10/30/20
Programming Language Concepts

1.

a. Given the start statement we get S -> AB, next we get aBBB, followed by that we get aBBcBb, the final string we get is aBBcCb which is not the same as the given string aBBcbbCa. The first three letters of the string given in the question follows the rules, however cbb is not part of the rules since a capital let has to follow a lowercase c.
b. Given the start statement we get S -> BC, next we get cBbC, following that we get cCbbC, continuing we get cCbbC, next we get cAcbbC, following we get caBBcbbC, finally we get cacBbBcbbC. The reason it is not the string given is because everything is formulated correctly up until the second capital B (cacBbB…) the string follows the pattern of a c has to follow a capital B if the start of the string follow ca or cac, but the "cc" in cacBbBcc is not correct because a lower case c can not be followed by another lowercase c.
c. Given the start statement we get S -> BC, next we get CbC, next we get bCAbC, after that we get bbCACAbC, followed by bbCbCAbC, and finally bbCbAcAbc. As we can see this final string is not the same as the one, we were given which is bbCbABBb. We can tell the string is not able to be formulated since we need the first character of the string is b and the last is b as well. ABB is not able to be formulated given the grammar rules as well.

2.

a.



Phrases: aaBBcB (abCBB), AcB, aaBcB Ab, aBB, bCB, cBB, a, c,

Simple Phrases: Ac, aBB, a, c

Handle: aBB (Op1) and Ac (Op2)

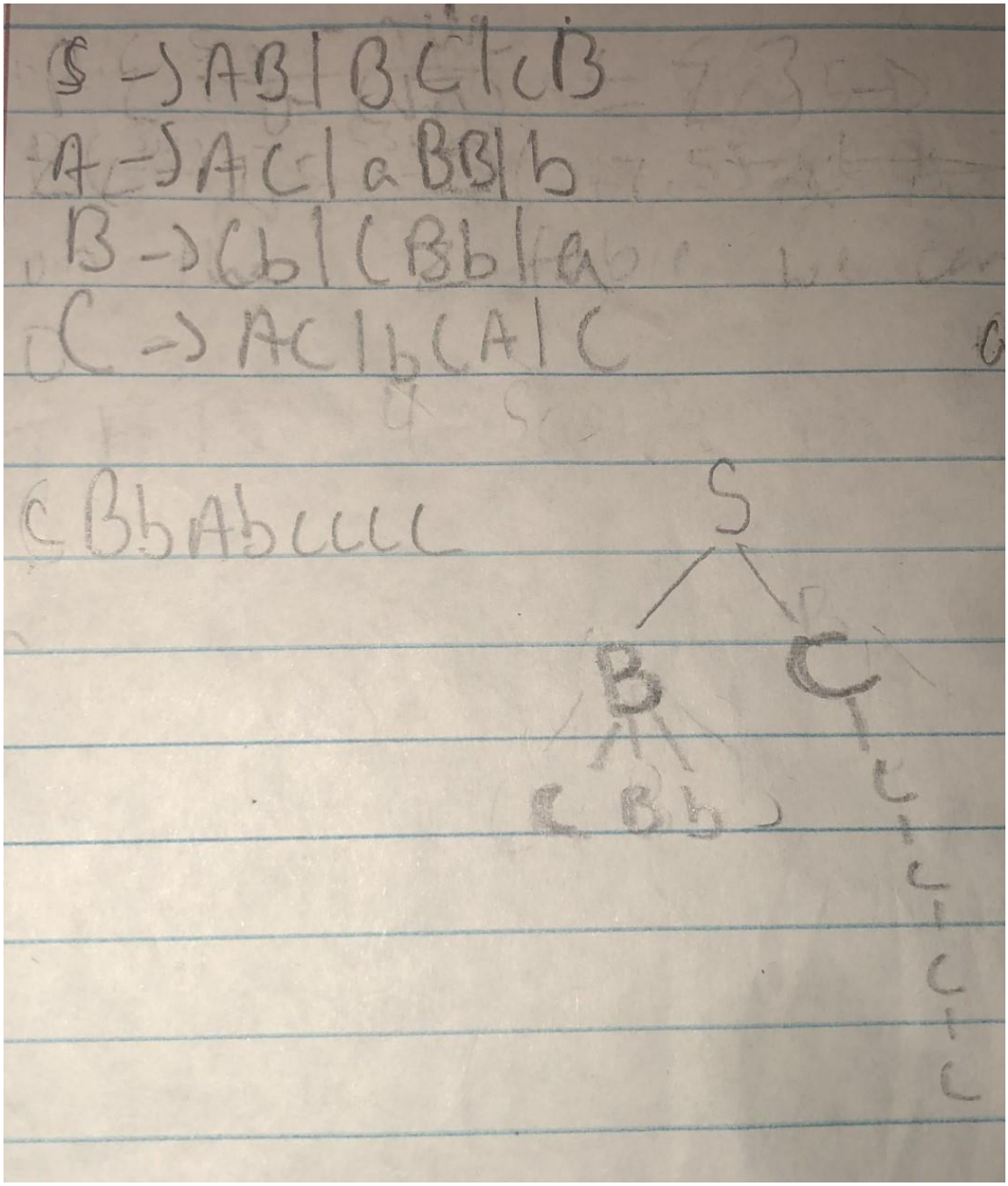
Not possible, we see that the first two characters are aa so we either have to choose aBB for A and then choose a from B or start from b from A and convert it to a from B. Even choosing either of

those choices, we are not able to form the string aaBcbcba because we can get to aaBc but we can not obtain aaBcb since we are not able to convert it. So we can follow either path:

S -> AB -> AcB -> aBBcB -> aaBcB or S -> AB -> AB -> aBBB -> aaBB

From C we can see we can not choose AC because the last character in our string is an a but we are given Ac. We can not choose bCA since there is no capital C or A and we can not convert it, and finally we can not choose c because the string does not end with a c so this is not a phrase that can be generated from the grammar rules given. So cbcba and bcba can not be created rendering this phrase invalid.

b.



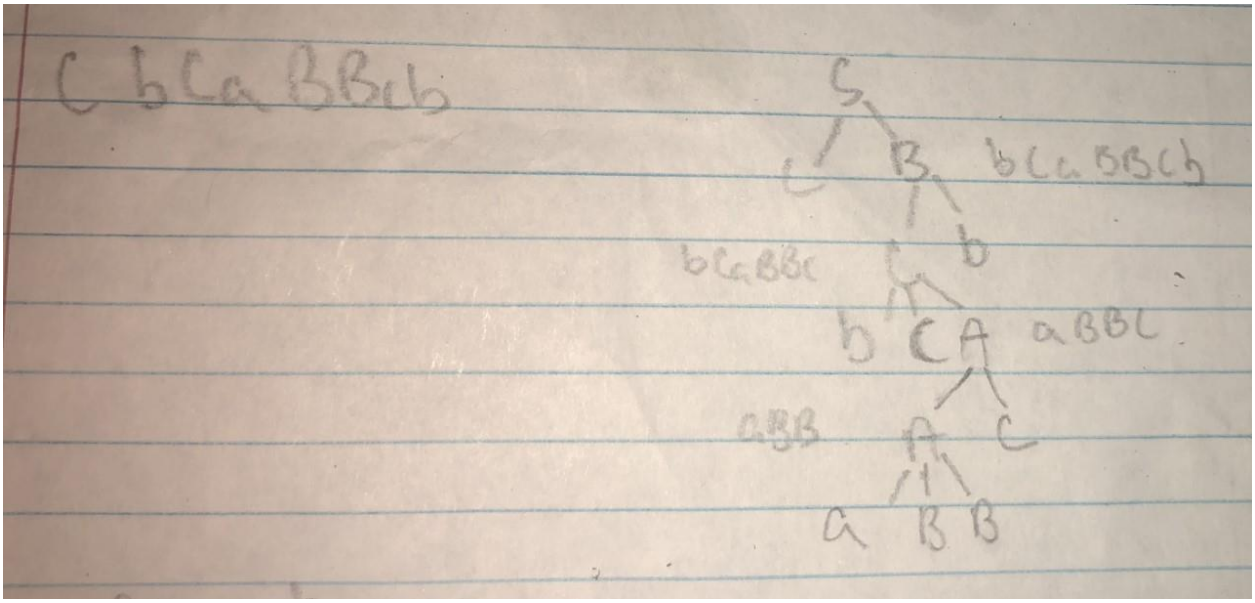
Phrases: cBbAcccc, cBbAc, cBbC, cBb, BC, cBb, c (c1, c2, c3, c4)

Simple Phrases: cBb, c (c1, c2, c3, c4)

Handle: cBb

This is not possible. If we start S from BC, from B we choose cBb which makes the string cBbC then choose Ac from A we get cBbAc and then for C we can just use c four times and we get cBbAcccc. However the desired string is cBbAbcccc and since we are not able to get b in-between A and c because if we were able to put a b there; the string would have to end with an a which it doesn't, so this string can not be generated.

c.



Phrases: aBB, aBBc, bcaBBc, bcaBBcb, bCA, aBB, Cb ,cB

Simple Phrase(s): aBB

Handle: aBB

This is possible. If we start with S -> cB, we can then use B -> Cb which would convert B to a C, after that we can use C -> bCA, next we us A -> Ac and finally A -> aBB. So we would get Cb, cCb, cbCAb, cbCAcb, and finally cbCaBBcb

3. Syntax analyzers or parsers usually have a formal description of the syntax of programs. Common examples include context – free grammars, BNF, or EBNF. There are two general types of parsers which are top-down which starts from the root (top) and goes down to the leaves and bottom-up which starts from the leaves (bottom) and makes its way to the root. Disadvantage of top-down parsing is that left recursion disallows top down parsing

The problem with Bottom-up parsers it accepts left – recursive grammar and doesn't include meta symbols which are used to specify extensions to BNF. It is also hard to generate the handle for the right sentential form since the definition of a handle doesn't always generate the proper handle. Using the definition of a phrase and simple phrase below. Phrases and simple phrases help generate

the rightmost sentential form since “the handle of any rightmost sentential form is its leftmost simple phrase which is an intuitive wat to find the handle of a right sentential form.

Definition: β is a **phrase** of the right sentential form γ if and only if
$S => *\gamma = \alpha_1 A \alpha_2 => + \alpha_1 \beta \alpha_2$

In this definition, $=> +$ means one or more derivation steps.

Definition: β is a **simple phrase** of the right sentential form γ if and only if $S => *\gamma = \alpha_1 A \alpha_2 => + \alpha_1 \beta \alpha_2$

The purpose of a parser is to help create a parse tree, but given a parse tree there is not need for the parser.

Pushdown Automaton is a context – free language recognizer and Recursive-decent parser is an example of a PDA.

LR Parsers are one of the algorithms bottom-up parsing uses. They use a parsing table to build a special programming language. The three advantages of LR parsers are: they are versatile and can be built for any language, they detect syntax compile errors when going through a left-to-right scan, and lastly LR class of grammars are a proper superset of LL parsers. The only disadvantage of LR parsing is that it is difficult to produce parsing tables especially hand-written ones.

The role of parsers is to establish the symbols and rules of a programming language so when a program compiles, it meets the requirements and regulations that are set in the programming language. Source code is able to be read by the computer given a proper representation. A parser also generates a parse tree which facilitates in creating intermediate code.

4.

An attribute grammar is similar to Context – Free Grammar; the difference is that it has attributes ad attribute evaluation rules called semantic functions (attribute computation functions) which allows it to target semantics and syntax unlike BNF. It uses grammar symbols called terminal and nonterminal symbols which have values assigned to them. When it comes to compilation, they help to see if a program follows rules and is correct. Syntax and semantic rule as well as the predicate.

Operational Semantic(s) is when certain aspects of a program (the state(s)) are proved by constructing proofs using logical statements about its execution and procedures compared to Denotational statements which use mathematical proofs. We see what changes happen from each state. Operational semantics affect the compilation process by giving errors in a compiler. When given sudo code, we are able to distinguish whether or not a line of code will run through the compiler and gives us a better understanding what each line of code does whether that may be using data structures or for, while, if-else loops until each line of code is proved. At a higher level we use natural operational semantics which we prove the logic of each line of code until we have the desired final result executed. At a lower level, we use structural operational semantic for a more precise meaning and the changes in the state of the program after each line is executed.

Denotational Semantics – similar to operational semantics but uses mathematical proofs (denotations) instead of logical statements for loops and source code and does not include a virtual machine. It is better to prove the correctness of a program and can help with language design but is hard to use. We can see the change of the current variable after proving the denotation. We use syntactic and semantic domains as well as the range. A domain is a collection of values that a proper parameters to the function and the range is a collection of objects which are mapped to the function. Denotational semantics does not use a virtual machine but uses mathematical proofs instead and helps the compiler generate and optimize code. Its main use is to correct implementations and would be the equivalent to compile errors in an IDE since it would see if each line of code is able to be run given the circumstances of the current code.

Axiomatic Semantics – proving the weakest precondition that can not be reasoned by an axiom (statement that is theorized to be true) (inference rule). Using the inference rule, we try to prove the precondition is true so it can become the postcondition, and once that it is proven true, the next statement becomes the precondition with the statement that was proven becomes the postcondition. Axiomatic semantics is great for a compiler that has strict specifications like making sure each loop and line of code follows each condition and provides the desired result. The axioms would be each section of code (being a loop or statement) and the precondition would be the first line and we would prove it (say a > b x = 2(a) + b given a = 1 and b = 4) and once that line is true, that would become the post condition and the next line would become the precondition and so on until we get to the last axiom and we see if it is the weakest pre-condition.

5.

<switch_stmt> --> switch"("(<expr>|<identifier>) ")""{" <body> "}"

<body> --> case<literal>:<stmt> {<stmt>;}{case<literal>:<stmt> {<stmt>;} [default: <stmt> {<stmt>;}]

CFG:

V = { <switch_stmt>, <body>, switch, <expr>, <identifier>, <literal>, <stmt> }

E = { default, case, ; , “”, :, (,),<,> }

R = [

switch_stmt => switch (<expr>|<identifier>) { <body> } |() | {}

body => case <literal> : <stmt> {<stmt>;}; | case<literal>:<stmt> {<stmt>;}; | default: <stmt> {<stmt>;} | {}

S = switch_stmt

BNF:

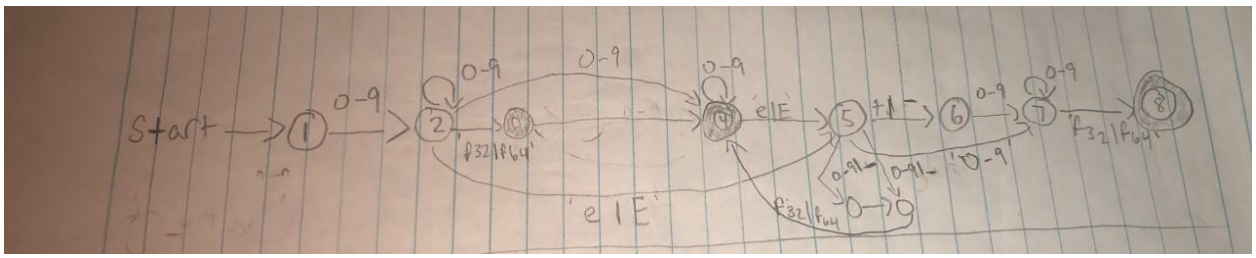
<S> -> <switch_stmt> -> switch(<A>) | body(<B>){<C>}

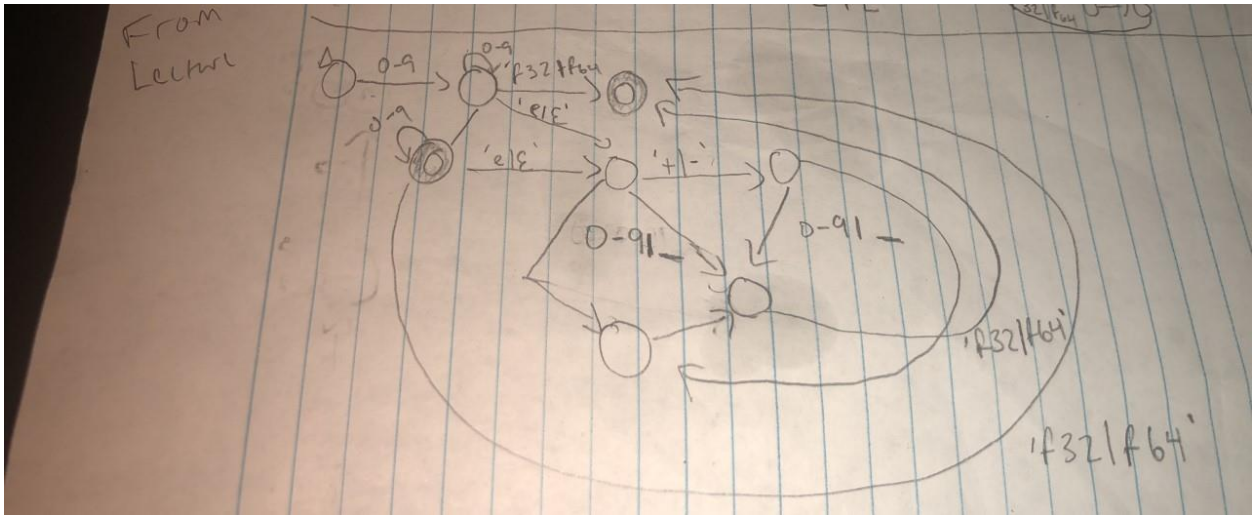
<A> -> <expr> | <identifier> | <body>

<B> -> case <literal> : <stmt>{<stmt>;} | case <literal> : <stmt>{<stmt>;} <B>

<C> -> default : <stmt>{<stmt>}

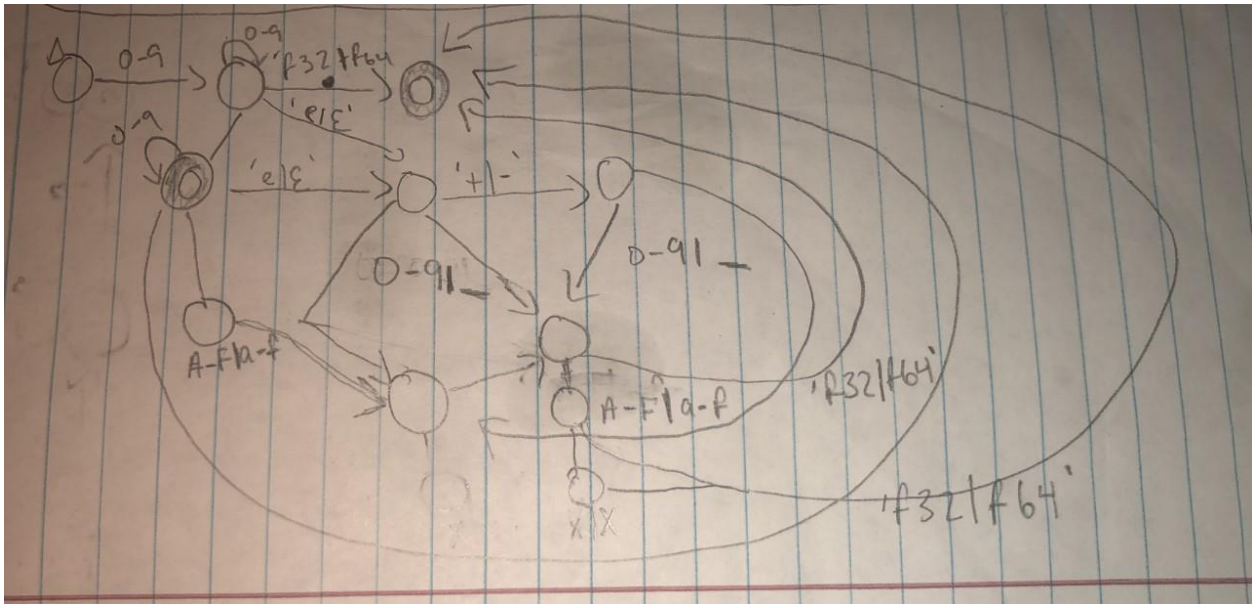
6.





The bottom state diagram is similar to the one completed in lecture.

7.



Key for 6 and 7

Hexadecimal - Digit ::= [0-9 |A-F |a-f]

Hexadecimal Digit Sequence ::= [X |x]

Digit ::= [0 – 9]

Digit and float ::= [0 – 9 | _]

Exponent ::= [“e” | “E”]

Sign ::= [“+” | “-“]

32 / 64 float digit ::= [“f32” | “f64”]

 ::= State

 ::= Final State

➔ ::= Transition

Δ ::= Start

References

Golang.org. 2020. *The Go Programming Language Specification - The Go Programming Language*. [online] Available at: <https://golang.org/ref/spec#Integer_literals> [Accessed 29 October 2020].

Sceweb.sce.uhcl.edu. 2020. *Module1*. [online] Available at: <https://sceweb.sce.uhcl.edu/helm/WEBPAGE-C/my_files/TableContents/Module-7/module7page.html> [Accessed 29 October 2020].

Sebesta, R., n.d. *Concepts Of Programming Languages*.

Www2.cs.sfu.ca. 2020. [online] Available at: <https://www2.cs.sfu.ca/CourseCentral/383/dma/notes/chapter3_semantics.pdf> [Accessed 29 October 2020].