Name: Spencer Milbrandt
Partner: None
Principles of Programming Languages
Lab 2 Writeup

2. Grammars: Synthetic Examples.

(a) Describe the language defined by the following grammar:

S ::= ABA
A ::= a | aA
B ::= ε | bBc | BB

S is an expression of the expressions A and B.
A is a right recursive and associative tree which consists of a string of a's with a as the terminal character.
B consists of epsilon as the terminal character if there is less than one b in the recursive sequence. If there is one b in the recursive sequence then there is a b and c.

(b) Consider the following grammar:

S ::= AaBb
A ::= Ab | b
B ::= aB | a

Which of the following sentences are in the language generated by this grammar? For the sentences that are described by this grammar, demonstrate that they are by giving **derivations.**

1. baab

S ::= AaBb => baBb because A => b
=> baab because B => a

The above derivation shows that baab is in the language generated by the above grammar.

2. bbbab

S ::= AaBb => AbaBb because A => Ab
=> AbbaBb
=> bbbaBb because A => b

=> bbbaab because B => a

The above grammar generates a language that ensures that all the sentences derived from contain more than one ‘a’. Since B => aB or a and S ::= AaBb, the grammar of S already contains a single ‘a’ followed by the grammar B which adds one or more a’s to any resulting sentence.

L(S) = { (b)^n (a)^m (b)^k; n >= 1, m > 1, k = 1 }

Therefore, bbbab is not in the language generated by the above grammar.

3. bbaaaaa

S :: = AaBb => AbaBb because A => Ab
=> bbaBb because A => b
=> bbaaBb because B => aB
=> bbaaaBb
=> bbaaaaBb
=> bbaaaaab because B => a

The above grammar generates a language that ensures that all sentences derived from it terminate in 'b' not 'a'.

L(S) = { (b)^n (a)^m (b)^k; n >= 1, m > 1, k = 1 }

Therefore, bbaaaaa is not in the language generated by the above grammar.

4. bbaab

S ::= AaBb => AbaBb because A => b
=> bbaBb
=> bbaab because B => a

The above derivation shows that baab is in the language generated by the above grammar.

(from Sebesta, Chapter 3)

(c) Consider the following grammar:

S ::= aScB | A | b

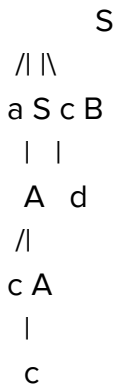
A ::= cA | c
B ::= d | A

Which of the following sentences are in the language generated by this grammar? For the sentences that are described by this grammar, demonstrate that they are by giving **parse trees.**

1. abcd

The above parse tree shows that abcd is in the language generated by the above grammar.

2. acccbd



The above parse tree shows the resulting sentence of acccd since b is not in the grammar B or the grammar A, so it can be concluded that acccbd cannot be obtained by using the language generated by the above grammar. Therefore, acccbd is not in this language.

3. acccbcc

```
c A   c A
 |    |
 c    c
```

The above parse tree shows a resulting sentence of accccc since b is not in grammar B or in grammar A, so it can be concluded that acccbd cannot be obtained by using the language generated by the above grammar. Therefore, acccbcc is not in this language.

4. acd

```
          S
 /| |\
a S c B
 /|   |
A b   d
|
c
```

The above parse tree shows the resulting sentences as either accd or abcd, so it can be concluded that acd it not in the language generated by the above grammar.

5. accc

```
          S
 /| |\
a S c B
 |   |
 A   A
 |   |
 c   c
```

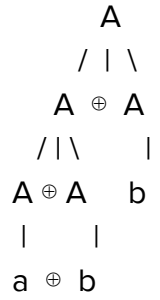
The above parse tree shows that accc is in the language generated by the above grammar.

(from Sebesta, Chapter 3)

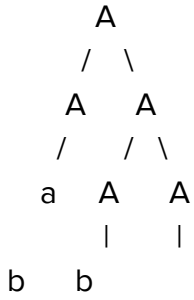
(d) Consider the following grammar:

A ::= a | b | A⊕A

Show that this grammar is ambiguous.



(a ⊕ b) ⊕ b



a ⊕ (b ⊕ b)

The grammar is ambiguous because the graph is able to reach the end in two different sequences.

(e) Let us ascribe a semantics to the syntactic objects A specified in the above grammar from part d. In particular, let us write

A ⇓ n

for the judgment form that should mean A has a total n a symbols where n is the metavariable for natural numbers. Define this judgment form via a set of inference rules. You may rely upon arithmetic operators over natural numbers. Hint: There should be one inference rule for each production of the non-terminal A (called a syntax-directed judgment form).

a & b is in A ⊕ A and b is in A

(a ⊕ b) ⊕ b

a is in A and b is in A
---
a ⊕ (b ⊕ b)

3. Grammars: Understand a Language.

(a) Consider the following two grammars for expressions e. In both grammars, operator and operand are the same; you do not need to know their productions for this question.

e ::= operand | e operator operand

e ::= operand esuffix
esuffix ::= operator operand esuffix | ε

i. Intuitively describe the expressions generated by the two grammars.

The expressions generated by both grammars can be the same although they are generated differently.

Observation => The language generated by both grammars is:

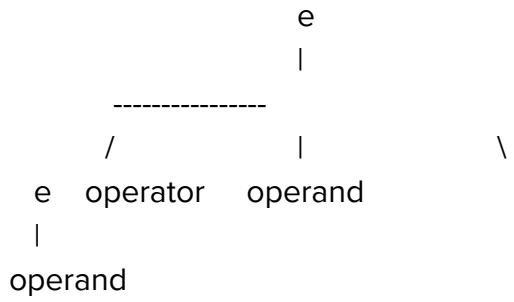
L(e) = { (operand)^n (operator operand)^m; n = 1 m >= 0 }

ii. Do these grammars generate the same or different expressions? Explain.

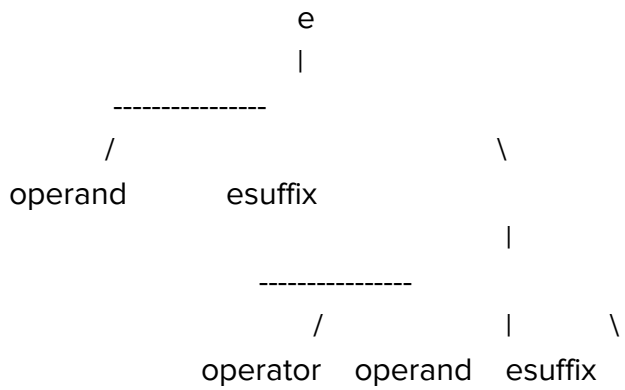
These grammars can generate the same expression but the latter uses left associativity (expands to the left) whereas the former uses right associativity (expands to the right). Therefore, these grammars do not generate the same expression.

Explanation using parse trees:

For e ::= operand | e operator operand, we can have the following which expands to the left:



For e ::= operand esuffix and esuffix ::= operator operand esuffix | epsilon where epsilon signifies "Empty", we can have the following which expands to the right:



|
epsilon

(b) Write a Scala expression to determine if '–' has higher precedence than '<<' or vice versa. Make sure that you are checking for precedence in your expression and not for left or right associativity. Use parentheses to indicate the possible abstract syntax trees, and then show the evaluation of the possible expressions. Finally, explain how you arrived at the relative precedence of '–' and '<<' based on the output that you saw in the Scala interpreter.

On the Scala command prompt, the following expressions were entered:

```
(1)  1 << 2 - 1
    returned 2

(2) 2 - 1 << 1
   returned 2
```

Experimentation with parentheses as follows:

```
(1) (1 << 2) - 1
    returned 7

(2) 2 - (1 << 1)
    returned 0
```

Therefore, - takes precedence over <<.

(c) Give a BNF grammar for floating point numbers that are made up of a fraction (e.g., 5.6 or 3.123 or -2.5) followed by an optional exponent (e.g., E10 or E-10). The exponent, if it exists, is the letter 'E' followed by an integer. For example, the following are floating point numbers: 3.5E3, 3.123E30, -2.5E2, -2.5E-2, and 3.5. The following are not examples of floating point numbers: 3.E3, E3, and 3.0E4.5. More precisely, our floating point numbers must have a decimal point, do not have leading zeros, can have any number of trailing zeros, non-zero exponents (if it

exists), must have non-zero fraction to have an exponent, and cannot have a '-' in front of a zero number. The exponent cannot have leading zeros. For this exercise, let us assume that the tokens are characters in the following alphabet Σ:

Σ def = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,E, -,.}

Your grammar should be completely defined (i.e., it should not count on a non-terminal that it does not itself define).

```
<F> ::=  <Neg><term>.<N><exp>
<term> ::= <I><N> | Empty
<exp> ::= E<Neg><I><N> | Empty
<N> ::= <I><N> | <Z><N> | Empty
<Neg> ::= - | Empty
<I> ::= 1 | 2 | 3 | ... | 9
<Z> ::= 0
```