Kevin Gutierrez – CSCI 330 – Hwork 4

Sebesta Chapter 3 Review Questions:

1. **Define syntax and semantics.**
   Syntax is the form of a languages expressions, statements, and program units. Semantics is the meaning of the expressions, statements, and program units

2. **Who are language descriptions for?**
   Program descriptions are for initial evaluators, implementors, and users.

3. **Describe the operation of a general language generator.**
   A language generator works by generating sentences of a language, allowing the user to compare syntax with the structure of a generator.

4. **Describe the operation of a general language recognizer.**
   A language recognizer works by checking if the given in put string is in the language

5. **What is the difference between a sentence and a sentential form?**
   A sentence is a string of the language. Sentential form is each of the strings in a derivation.

6. **Define a left-recursive grammar rule.**
   <expr> -> <expr> + <expr> | <var>
   <var> -> A | B | C

7. **What three extensions are common to most EBNFs?**
   1. An optional part of the RHS, delimited by brackets
   2. The use of braces in a RHS to indicate that the enclosed part can be repeated indefinitely or left out altogether
   3. A replacement of the recursion by a form of implied iteration, the part enclosed within braces can be iterated any number of times

8. **Distinguish between static and dynamic semantics.**
   Static semantics require analysis at compile time only
   Dynamic semantics define the behavior of the program when it's executed, in other works, runtime checks such as exception handling, runtime type checking, and related operations

9. **What purpose do predicates serve in an attribute grammar?**
   They specify semantic rules and ensure that the necessary rules are met for semantic conditions, error checking, and contextual validity

10. What is the difference between a synthesized and an inherited attribute?
    A synthesized attribute is used to pass semantic info up a parse tree, while an inherited attribute is used to pass semantic info down & across the parse tree

11. **How is the order of evaluation of attributes determined for the trees of a given attribute grammar?**
    It's determined by the dependencies between attributes as specified in the grammars semantic rules, done by using synthetic and inherited attributes

12. **What is the primary use of attribute grammars?**
    The primary use is to define the semantics of programming languages.

Sebeta Chapter 3 Problem Set

1. **The two mathematical models of language description are generation and recognition. Describe how each can define the syntax of a programming language.**
   Recognition is used to check if the string or sentence is defined by the grammar. Generation is used to specify the language by generating all the possible syntactically correct sentences. These two models, used in conjunction, can define the syntax of the language.

2. **Write EBNF descriptions for the following (refer to https://www.w3schools.com/ for any language details):**
   1. A C++ **while** statement
      <while_stmt> -> while (<condition>) {stmt}

   2. A C++ **struct** (structure) declaration statement1
      <struct> -> <identifier> <member_list>
      <member_list> -> <type> <identifier> {, <type> <identifier>}
      <type> -> int | double | float | char

   3. A C **switch** statement
      <switch> -> switch (<expression>) <case_list>
      <case_list> -> <case> { <case> } [<default_case>]
      <default_case> -> <statement>
      <case> -> case <const> : <statement_list>
      <statement_list> -> <statement> {, <statement>}
      <const> -> int | char

   4. C **float** literals
      <float> -> [<sign>]<integer_component> [<fractional_component>]
      <integer_component> ->[<sign>] <digit> {, <digit> }
      <fractional_component> -> <digit> {, <digit> }
      <sign> -> + | -
      <digit> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

3. **Rewrite the BNF of Example 3.4 to do each of these:**
   (a) give + precedence over * and
   (b) force + to be right associative.

<assign> -> <id> = <expr>
<id> -> A | B | C
<expr> -> <term> + <expr>
| <term>
<term> -> <factor>
| <factor> * <term>
<factor> -> (<expr>)
| <id>

Since term now appears before <expr> addition in the hierarchy, addition has precedence over *
Since expr is now defined as term + expr, (such that expr is on the right hand side), the addition expression is now right associative

4. **Rewrite the BNF of Example 3.4 to add the ++ and -- unary operators of Java.**

<assign> -> <id> = <expr>
<id> -> A | B | C
<expr> -> <expr> + <term>
| <term>
<term> -> <term> * <factor>
| <factor>
<factor> -> (<expr>)
| <unary_operator> <identifier>
| <id>
<unary_operator> -> ++ | --

5. **Write a BNF description of the Boolean expressions of Java, including the three operators &&, ||, and ! and the relational expressions.**

<boolean_expression> -> <boolean_expression> || <term>
| <term>
<term> -> <term> && <factor>
| <factor>
<factor> -> ! <factor>
| <relational_expression>
| true
| false
| (<boolean_expression>)
<relational_expression> -> <id> <relational_operator> <id>
<relational_operator> > | < | >= | <= | == | !=
<id> A | B | C

Notice that AND is defined first, then OR, then NOT. This is on purpose, as java places highest precedence on a logical NOT, then a logical AND, and finally, a logical OR

6. Using the grammar in Example 3.2, show a leftmost derivation for each of the following statements:

1. **A = A * (B + (C * A))**

<assign> => <id> = <expr>
=> A = <expr>
=> A = <id> * <expr>
=> A = A * <expr>
=> A = A * (<expr>)
=> A = A * (<id>+ <expr>)
=> A = A * (B + <expr>)
=> A = A * (B + (<expr>))
=> A = A * (B + (<id> * <expr>))
=> A = A * (B + (C * <expr>)
=> A = A * (B + (C * <id>))
=> A = A * (B + (C * A))

2. **B = C * (A * C + B)**

<assign> => <id> = <expr>
=> B = <expr>
=> B = <id> * <expr>
=> B = C * <expr>
=> B = C * (<expr>)
=> B = C * (<id> * <expr>)
=> B = C * (A * <expr>)
=> B = C * (A * <id> + <expr>)
=> B = C * (A * C + <expr>)
=> B = C * (A * C + <id>)
=> B = C * (A * C + B)

3. **A = A * (B + (C))**

<assign> => <id> = <expr>
=> A = <expr>
=> A = <id> * <expr>
=> A = A * <expr>
=> A = A * (<expr>)
=> A = A * (<id> + <expr>)
=> A = A * (B + <expr>)
=> A = A * (B + (<expr>))
=> A = A * (B + (<id>))
=> A = A * (B + (C))

7. **Using the grammar in Example 3.4, show a leftmost derivation for each of the following statements:**

1. **A = (A + B) * C**

<assign> => <id> = <expr>
=> A = <expr>
=> A = <term> * <factor>
=> A = (<expr>) * <factor>
=> A = (<expr> + <term>) * <factor>
=> A = (<id> + <term>) * <factor>
=> A = (A + < factor >) * <factor>

=> A = (A + < id >) * <factor>
=> A = (A + B) * <id>
=> A = (A + B) * C

2. **A = B + C + A**

<assign> => <id> = <expr>
=> A = <expr>
=> A = <term> * <factor>
=> A = (<expr>) * <factor>
=> A = (<expr> + <term>) * <factor>
=> A = (<id> + <term>) * <factor>
=> A = (A + < factor >) * <factor>
=> A = (A + < id >) * <factor>
=> A = (A + B) * <id>
=> A = (A + B) * C

3. **A = A * (B + C)**

<assign> => <id> = <expr>
=> A = <expr>
=> A = <term> * <factor>
=> A = <factor> * <factor>
=> A = <id> * <factor>
=> A = A * (<expr>)
=> A = A * (<expr> + <term>)
=> A = A * (<id> + <term>)
=> A = A * (B + <factor>)
=> A = A * (B + <id>)
=> A = A * (B + C)

4. **A = B * (C * (A + B))**

<assign> => <id> = <expr>
=> A = <expr>
=> A = <term> * <factor>
=> A = <factor> * <factor>
=> A = <id> * <factor>
=> A = B * (<expr>)
=> A = B * (<term> * <factor>)
=> A = B * (<factor> * <factor>)
=> A = B * (<id> * <factor>)
=> A = B * (C * (<expr>))
=> A = B * (C * (<expr> + <term>))
=> A = B * (C * (<id> + <term>))
=> A = B * (C * (A + <factor>))
=> A = B * (C * (A + <id>))
=> A = B * (C * (A + B))

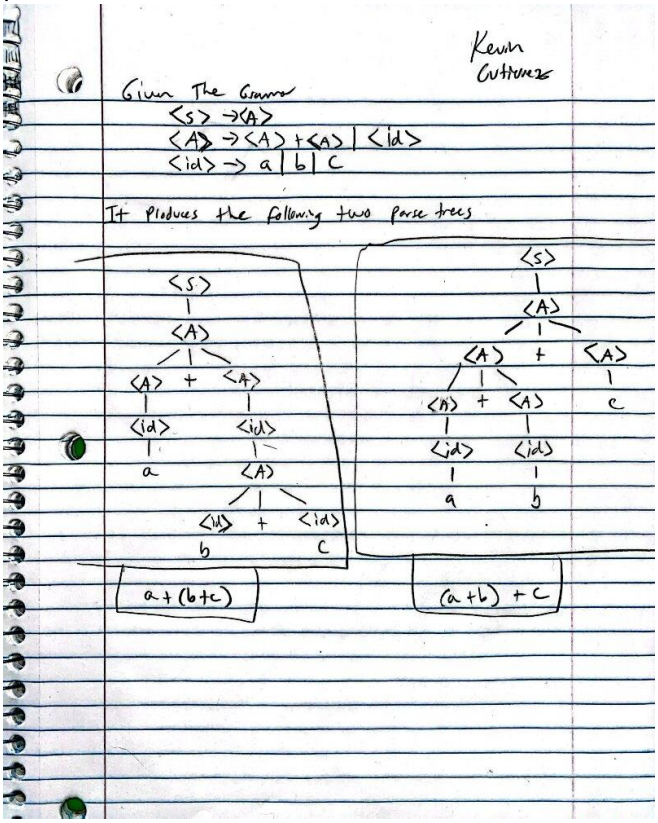
8. **Prove that the following grammar is ambiguous:**
   - <S> → <A>
   - <A> → <A> + <A> | <id>

- <id> → a | b | c

  The grammar is ambiguous because it produces more than one parse tree. This is because of the nonterminal definition

  <A> → <A> + <A> | <id>

  which allows for different groupings for addition on <A>. This can be seen by the two parse trees:



9. **Modify the grammar of Example 3.4 to add a unary minus operator that has higher precedence than either + or *.**

   unary minus just changes the sign of a number

   <assign> -> <id> = <expr>
   <id> -> A | B | C
   <expr> -> <expr> + <term>
   | <term>
   <term> -> <term> * <factor>
   | <factor>
   <factor> -> - <factor>
   | (<expr>)
   | <id>

10. Describe, in English, the language defined by the following grammar:

- <S> → <A> <B> <C>
- <A> → a <A> | a
- <B> → b <B> | b
- <C> → c <C> | c

It defines a nonterminal, <S>, as the combination of the letters a, b, c, in that order, such that each letter shows up *at least once* in the string. These are all valid strings

abc
aabc
aaaaaaaabcccc
abbbbbbccccc
abccccc
abc

These are not valid strings
bac
bc
ac

Also, grammar can be described by the following regular expression:
^a{1}b{1}c{1 }$

It works this way because of the recursively of the definition of <A>, <B>, and <C>

Note: I use regular expressions to describe the behavior of this grammar for when I review this material for the exam. I understand regular expressions more clearly than grammars currently

11. Consider the following grammar:
    - <S> → <A> a <B> b
    - <A> → <A> b | b
    - <B> → a <B> | a

Which of the following sentences are in the language generated by this grammar?

4. Baab **This is a valid sentence**
5. bbbab
6. bbaaaaa
7. bbaab **This is a valid sentence**

A regular expression that would describe this grammar is

^b{1,}aa{1,}b$

Note: I use regular expressions to describe the behavior of this grammar for when I

review this material for the exam. I understand regular expressions more clearly than grammars currently

12. Consider the following grammar:
    - <S> → a <S> c <B> | <A> | b
    - <A> → c <A> | c
    - <B> → d | <A>

Which of the following sentences are in the language generated by this grammar?

abcd	**This is a valid sentence**
acccbd	
acccbcc	
acd	
accc	**This is a valid sentence**

Sebesta Chapter 4 Review Questions

1. **What are three reasons why syntax analyzers are based on grammars?**
   1. BNF descriptions of the syntax of programs is clear and concise, both for humans and software systems that use them
   2. BNF can be used as the direct basis for the syntax analyzer.
   3. Implementations based on BNF are relatively easy to maintain because of their modularity

2. **Explain the three reasons why lexical analysis is separated from syntax analysis.**
   1. Simplicity, because lexical analysis is less complex than syntax analysis, lexical analysis can be done separately. Separating both makes the syntax analyzer smaller and less complex

   2. Efficiency, separation of syntax and lexical analysis allows for selective optimization, which is important because optimization of each yields greater or lesser gains (syntax analysis optimization is not as optimizable as lexical analysis)

   3. Portability, Lexical analysis is somewhat platform dependent due to file I/O, while syntax analysis can be platform independent

3. **Define *lexeme* and *token*.**
   Lexeme – a logical grouping of characters
   Token – Internal codes for categories of groupings of characters

4. **What are the primary tasks of a lexical analyzer?**
   To extract lexemes from a given input string and produce the corresponding tokens, perform syntactic error detection, and insert lexemes for user defined names into the symbol table

5. **Describe briefly the three approaches to building a lexical analyzer.**
   1. Write a formal description of the token patterns of the language using a descriptive language related to regular expressions (regex), and use the descriptions as an input to a software tool that generates a lexical analyzer.

   2. Design a state transition diagram that describes the token patterns of the language and write a program that implements the diagram

   3. Design a state transition diagram that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram

6. **What is a state transition diagram?**
   A state diagram is a directed graph, such that the nodes of the graph are labeled with state names, and the arcs are labeled with the input characters that cause the transitions among the states (nodes)

7. **Why are character classes used, rather than individual characters, for the letter and digit transitions of a state diagram for a lexical analyzer?**
   Because using character classes results in a significant decrease in the amount of state transitions needed (having a digit char class results in avoiding 10 state transitions for each int from 0 to 9 inclusive). It significantly decreases complexity.

8. **What are the two distinct goals of syntax analysis?**
   1. Checking the input program to determine whether it is syntactically correct and produce a diagnostic message, then return to the analysis of the program
   2. Produce a complete parse tree, or at least trace the structure of the complete parse tree

9. **Describe the differences between top-down and bottom-up parsers.**
   A top-down parser has the tree built from the root downward to the leaves, and in bottom-up, the tree is built from the leaves up to the root. In other words, the top-down parser is a leftmost derivation. A bottom-up parser is a reverse of a rightmost derivation.

10. **Describe the parsing problem for a top-down parser.**
    The parsing problem for a top-down parser is about making the correct parsing decisions to expand the leftmost nonterminal such that it matches the input string, ensuring a valid leftmost derivation of the input according to the grammar rules. This is done by recursive-decent parsing, or using parsing tables for syntax analysis, using the BNF description of the syntax of the language.