John Pennig

Homework 4

1. Chapter 3 Review Questions: 1, 2, 3, 4, 5, 6, 8, 11, 12

   1. Syntax of a language is the form of its expressions, statements, and program units. Semantics is the meaning of the expressions, statements, and programming units.

   2. Language descriptors are for users, implementors, and evaluators.

   3. A language generator is a device which can be used to generate the sentences of a language.

   4. A language recognizer is a device which is used to filter out valid inputs from non-valid from a given string.

   5. A sentence is a sentential form that only has nonterminals. Sentential form is each of the strings in a derivation including terminals.

   6. A left-recursive grammar rule is when a symbol on the left-hand side also appears at the beginning of the right-hand side.

   8. Static semantics refers to the proper form of programs and includes type checking. Dynamic semantics has to do with the meaning of statements, expressions, and program units of a language. Static semantics occurs at compilation time and dynamic semantics occurs during runtime.

11. The order of attributes evaluated is determined by synthesized or inherited attributes. Synthesized attributes are evaluated from the leaves of a tree upwards and inherited attributes are evaluated from the root down.

12. Attribute grammar is used to define the semantics of a language.

2. Chapter 3 Problem set Questions: 1, 2, 3, 6, 7, 8, 10, 11, 12

1. Recognition reads a sentence and determines if it is valid for the given alphabet. Generation has to do with the rules of the language such as syntax and generates the valid structure.

2. 1. while = “while”, “(“, expression, “)”, statement;

2. struct = “struct”, identifier, “{“, statement, “}” ;

3. switch = “switch”, “(“, expression, “)”, “{“, “{case}”, “{default}”, “}” ;

4. float = [“+” | “-“], integer, “.”, integer;

3. <assign> → <id> = <expr>

<id> → A | B | C

<expr> → <term> + <expr>

| <term>

<term> → <factor> * <term>

| <factor>

<factor> → (<expr>)

| <id>

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

<assign> → <id> = <expr>

→ A = <expr>

→ A = <term>

→ A = <term> * <factor>

→ A = <factor> * <factor>

→ A = (<expr>) * <factor>

→ A = (<expr> + <term>) * <factor>

→ A = (<term> + <term>) * <factor>

→ A = (<factor> + <factor>) * <factor>

→ A = (<id> + <id>) * <factor>

→ A = (A + B) * <id>

→ A = (A + B) * C

2. A = B + C + A

<assign> → <id> = <expr>

→ A = <expr>

→ A = <expr> + <term>

→ A = <expr> + <factor>

→ A = <expr> + <id>

→ A = <term> + <id>

→ A = <factor> + <id>

→ A = <id> + <id>

→ A = B + <expr>

→ A = B + <expr> + <term>

→ A = B + <term> + <term>

→ A = B + <id> + <id>

→ A = B + C + A

3. A = A * (B + C)

<assign> → <id> = <expr>

→ A = <expr>

→ A = <term>

→ A = <term> * <factor>

→ A = <factor> * <factor>

→ A = <id> * <factor>

→ A = A * (<expr>)

→ A = A *(<expr> + <term>)

→ A = A * (<term> + <term>)

→ A = A * (<factor> + <factor>)

→ A = A * (<id> + <id>)

→ A = A * (B + C)

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

<assign> → <id> = <expr>

→ A = <expr>

→ A = <term>

→ A = <term> * <factor>

→ A = <factor> * <factor>

→ A = <id> * <factor>

→ A = B * (<expr>)

→ A = B * (<term>)

→ A = B * (<term> * <factor>)

→ A = B * (<factor> * <factor>)

→ A = B * (<id> * (<expr>))

→ A = B * (C * (<expr> + <term>))

→ A = B * (C * (<term> + <term>))

→ A = B * (C * (<factor> + <factor>))

→ A = B * (C * (<id> + <id>))

→ A = B * (C * (A + B))

8. Considering a string such as A + B + C: We can create two distinct left-most derivations.

<S> → <A>

<A> → <A> + <A>

→ <A> + <A>

→ <id> + <A>

→ A + <A>

→ A + <A> + <A>

→ A + <id> + <id>

→ A + B + C

Or we could also have

<S> → <A>

<A> → <A> + <A>

→ <A> + <A> + <A>

→ <id> + <id> + <id>

→ A + B + C

10. This language consists of strings with one or more a's followed by one or more b's ending with one or more c's.

11. 4. baab: Yes a valid string according to the grammar.

    5. bbbab : No, not valid, does not have at least two consecutive a's in the middle.

    6. bbaaaaa: No not valid, does not end with b.

    7. bbaab: Yes, a valid string according to the grammar.

12. Sentences in the language: abcd, accc

3. Chapter 4 Review Questions: 1-10

1. Three reasons that syntax analyzers are based on grammars is that BNF descriptions of the syntax of programs are clear and concise. Second, the BNF description of the language can be used as a basis of the syntax analyzer. Finally, implementation on BNF are easy to maintain due to modularity.

2. Lexical analyzers and syntax analyzers are separated due to simplicity, efficiency, and portability of lexical analyzers compared to syntax analyzers.

3. A lexeme is a logical grouping of characters collected by the lexical analyzer, and tokens are the internal codes for the categories of logical groupings.

4. Lexical analyzers skip white space and comments, insert lexemes for user-defined names into the symbol table, and detect syntactic errors in the code.

5. 1. Write a formal description of the tokens in a language that will be used as input to a program to generate a lexical analyzer.

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

6. A state transition diagram is a directed graph.

7. The lexical analyzer does not care which digit or character is being used, only whether it is a character or a digit.

8. Check the program and make sure it is syntactically correct. If it is found to be incorrect, the analyzer must produce an error message and recover. Also, the syntactic analyzer must produce a complete parse tree.

9. A top-down parse tree is built from the root down, a bottom-up parse tree is built from the leaves up.

10. The parsing problem for a top-down parser is to determine which rule to apply to the current nonterminal to correctly choose the next node in a parse tree.