Thomas Herold

Professor Ramnath

CSCI 330-54

26 February 2025

Homework 4

Chapter 3 Review Questions:

1. Syntax - The form of its expressions, statements, and program units for a programming language.

Semantics – The meaning of those expressions, statements, and program units.

2. Language descriptions are intended for language designers, compiler writers, and programmers.

3. A language generator produces strings that belong to a language by applying production rules of a formal grammar.

4. A language recognizer determines whether a given string is a valid sentence in a language.

5. A sentence is a sentential form that consists **only of terminal symbols**. A sentential form is not a sentence, as it is a string of symbols which are **both terminal and nonterminal**. Although, both are derived from the start symbol of a grammar

6. A rule is said to be left recursive when a grammar rule has its LHS (left-hand side) appearing at the beginning of its RHS (right-hand side). An example would be as such: A => A + B | B. A can left recursively derive another string starting with A.

8. Static semantics – deals with the legal forms of programs, focusing on syntax rather than semantics. Static semantics govern the structure of programs, like type checking and variable declarations.

Dynamic semantics – deals with the rules that define the behavior of programs during execution. It governs how expressions are evaluated and how control statements operate.

11. The order of evaluation of attributes is determined by the dependencies between attributes. Attributes that depend on other attributes must be evaluated after those attributes. Determining attribute evaluation order for the general case of an attribute grammar is a complex problem, requiring the construction of a dependency graph to show all attribute dependencies.

12. Attribute grammars are devices used to describe more of the structure of a programming language than can be described with context-free grammars. They extend context-free grammars by associating attributes with grammar symbols and defining rules to evaluate said attributes. They help in specifying syntax and static semantics of a language.

<u>Chapter 3 Problem Set Questions:</u>

1. Generation specifies a set of production rules that can generate valid strings in the language. Starting with an initial symbol, the rules will expand non-terminal symbols into terminal symbols. Generation can be used to design compilers and interpreters to generate code/parse a given input.

Recognition specifies a set of rules that recognizes whether a given string belongs to the language. Recognition is fundamental for syntax analysis for a compiler, verifying that the code fits the language.

2. a-d (1-4) are done as follows:

a. A C++ while statement

<while_statement> → while (<expression>) <statement>

<expression> → <term> { (+ | -) <term> }

<term> → <factor> { (* | /) <factor> }

<factor> → (<expression>) | <id> | <number> | <comparison>

<statement> → <assignment> | <while_statement> | <if_statement> | <expression>

<comparison> → <expression> <compare> <expression>

<compare> → > | < | == | != | >= | <=

b. A C++ struct (structure) declaration statement

<structure_declaration> → struct <id> { <member_declarations> }

<member_declaration> → <type> <id>

<type> → int | float | double | char

c. A C switch statement

<switch_statement> → switch (<expression>) { <case_clauses> }

<case_clause> → case <constant_expression> : <statement> { break; }

<constant_expression> → <id> | <number>

<statement> → <assignment> | <switch_statement> | <if_statement>

// (Things like <expression> can be done with grammar rules in above questions.)

d. C float literals

<float_literal> → float <id> = <float_value>

<float_value> → [+ | -] <digits> . <digits> [(e | E) [+ | -] <digits>]

<digits> → <digit> { <digit> }

<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

3.

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <term> | <term> + <expr>

<term> → <factor> | <term> * <factor>

<factor> → (<expr>) | <id>

6. a-c (1-3) are done as follows:

a. 1. <assign> → <id> = <expr>

→ A = <id> * <expr>

→ A = A * (<expr>)

→ A = A * (<id> + <expr>)

→ A = A * (B + (<expr>))

→ A = A * (B + (<id> * <expr>))

→ A = A * (B + (C * <id>))

→ A = A * (B + (C * A))

b. 2. <assign> → <id> = <expr>

→ B = <id> * <expr>

→ B = C * (<expr>)

→ B = C * (<id> * <expr>)

→ B = C * (A * <id> + <expr>)

→ B = C * (A * C + <id>)

→ B = C * (A * C + B))

c. 3. <assign> → <id> = <expr>

→ A = <id> * <expr>

→ A = A * (<expr>)

→ A = A * (<id> + <expr>)

→ A = A * (B + (<expr>))

→ A = A * (B + (<id>))

→ A = A * (B + (C))

7. a-d (1-4) are done as follows:

a. 1. <assign> → <id> = <expr>

→ A = <term>

→ A = <term> * <factor>

→ A = <factor> * <factor>

→ A = (<expr>) * <factor>

→ A = (<expr> + <term>) * <factor>

→ A = (<term> + <factor>) * <factor>

→ A = (<id> + <id>) * <id>

→ A = (A + B) * C

b. 2. <assign> → <id> = <expr>

→ A = <expr> + <term>

→ A = <expr> + <term> + <term>

→ A = <term> + <factor> + <factor>

→ A = <factor> + <id> + <id>

→ A = <id> + C + A

→ A = B + C + A

c. 3. <assign> → <id> = <expr>

→ A = <term>

→ A = <term> * <factor>

→ A = <factor> * (<expr>)

→ A = <id> * (<expr> + <term>)

→ A = A * (<expr> + <term>)

→ A = A * (<term> + <factor>)

→ A = A * (<factor> + <id>)

→ A = A * (<id> + C)

→ A = A * (B + C)

d. 4. <assign> → <id> = <expr>

→ A = <term>

→ A = <term> * <factor>

→ A = <factor> * (<expr>)

→ A = <id> * (<term>)

→ A = B * (<term> * <factor>)

→ A = B * (<factor> * (<expr>))

→ A = B * (<id> * (<expr> + <term>))

→ A = B * (C * (<term> + <factor>))

→ A = B * (C * (<factor> + <id>))

→ A = B * (C * (<id> + B))

→ A = B * (C * (A + B))

8. The grammar is ambiguous because we can derive a + b in multiple ways:

<S> → <A> → <A> + <A> → <id> + <A> → a + <A> → a + <id> → a + b

<S> → <A> → <A> + <A> → <A> + <id> → <A> + b → <id> + b → a + b

This is done by swapping the order in which the <A>'s are turned into <id>'s which are then turned into a or b. a + b + c could be generated as such: S → <A> + <A>, which could turn into <A> + <A> → <id> + <A> **OR** <A> + <A> → <A> + <A> + <A>. Both can generate a termination string of a + b + c.

10. This grammar will generate a string of a, b, and c. There will be at least one of each different character, meaning that zero of any of the three characters is not possible. The smallest string is three characters, abc, and the largest string can be infinitely long. Every string will start with any chosen amount of a's, followed by with any chosen amount of b's, and ending with any chosen amount of c’s. No character will be able to have the same character before and after it. Ex. abac is not possible since after the first a switches to a b, no more a’s will exist in said string.

11. Sentences that can be generated by this grammar: **a. baab, c. bbaaaaa, and d. bbaab**

12. Sentences that can be generated by this grammar: **a. abcd, e. accc**

<u>Chapter 4 Review Questions:</u>

1. Syntax analyzers are based on grammars for these three reasons:

- BNF descriptions of the syntax of programs are clear and concise, both for humans and for software systems that use them.
- The BNF description can be used as the direct basis for the syntax analyzer.
- Implementations based on BNF are relatively easy to maintain because of their modularity.

2. The three reasons why lexical analysis is separated from syntax analysis are simplicity, efficiency, and portability.

3. Lexeme – The lowest-level syntactic unit.

Token – Categorizes lexemes that share common attributes.

4. The primary tasks of a lexical analyzer are:

- Identify substrings in the input and group them into lexemes.
- Assign a token to each lexeme.
- Skip irrelevant characters such as whitespace and comments.
- Detect and report lexical errors in the input.
- Construction of the symbol table, which acts as a database of names for the compiler.

5. Three approaches to building a lexical analyzer:

1. Write token patterns using regular expressions, then use a tool (such as UNIX system’s lex) to generate the analyzer.
2. Design a state transition diagram that describes the token patterns of the language using a descriptive language related to regular expressions.

3. Do the same as #2, but instead hand-construct a table to implement the transitions rather than using a descriptive language.

6. A state transition diagram (or just state diagram) is a directed graph. The nodes of a state diagram are labeled with state names. The arcs are labeled with the input characters that cause the transitions among the states. An arc may also include actions the lexical analyzer must perform when the transition is taken.

7. Character classes are used instead of individual characters to simplify the state transition diagram. Grouping similar characters into a class lets the lexical analyzer handle them with a single transition. Ex. There are 52 possible letters, which would require **52** transitions from the transition diagram's initial state. Instead, a character class can be used to make a **single** transition on the first letter of any name.

8. Two distinct goals of syntax analysis:

- Check the input of the program to determine whether it is syntactically correct.
- Produce a complete parse tree or at least trace the structure of the complete parse tree, for syntactically correct input.

9. Top-down parser – The parse tree is built from the root downward to the leaves.

Bottom-up parser – The parse tree is built from the leaves upward to the root.

10. The parsing problem for a top-down parser is about choosing the correct rule to expand the leftmost nonterminal in a sentence. This type of parser can face the problem of left recursion, which can require grammar transformations.