Homework 4 CSCI 330
Sagar Neupane
Due date: Tuesday February 25th
Ramnath, Sarnath

1. Define syntax and semantics.
- Syntax: The structure or form of a language's statements, expressions, and program units.
- Semantics: The meaning of those expressions, statements, and program units.

2. Who are language descriptions for?
- Initial evaluators, implementors, users

3. Describe the operation of a general language generator.
- Based on a language's syntax, a language generator generates valid sentences in that language. It creates a new sentence in the language each time it is turned on.

4. Describe the operation of a general language recognizer.
- A language recognizer takes a string of text and checks if it follows the rules of the language. It works like a checker, either allowing the input or rejecting it according to set rules.

5. What is the difference between a sentence and a sentential form?
- A sentence is a fully derived string composed entirely of terminal symbols, representing a valid expression in the language.

  A sentential form is a step in creating a sentence that includes both final words and symbols that still need to be changed before making a complete sentence.

6. Define a left-recursive grammar rule.
- If the left-hand side (LHS) of a grammar rule appears as the leftmost symbol in its right-hand side (RHS), then the rule is left-recursive. This may cause parsers to recur infinitely.

7. What three extensions are common to most EBNFs?

- Square brackets [] indicate optional elements, which mean that the contained portion may appear zero or once.

  Repetitions: The enclosed part can be repeated zero or more times, as indicated by the curly braces { } around it.

  Alternatives: A choice between several alternatives, enclosed in parenthesis ( ) and divided by a vertical bar |.

8. Distinguish between static and dynamic semantics.

- Static semantics check rules before the program runs, like ensuring the types of variables are correct and that variables are used in the right places, without looking at how the program behaves when it's actually running.
  In contrast, dynamic semantics explain what happens when programs run, showing how different parts work and interact while the program is being executed. Static semantics checks if the code is correct before it runs, while dynamic semantics explains what the code does while it is running.

9. What purpose do predicates serve in an attribute grammar?

- Predicates functions is used to state the static semantic rules of the language.

10. What is the difference between a synthesized and an inherited attribute?

- Synthesized attributes are calculated based on the values of a node's children using certain rules. They can also include attribute values passed down from above. Essentially, they help to send meaningful information up the structure of the tree. On the other hand, inherited attributes come from parent nodes (or from sibling nodes) to give information to lower-level nodes. This separation helps a computer program, like a compiler, to give meaning to code parts and check rules about that meaning that aren't clearly stated in the code.

11. How is the order of evaluation of attributes determined for the trees of a given attribute grammar?

- The parse tree is evaluated using its BNF grammar, and each node can have no attribute values attached to it.

12. What is the primary use of attribute grammars?

- Attribute grammars are mainly used to explain static rules in programming languages, like making sure the types are correct and checking where variables can be used.

Problem sets

1. The two mathematical models of language description are generation and recognition. Describe how each can define the syntax of a programming language
   - A language generator sets the rules for a programming language by creating all the valid sentences that can be made in that language. It uses standard grammar rules, like context-free grammars (CFGs), to create programs that are written correctly. This method helps us understand how a language is built and forms the base for creating languages.
   - A language recognizer checks if a given string follows the rules of the language. It works like a filter that lets in programs that are written correctly and keeps out the ones that are not. Recognizers are often used in compilers and interpreters. They help check if the input programs follow the rules of the language.

2. Write EBNF descriptions for the following (refer to https://www.w3schools.com/ for any language details):
   a. A C++ **`while`** statement

<while-stmt> ::= "while" "(" <condition> ")" <statement>

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

<struct-decl> ::= "struct" <id> "{" { <type> <id> ";" } "}" ";"

c. A C **`switch`** statement

<switch-stmt> ::= "switch" "(" <expression> ")" "{" { "case" <constant> ":" { <statement> } } ["default" ":" { <statement> }] "}"

d. C **`float`** literals

<float-literal> ::= <real> <suffix>

| <real> <exponent> <suffix>

| <integer> <exponent> <suffix>

<real> ::= <integer> "." <integer>

| "." <integer>

<exponent> ::= ("e" | "E") ["+" | "-"] <integer>

<suffix> ::= "" | "f" | "F" | "l" | "L"

<integer> ::= [0-9]+

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

Here the Original BNF from 3.4 is given below-

<assign> -> <id> = <expr>

<id> -> A | B | C

<expr> -> <expr> + <term> | <term>

<term> -> <term> * <factor> | <factor>

<factor> -> (<expr>) | <id>

+ takes precedence over * when the operators for and are switched. To make + right associative, I make sure that for, the parse tree expands from the right instead of the left.

The new BNF is

<assign> -> <id> = <expr>

<id> -> A | B | C

<expr> -> <expr> * <term> | <term>

<term> -> < factor > + <term> | <factor>

<factor> -> (<expr>) | <id>

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> | <id>

<unary> → "++" <factor> | "--" <factor> | <factor> "++" | <factor> "--"

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

<boolean_expr> ::= <boolean_term> | <boolean_expr> "||" <boolean_term>

<boolean_term> ::= <boolean_factor> | <boolean_term> "&&" <boolean_factor>

<boolean_factor> ::= "!" <boolean_factor> | <relational_expr> | "(" <boolean_expr> ")"

<relational_expr> ::= <arithmetic_expr> <relational_op> <arithmetic_expr>

<relational_op> ::= "==" | "!=" | "<" | "<=" | ">" | ">="

<arithmetic_expr> ::= <identifier> | <number> | <arithmetic_expr> <arith_op> <arithmetic_expr>

<arith_op> ::= "+" | "-" | "*" | "/"

<identifier> ::= <letter> { <letter> | <digit> }

<number> ::= <digit> { <digit> }

<letter> ::= "a" | "b" | ... | "z" | "A" | "B" | ... | "Z"

<digit> ::= "0" | "1" | ... | "9"

6. Using the grammar in Example 3.2, show a leftmost derivation for each of the following statements:
   a. `A = A * (B + (C * A))`

      <assign> → <id> = <expr>

      A = <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. `B = C * (A * C + B)`

<assign> → <id> = <expr>

B = <expr>

B = <id> * <expr>

B = C * (<expr>)

B = C * (<id> * <expr>)

B = C * (A * <expr>)

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

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

B = C * (A * C + B)

c. `A = A * (B + (C))`

<assign> → <id> = <expr>

A = <expr>

A = <id> * <expr>

A = A * (<expr>)

A = A * (<id> + <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:
   a. `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> + <term>) * <factor>

→ A = (<id> + <term>) * <factor>

→ A = (A + <term>) * <factor>

→ A = (A + <factor>) * <factor>

→ A = (A + <id>) * <factor>

→ A = (A + B) * <factor>

→ A = (A + B) * <id>

→ A = (A + B) * C

b. `A = B + C + A`

<assign>

→ <id> = <expr>

→ A = <expr>

→ A = <expr> + <term>

→ A = <expr> + <term> + <term>

→ A = <term> + <term> + <term>

→ A = <factor> + <term> + <term>

→ A = <id> + <term> + <term>

→ A = B + <term> + <term>

→ A = B + <factor> + <term>

→ A = B + <id> + <term>

→ A = B + C + <term>

→ A = B + C + <factor>

→ A = B + C + <id>

→ A = B + C + A

c. `A = A * (B + C)`

<assign>

→ <id> = <expr>

→ A = <expr>

→ A = <term>

→ A = <term> * <factor>

→ A = <factor> * <factor>

→ A = <id> * <factor>

→ A = A * <factor>

→ A = A * (<expr>)

→ A = A * (<expr> + <term>)

→ A = A * (<term> + <term>)

→ A = A * (<factor> + <term>)

→ A = A * (<id> + <term>)

→ A = A * (B + <term>)

→ A = A * (B + <factor>)

→ A = A * (B + <id>)

→ A = A * (B + C)

d. `A = B * (C * (A + B))`

<assign>

→ <id> = <expr>

→ A = <expr>

→ A = <term>

→ A = <term> * <factor>

→ A = <factor> * <factor>

→ A = <id> * <factor>

→ A = B * <factor>

→ A = B * (<expr>)

→ A = B * (<term>)

→ A = B * (<term> * <factor>)

→ A = B * (<factor> * <factor>)

→ A = B * (<id> * <factor>)

→ A = B * (C * <factor>)

→ A = B * (C * (<expr>))

→ A = B * (C * (<expr> + <term>))

→ A = B * (C * (<term> + <term>))

→ A = B * (C * (<factor> + <term>))

→ A = B * (C * (<id> + <term>))

→ A = B * (C * (A + <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

   We must show that a string can be generated in two or more different ways, producing distinct parse trees, to establish that the grammar is ambiguous.

   i) Left most derivation

   <S> => <A>

   => <A> + <A>

   => id + <A>

   => a + <A>

   => a + <A> + <A>

   => a + id + <A>

   => a + b + <A>

   => a + b + id

   => a + b + c

   ii) Left-most derivation

   <S> => <A>

   => <A> + <A>

   => <A> + <A> + <A>

   => id + <A> + <A>

   => a + <A> + <A>

   => a + id + <A>

=> a + b + <A>

=> a + b + id

=> a + b + c

Therefore, following grammar is ambiguous.

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

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <term> | <term> (* + has lower precedence than * *)

<term> →<term> * <factor> | <factor> (* * has higher precedence than + *)

<factor> →<unary> | (<expr>) | <id>

<unary> →"-" <factor> (* Unary minus has the highest precedence *)

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

- This grammar defines a language made up of strings that follow to a particular pattern:

  Generates one or more ‘a' characters.
  Any string in this language will therefore have the following form:

  One or more 'a' character, followed by
  One or more 'b' characters, followed by
  One or more 'c' characters.
  Example-

  abc

  abbbbbbbbbbc

  aaaaaaaaaabbcc

  aabcc

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
5. bbbab
6. bbaaaaa
7. bbaab

- baab, bbaab

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, acccbd, acccbcc, acd, accc.

- abcd, accc

Chapter 4 Review Questions

1. What are three reasons why syntax analyzers are based on grammars?

   Clear and Simple: BNF (Backus-Naur Form) explains syntax in a way that is easy to understand for both people and computer programs.

   Direct Basis for Syntax Analyzers: Using BNF directly helps create syntax analyzers.

   Easy to Maintain: Systems built using BNF are organized in parts, which makes them simpler to take care of.

2. Explain the three reasons why lexical analysis is separated from syntax analysis.

Simplicity: Methods for analyzing words are easier than methods for analyzing sentences. By keeping them separate, the sentence analyzer becomes smaller and easier to understand.

Efficiency: Analyzing words takes a lot of time during compilation, and separating it allows for specific improvements to the word analysis process.

Portability: The lexical analyzer works with input files and may depend on the platform it runs on, while the syntax analyzer can work on any platform. Being separated makes it easier to carry.

3. Define *lexeme* and *token*.

   Lexeme: A sequence of characters in a program that fits a specific token pattern.

   Token: A label given to a word, usually shown as a number to make it easier to analyze language and grammar.

4. What are the primary tasks of a lexical analyzer?

   Take words from the input text and create matching labels.

   Ignore unnecessary blank spaces and comments.

   Add names chosen by the user into the symbol table.

   Find and report mistakes in the way tokens are written, like incorrect numbers.

5. Describe briefly the three approaches to building a lexical analyzer.

Using a lexical analyzer generator: Lexical analyzers are automatically generated by tools like as Lex, which define token patterns using regular expressions.

Using a diagram of state transition: Token patterns are described by a state diagram, and the diagram's implementation is accomplished by writing a program.

Table-driven implementation of a state transition diagram: A lexical analysis program uses the state transition diagram after it has been manually transformed into a transition table.

6. What is a state transition diagram?

   A directed graph that shows how a system changes states depending on input symbols is called a state transition diagram. It is frequently used in lexical analysis to identify programming language tokens.

7. Why are character classes used, rather than individual characters, for the letter and digit transitions of a state diagram for a lexical analyzer?

   Character classes are used instead of single characters in diagrams that show how a lexical analyzer works. This makes it easier to design and less complicated. Instead of making different transitions for every letter (A-Z, a-z) and every number (0-9), they are grouped into two classes: LETTER and DIGIT. This makes the diagram smaller and easier to handle, which helps it work better. Using character classes makes the text easier to read and maintain. It helps recognize tokens correctly without having to deal with each character separately all the time.

8. What are the two distinct goals of syntax analysis?

Finding and fixing syntax mistakes: The syntax checker needs to make sure the input program follows the correct rules and give helpful error messages if there are problem

Parse Tree Construction: It needs to create a full parse tree (or at least show how it's set up) to help with the next steps in the compiler.

9. Describe the differences between top-down and bottom-up parsers.

   Top-down and bottom-up parsers are different in the way they build parse trees. Top-down parsers begin at the top of the tree and work their way down to the bottom. They use the left part of the input to guess how to shape the tree based on the tokens they see. They use techniques like recursive descent and LL parsing, which help them make decisions about how to understand the text quickly.

   On the other hand, bottom-up parsers start from the smallest parts and work up to the whole by identifying the last steps of the process in the opposite order. They use methods called shift-reduce parsing, like LR parsing, which can work with more types of grammar, even those that have left recursion. Top-down parsers make decisions based on the next tokens they see, while bottom-up parsers figure out the right way to reduce by looking at parts of the input called handles.

10. Describe the parsing problem for a top-down parser.

    When we have a left sentential form, like (xAa), the parser needs to find out what the next step is in creating a leftmost derivation.

    It must pick the right production rule for A from several options.

    The parser decides based on the first piece of input that can be created from each rule of A.

If several RHSs start with the same word, it confuses the parser and makes it hard to understand.