CS224 – Grammars

Purpose: The syntax of a programming language can be described with a grammar and a particular string can be "parsed" using that grammar to see if it is syntactically correct and see the pieces that make up that syntax.

Knowledge: This activity will help you become familiar with the following content knowledge:

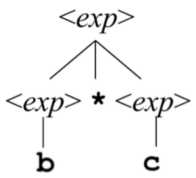
- How we describe grammars
- How to diagram the parse of a string with a parse tree.
- How a parse tree can describe operator precedence and associativity
- How some grammars can be ambiguous

Activity: With your group perform the following tasks and answer the questions. You will be reporting your answers back to the class in 1 hour.

1. A grammar can describe how tokens can be organized in legal strings. Each grammar is made up of "non-terminals" which can represent a node in the parse tree and "terminals" which are leaves in the parse tree. The following is a grammar representing an expression. In this grammar there is one non-terminal `<exp>` and the terminals are a, b, c, +, *, (, and). Choices in the grammar are indicated by `|`. So the following grammar indicates that a legal expression can be the sum of two expressions or, the product of two expressions, or an expression enclosed in parentheses. Finally an expression can simply be `a`, or `b` or `c`.

   ```
   <exp> ::= <exp> + <exp> | <exp> * <exp> | ( <exp> ) | a | b | c
   ```

   A parse of the expression `b * c` can be viewed as the following parse tree by using the rules of the grammar, first substituting `<exp>` with the rule `<exp> * <exp>` and then substituting the first `<exp>` with the rule `b` and then substituting the second `<exp>` with the rule `c` :

Draw the parse trees for each of the following:

```
a * b * c
( a + b )
( a + ( b ) )
```

2. Consider the following examples of Java declarations:

```
float a;
boolean a, b, c;
int a=1, b, c=1+2;
```

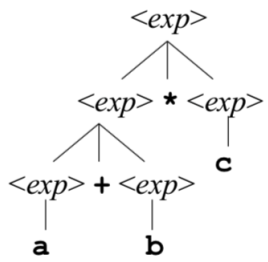
An incomplete grammar for Java declarations is given below:

```
<var-dec> ::= <type-name> <declarator-list> ;
<type-name> ::= boolean | byte | short | int | long | char | float | double
<declarator> ::= <variable-name> | <variable-name> = <expr>
```

Complete this grammar by giving the definition of the grammar for `<declarator-list>`
Hint: Use `<declarator>` and recursion.

3. Going back to the expression grammar,
`<exp> ::= <exp> + <exp> | <exp> * <exp> | ( <exp> ) | a | b | c`

This is a parse tree for `a + b * c`:



Give another parse tree for `a + b * c` using this grammar.

Which parse tree correctly represents the precedence of the operators? Why?

4. It would be great if the grammar forced the correct parse tree. Take a look at the modified grammar and give the parse tree for `a + b * c`:

```
<exp> ::= <exp> + <exp> | <term>
<term> ::= <term> * <term> | ( <exp> ) | a | b | c
```

   It forces the correct precedence! But all is not well. Give two parse trees for `a + b + c` with this grammar. Which one do you prefer? Would it make a difference if we replace the operator with division?

5. We would like the grammar to also correctly give the "associativity" of the operators. Performing (a + b) + c would be left-associativity, whereas a + (b + c) would be right-associativity.

   Take a look at another modification to the grammar and verify that it will give the normal (left-associative) meaning for both multiplication and addition:

```
<exp> ::= <exp> + <term> | <term>
<term> ::= <term> * <factor> | factor
<factor> ::=  ( <exp> ) | a | b | c
```

6. Starting with the grammar above, add a left-associative `&` operator at lower precedence than any of the others. Then add a right-associative `**` operator at higher precedence than any of the others.

   Verify your grammar is correct by drawing the parse trees for :

   (a) `a & b & c`

   (b) `a & b + c`

   (c) `a ** b ** c + d`

7. A well known problem with grammars for programming languages is called the "dangling else" problem. Consider the following grammar:

```
<stmt> ::= <if-stmt> | s1 | s2
<if-stmt> ::= if <expr> then <stmt> else <stmt> | if <expr> then <stmt>
<expr> ::= e1 | e2
```

   Draw two different parse trees with this grammar for:

```
if e1 then if e2 then s1 else s2
```

   It may not be clear which one the programmer intends. How can we solve this problem?