Parsing with ANTLR

Most of the examples below are based on the ANTLR Reference book, which is very readable (and on reserve for you at the library!). The ANTLR documentation is also quite good.

Ambiguity

Writing a perfectly unambiguous grammar is tricky. ANTLR resolves ambiguity using the following guidelines:

- Rules take precedence according to the order in which they were declared.
- Operators are left-associative by default
- Operators can be made right-associative using the option <assoc=right>

Consider these two grammars:

```
grammar ambiguity1;

expr : INT
   |<assoc=right> expr '^' expr
   | expr '*' expr
   | expr '+' expr;

INT : [0-9]+;
```

```
grammar ambiguity2;

expr : expr '+' expr
   | expr '*' expr
   | expr '^' expr
   | INT;

INT : [0-9]+;
```

Using the guidelines above, draw parse trees for the following inputs:

1. 1 + 2 * 3 + 1
2. 2 ^ 2 ^ 2
3. 2 ^ 2 * 3 ^ 1

Listeners and Visitors

When we parse an input, we probably want to do something more to it than just turn it into a parse tree. Many parser generators let us embed actions directly in grammars, but it's nicer to decouple the grammar from these actions. ANTLR provides two ways of doing this: listeners and visitors.

In the listener style, ANTLR parses an input and then walks its parse tree. When the tree-walker enters and exits each node, it calls a listener method. If we need to perform some action at this point, we can override that method to do it. You'll be working with listeners in homework 3.

In the visitor style, we don't use ANTLR's tree-walker. Instead, ANTLR generates a visitor interface, we implement that interface, and then call the visitor methods to visit the nodes of the parse tree directly. This way we can control the way in which the tree is traversed, and also can pass values from node to node, as visit methods return values (listener methods do not, so they need to use other methods of information sharing).

We can also differentiate nodes if we want the listeners for them to treat them differently by giving them labels. Consider this rule:

```
expr : INT
   | expr '*' expr
   | expr '+' expr;
```

When we generate listeners, we will get methods called `enterExpr` and `exitExpr`. If we want those expression types to behave differently, we could write some checks within those methods, but it would be nicer to separate them completely. Let's do that by adding labels (note: those aren't comments! Comments are marked off with //)

```
expr : INT # Int
   | expr '*' expr # Mult
   | expr '+' expr # Add
;
```

Now, we will get the enter listeners `enterInt`, `enterMult`, and `enterAdd` (and the corresponding exit listeners). You can see how the way you write your listeners might influence how you design your grammar.

Pull down the repo at https://github.com/jn80842/cse401section5 (a very simplified version of the homework3 starter kit). Try writing listeners such that your application will take in an arithmetic expression with no parentheses and return the expression fully parenthesized.