This lab provides a glimpse at the role grammars can play in computing by introducing ANTLR, a parser generator, along with a small application of the tool. It is due in class Monday, May 4 but earlier handins are welcome and encouraged.

While you may discuss ideas and strategies for problems with other students, you should always make the first attempt on a problem yourself. You may not collaboratively write solutions or copy a solution that one person in the group writes up. You also may not look for, copy, or use solutions from other sources, including from generative AI like ChatGPT, even if you make changes. There's no such thing as using someone else's solution for a problem "as an example" for writing your own.

Preliminaries

Part of this lab involves working with Java code, so it is recommended that you use Eclipse and create a project to hold all of your files (even the non-Java ones). Only one piece (running ANTLR to generate a parser) must be done on the Linux systems because it uses software installed there, but as with the regex lab, is it recommended for simplicity to do all of your work in Linux.

- Create a new Eclipse project named `parsing`. Make sure the "Create module-info.java file" box is *not* checked before clicking Finish in the dialog box.
- Create a text file in your project to hold your answers for the exercises: Start with File→New→Untitled Text File, add your name at the beginning of the file, and save it with File→Save. Name the file `parsing-exercises.txt` and be sure to save it in the top level of your project folder — choose the project folder as the parent folder in the Save As dialog.
- Import the files from `/classes/cs229/parsing` into the `src` folder in your project. (The Java files will be shown in the "default package" under `src` while the other file will just be in `src`.) There will be errors in the Java files; don't worry about that for now.

Handin

- Make sure your name is at the beginning of your `parsing-exercises.txt` file.
- Hand in your work by copying the entire project directory `~/cs229/workspace/parsing` to your handin directory `/classes/cs229/handin/`*`username`* (where *username* is replaced by your username).

In this lab you will be working with ANTLR (ANother Tool for Language Recognition), a parser generator. You can find out much more about ANTLR at its website `www.antlr.org` if you are interested.

1 Exploring Grammars With ANTLR Lab

ANTLR Lab is an online interface to ANTLR that lets you experiment with grammars. Access it at `lab.antlr.org`.

There are two steps to defining a grammar for ANTLR. First define the parser rules — these are the rules of the grammar. ANTLR's syntax is similar to the BNF syntax discussed in class.

1. Paste the following into the "Parser" tab on the left side of the ANTLR Lab page:

   ```
   parser grammar ExprParser;
   options { tokenVocab=ExprLexer; }

   expression
       : term ((PLUS | MINUS) term)*
       ;

   term
       : factor ((TIMES | DIV) factor)*
       ;

   factor
       : NUMBER
       | LPAREN expression RPAREN
       ;
   ```

   Also change the "Start rule" box to `expression` (since that's the starting rule of this grammar).

This is the ANTLR version of the simple expression grammar from section 4.2 (the bottom of page 191) in the book. (Compare it to the book's version to understand the syntax; note that `*` has the same meaning as in regular expressions. There's also one change, which is that the notion of identifiers has been removed — this grammar only supports numeric expressions.)

The next step is to define what the tokens `PLUS`, `MINUS`, `NUMBER`, etc mean. This is used by the lexer to group the individual characters of the input into the units that the parser will build a parse tree from.

2. Paste the following lexer rules into the "Lexer" tab on the left side of the ANTLR Lab page:

```
lexer grammar ExprLexer;

PLUS  : '+' ;
MINUS : '-' ;
TIMES : '*' ;
DIV   : '/' ;

LPAREN : '(' ;
RPAREN : ')' ;

NUMBER: [0-9]+ ;

WS : [ \t\r\n]+ -> skip ;
```

You'll notice some regular-expression-like syntax with the `[ ]`, `*`, and `+` symbols. The `-> skip` part after the pattern for whitespace tells the lexer to discard those tokens — whitespace separates tokens rather than being something parsed in the grammar.

Now test the grammar.

3. Paste (or type) each of the following in turn (one at a time, not all at once) into the "Input" panel on the right side of the ANTLR Lab page and click "Run":

```
3+4*5
3+(4*5)
(3+4)*5
((3+4)*5)+(6+2+1)*8
```

   In each case observe the parse tree and note how the tree structure reflects the operator precedence rules implemented through the structure of the grammar.

4. Try several illegal expressions (improperly balanced/nested parens, missing `*` operators (e.g. `2(3+4)`, and letters or other symbols). What happens in each case? Include the expressions you tried and a description of what happens in each case in `parsing-exercises.txt`.

2 More Grammars

Put your answers for these exercises (including the lexer and parser rules) into `parsing-exercises.txt`.

5. (a) Translate the following BNF grammar for a (very small) subset of Java syntax into lexer and parser rules for ANTLR.

$$\begin{aligned}
\langle name \rangle &::= \langle object_ref \rangle \ [\ \text{"."}\ \langle identifier \rangle\ ] \\
\langle object_ref \rangle &::= \langle identifier \rangle \mid \langle method_call \rangle \\
\langle method_call \rangle &::= \langle identifier \rangle\ \text{"("}\ \langle name \rangle\ [\ \text{","}\ \langle name \rangle\ ] \ldots\ \text{")"} \\
\langle identifier \rangle &::= \text{"}a\text{"} \mid \text{"}b\text{"} \mid \text{"}c\text{"} \mid \text{"}x\text{"} \mid \text{"}y\text{"} \mid \text{"}z\text{"}
\end{aligned}$$

   (b) Write down six things generated by the grammar above and use them to test your grammar in ANTLR Lab. (Be sure to update the "Start rule"!) Your examples should demonstrate all of the possibilities represented in the rules — make sure each of the different rules and variations appears in at least one parse tree.

6. Write lexer and parser rules for ANTLR for compound propositions made up of propositional variables, parentheses, and the logical operators `and`, `or`, and `not`. Propositional variables can be the lowercase letters p, q, r, and s.

7. Write lexer and parser rules for ANTLR for numeric literals in Java. Numeric literals include integers (e.g. 0, 1035, -47, +2) and floating point numbers (e.g. 17.3, 0.73, .50, 23.1e67, -1.34E-12, +0.2, 100E+100). Note that the only numbers that can start with 0 are the integer 0 and a floating point number between -1 and 1.

3 A Complete Application

The reason for a tool like ANTLR is to be able to generate code to handle scanning and parsing, which you can then build on to create an application. In these exercises you'll use the `antlr` command line tool to generate a parser from a grammar and see how that is used in a complete application.

The first step is defining the grammar. ANTLR Lab allowed the lexer and parser rules to be specified separately, but the command line parser generator tool uses a single file. The provided `Expr.g4` file (which should be in the `src` directory of your project) contains the parser and lexer rules from #1 and 2 with only a few minor syntax adjustments to reflect that both sets of rules are in the same file. You can open it in Eclipse to see what it looks like.

Next:

8. Run `antlr` to generate the code from this grammar: open a terminal window, change to the `src` subdirectory of your project (where you want the generated files to go):

   ```
   cd ~/cs229/workspace/parsing/src
   ```

   and run `antlr`:

   ```
   /classes/cs229/antlr/antlr Expr.g4
   ```

The `cd` command changes directories — the command line given assumes you set up your Eclipse workspace as originally directed, named your current project as specified, and imported the provided code as specified. If any of those steps didn't happen quite like that, you'll need to adjust the pathname.

If everything goes well, the `antlr` command completes quietly without any output displayed in the terminal. Use `ls` to view the contents of the directory — you should see something like the following:

```
ExprBaseListener.java  Expr.g4           ExprLexer.tokens   ExprVisitor.java
ExprBaseVisitor.java   Expr.interp       ExprListener.java
ExpressionApp.java     ExprLexer.interp  ExprParser.java
ExprEvalVisitor.java   ExprLexer.java    Expr.tokens
```

From the names, you can see that some of the files are part of the lexer and some are part of the parser. An application will use this code to build a parse tree for an input string. The "visitor" classes support working with the resulting parse tree.

The details of working with the generated parser are beyond the scope of this lab, so an application which uses the generated parser to read and evaluate numeric expressions has been provided.

8. Run the application:

   - First tell Eclipse to notice the newly generated code: Right-click on the top-level project directory in the Package Explorer and choose "Refresh". The generated files should now be visible in the project, and the errors in the provided code should have gone away.
   - Still in Eclipse, right-click on `ExpressionApp` and Run As→Java Application.
   - Try different (legal) expressions — the program will output a text representation of the parse tree and the result of evaluating the expression. (The parse tree is written in a nested-paren format where the root of a subtree

comes before the left and right subtrees, respectively. See if you can draw a picture of the tree.)

- Also try different (illegal) expressions. What happens?

Press Enter (without typing anything else) when prompted to enter an expression to quit the program.

9. Look at the main program (in `ExpressionApp.java`) to see how this all comes together. The core of the program is four steps: read a line of text, break it into tokens, build a parse tree, and finally evaluate the expression. The generated lexer and parse do the two middle steps. The last step is done by `ExprEvalVisitor`; keep reading to learn more about it.

With parse trees, and trees in general, a common thing to do is *traverse* the tree i.e. to go through each node of the tree in a systematic order, doing something for each node. The movement part of a traversal is always the same, but what is done at each node — printing it, adding up values, etc — depends on the task.

The visitor pattern defines a way to structure code so as to separate the common traversal part from the task-specific what-happens-for-each-node part, allowing the traversal part to be written just once (and, in this case, to be generated by ANTLR). The programmer then only has to define the task itself.

10. `ExprEvalVisitor` defines a visitor for evaluating the expression tree.

    - Look at the methods defined `ExprEvalVisitor`. They all have names of the form `visit`*`X`*, where *`X`* is the name of one of the non-terminals from the expression grammar — `Expression`, `Term`, and `Factor`. The parameter passed to each method is the parse tree node being visited.
    - Look at the body of `visitExpression`. Remember that the expression rule was the one involving `+` and `-` operators. How do you evaluate a `+` or `-` node in the parse tree? Go through the (two) children of the node, adding (or subtracting) their values. See if you can identify that happening in the code.
    - Look at the other two methods. `visitTerm` is similar to `visitExpression`, but what is going on in `visitFactor`? Can you connect each `if` statement to the different alternatives for the right side of the factor rule?