Assignment 2: Parsing

due: 2/20

In this assignment we will continue our work from assignment 1 to build a parser for MiniJava. The directory structure is going to be mostly the same as assignment 1 with a few additions.

- `syntaxtree` contains all of the different classes for nodes in our AST.
- `treedisplay` contains classes for displaying the syntax tree either in a GUI, or printing it out to the terminal.
- I've also included the generate, compile, and run scripts from the last assignment.

The `MJGrammar.java` in the `parse` directory contains a minimal set of definitions to parse `Test0.java` Your job will be to extend `MJGrammar.java` so that it can parse the entire MiniJava language and create the proper AST. The grammar for MiniJava is included in this file. You may need to modify parts of it to get it to work though.

There are some language constructs that are curiously missing from the syntaxtree directory

- if a class definition doesn't contains `extends`, Then you should assume it extends `Object`
- If an `if` statement doesn't contain an else, then you should add `else { }`
- All `for` loops should be translated into equivalent while loops
  example

  ```
  for ( a = 4 ; a <= 10 ; a++ ) { bodypart1() ; bodypart2() ; }
  ```

  should become

  ```
  { a = 4; while ( a <= 10 ) {{ bodypart1() ; bodypart2() ; } a++ ; } }
  ```

  If the middle clause is empty, it should be treated as `true`
- if an empty statement `;` is encountered treat it as an empty block `{ }`
- If a `x++;` is encountered treat it as `x = x + 1;`

- We don't have classes for `!=, >=, <=`, so treat them as `!(x == y)`, `!(x < y)`, and `!(x > y)` respectively.
- The expressions `-x` and `+x` should be treated as `0-x` and `0+x` respectively.
- If a method call does not have an explicit object, then assume the object is `this`.
- Treat all character literals as if they were integers with their ASCII value.

The setup for running this project is largely the same as assignment 1. However, the main program is `main/Main2.java`. This program handles a few optional arguments.

- -p will print a textual representation of the AST
- -pp will pretty print the program that it parsed.
- -w will display a GUI with a representation of the AST.

To display the GUI version you can run the following line where `"..."` is the contains the path to wranglr.jar

```
java -cp "..." main.Main2 -w testfile.java
```

I'm going to strongly recommend not attempting to do the entire grammar at once. Break it down into smaller pieces and get those working.

- pick a few grammar rules, such as simple expression with no operators, and get those working.
- make sure the rules are working before you add the semantic actions
- continue until your entire grammar is implemented
- Now, go back and add the semantic actions.

Breaking the tasks down like this will help with debugging, but it's also important because we have 60+ AST nodes. Its very difficult, if not impossible, to keep track of all of that while your working. Instead working on a few nodes at a time will allow you to build up the grammar without your head exploding. You will still get partial credit if your grammar parses correctly, even if it doesn't produce the correct syntax tree.

Extensions

The base assignment is worth 89%. You can extend the assignment with any of the following.

extension 1: +5%: Recognize Java's do-while statments. Since we don't have a DoWhile node, we need to produce a while node with the same meaning

```
do {
    x = x * 2;
    y = y - x;
} while(y < 0);
```

should become

```
while(true) {
    x = x * 2;
    y = y - x;
    if(!(y < 0)) break;
}
```

extension 2: +10%: Recognize Java's switch statement. There are Switch, Case, and Default classes in syntaxtree. Case and Default act as normal statements, so this example would contain 14 statements in the body.

```
switch(n) {
  case 1: x++; break;
  case 2: case 3: y++; break;
  default: x--; y--; break;
  case 100: x = 0; break;
}
```

Using the Lexer

The lexer from assignment 1 is already implemented for you to use. The .class files are in the lib directory. You can copy them over to the src/parse directory. There's no need to run wrangLR. The lexer files work in an odd way. The details aren't very important, but instead of recognizing the tokens directly, we strip all whitespace out and translate the tokens into a canonicalized form. This helps with compilation times for the MJGrammar.java file. It also means you can't use the lib directory as a solution to assignment 1.

You can actually copy your definition of the tokens into MJGrammar and it will work fine, but I recommend you avoid doing that for this assignment. On the off chance that your lexer has a bug, or just works differently than mine.

Grading and Handin

Currently your parser works a class defined like this.

```
class Def {
    public void exec() {
        int a = 10;
    }
}
```

This is a minimal program for MiniJava, and it is important that your parser continues to work for this program. If your parser can't compile this, then I can't test it, which is not good for your grade.

Turn in just your MJGrammar.java file to moodle. Please do not change the name of the program.