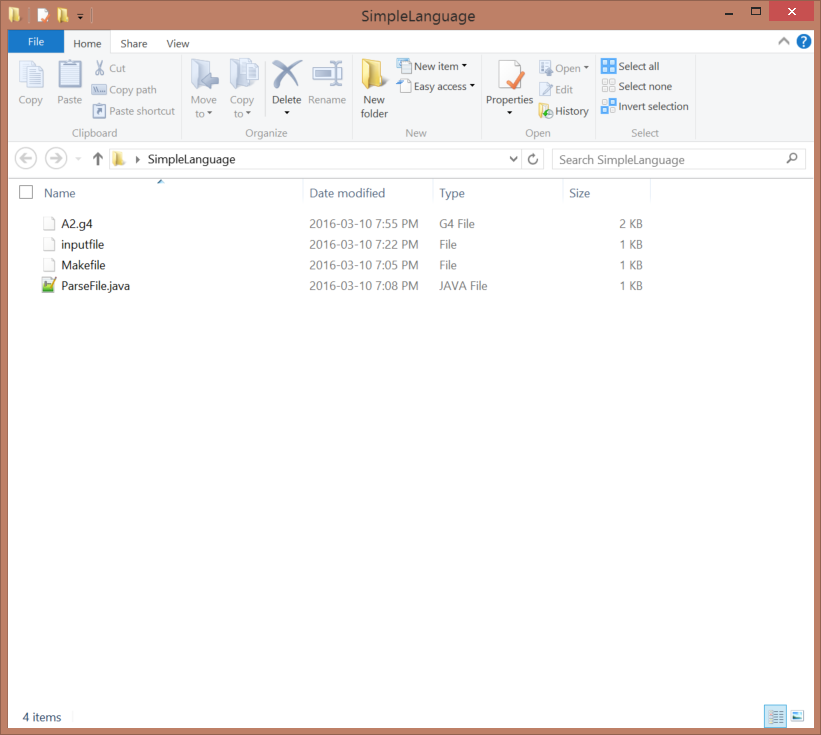
I have taken my own code for a similar assignment in the past and repurposed it to match the file structure of your assignment.

Grab the code located at <https://github.com/adelehedrick/Compilers2016/tree/master/SimpleLanguage>



Files:

* A2.g4 (this is the grammar with actions)
* Inputfile
* Makefile (this is for Windows, linux users will need to modify it)
* ParseFile.java (creates objects from the classes that Antlr creates from the grammar A2.g4)

## Inputfile

SimpleLanguage can only:

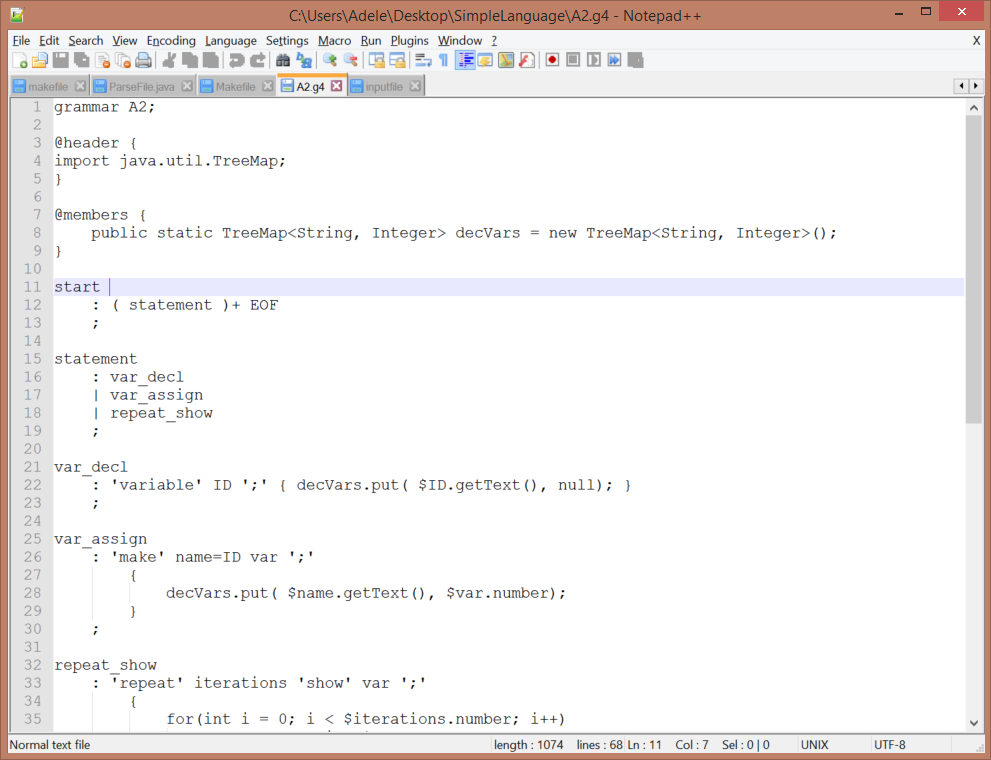
* Declare variables (line 1 and 2)
* Assign values to variables (line 3 and 5)
* Repeatably output values (line 4 and 6)

## ParseFile.java

This takes one command line argument, consisting of the filename of the input file, creates lexer and parser objects and runs the input on them.

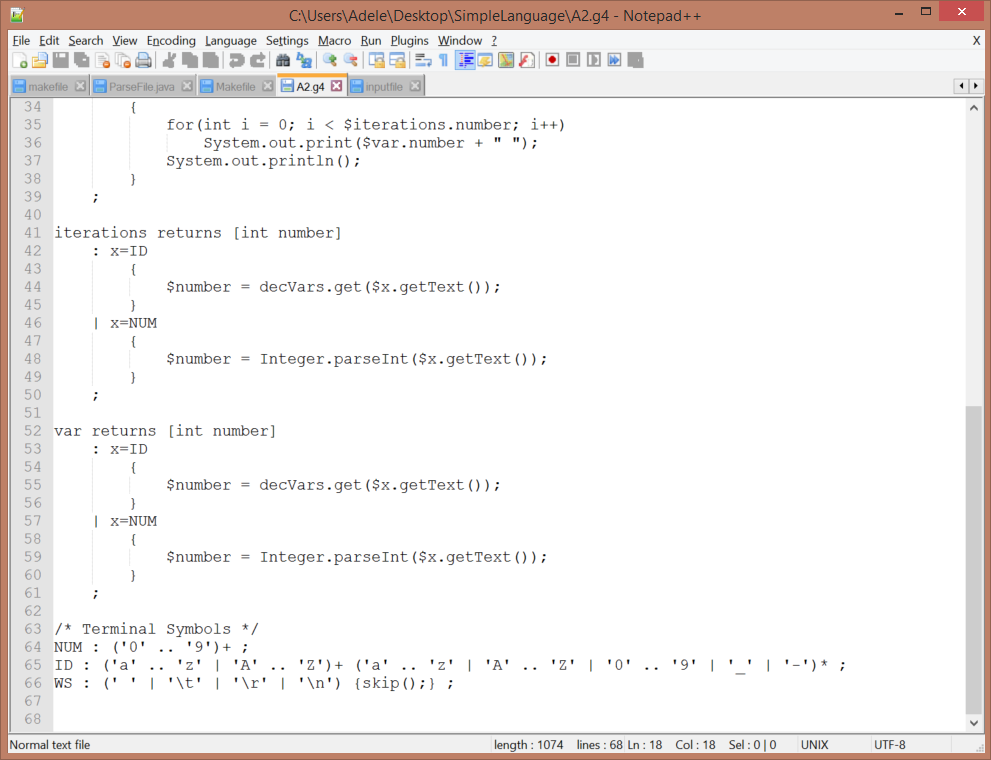
## A2.g4

Our grammar needs to be really smart, and by smart I mean it needs to DO THINGS, and remember variable names and values assigned to them. So using the magic of @header and @members we can embed a means of storing variable names and values for those variables in the parser. Back in 2011, the means I chose to go with was using a [TreeMap](https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html), which allows you to store key (strings) and values (in my case integers) into a dictionary like data structure.



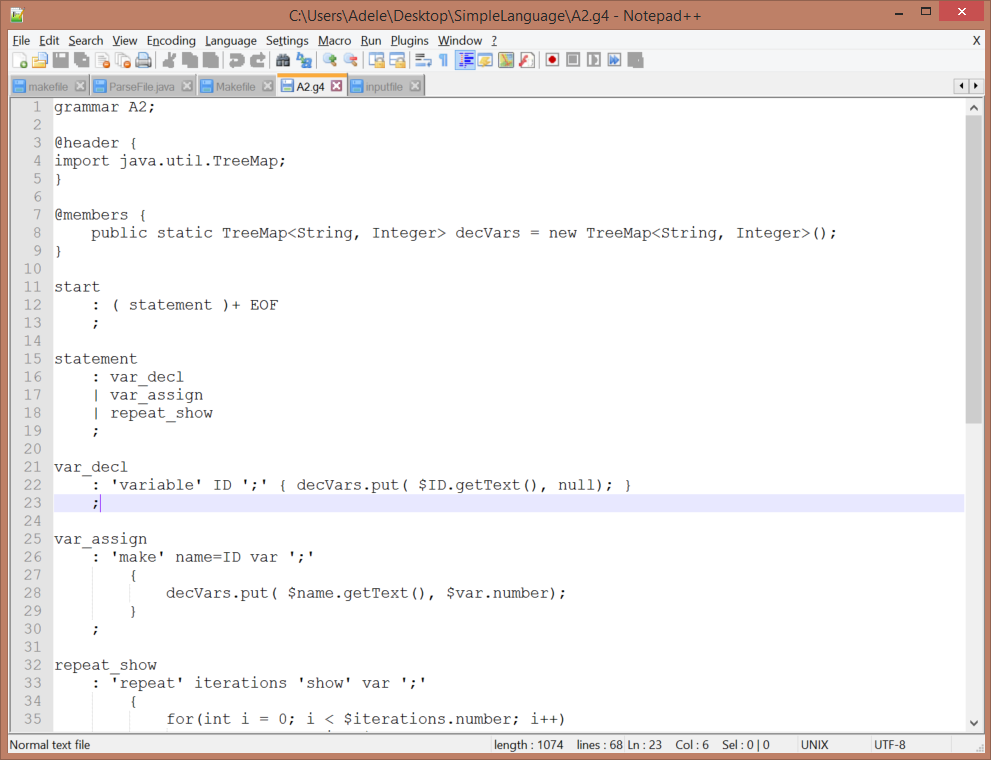
Now anywhere that you see “decVars” (which in my mind means “declared variables”), you can feel safe knowing that this object has been created and that it is in the comfort of our parser’s class.

At the very bottom of the file you will find the terminal symbols. There are more embedded in the grammar such as; ‘variable’, ‘make’, ‘repeat’, but they don’t need to be declared individually as they are a part of a production rule and never seen by themselves.



**Start** is the start symbol, and it consists of one or more **statements** followed by the end of file (EOF)

There are three types of **statements**; variable declaration, variable assignment, and repeat show



Now we get into the cool stuff… lights, camera, ACTION!!

Let’s look at the first action, *var\_decl*, on line 21 to 26:

var\_decl

: 'variable' ID ';'

{

decVars.put( $ID.getText(), null);

}

;

When a non literal is used in the production rule, it can then be used within the action (but a $ gets precedes it). In this action, we are putting a key/value pair into our decVars TreeMap. The key is the string ID and the value is null since we haven’t assigned anything to the variable yet.

We can look at the next rule for *var\_assign:*

var\_assign

: 'make' name=ID var ';'

{

decVars.put( $name.getText(), $var.number);

}

;

For some reason, I didn’t want to use the variable $ID within my action, so I then declared the variable name which is of type ID. var will be defined later on, but it basically returns a number that could be an integer or the integer value of a variable that is previously defined.

Repeat\_show is a fun rule and is a simple example of how you can create loops!

repeat\_show

: 'repeat' iterations=var 'show' x=var ';'

{

for(int i = 0; i < $iterations.number; i++)

System.out.print($x.number + " ");

System.out.println();

}

;

Now you are probably starting to get a little worried about these .numbers that you are seeing, but never fear, they are what is returned by the production of the rule they are attached to. The var rule has a return value called number!

Var returns [int number]

: x=ID

{

$number = decVars.get($x.getText());

}

| x=NUM

{

$number = Integer.parseInt($x.getText());

}

;

So var returns a number that could have literally been a number (which would be a string that has to be parsed as an integer) or it could be a variable that has previously been defined and stored in decVars. Either way, we will handle whatever type of number it is and literally return an integer number.