SubC Compiler

Neil Majumdar, Rahul Krushan, Henry Trollinger, Caleb Nale

**Abstract:**

SubC is a language based off of the Pascal programming language, but with some aspects of the C Programming language. We created the language because we were familiar with compiling Pascal, but wanted some of the convenience provided by C. Like C, it is not object orientated, and is therefore without classes or similar data structures.

**Description:**

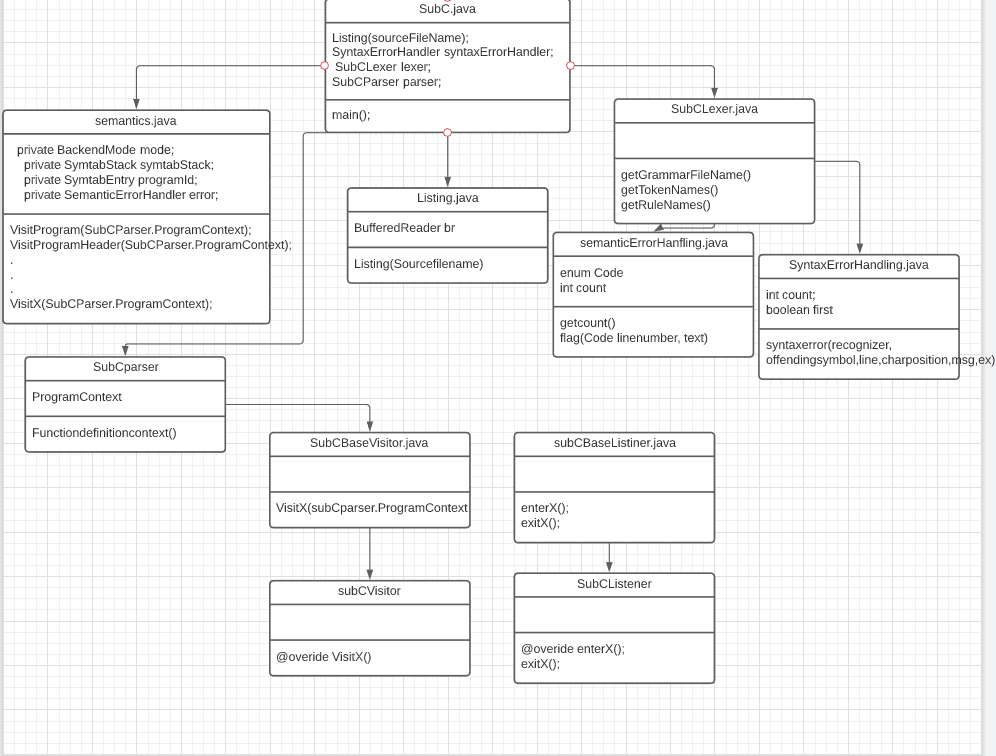
SubC shares many similarities with both C and Pascal. Our SubC files are labelled with .c, so our SubC files are stored as fileName.c. While we did not intend to make it this way, it shares many similarities with Small-C, another stripped down version of the C language. One major difference is that Small-C was stripped down to work on smaller computers, while ours is stripped down because we do not have the time or knowledge to fully replicate all of the C language. It currently supports four data types: Integers, Chars, Doubles, and Strings. It also supports If, switch, while, and for statements. In addition, functions passed by variables are allowed.

The compiler itself is also very similar to what we have worked with over the semester. For the most part, we were able to keep the code for the frontend and intermediate tier. We just had to change the code to reflect the change in our grammar. A few files, notably the antlr ones, we had to completely recreate for our new grammar.

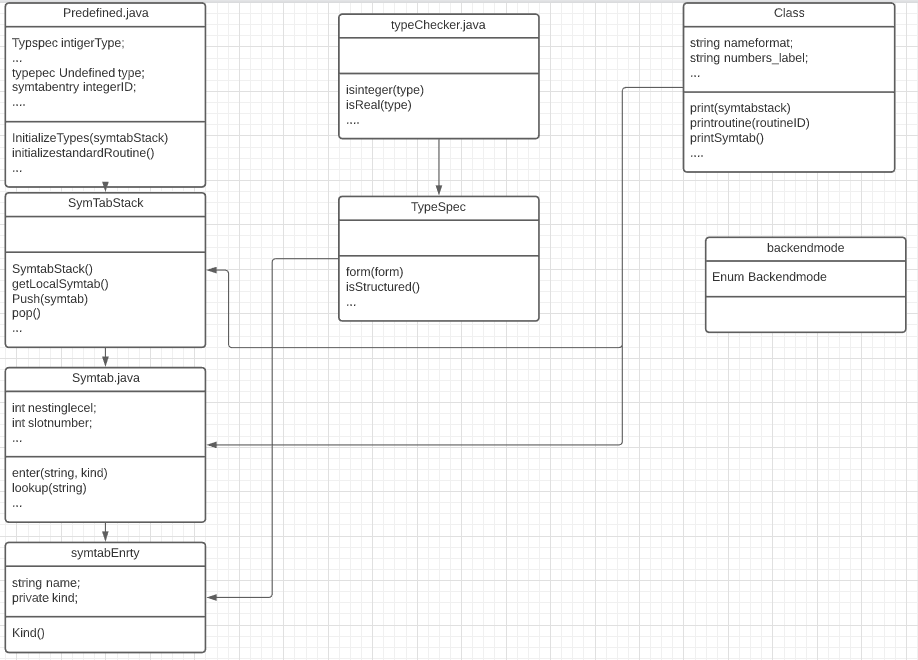
Brief disclaimer: we intended on getting the Mandelbrot algorithm to work in our language, but we ran into some issues getting arrays to work, which was a requirement for the mandelbrot program.

**UML Diagrams:**

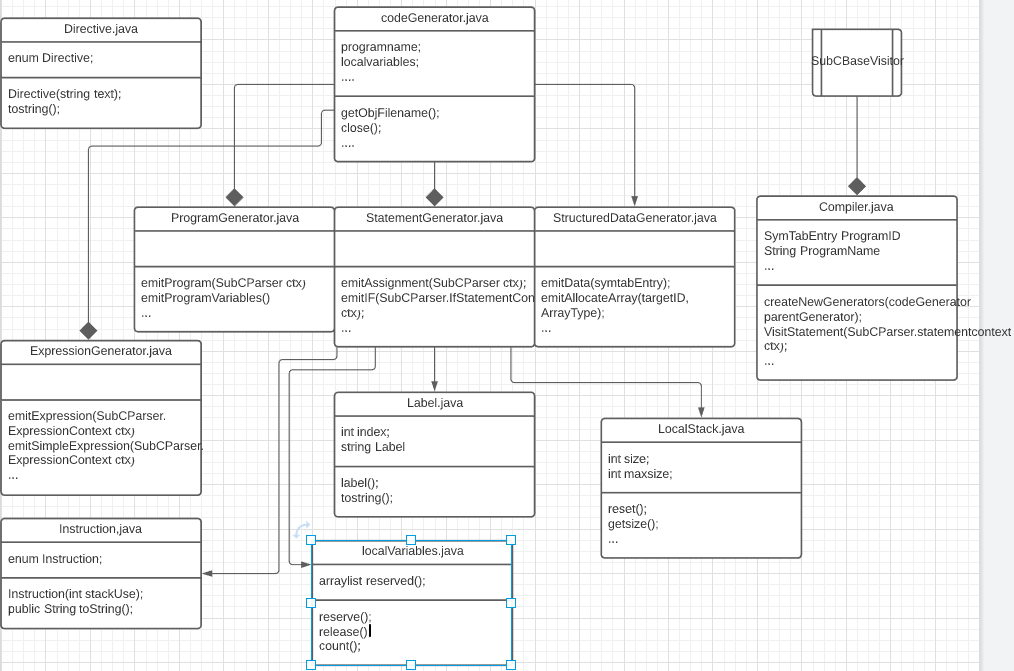
The following diagram displays the frontend and files generated by antlr4. Not all detail is contained in them, as many of these have too many functions to reasonably list in a diagram of this size. The important ones and their relations are listed. The frontend's job is to look for errors in the code. If an error occurs,semantically or in syntax, the program will fail to compile and give an error.



The next UML diagram is for the intermediate tier. The intermediate tier goes through the code again, this time putting it into s symbol table. Some more checking is done at this time, specifically type checking.



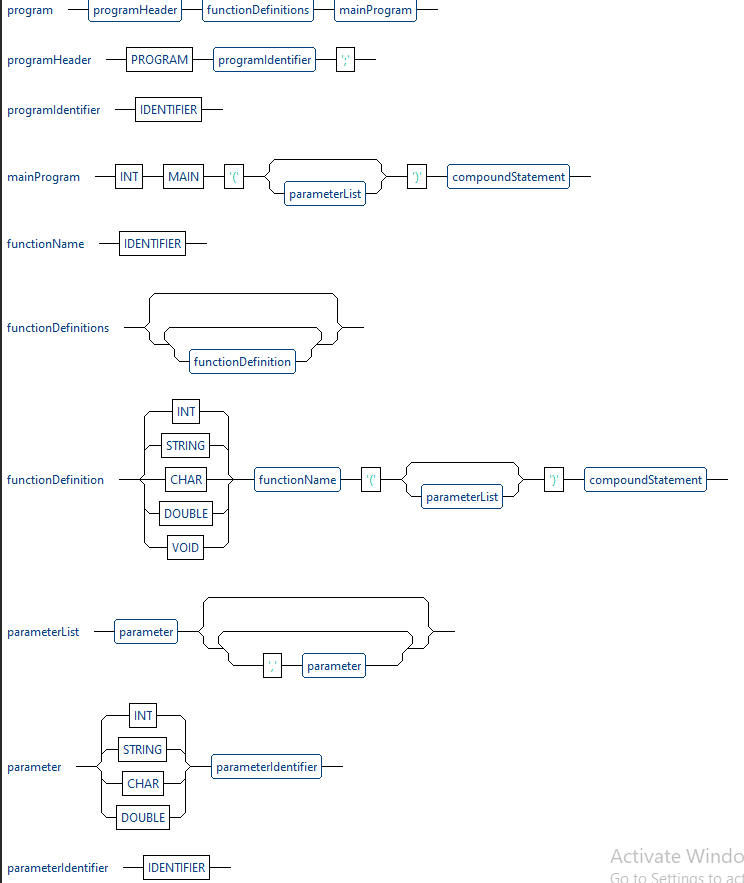
Finally, the backend is where most of the compiler's work is done. Here, we go through the symbol table created in the intermediate tier, and translate that into jasmin code. We also have classes to help keep track of variables used in jasmin code, as well as the stack. The backend uses a lot of classes from the Intermediate tier, but they are not displayed here to save space, as they are already showcased in the previous diagram.

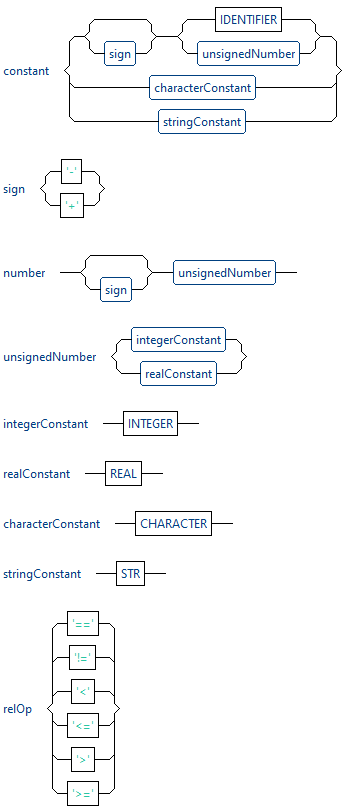
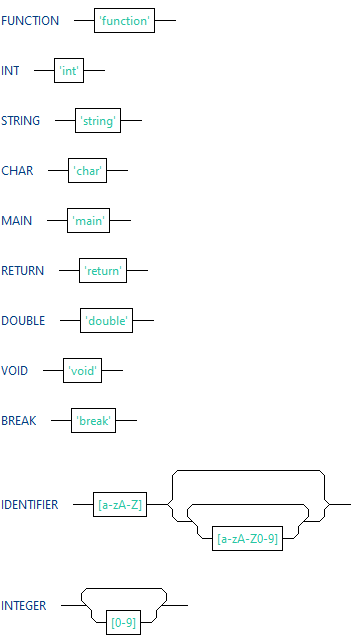
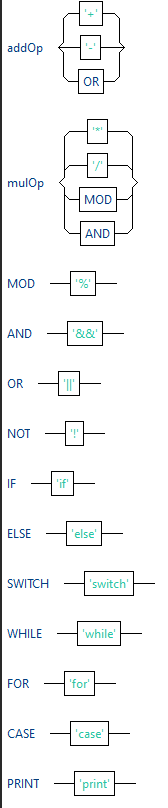
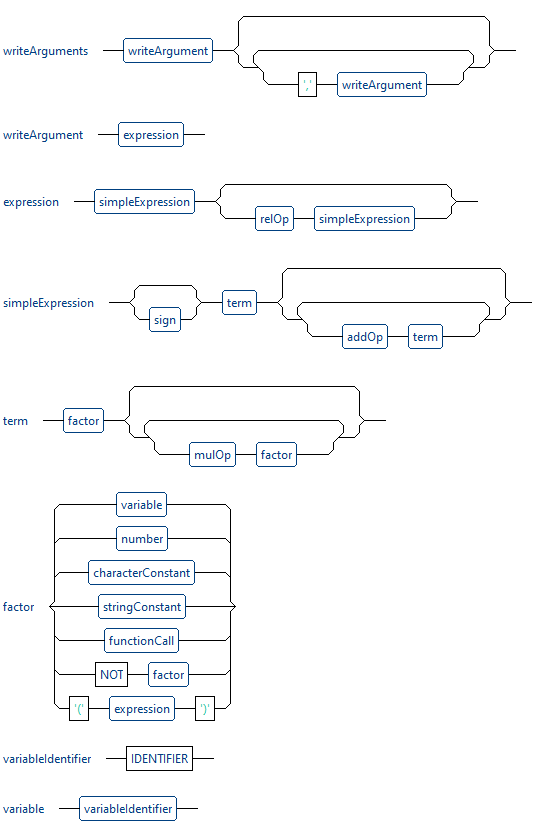
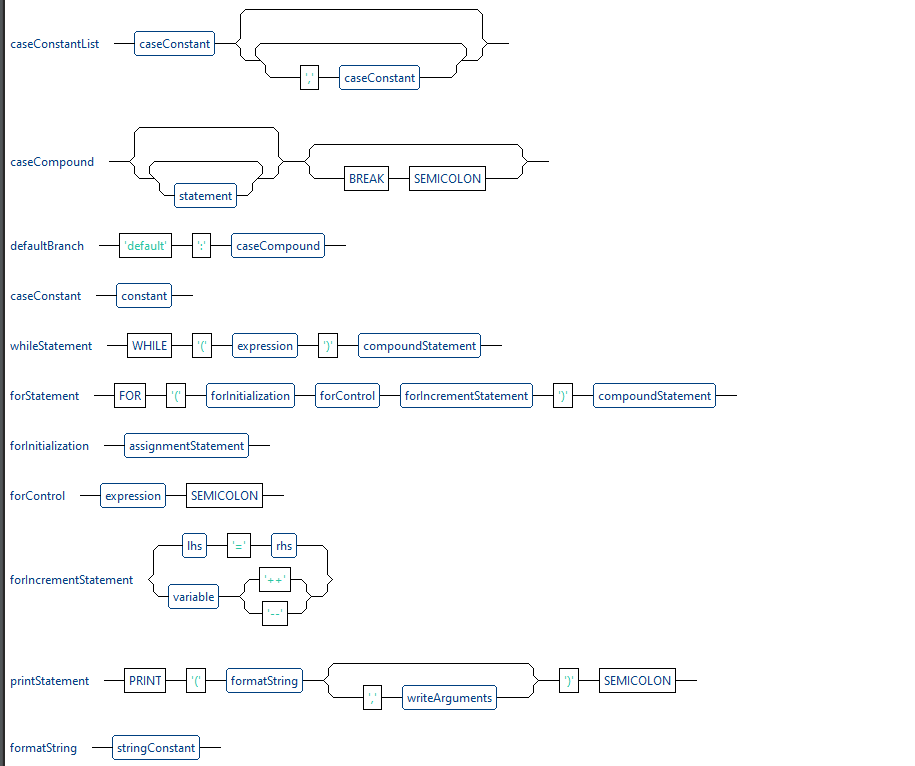
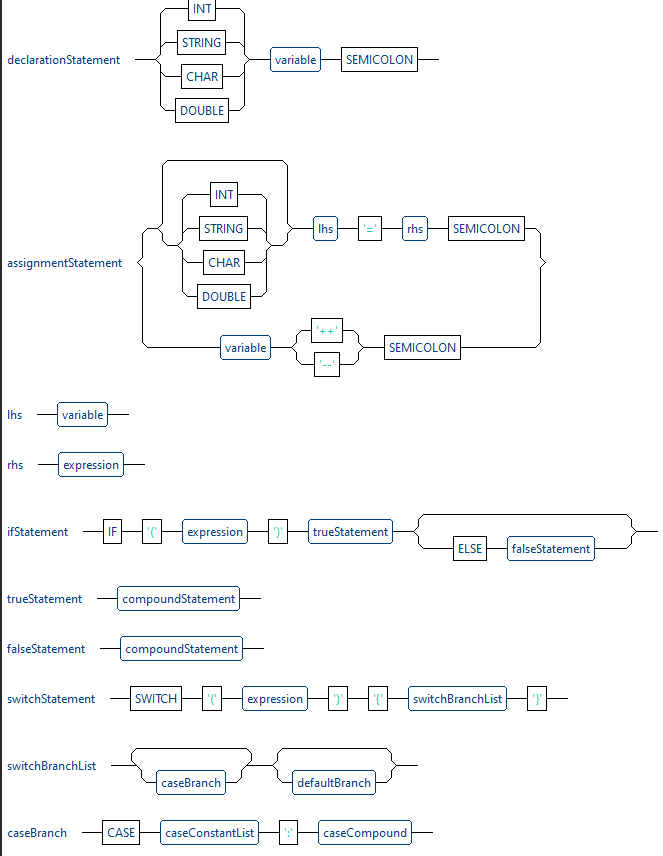
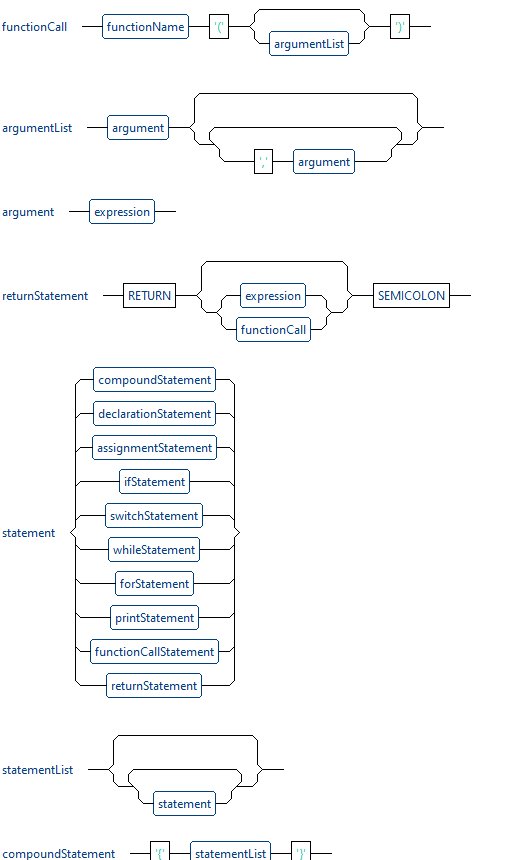


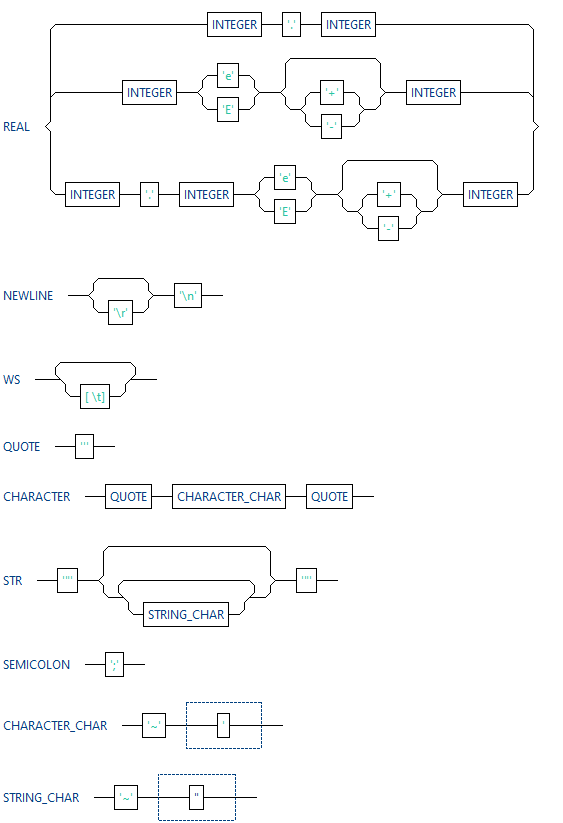
**Grammar:**

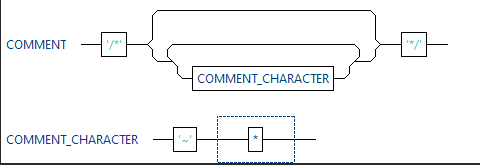
Our grammar is very similar to Pascal, but with some key differences, for example syntax for how a program would output data is a bit different. Instead of writing, we use print. Case statements are allowed multiple constants per case, function declaration must be followed by definition, all the global variables are declared in main, and you cannot pass by reference, only values. Additionally, all statements are considered as compound statements, even if they have only one variable. Below is the full grammar in syntax diagrams.

**Syntax Diagtams:**

****

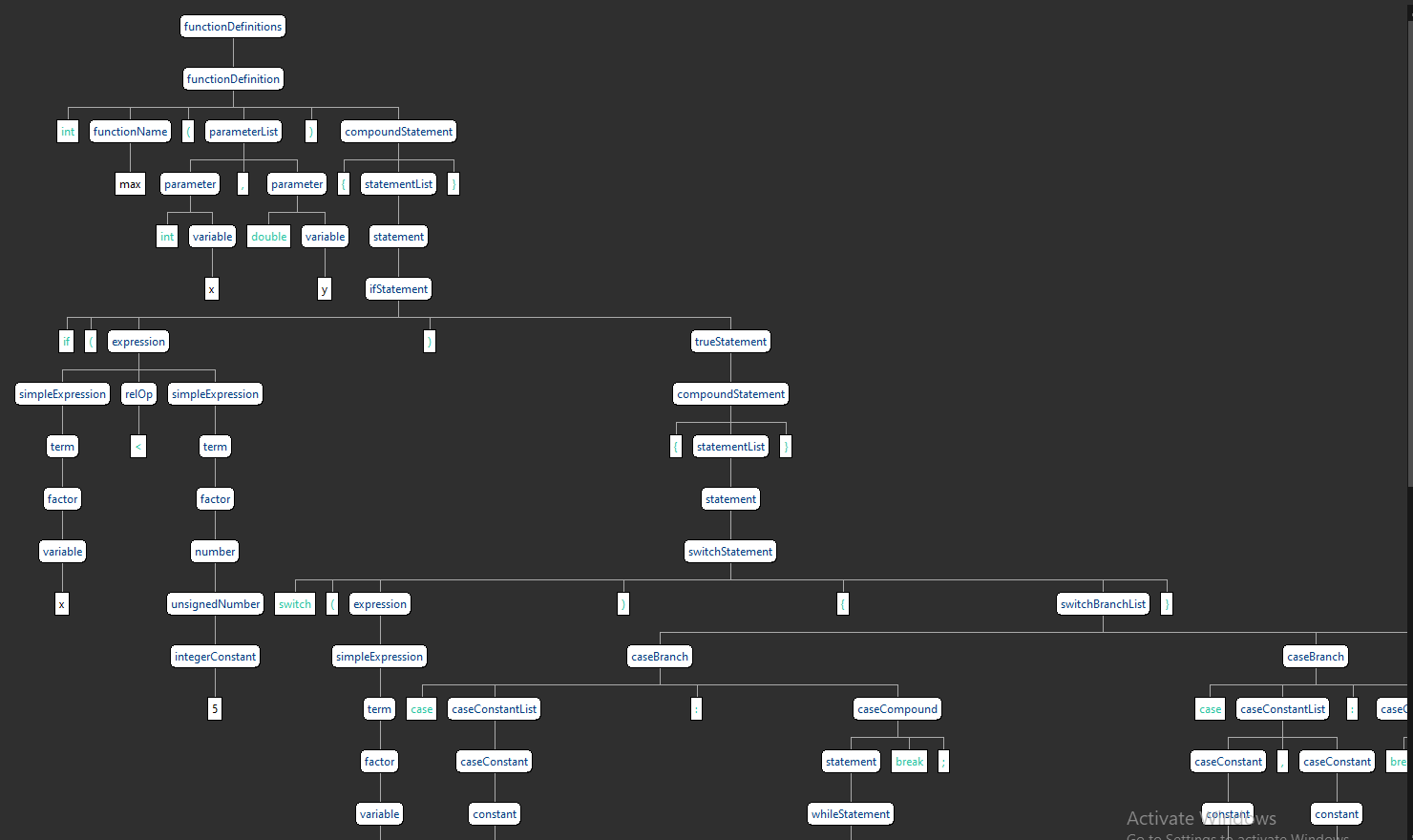
****

****

****

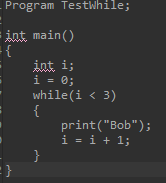
**Parse Tree example:**

The following parse tree is an example of what a switch statement looks like in our language.

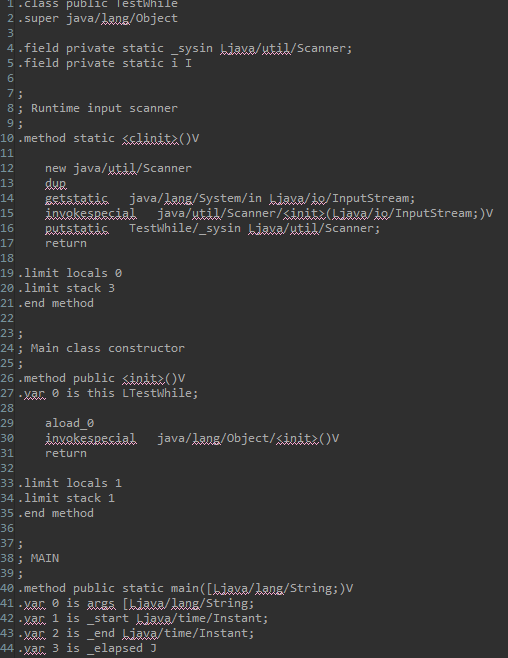
****

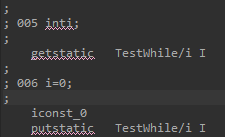
**Code Template:**

Let's take a while loop as an example for what jasmin code is generated by our compiler.Here is the TestWhile code in its entirety:



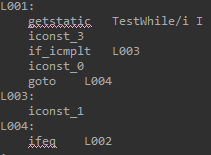
First off in our jasmin code, we have some code that we use in all our files, that does things such as setting up the stack, setting up a timer,naming the program, and set up the main function.



Up next, we have the code for the first two unique lines of code: “int i;” and “i = 0;”

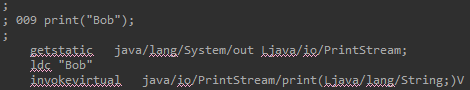
As you can see, we use getstatic to set up the variable, load the constant zero onto the stack, then put zero into I.

Next, we get into the while loop



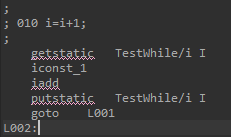
First, we put the current value of I on the stack, as well as the constant to compare it to, 3. We then compare them with if\_icmplt, which jumps to label 3 if I is less than 1. From there, we load 1 onto the stack. Otherwise, we load zero onto the stack. After label 4, we compare the values of I to three. If they are equal at this point, we jump to label two, which is outside the loop.

Otherwise, we move on to the print statement.



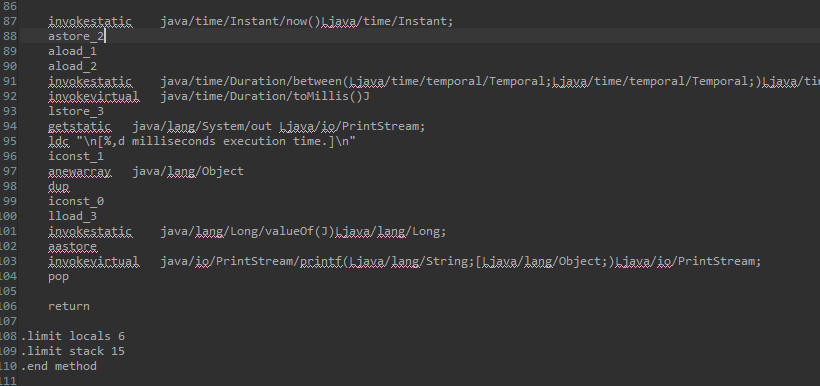
Here we get the print command loaded, then load the statement we want to print, in this case “Bob”. Finally, we invoke the java command to print.

Next up, we increment the value of I, so we can loop properly, then loop back to the first label

.

Notice that below that is label two, which indicates the code for the loop is over, as label two is the label we jump to earlier when I equals 3.

After that is just more boilerplate code that prints out the time it took for the program to run, and ends the program.

**Instructions to run:**

Nothing special is required to run our programs, just use the usual method of converting a file into a jasmin file and executing it. Just specify which file you want to compile in the run options of Eclipse, for example “-compile test.c”. You then should have the jasmin code. The following are the programs that we have included as examples to be run: TestFunction,c testC.C and TestShape.c. The latter is our nontrivial example.