Compiler Report

In addition to your code, you should also submit a report describing how you have implemented the compiler.

This report should explain the overall structure of your code. You should explain in more detail the parts of the code that you have altered and how these relate to the difference between the assessment language and the language used in the labs. If certain features have not been implemented or do not function correctly, you should acknowledge this.

You should also include a section in the report covering the testing that you have performed to check the various components. You should include all programs that you compiled to test your code, either as files or as an appendix to your report. I will not be providing any test programs – it is part of the assessment for you to be able to read what is an acceptable input in the language and to write short programs that meet this definition.

Your report should be no more than 3000 words. This does not include any words in tables, figures, code snippets, appendices etc. If you can cover everything in fewer words, there is no need to pad the report out. I would recommend however that you try to write around 500 words about each of the components you have implemented; if you write significantly less, you are unlikely to have described all the changes required in sufficient detail.

IS IT OK TO USE THE SAME COMMENTS YOU USED IN THE LAB FILES?

Explain briefly what the files that were added from the lab do and then explain in more detail what was changed in the files/grammar from the lab to the assessment. (And new files/methods)

# 1 Scanner

The first action the scanner will need to do is to read in the source code from a file, so I pulled in the FileReader, IFileReader and Position files from the labs; these allow the scanner to move through the file character by character while keeping a current position which is useful for error reporting. I also pulled the ErrorReporter file from the labs so the compiler can display helpful errors when compiling to help in debugging, and the Debugger file which allows you to write debugging logs when the DEBUG variable is set to true. I then pulled in the main class file, Compiler, from the labs and commented out the unnecessary code at this time: the parser, identifier, type checker, code generator, writer and the output file arguments. The Compiler class is what performs the compilation process and writes a message reporting on the success of the compilation.

With these files added I could start on the tokenization of the source code now, which is where the lab language and the assessment language starts to differ. I first pulled in all the Tokenization files from the lab into the solution, Token, TokenType and Tokenizer. Both the TokenType and Tokenizer files need to be altered to fit the new grammar in the assessment language.

To make the token types I went through the language specification and noted down all the appropriate terminals and non-terminals in order of appearance …

# 2 Parser

* Had to add the nothing token and then accept it in the parsing blank command (SingleCommand)
* Uses Comma instead of semicolon inbetween commands and declarations while using comma to separate the for command options

# 3 Semantic Analyser

# 4 Code Generator

# 5 Testing