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 the writing of 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, target code 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 started on the tokenization of the source code, 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. The Token class holds all of the information that each token needs, token type, spelling and position. The TokenType class holds the types of tokens that are in the source language including non-terminals, terminal reserved words, terminal punctuation and special tokens, and it has a mapping of keywords in the language to the token type enums. The Tokenizer class is where the scanning and tokenizing of the source code happens to turn the code into tokens. Both the TokenType and Tokenizer files needed to be altered to fit the new grammar in the assessment language.

The first step I took was to update the TokenType file as there were different and new token types. I first removed the unneeded types, Const, Semicolon and Is, then I moved Becomes to the reserved words as this was the new assign token. After that, I added the new types Nothing, Forever, For, Comma and Tilde. Nothing is used for blank commands; Forever for use in the while command; For for the new for command; Comma for replacing Semicolon in sequential commands or declarations, and for separating the grammar in the for parameter; and finally Tilde for replacing the Is token in constant declaration. Lastly I updated Keywords dictionary to include the new token types and remove the old ones.

The first difference in the grammar was that the identifier must have a second letter or digit, instead of just needing a letter in the labs; for this I updated the ScanToken method in

The differences in the grammar are …

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 …

* Changed scanner identifier so that it must have a letter AND a letter or digit
* Character literal uses “” instead of ‘’ and needs a graphic
* Graphic is either a letter or digit or operator or a single space as opposed to being any character like it was in the lab
* Comment is now $ instead of !
* TokenTypes are ordered in the way they show up in the grammer
* Added Tokens: Nothing, Forever, For, Comma, Tilde.
* Got rid of Tokens: Const, Semicolon, Is
* Changed Becomes to a Terminal reserved word tokentype
* Updated Keywords ImmutableDictionary to incompass the new TokenTypes and get rid of old ones.

# 2 Parser

* Added Parser file
* Had to add the nothing token(tokenizer) and then accept it in the parsing blank command method (SingleCommand)
* Uses Comma instead of semicolon in-between commands and declarations, also using comma to separate the for command options
* Updated single command identifier so that it uses ‘becomes’ then expression instead of ‘:=’.
* Updated while terminal to include the forever terminal when parsing
* Added for terminal
* Single declaration was changed to just identifier then, Tilde and expression, or Colon and type-denoter.
* Primary expression was changed so that the identifier can have brackets and parameter.

Abstract syntax tree nodes

* Added TreePrinter in IO
* Added Nodes files
  + Not needed:
  + Changed:
  + Added:
    - WhileForeverCommandNode (added new case to TreePrinter ToString)
    - ForCommandNode (added new case to TreePrinter ToString)
    - CallExpressionNode (added new case to TreePrinter ToString)
* (Temp is for IRuntimeEntity which is used for declaration nodes and error node)
* Added CodeGeneration: empty IRuntimeEntity and TriangleAbstractMachine classes, adding an empty Primative, Type enums and TypeSize Dictionary for it to be used for declaration nodes.

# 3 Semantic Analyser

* Added SemanticAnalysis files
* Added full TriangleAbstractMachine with the address at the bottom commented out from labs
* DeclarationIdentifier:
  + Changed:
  + Added:
    - PerformIdentificationOnWhileForeverCommand
    - PerformIdentificationOnForCommand
    - PerformIdentificationOnCallExpression
* TypeChecker:
  + Changed:
  + Added:
    - PerformTypeCheckingOnWhileForeverCommand
    - PerformTypeCheckingOnForCommand
    - PerformTypeCheckingOnCallExpression (Based it off CallCommand but added extra check for return value and set the expression type to function return type)
* StandardEnvironment:

# 4 Code Generator

Test Bracket Expression

* Added TargetCodeWriter to IO
* Added Address, CodeGenerator, Instruction, IRuntimeEntity, RuntimeKnownConstant, RuntimeUnknownConstant, RuntimeVariable, ScopeSizeRecorder, TargetCode to CodeGeneration
* Uncommented Address in TriangleAbstractMachine
* CodeGenerator:
  + Added:
    - GenerateCodeForWhileForeverCommand
    - GenerateCodeForForCommand
    - GenerateCodeForCallExpression (Maybe not returning value?)
* Added to Comiler for Generating code and adding in 2 extra parameters for output files
* Added out.tam and out.txt to arguments in project properties
* Does the StandardEnvir functions work?

# 5 Testing

* Test all commands
* Test all Declarations
* Test all Expressions
* Test all Parameters
* Test all types
* Test Char & Int Literals

Test\_program1:

* Let
* Sequential Declaration
  + Const Declaration
  + Integer and Char Var Declaration
* Sequential Command
  + Begin
  + Procedures (Call Commands)
    - put with expression parameter - Testing chars (including single space, operator and letters)
    - get with var parameter to assign a character to a declaration
    - puteol with blank parameter
    - put with ID expression parameter
    - getint with var parameter
  + If command with binary expression using ID Expression and > operator (and another using < operator)
    - While command with binary expression
      * Putint procedure with ID expression
      * Assign command for var declaration identifier
  + Nothing command (BlankCommand)
  + Call Expression (chr function)
  + For command
  + Let in let command
    - Declaring with same identifier in different scope
    - While forever command doing Fibonacci sequence
      * Unary expression in if command with bracket expression

Split up other test programs into smaller ones that have errors in them to test error reporting.