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 next step was to update the Tokenizer file. The first difference in the grammar for the scanner was that the identifier must have a second letter or digit, instead of just needing one letter in the labs; for this, I updated the ScanToken method to include a check for the second character in the identifier section to make sure it was either a letter or digit. Next, I changed the Semicolon check to a Comma check, then removed the assign (=) check in the colon check section. The next big difference was in the character literal grammar as the apostrophe in the labs was changed to a quotation mark for denoting a character, and the allowed graphic in the quotations was limited to either a letter or digit, operator, or a single space; this was all taken care of in the character literal section of the ScanToken method. Lastly the comment denoter was changed from an exclamation mark to a dollar sign; this was done in the SkipSeperators method.

Error reporting was done for tokens that appeared as error token types and in more detail for identifier and character literal tokenizing.

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

I first decided to pull only the Parser file into the solution from the labs which allowed me to first update the differences in the parsing grammar code before adding complexity with the tree nodes, though I will explain both at once below. The Parser checks that the syntax of the program is correct and creates a tree showing the structure of the program. I decided to work and order the parsing methods in the order that the grammar appeared in the language specification. To create the tree structure, nodes were used for representing an abstract syntax tree; these were all pulled from the labs. The TreePrinter file was also added to allow for the node tree to be printed once the parsing was complete. To allow the parsing to run the IRuntimeEntity and TriangleAbstractMachine classes were needed, which at this point were empty apart from three declarations in the TriangleAbstractMachine.

Working through the methods of the Parser I first updated the ParseCommand method so that it uses Comma instead of Semicolon for sequential commands. Next, I changed the ParseSkipCommand method to be called ParseBlankCommand and updated it to accept a Nothing token before returning. I then updated the ParseWhileCommand so that it would include the new Forever grammar by adding a check for if the current token was forever then it would just accept Forever and Do then parse a single command, and then renamed the method to ParseWhileOrForeverCommand. Since the while forever command did not need an expression I had to create a new node called WhileForeverCommandNode. To allow for the new node to be printed I also had to update the TreePrinter file ToString method, adding in a new case for the new node. To parse the new for command I created a new method called ParseForCommand which handles the accepting of tokens and parsing for the commands and expression according to the grammar. This also meant that I had to create a new node called ForCommandNode to hold the commands and expression, and like above created a new case in the TreePrinter ToString method for the new node to allow for printing. Once the new/changed methods were created/updated I then went back to the ParseSingleCommand method and updated the switch statement to add a case for Nothing and For, updated the While case to the new name of the parse while command method and changed the default case to report an error and return an error node.

The next section I had to adjust was the declarations. I updated the ParseDeclaration method to use Comma instead of Semicolon for sequential declarations then had to update both Const and Var declarations due to there no longer being a need for the Const and Var terminals in the assessment language grammar. I got rid of the ParseConstDeclaration and ParseVarDeclaration methods to avoid duplication of the position and identifier code and created and returned the Const and Var declaration nodes in the ParseSingleDeclaration method’s switch case. I also changed the switch cases and the token accepting inside them to Tilde and Colon.

The last change that needed to be done was in the ParsePrimaryExpression method to allow for a call expression which is new in the assessment grammar. To do this I had to add a check for a left bracket into the switch case for Identifier token type; I then accepted the brackets, parsed the parameter, and created and returned a new node. I created a new node for this expression called CallExpressionNode and updated the TreePrinter ToString method accordingly.

# 3 Semantic Analyser

To start off I first pulled in the four semantic analyser files from the labs, SymbolTable, StandardEnvironment, DeclarationIdentifier and TypeChecker. The SymbolTable is used in the identification stage for storing and retrieving identifiers that are known about to allow for quick look up and avoid repeating work; it also allows for different scopes through pushing and popping dictionary scopes from a stack. The StandardEnvironment holds all the global identifiers that need to be added to the symbol table at the global level so that they will always be in scope. The DeclarationIdentifier is the identification part of the compiler where it matches identifiers to their declarations; it discovers if there are any problems with out-of-scope variables or missing declarations. The TypeChecker carries out type checking for nodes in the compiler to make sure that the types of items such as commands and operands are appropriate for the context they appear in. The TriangleAbstractMachine file was also updated to include everything from the lab, expect for the CodeBase address, to allow the semantic analyser to work. To fit the new grammar, it was the DeclarationIdentifier and TypeChecker files that needed to be changed.

The only new change that needed to be added to the DeclarationIdentifier was the identification on the new nodes which I had added in the parsing stage, WhileForeverCommandNode, ForCommandNode and the CallExpressionNode. The method added for the while forever command was called PerformIdentificationOnWhileForeverCommand, and it performs identification on the WhileForeverCommandNode command. The next method was the PerformIdentificationOnForCommand for the identification of a for command node which performs identification on the node’s commands and expression. The last method added was the PerformIdentificationOnCallExpression method for identifying a call expression node; this performs the identification on the call expression’s identifier and parameter child nodes.

Similar to the changes above in the identification stage the only new change in the TypeChecker file was the addition of three new methods, PerformTypeCheckingOnWhileForeverCommand, PerformTypeCheckingOnForCommand and PerformTypeCheckingOnCallExpression. The type checking for the while forever command and for command was fairly simple, just needing to call PerformTypeChecking on the command child of the while forever command, and the expression and command children of the for command; though there was also a need to make sure that the type of the expression child in the for command was boolean. The call expression type checking was more complex as once I had done the type checking on the children nodes, I had to check that the identifier was a function and then check the arguments of the function to make sure that it had the correct number and type, making sure to also set a type for the call expression as well using the function’s return type.

* 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

To start off I added the Code Generation files, Address, Instruction, IRuntimeEntity, RuntimeKnownConstant, RuntimeUnknownConstant, RuntimeVariable, ScopeSizeRecorder, TargetCode and CodeGenerator.

* 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

GO THROUGH FILES TO CLEAN UP AND ADD COMMENTS

# 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.