**Declaration on Plagiarism**

**Assignment Submission Form**

This form must be filled in and completed by the student(s) submitting an assignment

Name(s): Jaime Kirk

Programme: Compiler Construction

Module Code: CA4003

Assignment Title: A Lexical and Syntax Analyser

Submission Date: 16/11/2020

Module Coordinator: David Sinclair

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I/We have read and understood the Assignment Regulations. I/We have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

I/We have read and understood the referencing guidelines found at

<http://www.dcu.ie/info/regulations/plagiarism.shtml>, <https://www4.dcu.ie/students/az/plagiarism>

and/or recommended in the assignment guidelines.

Name(s): Jaime Kirk\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date: \_\_15/11/2020\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

This assignment was to implement a lexical and syntax analyser using Antlr4 for a simple language called CAL. The parser has to take a file, e.g. test.cal, as input and output either “test.cal parsed successfully” or “test.cal has not parsed”.

The first step of implementing this was defining the grammar rules. These rules were tested in the terminal by running the command “grun assign prog -gui test.cal” which parsed the file and displayed a parse tree created using the grammar rules and displayed any errors which occurred.

The first grammar rule to define was the prog line which was “decl\_lst func\_lst main”. This rule is immediately called each time a file is parsed. “decl\_lst” defined as “decl\* | ;” is a list of all the declarations within a file. This includes variables which are defined by “Variable ID COLON type SEMI” so variables such as “variable i:integer;” will be picked up by this rule. An identifier, “ID”, is defined by “ID:Letter(Letter | NUMBER | Underscore)\*;” and type “type:Integer | Boolean | Void;”. Constants “Constant ID COLON type ASSIGN expr SEMI” are also included in the “decl\_lst”. An example of a constant is “constant five:integer := 5;”. The prog statement also includes “main” which is defined as “main begin decl\_lst stmblock End;”. This definition picks up the main part of the code. This must start with main begin then the “decl\_lst” for declarations and “stmblock” which picks up all the statements. An example statement is “i := 1;”. The main definition must finish with “end”.

A statement can be found in a number of different ways therefore it has multiple definitions. The first definition is “ID ASSIGN expr SEMI”. “ID LBR arglst RBR SEMI” is another definition which includes arglst which contains all the arguments in the statement. An example of this statement is “func();” where in this case the arglst is empty. The next possible statement is “Begin stmblock End” which must start with Begin and have a statement block in it. The next rule is to cover IF statements, “If cond Begin stmblock End Else Begin stmblock End”. The statement must begin with “if” then a condition followed by “begin” a statement block then “end” this is then followed by the else statement which starts with “else” then “begin” then another statement block which then is completed with “end”. While statements were defined next with the rule “While cond Begin stmblock End”. This rule is quite similar to the IF statement except there is no ELSE statement. The final rule defined for statements was a rule for the word “skip” which was defined “Skipping SEMI” where “Skipping: S K I P;” is used to make Skipping a reserved word.

Expressions are the next possible input that could be received that needed to be defined. The first rule is “frag bin\_arith\_op frag” which would pick up expressions such as “x + y”. A fragment is “x” and “y” and is defined by “frag: ID | MINUS ID | NUMBER | BOOL | LBR expr RBR | ID LBR arglst RBR;”. A “bin\_arith\_op” is defined by “bin\_arith\_op:ADD | MINUS;” so it can be “+” or “-”. The next definition is “ID LBR arglst RBR” includes a list of arguments so expressions such as “multiply (arg\_1, arg\_2);” with “arg\_1” and “arg\_2” being contained in the arg\_list. The final possible expression is that an expression can be a fragment.

Conditions were defined next. These are used when there are IF or WHILE statements. The first definition is “NEG cond” where “NEG: “~” ”. So a condition must have “~” before it in this case. The next definition is “LBR cond RBR” so a condition must have brackets around it. An example of this would be “(x < 0)”. “expr comp\_op expr” is the next definition which is a rule for when two expression are compared to each other so “x >= 0” would be accepted in this case. The final definition “cond (OR | AND) cond” is for when there are more than one condition and they must either both be true when the “AND: “&” ” operator or one of them true when “OR: “|”” is used. An example of this is “if y < 0 & x >= 0”.

To ensure that the language was case insensitive, I defined fragments for each letter of the alphabet e.g. “fragment A:'a' | 'A';” and then defined each of the reserved words “Variable: V A R I A B L E;” so that both upper and lower case characters would be accepted for each of the reserved words. All operators are easily defined by “NEG: '~';” which allows these symbols to be accepted and not cause errors when the file is being parsed.

Comments and white space are not to be included in the tree so they needed to be defined so that the parser would ignore them and move on. Comments were defined as “COMMENT:'/\*' (COMMENT|.)\*? '\*/' -> skip;” and “LINE\_COMMENT:'//' ~[\r\n]\* -> skip;” This will skip a comment if it appears in these formats. Examples of these are “// a simple comment” and “/\*This is a comment\*/”. White space is defined “[ \t\n\r]+ -> skip;” meaning any white space will be ignored like comments.

After the grammar was all defined and no errors were occurring when parsing files then java code was needed to run analyser. First is to check if a file has been provided so “if (args.length

* 0)” which if true then arg[0] is the input file. If an input file is given then using a try catch statement the file will try to parse the input file. Creates a new lexer object using “assignLexer lexer = new assignLexer (CharStreams.fromStream(is));”. Then I used “lexer.removeErrorListeners();” to remove existing listening then using lexer.addErrorListener(ThrowingErrorListener.INSTANCE); to use the one I had defined, ThrowingErrorListener which will throw a ParseCancellationException, so I can detect errors that occur. I used the antlr documentation to help write the ThrowingErrorListener file and adjusted my current code to work along with it. These statements are repeated for parser then a tree is created by “ParseTree tree = parser.prog();”. Then if the file is parsed then the code will print “inputFile + " wasS parsed successfully"” but if an exception is thrown then “inputFile + " has not parsed"” is printed.