University of Leeds **School of Computing**

COMP2932, 2019-2020

Compiler Design and Construction

Jack Compiler

Date: 14/05/20

1. Introduction

Three of the four phases of the Jack complier were attempted. I implemented the lexical analyser, the parser, the symbol table, and partially have implemented the semantic analyser. I followed the plan and met the milestones for submission on time for the lexer and parser phases. However, for the symbol table phase I did not meet the milestone with no individual submission made.

To develop the Jack compiler, I used C++ and developed the program in the CLion IDE on CentOS. There were a number of factors considered in deciding upon this approach. I am more familiar with the C language than the other high-level options of Java and Python and as such gave consideration to development in both C and C++. I decided that I wanted to implement the compiler via objects and thus opted for C++ however, the strict typing of C++ did lead to some hurdles in the symbol table phase that I may have not faced in Java or Python. I chose CLion for my development environment due to its inbuilt interfaces for both GCC and Git.

2. The Lexical Analyser

To implement the lexical analyser, I opted to store the characters from the .jack files and the resulting tokens in C++ vectors. I felt as though this was optimal due to its ease of implementation in C++. I was aware of but not concerned about the resulting extra storage overhead that this implementation results in as the relatively large size of memory on modern machines is amply sufficient.

To test the lexical analyser, I developed some unit tests using the Catch2 test framework. In these tests I call the lexical analyser upon a Jack file and then call the public access methods peekNextToken() and getNextToken() and test their return value upon what I perceived to be the next expected token. The files used were predominately from the provided Jack testing files however I did personally code some new files to test for general functionality and edge cases such as an end of file while scanning. There is also a tokenLog.txt file generated when debugging is run that prints out all tokens and their information designed for use in development and review for verification.

I do believe through the implementation and testing of the lexical analyser that it successfully removes white space and comments and that is does correctly extract all tokens from the source code as required.

3. The Parser

Following the instruction given in lectures I implemented a recursive descent parser based upon the grammar of the Jack language. For each of the grammar rules an accompanying method of the same title was developed each utilising chained selection statements to enforce the body of the rule.

I also developed some sister methods, isOperand(), isFactor(), etc, for some of the later grammar rules. In these methods, I coded the conditional statements that are used throughout the parser to check whether or not the current/next token is of the expected type before entering the corresponding recursive call.

To test the parser I attempted to develop another set of unit tests however, due to the nature of the parser and perhaps my inexperience in unit testing I was unable to replicate the sort of testing seen for the lexical analyser. I settled on calling the lexer upon a file and then calling the parser upon the lexer object. Though not as thorough as the lexer tests the parser tests do inform of what file failed and, alongside the error messages I coded, I was able to hone in on bugs and ensure consistency.

After thorough testing of the parser upon the given Jack files I do believe that the parser follows the structure of the given Jack grammar and that it throws syntax errors where appropriate.

4. The Symbol Table

To implement the symbol table, I created three classes: the symbol class, the table class, and the symbol table class. I went through multiple implementations of the symbol class in an attempt to be represent the end object better. The majority of the time spent went into attempting to use the symbol class as a parent to a set of ‘type’ based classes: type, variable and function. I failed to implement this in C++ due to constraints on the strict typing of vectors/arrays and issues with storing and accessing the arguments of a method when it was originally an instance of the table class. As a result, I stripped back my implementation to a single symbol class with only the fields I required to implement the semantic analyser’s requirements and replaced the table instance with a vector of symbols.

Following the diagram provided in the lecture slides - ‘General Structure of a Symbol Table’, and the description given in the text book the symbol has the fields name, type, kind, relativeAddress, arguments and initialised. Name and kind are used by all symbols, with type being used by functions and variables, and relative address, arguments and initialised being used by variables. Each field has their getters and setters. The table class is designed to represent a singular table within a symbol table. The table class has two fields: table and relativeAddresses. Table is a vector of symbols and the array of integers relativeAddress stores the current count of variable kinds: field, static, local and argument for assignment of the symbols relativeAddress and later use in the stacks. There is also an addSymbol() method and two virtual methods editSymbol() and findSymbol(). Finally, there is the symbol table class which is a child of the table class and is designed to represent the Jack symbol table with two fields: global, an instance of table for holding the class level symbols, and local, an instance of table for holding the subroutine level symbols. The symbol table class also has implementations of the virtual methods findSymbol() and editSymbol().

These classes and their setters and getters are utilised throughout the parser files recursive methods to implement the symbol table functionality. Upon the parser's execution a symbol table instance is created. When classDeclare() is called the global table is reset and subsequent class level fields and methods are appended to this table. Similarly, when subroutineDeclare() is called the local table is reset and the subsequent subroutine body contents are appended to the local table.

I am not contented with my implementation of the symbol table but due to the struggles that I faced in the development of it I decided to leave it as is and to attempt to develop the semantic analyser. As such, I opted for manual testing in an attempt to progress faster with the semantic analyser but am relatively confident that it does correctly represent the symbols given in the Jack files. There is also a symbolTableLog.txt file generated when debugging is run that prints out the global and local symbol tables at the end of the parsing for use in development and review for verification.

5. The Semantic Analyser

Built upon the symbol table the semantic analyser has been implemented through the use of the symbol table's classes and their getters in a number of selection statements throughout the parser's methods.

Based upon the provided list of semantic checking tasks that the compiler should be able to perform some amendments had to be made to the symbol table. This included the addition of the arguments and initialised field.

Of the ten semantic checking requirements I believe to have implemented nine fully:

- Variables must be declared before use and using variables from another class is permitted even when their declaration has not yet been encountered by the compiler.
- An identifier can only be declared in the same scope once.
- Variables must be initialised before they are used in expressions.
- The right had side of the assignment statement must be type compatible with the left-hand side.
- Expressions used as array indices must evaluate to an integer value.
- The called subroutine must have the same number and type of arguments as its declaration.
- A function should return a value compatible with the return type of the function.
- All code paths in a function must return a value.
- Must flag unreachable code.

And I have not implemented the following requirement:

- Subroutine calls must be resolvable i.e. If there is a call to method g in class G then g must exist.

If I had the time to implement this requirement, I would attempt to implement it by creating a two static fields in the symbol table class. One field to act as a queue where external subroutine calls are held and a program level table that contains the discovered classes and their subroutines. Comparisons can be made and any unresolved calls at the end of compilation could throw an error.

Due to the email and discussion regarding the type checking behaviour of the Jack language and how semantic errors should be handled ‘errors’ raised by the semantic analyser are displayed as warnings and do not cause an exit of the program.

Much like the symbol table targeted unit tests are missing for the semantic analyser. I undertook manual testing through the use of a number of the Jack program sets and through the generation of Jack programs with a range of incorrect semantics to test for the requirements.

6. Code Generation

Unfortunately, I have not been able to begin the implementation of the code generation phase. I believe had I have been at the university and not attempting to undertake distanced learning that I may have made more timely progress or have been more open to requesting and accessing help as I struggled with the later phases of the compiler. I am disappointed by this but fully intend to complete the compiler over the summer.

7. Compiler Usage Instructions

Information about the CMakeList.txt file:

There is a singular CMakeLists.txt provided at the root level of the submission. This cmake file sets the minimum required cmake version to 2.8 (the version found on Dec-10 machines via feng) and sets the required C++ version to 11.

Compiling the release build to view generally implementation and functionality:

1. Open the terminal in the root directory where the CMakeLists.txt file and release and debug directories are visible.
2. In the terminal run the command: cmake -DCMAKE_BUILD_TYPE=Release CMakeLists.txt
3. In the terminal run the command: make
4. In the terminal run the command and replace Set 1/Arrays with the desired Jack file directory for compiling: ./myjc “Set 1/Arrays”

After executing the compiler, you should see any warnings and errors thrown in the process of compiling the Jack files in the terminal and if an error was thrown a corresponding exit call.

Compiling the debug build to view unit tests:

1. Open the terminal in the root directory where the CMakeLists.txt file and release and debug directories are visible.
2. In the terminal run the command: cmake -DCMAKE_BUILD_TYPE=Debug CMakeLists.txt
3. In the terminal run the command: make
4. In the terminal run the command: ./myjc

After executing the compiler, you should see any warnings and errors thrown in the tests and if no error is encountered a Catch2 printout to the terminal stating the number of tests passed.

Fixing Potential Cmake Issues:

In following the given instructions and running them on the same dec-10/feng environment no errors are expected. However, if in some case an error is encountered here is some information on what could be potentially happening and how to fix it.

- If the ./myjc *directory* command returns no test cases matched *directory* and you are trying to run the release version make sure you called the cmake command with the -DCMAKE_BUILD_TYPE=Release. Or, if you are trying to run the debug version, make sure you only call ./myjc as no arguments are required and any arguments provided are used by the Catch2 test framework.
- If syntax errors are raised during the compilation of the lexer.cpp and parser.cpp files then the compiler is not using C++ version 11.
- If the cmake file is encountering compilation errors within itself ensure that the installed version is a minimum of 2.8 as expected on the dec-10/feng machines.