University of Leeds **School of Computing**
Compiler Design and Construction, COMP2932
Semester 2, 2021-2022

Compiler Project (70 marks = 70% of total module marks)

Important Note: This is an individual project, NOT a team project. Each student must implement their own compiler.

Submission Deadline: Friday 06/05/2022 at 11:00 pm (UK time)

1. The Brief

In this project, you will develop a compiler for the JACK programming language described in our textbook [1]. This involves implementing the lexical analyser, the syntactic analyser, the symbol table(s), and the code generation phases of the compiler. The target machine for the compiler is the virtual machine described in [1]. The compiler should produce virtual machine code that successfully runs on the VM emulator provided with the textbook (see this link https://www.nand2tetris.org/software). The specifications of both the JACK language and the target VM code are clearly laid out in [1]. For further information, resources, software tools and many other useful things see this site https://www.nand2tetris.org/ .

2. Compiler Details

2.1. The JACK source files

A JACK program is a collection of one or more classes. Each class is defined in its own file. JACK source files (you can call them class files) must have the *.jack* extension. All the source files of a single JACK program should be stored in the same directory.

You will develop a working compiler that accepts as input a directory containing one or more JACK source files. For each source file the compiler should produce an equivalent VM code file having the same name as the source file but with the **.vm** extension (vm=virtual machine). The target code files should be created in the same directory as the source files. After compilation, the directory will contain the .jack source files and the corresponding .vm files. To run the program using the provided VM emulator, copy the provided OS (library) files into the same program directory, then load the whole thing into the emulator.

2.2. Invoking the compiler

It should be possible to invoke your compiler at the command line. The compiler should accept one command line argument representing the name of the directory that contains the JACK source files (the JACK program). For example, if your compiler's executable file is called *myjc* and the JACK program directory is called *myprog*, the compiler would be invoked by typing this at the command line:

myjc myprog

2.3. Error Reporting

At any phase of the compilation process, if an error is encountered, the compiler should print out an informative error message, citing the line number where the error was encountered. The message

should also cite the token near which the error occurred. Here is an example of a typical error message:

```
Error, line 103, close to ";", an identifier is expected at this position
```

Uninformative generic messages such as "syntax error" must be avoided. Upon encountering an error, the compiler should report this error and immediately stop (yes, it is a short-tempered compiler!). The compiler is not required to attempt error recovery or report more than one error at a time.

2.4 The Lexical Analyser

You will write a lexical analyser (lexer) module that reads a JACK source file (having a .jack extension) and extracts all the tokens from this file. The lexer should reveal itself to other modules (i.e. the parser) through two main functions:

- *Token GetNextToken*(). Whenever this function is called it will return the next available token from the input stream, and the token is removed from the input (i.e. consumed).
- *Token* PeekNextToken(). When this function is called it will return the next available token in the input stream, but the token is not consumed (i.e. it will stay in the input). So, the next time the parser calls GetNextToken, or PeekNextToken, it gets this same token.

The lexer should successfully remove white space and comments from the input file and correctly extract all the tokens of the source code. It should not crash or become unstable if the source file contains any kind of lexical errors. One particular type of error to watch for is the unexpected end of file while scanning a string literal, or a multi-line comment.

Once you have finished developing the lexer, you must test its operation using a set of JACK source files which will be provided in due course.

2.5. The Grammar

The grammar of the JACK language is given in [1]. However, **we will write another version of the grammar**. The main purpose of writing a new version is to account for operator precedence in arithmetic expressions which is not currently accounted for in the original grammar described in [1].

2.6. The Parser

You will implement a recursive descent parser. A recursive descent parser is a collection of recursive functions that are easily developed from the grammar of the language. Once you have finished implementing the parser it should be thoroughly tested using the set of JACK source files. All kinds of syntax errors should be reported properly. However, as stated earlier, the compiler should stop on encountering the first error in a source file. It should not crash or become unstable when a syntax error is encountered.

2.7. The Symbol Table

You will implement a symbol table to store all program identifiers and their properties. A symbol in this context is any identifier defined in the program such as a variable or method name. We will explain symbol tables and their implementation in due course.

2.8. Code Generation

You will implement code generation in your compiler. However, this will not require a separate module. Code generation statements will be inserted to the parser functions at appropriate points.

3. Implementation and Planning

You will adhere to the specifications of the JACK language and the virtual machine code described in [1]. However, you will **not** follow the development guidelines, the templates, or the plan of the book. Instead, you will follow the plan detailed in the *Module Plan* spreadsheet available on Minerva.

You will write your compiler in C. You must write your compiler from scratch following the methods and guidelines explained in this module. For each phase, you will be provided with template files (both header and source files) for you to use in developing your code.

It is very important to keep to the plan and make sure that you submit each phase of the compiler project at the specified date. There are four submission dates, corresponding to the lexer, parser, symbol table, and the complete compiler respectively.

4. Submissions and Version Control

At each submission date, you must submit your current compiler code base (without executables) to Gradescope.

You may not be able to work ahead of the plan, mainly because we will be explaining the techniques and the implementation guidelines as we proceed with the teaching in lectures and tutorials.

Before submitting your code, you must make sure that you can compile and run your code on our school's Linux machines. You can remotely access your Linux account from **feng-linux.leeds.ac.uk**. You must thoroughly test your compiler on a Linux machine. So, if you develop your compiler on a Windows or Mac PC, you must not submit your project until you have made sure that it does compile and run on our Linux computers without any problems. When you submit your code to Gradescope, it will be auto-graded and you will be able to see your mark on Gradescope. Submissions that do not compile and run on Gradescope will score a zero mark, even if they work perfectly well on another platform. Instructions on how to upload the code for each phase of the compiler to Gradescope will be provided in due course.

Finally, please beware of plagiarism. It is much better to submit a partially finished project than to copy someone else's code. Gradescope can detect similarity between programs, and if it is proven that you have copied code from someone or somewhere, or allowed someone to copy your code, you may face very serious consequences.

Marking Scheme

The lexical analysis phase (the lexer) works correctly	(20 marks)
The syntactic analysis phase (the parser) works correctly	(20 marks)
The symbol tables are correctly implemented	(10 marks)
The code generation phase works correctly	(20 marks)

References

[1] Noam Nisan and Shimon Schocken, The Elements of Computing Systems, Building a Modern Computer from First Principles, MIT Press, 2005.

[2] Kenneth C. Louden, Compiler Construction: Principles and Practice, Course Technology, 1997.

[3] Alfred V. Aho et al, Compilers: Principles, Techniques, and Tools, 2nd Edition, Pearson; 2006.