**Assessment Brief**

|  |  |
| --- | --- |
| Module title | Compiler Design and Construction |
| Module code | COMP2932 |
| Assignment title | Compiler Project |
| Assignment type and description | This is a programming assignment. You will write a compiler for the JACK programming language. |
| Rationale | This coursework will help you learn the knowledge and skills needed to develop a complete compiler for a high level language. You will learn the skills associated with writing lexical analysers (lexers), syntactic analysers (parsers), semantic analysers, and the code generation phase of the compiler. |
| Word limit and guidance | You are not required to submit a report for this coursework. |
| Weighting | This coursework is worth 70% of the module mark. |
| Submission deadline | 10 May 2024 |
| Submission method | Electronic via Gradescope |
| Feedback provision | Electronic via Gradescope’s Autograder |
| Learning outcomes assessed | When you complete this coursework you will be able to:   * Write a lexical analyser for any high level programming language. * Understand context free grammers (CFGs), as well as extended CFGs. * Learn how to translate any CFG into a parser (syntactic analyser) for the language described the grammar. * Learn how to develop the necessary data structures, such as the symbol tables, and algorithms for the sematic analysis of a high level object oriented language. * Learn how to implement code generation in a compiler. |
| Module leader | Ammar Alsalka |
| Other Staff contact |  |

### 1. Assignment guidance

In this project, you will develop a compiler for the JACK programming language as described in the ‘Elements of Computing Systems’ 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. Assessment tasks

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.

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.

**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 with the project files.

**The Grammar**

The grammar of the JACK language is given in [1]. However, during the teaching of this module, 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].

**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.

**The Symbol Table**

You will implement a data structure comprising one or more symbol tables 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 during the teaching of this module.

**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. Code generation will be explained during the teaching of this module.

**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 which will be 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.

### 3. General guidance and study support

During the lectures and tutorials of this module we will explain in detail how each of the above tasks are performed. You will also be supported by lab tutors during the lab sessions of this module. Please make sure you attend all lectures and tutorials and that you use our labs when you work on the coursework so that you can get help and support from lab staff.

### 4. Assessment criteria and marking process

Your submissions will be autograded on Gradescope. The mark you see on Gradescope when you submit your code is the mark you get.

### 5. Presentation and referencing

You are not required to submit a report for this coursework.

### 6. Submission requirements

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 the School’s Linux machines. 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.

### 7. Academic misconduct and plagiarism

Leeds students are part of an academic community that shares ideas and develops new ones.

You need to learn how to work with others, how to interpret and present other people's ideas, and how to produce your own independent academic work. It is essential that you can distinguish between other people's work and your own, and correctly acknowledge other people's work.

All students new to the University are expected to complete an online [Academic Integrity tutorial and test](https://desystemshelp.leeds.ac.uk/student-guides/assessment/the-academic-integrity-tutorial-and-test/), and all Leeds students should ensure that they are aware of the principles of Academic integrity.

When you submit work for assessment it is expected that it will meet the University’s academic integrity standards.

If you do not understand what these standards are, or how they apply to your work, then please ask the module teaching staff for further guidance.

**By submitting this assignment, you are confirming that the work is a true expression of your own work and ideas and that you have given credit to others where their work has contributed to yours.**

### 8. Assessment/ marking criteria grid

**The lexical analysis phase (the lexer) works correctly (20 marks)**

**The syntactic analysis phase (the parser) works correctly (20 marks)**

**The semantic analysis phase works correctly (10 marks)**

**Code generation works correctly (20 marks)**

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