University of Leeds **School of Computing**
Compiler Design and Construction, COMP2932
Semester 2, 2019-2020

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

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

Submission Deadline: Monday 11/05/2020 at 10:00 am

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, the semantic analyser, 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 VM code that successfully run 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, and can be very 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. The Semantic Analyser

You will implement a semantic analyser to find and report possible semantic errors in the JACK program. However, this analyser will not be a standalone module. You will insert semantic analysis statements at suitable points in the parser functions.

Here is a list of the semantic checking tasks that your JACK compiler should be able to do. Most of these are very straightforward to achieve. A few of them, however, may require a little bit of resourcefulness.

1. Variables must be declared before they are used. For example, the following JACK code fragment is incorrect because variable y is not declared in the local scope of function f nor in the scope of class X:

```
class X
{
   field int x;
   function void f (int a)
   {
     let x = a/2;
     let y = x+1;            // wrong, y has not been declared
   }
}
```

   Using variables from another class is permitted even when their declaration has not yet been encountered by the compiler. However, these declarations must later be resolved when all the source code is compiled. Hence, this JACK code fragment is correct:

```
// in one file
class X
{
  function void f (int a)
  {
     var Y y;
     char c;
     let y = Y.new ();
     let c = y.v;
  }
}
// in another file
class Y
{
   field char v;
   // … etc
}
```

2. An identifier can only be declared in the same scope once (no redeclaration of the same identifier). This applies to all scopes whether it is the local scope of a subroutine, the scope of a class, or the entire program scope. For example, it is not allowed to declare two classes with the same name in the same program (albeit they are declared in different files).

3. Variables must be initialized (assigned a value) before they are used in expressions. Hence, the following JACK code fragment is incorrect, because variable x is being used before it is initialised:

```
method int m ()
{
  var boolean b;
  if (b) {                    // wrong, b is being used before it is initialised
  // … etc
 }
}
```

4. The right-hand side (RHS) of an assignment statement must be type compatible with the left-hand side (LHS), for example, if variable x is of type BankAccount (an object) and y is a char, then the following assignment statement is not allowed:

```
var BankAccount x;
var char y;
let x = BankAcount.new ();
let y= 100;
let x = y+1;    // wrong, LHS is a reference to an object but RHS is a char
```

5. Expressions used as array indices must evaluate to an integer value, hence the following is wrong:

```
var bool b;
var Array a;
let a = Array. new (10);
let b = false;
a[b] = 10;
```

6. Subroutine calls must be resolvable, i.e. if there is a call to method g in class G, then there must be a declaration of this method somewhere in the source code.

7. The called subroutine must have the same number and type of arguments as its declaration, for example the following function call is wrong:

```
function int g (int a, char b) {
//…etc
}
do g (3);  // wrong, g requires 2 arguments
```

8. A function should return a value compatible with the return type of the function, for example the following return statement is wrong:

```
function int g (int a, char b)
{
 BankAccount b;
 let b = BankAccount.new ();
 return b;  // b is a reference to an object and not an int type
```

```
}
```

9. All code paths in a function must return a value, for example the following function implementation should not be allowed:

```
Function int f(in a)
{
   If (a==0)
     return 0;
// wrong, what is the return value if a!=0
}
```

10. Unreachable code (e.g. following a non-conditional return in the body of a function).

```
Function int f(in a)
{
   If (a==0)
     return 0;
return 1;
let a = 4; // wrong, this part of the code will never be reached
}
```

Once the semantic analyser is complete, it should be thoroughly tested.

2.9. 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 can write your compiler in any of the following languages:

- C
- C++
- Java
- Python

You are allowed to use any standard library that comes with the programming language. But you are not allowed to use libraries that can perform entire compilation tasks (e.g. tokenization or parsing). **You are also not allowed to use complier generator tools**. You must write your compiler from scratch following the methods and guidelines explained in this module.

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 or the plan of the book. Instead, you will follow the plan detailed in the *Teaching and Project Plan* spreadsheet attached to this document (A copy of this spreadsheet is also available on Minerva). **It is very important to keep to the plan and make sure that you reach each milestone of the compiler project at the specified date**. Each milestone represents the completion of one phase of the compiler. Except for the final project submission deadline, there are three major milestones, corresponding to the lexical, syntactic, and semantic analysis phases respectively.

4. Submissions and Version Control

You must create a GitLab repository for your project and give me full access to it. My GitLab username is drsalka. You must regularly push commits to the repository *particularly at project milestones*.

At each milestone date, you must submit your current compiler code base (without executables) to Minerva with a brief description (200 - 300 words) of what you have achieved so far. It does not matter if your code at a certain milestone still contains some bugs or errors, as long as the main functionality required at this stage has been achieved. **10 marks have been assigned for achieving the milestones in due time, as detailed in the marking scheme below.**

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 compiler, you must make sure that you can compile your compiler on our school's Linux machines (e.g. DEC-10 lab machines). You must also thoroughly test your compiler on these machines. Submissions that do not compile and run on our Linux machines will score a zero mark, even if they work perfectly well on another platform. 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.

The final project code base should be committed to GitLab and submitted to Minerva. Before submitting to Minerva, clean your project directory of any object or executable files, compress it using ZIP, and upload to Minerva.

You are also required to write and submit a brief report (5-10 pages) at the end of the project. A template for the report 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. I will be using special software that 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	(6 marks)
The syntactic analysis phase (the parser) works correctly	(12 marks)
The semantic analysis phase (including the symbol table) works correctly	(16 marks)
The code generation phase works correctly	(10 marks)
The compiler code is well structured, efficient, clear, and properly commented	(6 marks)
Adherence to project milestones (the plan):	
• First milestone, the lexer, 21 February 2020	(3 marks)
• Second milestone, the parser, 6 March 2020	(4 marks)
• Third milestone, the symbol table, 20 March 2020	(3 marks)
Final project report	(10 marks)

Reference

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