Assignment 1 Description

Weighting and Due Date

- Marks for this assignment contribute 5% of the overall course mark.
- Marks for functionality will be awarded automatically by the web submission system.
- **Due dates: Milestone** - 11:55pm Tuesday of week 7, **Final** - 11:55pm Friday of week 7.
- **Late penalties:**The maximum mark awarded will be reduced by 25% per day / part day late. Marks above the cap will be discarded.
- **Core Body of Knowledge (CBOK) Areas:** abstraction, design, hardware and software, data and information, and programming (see here (Links to an external site.)Links to an external site. for a detailed description of the CBOK).

Project Description

In this assignment you will implement two versions of a tokeniser that with minor changes could be used to complete variations of projects 6, 10 and 11 in the nand2tetris course. A detailed description of the requirements are shown below. The exectuable programs, **tokeniser-m** and **tokeniser-f** will read text from standard input and produce a list of all tokens in the text on standard output.

SVN Repository

You must create a directory in your svn repository named: **<year>/<semester>/cs/assignment1**. This directory must only contain the following files and directories - the web submission system will check this:

- **Makefile** - this file is used by **make** to compile your programs - **do not modify this file**.
- **tokensm.cpp**, **tokensm.h** C++ source files for class ***tokensm***
- **tokensf.cpp**, **tokensf.h** C++ source files for class ***tokensf***
- **my*.cpp** C++ source files with names that start with **my**
- **my*.h** C++ include files with names that start with **my**
- **lib** - this directory contains precompiled programs and components - **do not modify this directory**.
- **includes** - this directory contains **.h** files for precompiled classes - **do not modify this directory**.
- **testsm** - this directory contains sample test data for the ***tokensm*** class, it can be used to store any extra files you need for testing
- **testsf** - this directory contains sample test data for the ***tokensf*** class, it can be used to store any extra files you need for testing

Note: if the **lib.a** and **.o** files in the **lib** directory do not get added to your svn repository you will need explicitly added them using:

```
% svn add lib/lib.a lib/*.o
```

Submission and Marking Scheme

This assignment has two *assignments* in the web submission system named: ***Assignment 1 - Milestone Submissions*** and ***Assignment 1 - Final Submissions***.

Assignment 1 - Milestone Submissions: due 11:55pm Tuesday of week 7

The marks awarded by the web submission system for the milestone submission contribute up to 20% of your marks for assignment 1. Your milestone submission mark, after the application of late penalties, will be posted to the myuni gradebook.
Your programs must be written in C++ and will be tested using the set of test files that are attached below. Although a wide range of tests may be run, marks will only be recorded for those tests that use the program **tokeniser-m**. Your programs will be compiled using the **Makefile** included in the zip file attached below. Any **.h** or **.cpp** files that you create, in addition to the skeletons provided, must have names that start with **my**. The **Makefile** will use all of the **my*.cpp** files in your svn directory as part of the **tokeniser-m** and **tokeniser-f** programs that it compiles.

Assignment 1 - Final Submissions: due 11:55pm Friday of week 7

The marks awarded for the final submission contribute up to 80% of your marks for assignment 1.
Your mark for the final submission will be either the marks awarded by the web submission system or, if they exceed either the marks awarded for your logbook or the marks awarded for your coding, your final submission mark will be the geometric mean of these three marks. Your final submission mark, after the application of late penalties, will be posted to the myuni gradebook.
Important: the logbook must have entries for all work in this assignment, including your milestone submissions. See "Assessment of Practical Assignments" for the logbook rubric.

Automatic Marking

The automatic marking will compile and test both of your tokenisers in exactly the same way as for the milestone submission. The difference is that marks will be recorded for **all** of the tests and there may be a number of additional ***secret*** tests.

Code Review Marking

The marks for your coding are based on a manual review of your code that will look at your coding style, your commenting and program structure. We expect that all code that you write will be laid out consistently, it will be easy to read, it will have a clear structure, it will not be a single main function, each function will be prefixed by an appropriate comment and any key logic or data structures will also be appropriately commented.
Note: the way in which code is written is extremely important in the real world. One of the major costs of software development is maintenance. That is, taking existing code that needs bugs fixed or modified in order to add new functionality. If code is poorly structured, does not make good use of abstractions, is badly formatted, lacks comments or the

comments are confusing, it makes the task of maintaining the code very unpleasant and more error prone than it should.
To reflect the importance of how code is written, the code review assessment may be used to limit your final submission mark, as described above. See "Assessment of Practical Assignments" for the code review rubric.

Tokenisers

Background

The primary task of any language translator is to work out how the structure and meaning of an input in a given language so that an appropriate translation can be output in another language. If you think of this in terms of a natural language such as English. When you attempt to read a sentence you do not spend your time worrying about what characters there are, how much space is between the letters or where lines are broken. What you do is consider the words and attempt to derive structure and meaning from their order and arrangement into English language sentences, paragraphs, sections, chapters etc. In the same way, when we attempt to write translators from assembly language, virtual machine language or programming language into another form, we attempt to focus on things like keywords, identifiers, operators and logical structures rather than individual characters.
The role of a tokeniser is to take the input text and break it up into tokens (words in natural language) so that the assembler or compiler using it only needs to concern itself with higher level structure and meaning. This division of labor is reflected in most programming language definitions in that they usually have a separate syntax definition for tokens and another for structures formed from the tokens.
The focus of this assignment is writing a tokeniser to recognise tokens that conform to a specific set of rules. The set of tokens may or may not correspond to a particular language because a tokeniser is a fairly generic tool. After completing this assignment we will assume that you know how to write a tokeniser and we will provide you a working tokeniser to use in each of the remaining programming assignments. This will permit you to take the later assignments much further than would be otherwise possible in the limited time available.
Objective
Write two C++ classes, ***tokensm*** and ***tokensf***, that provide a tokeniser for the programs **tokeniser-m** and **tokeniser-f**, respectively. The classes, ***tokensm*** and ***tokensf***, read text character by character using a function that is supplied at runtime, ***read_char()***, and return the next recognised token in the input. The tokens recognised by the classes, ***tokensm*** and ***tokensf***, are specified in separate tables below.

Contract

1. You must implement the class ***tokensm*** to provide the tokeniser for **tokeniser-m**.
2. The ***tokensm*** class and its include file must be placed in the **tokensm.cpp** and **tokensm.h** files respectively.
3. For each **.txt** file in the **tests** directory, the output of **tokeniser-m** must match the corresponding **.tokensm** output file.
4. You must implement the class ***tokensf*** to provide the tokeniser for **tokeniser-f**.

5. The ***tokensf*** class and its include file must be placed in the **tokensf.cpp** and **tokensf.h** files respectively.
6. For each **.txt** file in the **tests** directory, the output of **tokeniser-f** must match the corresponding **.tokensf** output file.

Writing Your Programs

In order to simplify testing we have provided two precompiled main programs, **mainm.o** and **mainf.o**, that will access your implementations. The are stored in the **lib** sub-directory together with a library **lib.a** that implements the ***lookup_keyword()*** function described below. The two main programs use either the ***tokensm*** or ***tokensf*** class respectively to read a text file and output the tokens that they find. Your tokeniser classes should not attempt to read input in any other way and, once complete, must not produce any output of their own.

Compiling Your Programs

In addition to the files provided in the zip file attached below, if you wish to create your own **.h** and **.cpp** files, their names must all start with **my**. Your programs will be compiled using the **Makefile** in the zip file attached below using the command:

```
% make
```

This will attempt to compile both versions of your tokeniser program using all of the **my*.cpp** files as part of both programs. If you only wish to compile one of your programs you can specify this on the command line, eg:

```
% make tokeniser-m
```

or

```
% make tokeniser-f
```

Note: Do not modify the **Makefile** or the sub-directories **includes** and **lib**. They will be replaced during testing by the web submission system.

Running Your Programs

Your program will be tested by the web submission system as follows. If we have an input test file named **somefile** then there will be matching output files **somefile.tokensm** and **somefile.tokensf**. The **tokeniser-m** program can be compiled and tested using the commands:

```
% make tokeniser-m
% cat somefile | ./tokeniser-m | diff - somefile.tokensm
```

The **tokeniser-f** program could be compiled and tested using the commands:

```
% make tokeniser-f
% cat somefile.txt | ./tokeniser-f | diff - somefile.tokensf
```

In each case the make command only compiles the specified program, **tokeniser-m** or **tokeniser-f**. The second command runs the compiled program with the file **somefile** connected to its standard input, (**cin**), and connects its standard output (**cout**) to the standard input of ***diff***. ***diff*** then compares the output of the program with the specified test output file. If ***diff*** does not report any differences, then the program is working correctly. If you wish to save the output before checking the output you could run and test the programs as follows:

```
% cat input00.txt | ./tokeniser-m > somefile
% diff somefile test00.tokensm
```

If you wish to run your programs against all of the supplied tests you can run the following commands to test the **tokeniser-m** or **tokeniser-f** programs respectively:

```
% make test-tokeniser-m
Testing student tokensm class against .tokensm files
Checking "./tokeniser-m < testsm/Add.asm | diff - testsm/Add.asm.tokensm" - test passed
...
% make test-tokeniser-f
Testing student tokensf class against .tokensf files
Checking "./tokeniser-f < testsf/Add.asm | diff - testsf/Add.asm.tokensf" - test passed
...
```

Tokeniser-m

Your implementation of the class ***tokensm*** in the files **tokensm.h** and **tokensm.cpp** must recognise the following tokens:

Token		**Definition**	**Example Token**	**Token Value**
"identifier"	::=	letter **(** letter **|** digit **)***	_he82mUch	"_he82mUch "
"number"	::=	**(** '0' **|** **(** digit19 digit* **))** **[** '.' digit* **]**	**17.05**	"17.05"
"punctuation"	::=	';' **|** ':' **|** '!' **|** ',' **|** '.' **|** '=' **|** '{' **|** '}' **|** '(' **|** ')' **|** '[' **|** ']' **|** '@'	;	";"
Additional Rules		**Definition**	**Example Text**	
letter	::=	'a'-'z' **|** 'A'-'Z' **|** '_'	"C"	
digit19	::=	'1'-'9'	"1"	
digit	::=	'0'-'9'	"0"	

Notes:

- *all input must be read using the function **read_char()***
- if an error occurs or the end of input is reached, return the token as "?" with value "?"
- it is an error to find a character that cannot be part of token or is not a space " ", tab "\t", carriage return "\r" or newline "\n"
- letter, digit19 and digit are never returned as token classes
- all tokens must be contiguous characters in the input
- when searching for the start of the next token all spaces and newlines are ignored
- in a definition the or operator **|** separates alternative components
- in a definition the round brackets **( )** which are not inside single quotes are for grouping components of token

- in a definition the square brackets **[ ]** which indicates that the enclosed components may appear 0 or once
- in a definition the star character ***** indicates that the preceding component of a token may appear 0 or more times

Tokeniser-f

Your implementation of the class ***tokensf*** in the files **tokensf.h** and **tokensf.cpp** must recognise the following tokens:

Token		**Definition**	**Example Token**	**Token Value**
"identifier"	::=	letter **(** letter **|** digit **)***	_he82mUch	"_he82mUch"
"number"	::=	**(** '0' **|** **(** digit19 digit* **))** **[** '.' digit* **]**	**17.05**	"17.05"
"punctuation"	::=	';' **|** ':' **|** '!' **|** ',' **|** '.' **|** '=' **|** '{' **|** '}' **|** '(' **|** ')' **|** '[' **|** ']' **|** '@'	;	";"
"keyword"	::=	'if' **|** 'while' **|** 'else' **|** 'class' **|** 'int' **|** 'string'	if	"if"
Additional Rules		**Definition**	**Example Text**	
letter	::=	'a'-'z' **|** 'A'-'Z' **|** '_'	"C"	
digit19	::=	'1'-'9'	"1"	
digit	::=	'0'-'9'	"0"	

Notes:

- all input must be read using the function ***read_char()***
- if an error occurs or the end of input is reached, return the token class "?" with value "?"
- it is an error to find a character that cannot be part of token or is not a space " ", tab "\t", carriage return "\r" or newline "\n"
- single line comments start with "//" and finish at the next newline character "\n"
- adhoc comments start with "/*" and continue until the first "*/", the shortest adhoc comment is "/*/"
- letter, digit19 and digit are never returned as token classes
- keyword tokens are only to be recognised by interpreting an identifier token
- use the ***lookup_keyword()*** function to check if an identifier is actually a keyword
- all tokens must be contiguous characters in the input
- when searching for the start of the next token all spaces, newlines and comments are ignored

- in a definition the or operator **|** separates alternative components
- in a definition the round brackets **( )** which are not inside single quotes are for grouping components of token
- in a definition the square brackets **[ ]** which indicates that the enclosed components may appear 0 or once
- in a definition the star character ***** indicates that the preceding component of a token may appear 0 or more times

Tests

In addition to the test files in the zip file attached below, we will use a number of ***secret*** tests that may contain illegal characters or character combinations that may defeat your tokenisers. **Note:** these tests are ***secret***, if your programs fail any of these ***secret*** tests you **will not** receive any feedback about these ***secret*** tests, even if you ask!

Startup Files

- assignment1.zip