Assignment 1 Description

Assignment 1 - Tokeniser

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:** For each part, the maximum mark awarded will be reduced by 25% per day / part day late. If your mark is greater than the maximum, it will be reduced to the maximum.
- **Core Body of Knowledge (CBOK) Areas:** abstraction, design, hardware and software, data and information, and programming.

Project Description

In this assignment you will implement a tokeniser that with minor changes could be used to complete variations of projects 6, 7, 8, 10 and 11 in the nand2tetris course. A detailed description of the requirements are shown below. The exectuable programs, **tokens** and **tokens-context** will read text from standard input and produce a list of tokens in the text on standard output.

Note: you should complete workshop 05 before attempting this assignment and review the **EBNF, Languages and Parsing (https://myuni.adelaide.edu.au/courses/64329/pages/ebnf-languages-and-parsing)** page.

SVN Repository

Note: this assignment assumes that you have already created directories for every assignment, workshop, project and exam in your svn repository, as described on the [Startup Files for Workshops and Assignments (https://myuni.adelaide.edu.au/courses/64329/pages/startup-files-for-workshops-and-assignments)](https://myuni.adelaide.edu.au/courses/64329/pages/startup-files-for-workshops-and-assignments) page.

1. If required, checkout a working copy of the assignment1 directory from your svn repository.
2. Change directory to the working copy of the assignment1 directory.
3. Copy the latest startup files from the "**View Feedback**" tab of the "**Assignment 1 - Submit Here**" assignment into the updates sub-directory. **Do not unzip the file.**
4. Run the following command to place the assignment's startup files in the correct locations, this will automatically add them to svn and then commit them to your svn repository:
   ```
   % make install
   ```
5. Goto the Web Submission System and make a submission to the "**Assignment 1 - Submit Here**" assignment.

Assignment 1 Files and Directories

In addition to the generic **Makefile** and **updates** sub-directory, the assignment1 directory should now contain the following files and directories:

- **tokens** - executable script that will run your compiled **tokens** program.
- **tokens.cpp** C++ source file containing the **main()** function for **tokens.**
- **tokens-context** - executable script that will run your compiled **tokens-context** program.
- **tokens-context.cpp** C++ source file containing the **main()** function for **tokens-context.**
- **tokeniser.cpp** C++ source file containing the **next_token()** function.
- **tokeniser-context.cpp** C++ source file containing support functions.
- **tokeniser-extras.cpp** C++ source file containing support functions.
- **bin** - this directory contains precompiled programs and scripts.
- **includes** - this directory contains **.h** files for the library.
- **lib** - this directory contains precompiled library components.
- **originals** - this directory contains the original versions of **tokeniser.cpp**, **tokeniser-context.cpp** and **tokeniser-extras.cpp**

- **tests** - this directory contains test data.

Note: you need to edit the **tokeniser.cpp**, **tokeniser-context.cpp** and **tokeniser-extras.cpp** files to complete this assignment. All the other files are automatically regenerated every time you run **make**, they must not be changed or added to **svn**.

Note: if a newer version of the startup files is made available, it must be placed in the **updates** sub-directory and added to **svn**. The next time **make** is run, all of the files will be updated except for **tokeniser.cpp**, **tokeniser-context.cpp** and **tokeniser-extras.cpp**

Submission and Marking Scheme

Submissions for this assignment must be made to the web submission system (https://cs.adelaide.edu.au/services/websubmission) assignment named: **Assignment 1 - Submit Here**. The assessment is based on "Assessment of Programming Assignments (https://myuni.adelaide.edu.au/courses/64329/pages/assessment-of-programming-assignments)".

Notes:

- the marks for the Milestone Tests will be shown by the **Assignment 1 - Milestone** assignment
- the marks for the Final Tests will be shown by the **Assignment 1 - Final** assignment
- the participation marks for this assignment will be shown by the **Assignment 1 - Pre Milestone** assignment

Your tokeniser program must be written in C++. Your **tokens** and **tokens-context** programs will be compiled using the **Makefile** and the **tokens.cpp** or **tokens-context.cpp** files included in startup files together with the **tokeniser.cpp**, **tokeniser-context.cpp** and **tokeniser-extras.cpp** files in your svn directory. Your programs will then be tested using the set of test files that are included in the startup files. In addition a number of ***secret*** tests will also be run. **Note**: if your program fails any of these ***secret*** tests you **will not** receive any feedback about these ***secret*** tests, even if you ask!

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

A mark out of 100 for the Milestone Tests will be awarded by the **web submission system (https://cs.adelaide.edu.au/services/websubmission)** and will contribute up to 20% of your marks for this assignment. The marks awarded will be automatically modified by a code review program, as described on the **Assessment of Programming Assignments (https://myuni.adelaide.edu.au/courses/64329/pages/assessment-of-programming-assignments)** page. The Milestone Tests test the milestone token definitions shown below using your **tokens** program.

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

A mark out of 100 for the Final Tests will be awarded by the **web submission system (https://cs.adelaide.edu.au/services/websubmission)** and will contribute up to 80% of your marks for this assignment. The marks awarded will be automatically modified by a code review program, as described on the **Assessment of Programming Assignments (https://myuni.adelaide.edu.au/courses/64329/pages/assessment-of-programming-assignments)** page. The tests of the **tokens-context** program will contribute 25% of the marks for the Final Tests.

Your Final Assignment Mark will be the geometric mean of two components, the weighted marks, **MF**, for the Milestone Tests, **M**, and Final Tests, **F**, and a mark for your logbook, **L**. It will be limited to 10% more than the marks for the Final Tests. ie

MF = M * 0.2 + F * 0.8

FinalMark = MIN(1.1 * MF, SQRT(MF * L))

NOTE - A logbook mark of 0 results in a Final Assignment mark of 0.

Logbook Marking

Important: the logbook must have entries for all work in this assignment, including your milestone submissions. See "**Assessment - Logbook Review (https://myuni.adelaide.edu.au/courses/64329/pages/assessment-logbook-review)** " for details of how your logbook will be assessed.

Assignment Weighting and Workload

Depending on your individual programming skills and your understanding of the course material, the assignment weighting may not reflect the workload required to complete it. It is important to recognise this and to make sure that you do not spend an excessive amount of time trying to complete everything at the expense of your other studies. It is important to know when to stop!

Assignment 1 - Participation Marks

Any submissions to assignment 1 that are made before the due date for Milestone Submissions may be awarded up to 5 participation marks. The participation marks will be the marks awarded for the Final Tests divided by 20. Your logbook review mark will not affect the participation marks. These participation marks will be allocated to week 7.

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 a 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.

Writing Your Program

You are required to complete the implementation of the C++ files **tokeniser.cpp**, **tokeniser-context.cpp** and **tokeniser-extras.cpp** which are used to compile the programs **tokens** and **tokens-context**.

Tokeniser

The **tokeniser.cpp** file is where you will complete the implementation of a function, ***parse_token(),*** that will parse the program's input character by character and return the next recognised token. The only functions that should be required to achieve this are, ***read_next_char()***, ***next_char_isa()***, ***next_char_mustbe()*** and ***did_not_find_char()***. These functions and supporting character groups are described in the file **includes/tokeniser-extras.h** and operate as they did in Workshop 05. The tokens that must be recognised in the milestone and final submissions are specified in the file **includes/tokeniser.h**.

Tokeniser Extras

The **tokeniser-extras.cpp** file is where you will implement the support functions ***char_isa()***, ***classify_spelling()*** and ***correct_spelling()***.

The ***char_isa()*** function is a passed a character and it checks if that character matches a second character or group of characters. There should be a group of possible start characters for every rule in the token grammar. To illustrate a possible code structure the checking of membership of the cg_wspace group has been already implemented. You do not need to explicitly check any groups consisting of a single character, these cases are all caught by the initial if statement. The parsing functions
next_char_isa() and ***next_char_mustbe()*** both depend on ***char_isa()*** to operate correctly.

The ***classify_spelling()*** function is called by ***new_token()*** to work out what kind of token was parsed. In most cases this can be achieved by simply looking at the first character of the spelling. However, given that we already know that we have a properly constructed token any other checking will also be very simple.

The ***correct_spelling()*** function is called by ***new_token()*** to modify the spelling that it includes in the new token object. Depending on what is using the tokens, it may be convenient to make minor adjustments such as removing "" around a string or ( ) around a label. If there are specific modifications required, these will be described in file **includes/tokeniser.h**.

Tokeniser Context

When errors occur in parsing it is nice to be able to show where the error occurred with some context. In order to do this the input needs to be remembered as it is read so we can later reconstruct the input around the position of an error. In our case the ***new_token()*** function also uses all characters remembered since it was last called as the initial token spelling. The functions to support the input memory are defined in the file **includes/tokeniser-context.h** and must be implemented in the file **tokeniser-context.cpp**.

The ***remember()*** function can be passed any legal unicode code-point which are integer values in the range 0 to 0x10FFFF with the exception of the range 0xD800 to 0xDFFF. The function will need to remember the UTF-8 encoding of the value which can be up to 4 one-byte characters long. Fortunately, C++

treats strings as sequences of one-byte characters so each remembered unicode character will be from 1 to 4 characters in a C++ string. The column numbering is in bytes so you can use C++ string lengths and ignore the actual number of unicode characters they represent. The character sequence is only interpreted as UTF-8 when it is displayed, for example by a terminal. See https://en.wikipedia.org/wiki/UTF-8 (https://en.wikipedia.org/wiki/UTF-8) for more details.

The necessary arithmetic required to construct the byte values for the UTF-8 encoding is described in the comments in the **tokeniser-context.cpp** file. You may wish to review the Bitwise Operators (https://myuni.adelaide.edu.au/courses/64329/pages/bitwise-operators) page before attempting the encoding.

The ***remembered_line()*** function returns a specific line from the text built up by calls to the ***remember()*** function. For example, if we have the following calls:

```
remember('a') ; remember('b') ; remember('\n') ; remember('z') ;
```

The call remembered_line(1) would return "ab\n" and the call remembered_line(2) would return "z". The second line does not have a newline character because the last remembered character is 'z'.

Compiling

Your **tokens** and **tokens-context** programs can be compiled using the **Makefile** in the startup files using the command:

```
% make notest
```

Notes:

- The only files you are allowed to edit are **tokeniser.cpp**, **tokeniser-context.cpp** and **tokeniser-extras.cpp**. All other files may be automatically regenerated every time you run **make** and are not used by the web submission system (https://cs.adelaide.edu.au/services/websubmission) 's test scripts.

- You must use **make** to compile and test your programs so that the **updates/log** file can help record your progress and provide some additional information, such as source file changes, that can be later integrated into your logbook.

Testing Your Program

The **tests** directory contains some **tests.csv** files that list a set of test inputs and expected outputs that can be used to test your programs. In order to pass a test, the output of your **tokens** and **tokens-context** programs must match the expected output, any error output must also match and the programs must exit with the expected exit status. You can compile and test both programs simply by running make with no arguments:

```
% make
```

The testing will not show you any program output but it will show, the test-id, the program being run, any command line arguments, whether or not the output, error output and exit status were what was expected and possibly a short description of the test. Unless explicitly stated elsewhere, the exit status for your programs should always be 0.

Test Details

If you want to see the actual output or other details of the tests, there are a few options you can try.

- **make test test-id**
  This will only run the test(s) with the matching test-id. If you do not supply a test-id it runs all of the tests.
- **make live test-id**
  If the test-id matches a single test, the test is run as before but then a final run is made that outputs directly to the terminal.
- **make show test-id**
  This does not run any tests but it will show the differences between expected and actual outputs. If no test-id is given all tests are shown.
- **make Show test-id**
  This does not run any tests and is a more detailed version of show that lists the pathnames of all files used in testing as well as differences between

your program output with and without trace writes and error log messages. The test scripts normally disable all trace writes and error log messages so that your debugging output does not interfere with testing. However, this viewing option lets you see you all of your debugging output. If no test-id is given all tests are shown.

- **make less test-id**
  This is the same as **make show** but the output is piped through less so you can view it page by page.
- **make Less test-id**
  This is the same as **make Show** but the output is piped through less so you can view it page by page.

Milestone Tokens

Your milestone submission will only be awarded marks for tests that require the correct recognition of the milestone tokens described in the **includes/tokeniser.h** file.

Note: the **includes/tokeniser.h** file describes

- the grammar rules for all tokens,
- the tokeniser interface functions that must be implemented,
- the rules for differentiating identifiers and keywords (not required for the milestone),
- the rules for modifying the spelling of a token (not required for the milestone) and
- the rules governing error handling.

Final Submission Tokens

Your final submission will be awarded marks for tests that require the correct recognition of all tokens described in the **includes/tokeniser.h** file.

Tests

In addition to the test files in the startup files, the web submission system will use a number of ***secret*** tests that may contain illegal characters or character combinations that may defeat an incorrect tokeniser. The ***secret*** tests may also check whether or not you have followed the required approach for keyword recognition. **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!