**Workshop 07 - Writing an Assembler**

**Writing an Assembler**

[**Pracmarker Link**](https://cs.adelaide.edu.au/services/pracmarker/?sub_year=2016&sub_period=s2&sub_course=cs&sub_assign=workshop07)

In assignment 3 you will be writing your own Hack assembler. However, we require you to read the source files, manage the symbols and produce the output in using particular classes. You will have an opportunity to implement these classes yourself but some of the tests we will run will also use our implementations.

There are five classes that must be used, ***asmtokens* (T)**, ***symbols\_int* (S)**, ***symbols\_string* (S)**, ***asmxml* (X)** and ***asmcode* (X)**. In this workshop you will first download and a program that can be made up of precompiled versions of these classes that break an assembly language program into tokens. Once you have the programs working with the precompiled classes, you can attempt to implement your own version of the classes ***asmtokens*** and ***symbols\_string***, then use them to replace the precompiled versions.

**Step 1 - Download, compile and run the precompiled assembler classes**

The first exercise is to run a main program that uses pre-compiled versions of the classes to read a file of Hack assembly language and print out tokens in XML.

First download the zip file attached below.

After expanding the file compile the main program using the command:

% make

This will compile four versions of the main.cpp file using different combinations of precompiled implementations of the five classes:

* **X** - this version uses precompiled versions of ***asmxml*** and ***asmcode***.
* **XS** - this version uses precompiled versions of ***symbols\_int***,***symbols\_string****,* ***asmxml*** and ***asmcode***.
* **XT** - this version uses precompiled versions of ***asmtokens***, ***asmxml*** and ***asmcode***.
* **XST** -  this version uses precompiled versions of ***symbols\_int***,***symbols\_string****,* ***asmtokens***, ***asmxml*** and ***asmcode***.

Now run each version of the compiled program and give it some assembler input using pipes or shell redirection. Here is an example using all the precompiled classes and provided a single C instruction as input:

% echo "MD=A" | ./XST

<tokens>

<dest>MD</dest>

<equals>=</equals>

<dest-comp?>A</dest-comp?>

</tokens>

Number of tokens read:

0000000000000011

The **X** and **XS** versions of the program do not work correctly because they are using a skeleton implementation of the tokeniser that always returns "**?**" as the next token. The **XT** version of the program does not work because ***asmtokens*** is using a skeleton implementation of ***symbols\_string*** that cannot add symbols or find them in its symbol table.

**Step 2 - Web Submission Testing**

If you want to see what kind of tests the web submission system will run for the later assignments, place the tokeniser files in your svn repository in a directory named **<year>/<semester>/cs/workshop07** and make a submission to the **Workshop 07** assignment in the Web Submission System. The test script will first replace all of your files except for ***main.cpp***, ***asmtokens.cpp*** and ***asmsymbols.cpp*** and then it will run a number of tests using programs **X**, **XS**, **XT** and **XST**. The expected output for each Hack assembly file in the Nand2Tetris project06 directories is included in the zip file attached below.

**Step 3 - Write your own tokeniser**

Now that you know how to run the main program and produce XML output, you should attempt to write your own tokeniser for the Hack assembler language. To do this you need to complete the skeleton implementation provided in the file ***asmtokens.cpp***.

There are two restrictions on your implementation of the ***asmtokens*** class. Firstly, you must read the input by calling ***cin.get()***, you**must not**use***cin.getline()***. Secondly, you must recognise the *C-component* token and use an internal ***symbols\_string*** symbol table to match it to a legal *dest*, *dest-comp?*, *comp*, *jump* or *null* token value. If a match is not found then a legal token cannot be formed from the next input. The skeleton implementation provided in the file ***asmtokens.cpp*** creates the internal symbol table and populates it.

**Tokens to Recognise**

The token classes that you must recognise are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Token** |  | **Definition** | **Example Token** | **Token Value** |
| "address" | ::= | '@' ( name | number ) | **@hello** | "hello" |
| "label" | ::= | '(' name ')' | **(END)** | "END" |
| "dest" | ::= | 'MD' | 'AM' | 'AD' | 'AMD' | **AM** | "AM" |
| "dest-comp?" | ::= | 'A' | 'M' | 'D' | **A** | "A" |
| "comp" | ::= | '0' | '1' | '-1' | '!D' | '!A' | '-D' | '-A' | 'D+1' | 'A+1' | 'D-1' | 'A-1' | 'D+A' | 'D-A' | 'A-D' | 'D&A' | 'D|A' | '!M' | '-M' | 'M+1' | 'M-1' | 'D+M' | 'D-M' | 'M-D' | 'D&M' | 'D|M' | **D-A** | "D-A" |
| "jump" | ::= | 'JMP' | 'JLT' | 'JLE' | 'JGT' | 'JGE' | 'JEQ' | 'JNE' | **JGT** | "JGT" |
| "semi" | ::= | ';' | **;** | ";" |
| "equals" | ::= | '=' | **=** | "=" |
| "null" | ::= | 'NULL' | **null** | "null" |
| **Additional Rules** |  | **Definition** | **Example Text** | |
| C-component | ::= | ( letter | digit | operator )( letter | digit | operator )\* | "D+M" | |
| name | ::= | letter ( letter | digit )\* | "\_he:82m.Uch$" | |
| number | ::= | digit digit\* | "99" | |
| letter | ::= | 'a'-'z' | 'A'-'Z' | '$' | '\_' | ':' | '.' | "$" | |
| digit | ::= | '0'-'9' | "1" | |
| operator | ::= | '-'|'+'|'!'|'&'|'|' | "+" | |

**Notes:**

* C-component, name, number, letter, digit and operator are never returned as token classes
* all letters in a C-component must be converted to upper case before searching for a matching token
* all tokens must be contiguous characters in the input
* when searching for the start of the next token all whitespace and comments are ignored
* newlines are not significant to the tokeniser so more than one instruction can be on the same line
* your tokeniser should have a member variable to record the next character to be read
* in a definition the round brackets **( )** which are not inside single quotes are for grouping components of token
* in a definition the star character **\*** indicates that the preceding component of a token may appear 0 or more times

**Testing Your Tokeniser**

Once you have something that might work, compile your tokeniser using this command:

% make

Once your tokeniser compiles, you can run it using the **XS** program and hopefully you will see:

% echo "D=1" | ./XS

<tokens>

<dest-comp?>D</dest-comp?>

<equals>=</equals>

<comp>1</comp>

</tokens>

Number of tokens read:

0000000000000011

**Step 4 - Implement your own symbol tables**

When implementing assemblers and compilers there is a need to record and keep track of a range of different symbols and their values. These symbols may represent addresses in memory, constant values, components of an instruction, variables, classes, pointers, functions, and so on. Regardless of the values associated with the symbols, all symbol tables need to provide two basic operations, *insert()*and*lookup()*. Exactly how these two functions behave can vary somewhat depending on the kinds of values they may store, how many values they may store, what they do when an attempt is made to insert a symbol that is already in the symbol table and what they do if a lookup cannot find a symbol.

In the context of writing an assembler, one obvious use of a symbol table is to record the locations of variables and the destinations of jumps. To support this we need to be able to create a symbol table that can record an association between variable names or labels (string) with an address (int). The ***symbols\_int* (S)** class attached below provides a symbol table for associating string values to int values.

Another use of symbol tables in an assembler is to record the bit patterns that are associated with components of an instruction. For example, if a C-instruction contains a jump less than command, "JLT", then we need to include the bit pattern "100" (bits d1,d2,d3) in the assembled instruction. If a C-instruction includes the arithmetic operation, "D+M", then we need to include the bit pattern "1000010" (bits a,c1,c2,c3,c4,c5,c6) in the assembled instruction. A symbol table is a convenient way of recording which bit pattern implements each token that makes up an instruction. In this case we have a choice of how to represent the bit patterns. We could use int values to record the bit patterns ("100" = 4 or 0x4 or 04, "1000010" = 66 or 0x42 or 0102) and make use of left shift (**<<**) and bitwise or (**|**) operations to assemble instructions. Alternatively, we could record the bit patterns as strings of 0s and 1s. The ***symbols\_string* (S)** class attached below provides a symbol table for associating string values to string values.

Symbol tables can also be used in tokenisers when different kinds of tokens may be formed using the same rules. For example, a reserved word in most programming languages satisfies the lexical rules for an identifier. To deal with this, most tokenisers do not look for reserved words directly, rather they check each identifier just in case it is actually a reserved word. A symbol table is a convenient way of remembering which names are actually reserved words.

**Implementing symbols\_string**

Now that you know how to run the main program and produce XML output, you should attempt to write your own ***symbols\_string*** class that can be used by the tokeniser. To do this you need to complete the skeleton implementation provided in the file ***asmsymbols.cpp***.

**Testing Your Symbol Table**

Once you have something that might work, compile your symbol table using this command:

% make

Once your symbol table compiles, you can run it using either the **XT** program or, if you have a working tokeniser, using the **X** program, and hopefully you will see:

% echo "D=1" | ./XT

<tokens>

<dest-comp?>D</dest-comp?>

<equals>=</equals>

<comp>1</comp>

</tokens>

Number of tokens read:

0000000000000011

**The Precompiled Classes**

**Class asmtokens**

***asmtokens*** **(T)**has a default constructor and two public member functions named *next\_token()* and *token\_value()* that returns a string. Each time *next\_token()* is called the next token read from standard input (**cin**) is returned as a string value. Tokens are only returned as a category name, the specific characters that make up the token are obtained by calling *token\_value().* Incomplete tokens finish when a white space character (spaces, tabs, carriage returns and newlines) or the start of a comment is encountered. All white space and comments are ignored. Single line comments start with "**//**", and end at the end of the current line. Multi line comments start with "**/\***" and end at the first occurrence of "**\*/**" after the starting "**/**" character. Once the end of the input is reached or a legal token cannot be formed from the next input, both*next\_token()* and *token\_value()*, and all future calls to these functions, will return the string "**?**".

**Classes symbols\_int and symbols\_string**

The ***symbols\_int*(S)**and ***symbols\_string*(S)**classes have a static function *newtable()* that returns a pointer to a new empty symbol table. All other public member functions are pure virtual and the actual class implementing the symbol table is hidden. A new symbol can be added to a symbol table using *insert()* which takes the name of the symbol and its associated value. If the symbol is not present in the symbol table it will be added together with the value and then the *insert()* function will return**true**. If the symbol is already in the symbol table the *insert()* function returns **false**. The value associated with a symbol can be retrieved using the *lookup()* function. If the symbol is present in the table its associated value will be returned. If the symbol is not present then the value **-1** or **""** will be returned as appropriate. The *display()*function can be used to display the contents of a symbol table for the purposes of debugging and testing. Finally, the *deleteme()* function is provided to avoid problems with attempting to use **delete** on a pointer to an abstract class.

**Class asmxml**

***asmxml*(X)**has a default constructor and three public member functions, *open\_node()*, *node\_text()* and *close\_node()* each of which takes a string parameter and writes some text to standard ouput (**cout**). *open\_node()* takes the name of an XML tag as its parameter and writes out the tag names enclosed in "**<**" and "**>**". *close\_node()* takes the name of an XML tag as its parameter and writes out the tag name enclosed in "**</**" and "**>**". *node\_text()* simply writes out its parameter after translating the characters**<**, **>**, **&**, **'** and **"** into **&lt;**, **&gt;**, **&amp;**, **&apos;** and **&quot** respectively. *open\_node()* and *close\_node()* remove these characters from their parameters before printing their output.

**Class asmcode**

***asmcode*(X)**has a default constructor and two public member functions named *output\_0s\_and\_1s()* and *output\_16bits()*. The function *output\_0s\_and\_1s()* takes a string parameter that is assumed to be a string consisting of exactly sixteen 0s or 1s that represents a **Hack** machine instruction. This string will be written to standard output (**cout**) followed by an end of line marker (**endl**). The function *output\_16bits()* takes an int parameter and it is assumed that the least significant 16 bits of the int are a **Hack** machine instruction. This 16 bit value is written to standard output (**cout**) as a string of 0s and 1s followed by an end of line marker (**endl**).