# Workshop 06 - Tokeniser

### Writing aTokeniser

##### [Pracmarker Link](https://cs.adelaide.edu.au/services/pracmarker/)

In this workshop you are going to implement a single class named ***w6tokens***, that will be used to break a text file into tokens. Your implementation of ***w6tokens*** will be used by a driver program, ***w6tokeniser***, to display a list of the tokens found in a given source file.

#### Step 1 - Download, compile and run the precompiled classes

The first exercise is to download, compile and run a copy of the ***w6tokeniser*** program that has a skeleton implementation of the class ***w6tokens***. The skeleton always returns "**?**" instead of actually reading any tokens.

First download the zip file attached below.

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

% make

This will compile the skeleton file ***w6tokens.cpp*** and link it to precompiled ***.o*** files in order to create an executable program named ***w6tokeniser***.

Now run ***w6tokeniser*** and give it some text input using pipes or shell redirection. Here is an example of the output produced when a single line of Hack Assembly language is input:

% echo "MD=A" | ./w6tokeniser

tokens:

Number of tokens read: 0

This does not work correctly because the skeleton implementation of the tokeniser always returns "**?**" as the next token.

#### Step 2 - Write the tokeniser

Now that you know how to compile and run the main program, you should attempt to write your own tokeniser to recognise the tokens shown below. To do this you need to complete the skeleton implementation provided in the file **w6*tokens.cpp***.

There are two restrictions on your implementation of the **w6*tokens*** class. Firstly, you must read the input by calling the function ***nextch()*** that is passed to the ***w6tokens*** constructor. You **must not** use any other form of input. Secondly, you must only recognise the *name* and number tokens and use an internal lookup table to match a name token to a legal *jump* or *null* token value. When you look for the start of the next token, any character that cannot start a name or number is ignored. The skeleton implementation provided in the file **w6*tokens.cpp*** creates the lookup table and populates it.

#### Tokens to Recognise

The token classes that you must recognise are as follows:

| **Token** |  | **Definition** | **Example Token** | **Token Value** |
| --- | --- | --- | --- | --- |
| "name" | ::= | letter ( letter | digit )\* | **\_he:82m.Uch$** | "\_he:82m.Uch$" |
| "number" | ::= | digit digit\* | **99** | "99" |
| "jump" | ::= | 'JMP' | 'JLT' | 'JLE' | 'JGT' | 'JGE' | 'JEQ' | 'JNE' | **JGT** | "JGT" |
| "null" | ::= | 'NULL' | **nULl** | "NULL" |
| **Additional Rules** |  | **Definition** | **Example Text** |  |
| letter | ::= | 'a'-'z' | 'A'-'Z' | '$' | '\_' | ':' | '.' | "$" | |
| digit | ::= | '0'-'9' | "1" | |

##### Notes:

* letter and digit are never returned as token classes
* all letters 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, ignore all characters that cannot start a name or number token
* when the end of input is reached or an error occurs, return the token "**?**" with value "**?**"
* 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 and hopefully you will see (the "**=**" and "**;**" are ignored, "**Jmp**" is converted to "**JMP**"):

% echo "Dog=1;Jmp" | ./w6tokeniser

Tokens:

name: Dog

number: 1  
jump: JMP

Number of tokens read: 3

#### Step 3 - Implement your own lookup table

Lookup tables can 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 look for identifiers and then check each identifier just in case it is actually a reserved word. A lookup table is a convenient way of remembering which names are actually reserved words.

#### Implementing lookup

Now that you have a working tokeniser, you should attempt to write your own lookup table to replace the one in the skeleton **w6tokens** class. This means that you need to change any calls to the functions ***w6insert()*** and **w6lookup()** to be calls to your new functions.

#### Testing Your Lookup Table

Once your lookup table is working you can recompile the ***w6tokeniser*** program and it should behave exactly as it did before. If you created separate **.cpp** and **.h** files for a lookup table class you may need to modify the **Makefile**.

% echo "Dog=1;Jmp" | ./w6tokeniser

Tokens:

name: Dog

number: 1  
jump: JMP

Number of tokens read: 3