# Workshop 08 - Virtual Machine Translation

### The Jack Virtual Machine

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

#### Translating VM Code

Completing all of this workshop would be the equivalent of projects 07 and 08. However, just attempting to complete the steps for a small subset of the VM language should expose you to the keys ideas required to translate Jack Virtual Machine Code into Hack Assembly Language.

To simplify the task we have provided two precompiled classes named ***vmtokens***, that implements a tokeniser for the VM language and ***vmxml***, that can be used to produce XML formatted output. You do not need to provide your own implementations of either of these classes, just use the precompiled versions to implement your VM Code translator.

#### Step 1 - Download, compile and run the VM Code tokeniser

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

First download the zip file attached below.

After expanding the file compile the programs using the command:

% make

This will compile the files **tokens.cpp**, **parser.cpp** and **translator.cpp** using the precompiled implementations of the above classes and produce three executable programs, **tokens**, **parser** and **translator**. Initially all three **.cpp** files are identical and print out Jack VM code tokens in XML.

Now run **tokens**and give it some VM code input using pipes or shell redirection. Here is an example that is provided a single VM command as input:

% echo "push constant 43" | ./tokens

<tokens>

<stack>push</stack>

<segment>constant</segment>

<number>43</number>

</tokens>

Number of tokens read: 3

##### Recognised Tokens

The following table describes the structure of the tokens that are recognised:

|  |  |  |
| --- | --- | --- |
| **Token** |  | **Definition** |
| op | ::= | 'add' | 'and' | 'eq' | 'gt' | 'lt' | 'neg' | 'not' | 'or' | 'sub' | 'return' |
| jump | ::= | 'goto' | 'if-goto' | 'label' |
| func | ::= | 'call' | 'function' |
| stack | ::= | 'pop' | 'push' |
| label | ::= | ('a'-'z' | 'A'-'Z' | '\_' | ':' | '.')('a'-'z' | 'A'-'Z' | '0'-'9' | '\_' | ':' | '.')\* |
| segment | ::= | 'argument' | 'constant' | 'local' | 'pointer' | 'static' | 'temp' | 'that' | 'this' |
| number | ::= | ('0'-'9')('0'-'9')\* |

##### Example Input

push constant 5

push static 6

add

##### Example Output

<tokens>

<stack>push</stack>

<segment>constant</segment>

<number>5</number>

<stack>push</stack>

<segment>static</segment>

<number>6</number>

<op>add</op>

</tokens>

#### Step 2 - Parsing VM Code

Write a parser that can read Hack Virtual Machine Code using the precompiled classes and produce a list of VM commands represented as XML. You may assume that the Virtual Machine Code is correct. Edit the **parser.cpp** file to complete this step of the workshop. You can limit yourself to just generating XML for the example.

The following table describes the structure of the commands:

|  |  |  |
| --- | --- | --- |
| **Node** |  | **Definition** |
| class | ::= | (an-op | a-jump | a-func | a-stack)\* eof |
| an-op | ::= | op |
| a-jump | ::= | jump label |
| a-func | ::= | func label number |
| a-stack | ::= | stack segment number |

##### Example Input

push constant 5

push static 6  
add

##### Example Output

<class>

<a-stack>

<stack>push</stack>

<segment>constant</segment>

<number>5</number>

</a-stack>  
 <a-stack>  
 <stack>push</stack>

<segment>static</segment>  
 <number>6</number>

</a-stack>  
 <an-op>

<op>add</op>

</an-op>

</class>

#### Step 3 - VM Translator.

Write a translator that can read Jack Virtual Machine Code using the parser in Step 2 and produce the equivalent Hack Assembly Code. You may assume that the Virtual Machine Code is correct. Edit the **translator.cpp** file to complete this step of the workshop. You can limit yourself to just generating code for the example.

Notes:

* Assume that the code is being read from a file named ***SS.vm***.
* Assume that the initial function name is ***SS.Unknown***.
* Each VM command must be output as a comment on the line before the instructions that implement it.
* Function names are translated into an assembly language label of the same name.
* VM label names are translated into an assembly language label that is prefixed by the name of the current function and the character '$', eg inside the VM function ***SS.FF***, the label ***LL*** would be translated into ***SS.FF$LL***.
* Static variables, referred to by offsets into the static segment, are translated in an assembly language symbol that prefixes the offset with the current filename and the character '.', eg inside the file ***SS.vm***, references to ***static 6*** would be translated into references to ***SS.6***.

##### Example Input

push constant 5

push static 6  
add

##### Example Output

// push constant 5

@5

D=A

@SP

AM=M+1

A=A-1

M=D

// push static 6  
 @SS.6

D=M

@SP

AM=M+1

A=A-1

M=D

// add

@SP

AM=M-1

D=M

A=A-1

M=D+M