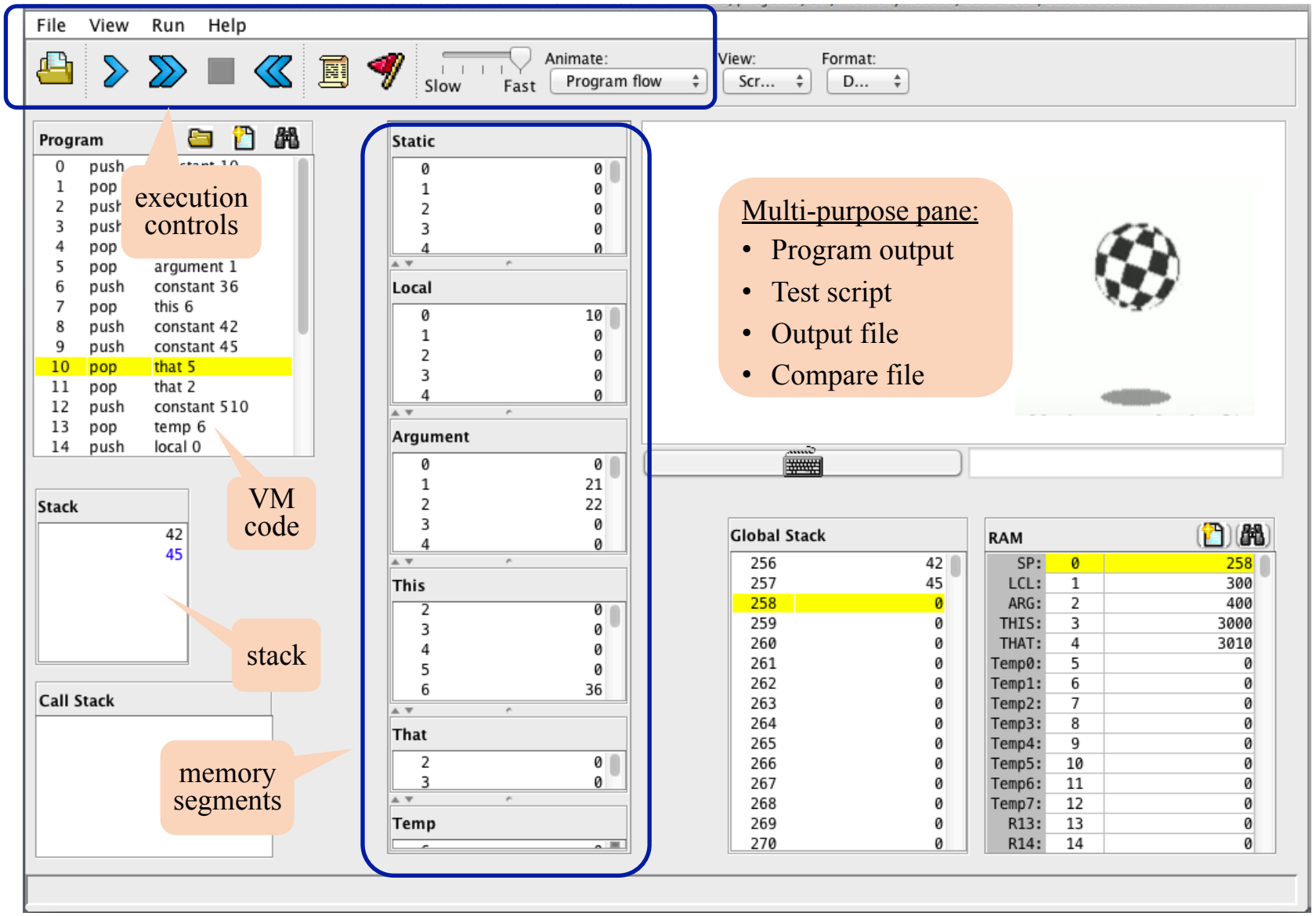
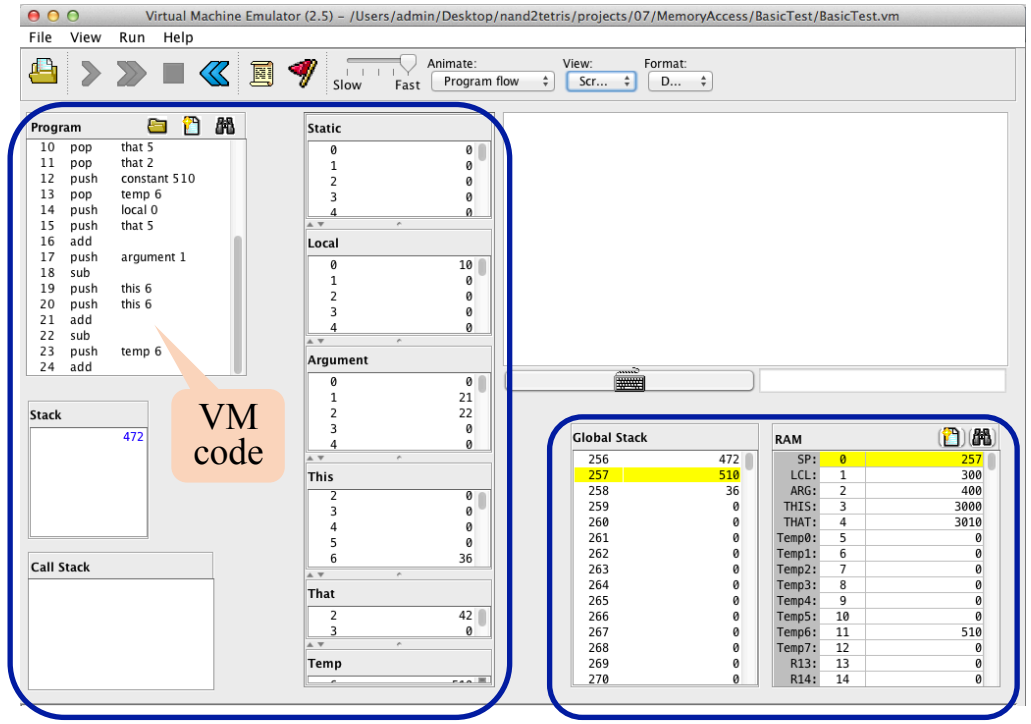
The VM Emulator



Example

BasicTest.vm

```
...
push constant 10
pop local 0
push constant 21
push constant 22
pop argument 2
pop argument 1
push constant 36
pop this 6
...
```



abstraction

implementation

Things to watch for:

(during the code's execution)

- Stack state
- Memory segments states

How the stack and memory segments are realized on the host platform

Test script

BasicTest.vm

```
...
push constant 10
pop local 0
push constant 21
push constant 22
pop argument 2
pop argument 1
push constant 36
pop this 6
...
```

BasicTestVME.tst

```
load BasicTest.vm,
output-file BasicTest.out,
compare-to BasicTest.cmp,
output-list RAM[256]%D1.6.1 RAM[300]%D1.6.1 ...




repeat 25 {        // BasicTest.vm has 25 instructions
  vmstep;
}

// Outputs some values, as specified by the output-list
// (stack base + selected values from the tested mem. segments)
output;
```

There's no need to delve into the code of test scripts

BasicTest.out

```

```

BasicTest.cmp

```
|RAM[256]|RAM[300]|RAM[401]|RAM[402]|RAM[3006|RAM[3012|RAM[3015|RAM[11] |
|    472 |     10 |     21 |     22 |     36 |     42 |     45 |    510 |
```

Test script

BasicTest.vm

```
...
push constant 10
pop local 0
push constant 21
push constant 22
pop argument 2
pop argument 1
push constant 36
pop this 6
...
```

BasicTestVME.tst

```
load BasicTest.vm,
output-file BasicTest.out,
compare-to BasicTest.cmp,
output-list RAM[256]%D1.6.1 RAM[300]%D1.6.1 ...




repeat 25 {        // BasicTest.vm has 25 instructions
  vmstep;
}

// Outputs some values, as specified by the output-list
// (stack base + selected values from the tested mem. segments)
output;
```

There's no need to delve into the code of test scripts

if (.out == .cmp)
  the test is successful
else
  error

BasicTest.out

```
|RAM[256]|RAM[300]|RAM[401]|RAM[402]|RAM[3006|RAM[3012|RAM[3015|RAM[11] |
|    472 |     10 |     21 |     22 |     36 |     42 |     45 |    510 |
```

BasicTest.cmp

```
|RAM[256]|RAM[300]|RAM[401]|RAM[402]|RAM[3006|RAM[3012|RAM[3015|RAM[11] |
|    472 |     10 |     21 |     22 |     36 |     42 |     45 |    510 |
```

Some missing elements

BasicTest.vm

```
function Foo.bar
...
push constant 10
pop local 0
push constant 21
push constant 22
pop argument 2
pop argument 1
push constant 36
pop this 6
...
return
```

BasicTestVME.tst

```
load BasicTest.vm,
output-file BasicTest.out,
compare-to BasicTest.cmp,
output-list RAM[256]%D1.6.1 RAM[300]%D1.6.1 ...

set sp 256,        // stack pointer
set local 300,     // base address of the local segment
set argument 400,  // base address of the argument segment
set this 3000,     // base address of the this segment
set that 3010,     // base address of the that segment

repeat 25 {        // BasicTest.vm has 25 instructions
  vmstep;
}

// Outputs some values, as specified by the output-list
// (stack base + selected values from the tested memory segments)
output;
```

There's no need to delve into the code of test scripts

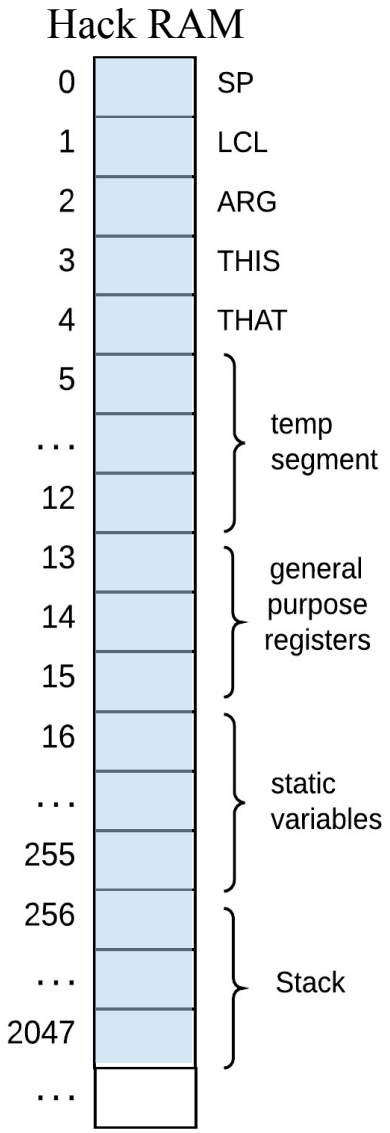
Initialization is handled manually by the supplied test script

Some missing elements

- `function` / `return` "envelope"
- Initializing the stack and the memory segments on the host RAM

  (both will be added in project 8)

Standard VM mapping on the Hack platform



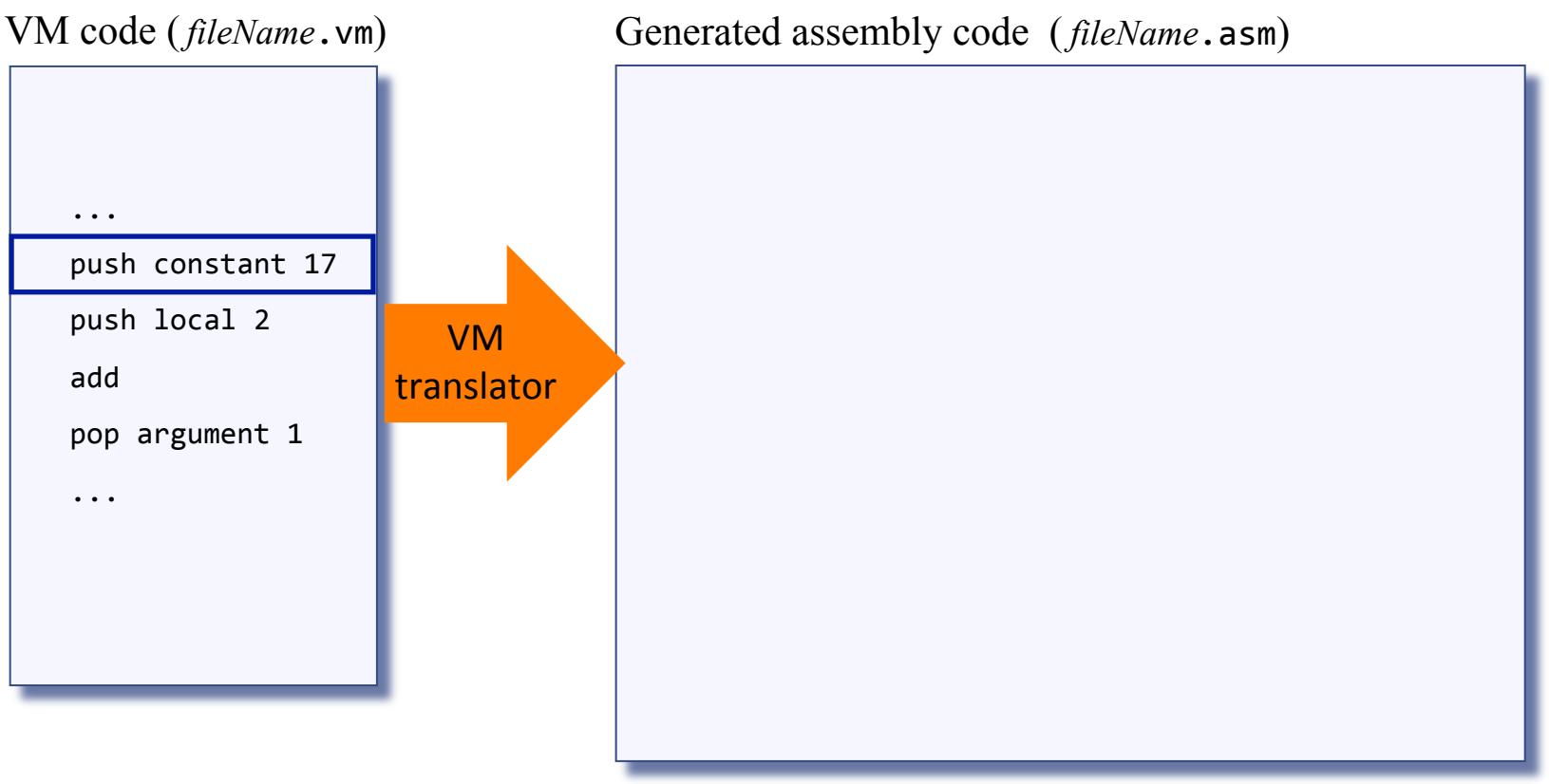
In order to realize this mapping, the VM translator should use some special variables / symbols:

Symbol	***Usage***
`SP`	This predefined symbol points to the memory address within the host RAM just following the address containing the topmost stack value.
`LCL`, `ARG`, `THIS`, `THAT`	These predefined symbols point, respectively, to the base addresses within the host RAM of the virtual segments `local`, `argument`, `this`, and `that` of the currently running VM function.
`R13`–`R15`	These predefined symbols can be used for any purpose.
`Xxx.`*i* symbols	The `static` segment is implemented as follows: each static variable *i* in file `Xxx.vm` is translated into the assembly symbol `Xxx.`*i*. In the subsequent assembly process, these symbolic variables will be allocated to the RAM by the Hack assembler.

Implementation note:

The standard mapping will be extended in project 8, when we'll complete the VM translator's implementation.

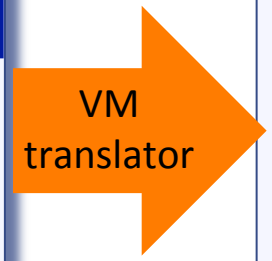
The VM translator



The VM translator

VM code (*fileName*.vm)

```
...
push constant 17
push local 2
add
pop argument 1
...
```



Generated assembly code (*fileName*.asm)

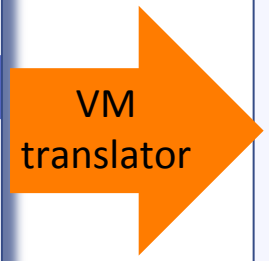
```
...
// push constant 17
@17
D=A
```

... additional assembly commands that complete the implementation of `push constant 17`

The VM translator

VM code (*fileName*.vm)

```
...
push constant 17
push local 2
add
pop argument 1
...
```



Generated assembly code (*fileName*.asm)

```
...
// push constant 17
@17
D=A
```

... additional assembly commands that complete the implementation of `push constant 17`

The VM translator

VM code (*fileName*.vm)

```
...
push constant 17
push local 2
add
pop argument 1
...
```

VM translator

Generated assembly code (*fileName*.asm)

```
...
// push constant 17
@17
D=A
... additional assembly commands that complete the
    implementation of push constant 17
// push local 2
... generated assembly code that implements push local 2
```

The VM translator

VM code (*fileName*.vm)

```
...
push constant 17
push local 2
add
pop argument 1
...
```

VM translator

Generated assembly code (*fileName*.asm)

```
...
// push constant 17
@17
D=A
... additional assembly commands that complete the
    implementation of push constant 17
// push local 2
... generated assembly code that implements push local 2
```

The VM translator

VM code (*fileName*.vm)

```
...
push constant 17
push local 2
add
pop argument 1
...
```

VM translator

Generated assembly code (*fileName*.asm)

```
...
// push constant 17
@17
D=A
... additional assembly commands that complete the
    implementation of push constant 17
// push local 2
... generated assembly code that implements push local 2
// add
... generated assembly code that implements add
```

The VM translator

VM code (*fileName*.vm)

```
...
push constant 17
push local 2
add
pop argument 1
...
```

VM translator

Generated assembly code (*fileName*.asm)

```
...
// push constant 17
@17
D=A
... additional assembly commands that complete the
    implementation of push constant 17
// push local 2
... generated assembly code that implements push local 2
// add
... generated assembly code that implements add
```

The VM translator

VM code (*fileName*.vm)

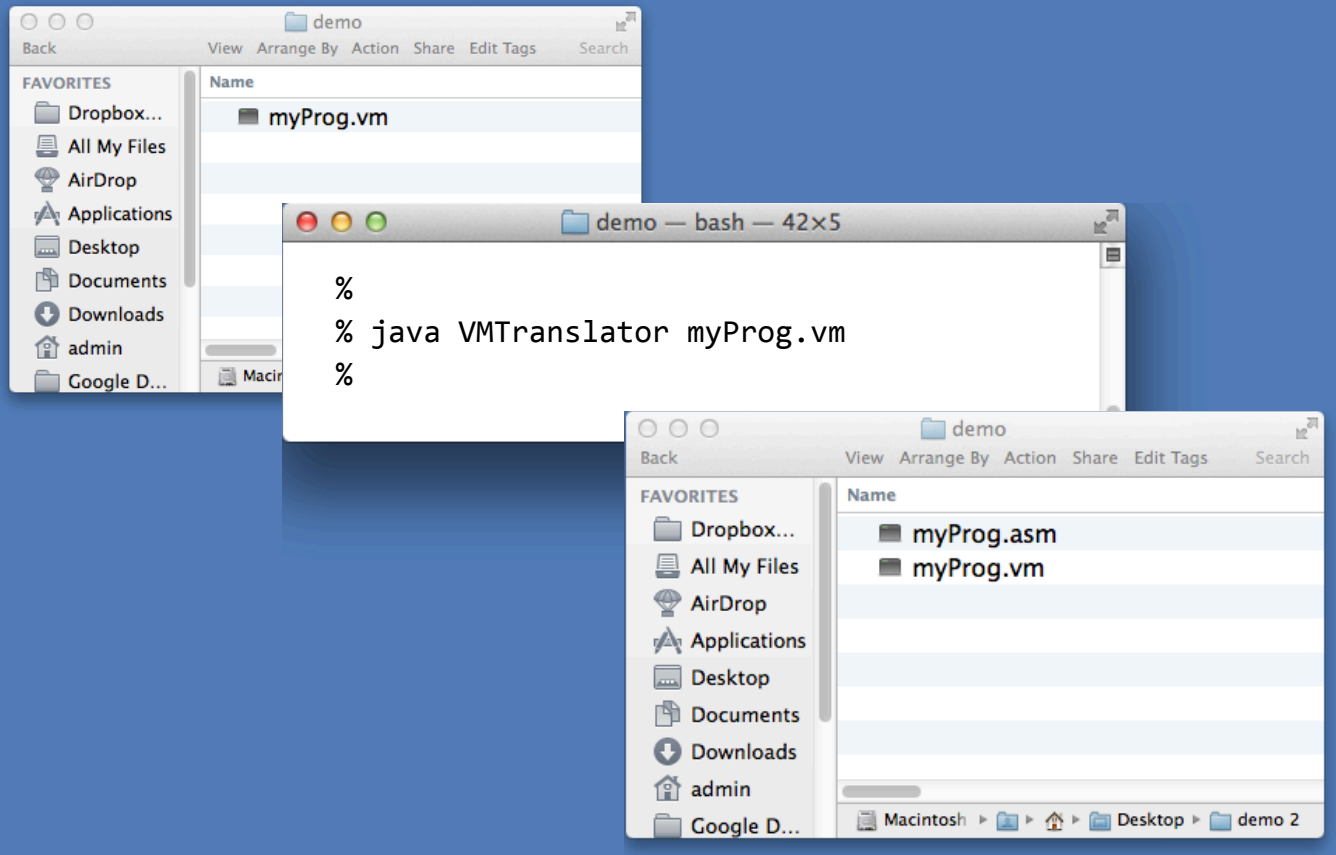
```
...
push constant 17
push local 2
add
pop argument 1
...
```

VM translator

Generated assembly code (*fileName*.asm)

```
...
// push constant 17
@17
D=A
... additional assembly commands that complete the
    implementation of push constant 17
// push local 2
... generated assembly code that implements push local 2
// add
... generated assembly code that implements add
// pop argument 1
... generated assembly code that implements push argument 1
...
```

The VM translator: usage



Implementation

Proposed design:

- `Parser`: parses each VM command into its lexical elements
- `CodeWriter`: writes the assembly code that implements the parsed command
- `Main`: drives the process (`VMTranslator`)

`Main` (`VMTranslator`)

Input: *fileName*`.vm`

Output: *fileName*`.asm`

`Main` logic:

- Constructs a `Parser` to handle the input file
- Constructs a `CodeWriter` to handle the output file
- Marches through the input file, parsing each line and generating code from it

Parser

- Handles the parsing of a single `.vm` file
- Reads a VM command, parses the command into its lexical components, and provides convenient access to these components
- Ignores all white space and comments

Routine	**Arguments**	**Returns**	**Function**
Constructor	Input file / stream	—	Opens the input file/stream and gets ready to parse it.
`hasMoreCommands`	—	Boolean	Are there more commands in the input?
`advance`	—	—	Reads the next command from the input and makes it the *current command*. Should be called only if `hasMoreCommands()` is true. Initially there is no current command.

Parser

- Handles the parsing of a single `.vm` file
- Reads a VM command, parses the command into its lexical components, and provides convenient access to these components
- Ignores all white space and comments

Routine	Arguments	Returns	Function
`commandType`	—	`C_ARITHMETIC`, `C_PUSH`, `C_POP`, `C_LABEL`, `C_GOTO`, `C_IF`, `C_FUNCTION`, `C_RETURN`, `C_CALL`	Returns a constant representing the type of the current command. `C_ARITHMETIC` is returned for all the arithmetic/logical commands.
`arg1`	—	string	Returns the first argument of the current command. In the case of `C_ARITHMETIC`, the command itself (`add`, `sub`, etc.) is returned. Should not be called if the current command is `C_RETURN`.
`arg2`	—	int	Returns the second argument of the current command. Should be called only if the current command is `C_PUSH`, `C_POP`, `C_FUNCTION`, or `C_CALL`.

CodeWriter

Generates assembly code from the parsed VM command:

Routine	**Arguments**	**Returns**	**Function**
Constructor	Output file / stream	—	Opens the output file / stream and gets ready to write into it.
`writeArithmetic`	command (string)	—	Writes to the output file the assembly code that implements the given arithmetic command.
`WritePushPop`	command (`C_PUSH` or `C_POP`), `segment` (string), `index` (int)	—	Writes to the output file the assembly code that implements the given command, where command is either `C_PUSH` or `C_POP`.
Close	—	—	Closes the output file.

More routines will be added to this module in Project 8,
when we complete the implementation of the VM translator.

The big picture

VM language:

Arithmetic / Logical commands:

```
add
sub
neg
eq
gt
lt
and
or
not
```

Memory access commands:

`pop` *segment i*

`push` *segment i*

Project 7

Branching commands:

`label` *label*

`goto` *label*

`if-goto` *label*

Function commands:

`function` *functionName nVars*

`call` *functionName nArgs*

`return`

Project 8

Development Plan

Objective: build a basic VM translator that handles the VM language *stack arithmetic* and *memory access* (`push`/`pop`) commands

Contract

- Write a VM-to-Hack translator, conforming to the *Standard VM-on-Hack Mapping*
- Use your VM translator to translate and test the supplied `.vm` programs, yielding corresponding `.asm` programs
- When executed on the supplied CPU emulator, the generated `.asm` programs should deliver the same results mandated by the supplied test scripts and compare files.

Test programs

- `SimpleAdd`
- `StackTest`
- `BasicTest`
  - `BasicTest.vm`
  - `BasicTest.tst`
  - `BasicTest.cmp`
  - `BasicTestVME.tst`
- `PointerTest`
- `StaticTest`

`BasicTest.vm` (example)

```
...
push constant 510
pop temp 6
push local 0
push that 5
add
push argumer
sub
...
```

`BasicTest.asm`

```
...
// push constant 510
@510
D=A
...
```

Development Plan

Objective: build a basic VM translator that handles the VM language
stack arithmetic and *memory access* (`push`/`pop`) commands

For each test *xxx*`.vm` program:

0. (optional) load *xxx*`VME.tst` into the VM emulator; run the test script and inspect the program's operation
1. use your translator to translate *xxx*`.vm`; The result will be a file named *xxx*`.asm`
2. inspect the generated code; If there's a problem, fix your translator and go to stage 1
3. Load *xxx*`.tst` into the CPU emulator
4. Run the test script, inspect the results
5. If there's a problem, fix your translator and go to stage 1.

Test programs

- `SimpleAdd`
- `StackTest`
- `BasicTest`
  - `BasicTest.vm`
  - `BasicTest.tst`
  - `BasicTest.cmp`
  - `BasicTestVME.tst`
- `PointerTest`
- `StaticTest`

`BasicTest.vm` (example)

```
...
push constant 510
pop temp 6
push local 0
push that 5
add
push argumer
sub
...
```

`BasicTest.asm`

```
...
// push constant 510
@510
D=A
...
```

Perspective

(A subset of historical notes and additional issues)

- History of VMs and two-tier compilation:
  - p-code
  - Sun
  - Cellphones
- How close is our VM to Java's JVM?
- Efficiency and optimization
- Different VM implementations:
  - Stack machine
  - Register machine
  - Other approaches.