CS210-001: Assignment 4
Fall 2019 (201930)

Due Date and Time: Thursday, October 10, 2019 at 11:20 AM

(100 marks) A *stack machine* is a simple implementation model for executing computer programs. In fact, in some development environments, there are compilers that translate a high-level computer language into a set of instructions to be executed using the stack machine model (e.g., the Java virtual machine instruction set). A stack machine consists of a stack, instructions, operands, a symbol table, and an evaluation model.

Assume that we have a stack machine that supports the following instructions from Hilderman's Algorithm Language (known as `HAL`) and the associated evaluation criteria for each instruction:

a. `declare` *x*: Inserts an integer variable named *x* in the symbol table and initializes it to zero (e.g., `declare length`). A variable must be declared before it can be used in an instruction. A valid variable name may consist of letters and numbers, but it must start with a letter.
b. `read`: Pushes a literal integer value read from the keyboard onto the stack (e.g., `read`).
c. `push` *a*: Pushes the literal integer value *a* onto the stack (e.g., `put 8`).
d. `get` *x*: Locates the variable *x* in the symbol table and pushes its value onto the stack (e.g., `get length`).
e. `add`: Pops the top two values off the stack, adds them, and pushes the result back onto the stack (e.g., `add`).
f. `multiply`: Pops the top two values off the stack, multiplies them, and pushes the result back onto the stack (e.g., `multiply`).
g. `subtract`: Pops the top two values off the stack, subtracts the second operand from the first, and pushes the result back onto the stack (e.g., `subract`).
h. `divide`: Pops the top two values off the stack, divides the first operand by the second, and pushes the result back onto the stack (e.g., `divide`).
i. `set` *x*: Pops the top value off the stack and assigns it to the variable *x* in the symbol table.
j. `write`: Pops the top value off the stack and writes it to the screen at the current cursor position (e.g., `write`).
k. `writestring` *s*: Writes the string *s* to the screen at the current cursor position (e.g., `writestring The volume of the box is:`).
l. `writenl`: Writes a new line character to the screen and positions the cursor at the beginning of the next line (e.g., `writenl`).
m. `compare`: Pops the top two values off the stack and compares them (e.g., `compare`). The result of the comparison is stored in a special integer variable named `status`. Specifically, if the two values are equal, it assigns `1` to `status`. Otherwise, it assigns `0`.

n. `jumpequal` *a*: If the value of `status` is `1`, it assigns the value `-1` to `status` and branches to the instruction indicated by the literal integer value *a* (e.g., `jumpequal 3`). Otherwise, it just assigns the value `-1` to `status`.
o. `jump` *a*: Unconditionally branches to the instruction indicated by the literal integer value *a* (e.g., `jump 3`).
p. `end`: Stops the program.

As you can see from the instructions described above, the operands are limited to variable names and literal integer values (except for `writestring`). All arithmetic operations are integer operations. Together, all of the instructions and the associated evaluation criteria describe the evaluation model.

PART 1

Implement a program that uses a stack machine to interpret a program written in the `HAL` instructions given above. The input to your interpreter program will be a file containing a `HAL` program. Your interpreter program should prompt the user for the name of the file containing the `HAL` program. You can assume the file contains a complete and correct `HAL` program. That is, you don't need to worry about handling syntax errors. An example `HAL` program is shown below. Here, each line of the program is associated with a unique, sequential integer (i.e., the line number), where the first line number is `0`.

```
0 declare a
1 declare b
2 declare c
3 push 3
4 set b
5 get a
6 read
7 add
8 set a
9 push 1
10 get b
11 subtract
12 set b
13 get b
14 get c
15 compare
16 jumpequal 18
17 jump 5
18 writestring the sum is:
19 get a
20 write
21 writenl
22 end
```

Before interpreting a HAL program, the instructions in the file should be read into a program array implemented as an *unsorted array-based list* defined as a C++ class, where each element consists of a `struct` containing two members: (1) a `string` operation member and (2) a `char*` operand member. For example, the following program array contains the example HAL program shown above. Here, each line of the program is associated with a unique, sequential integer (i.e., the array index doubles as the line number, so the line number from the input file does not need to be stored in the program array). A special variable of type `int` named `pc` (i.e., program counter) should be used to indicate the next instruction to execute.

0	declare	->	a
1	declare	->	b
2	declare	->	c
3	push	->	3
4	set	->	b
5	get	->	a
6	read	Ø	
7	add	Ø	
8	set	->	a
9	push	->	1
10	get	->	b
11	subtract	Ø	
12	set	->	b
13	get	->	b
14	get	->	c
15	compare	Ø	
16	jumpequal	->	18
17	jump	->	5
18	writestring	->	the sum is:
19	get	->	a
20	write	Ø	
21	writenl	Ø	
22	end	Ø	
⋮	⋮	⋮	⋮

A HAL program always starts executing at line number `0`. Prior to executing an instruction, your interpreter should print the line number to the screen. For example, when interpreting the sample HAL program given above, your interpreter would print

`0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 5 6 7 8 9 ...`
meanwhile, a miracle happens `... 22`

The symbol table should be implemented as a *sorted array-based list* defined as a C++ class, where each element consists of a `struct` containing two members: (1) a `string` symbol member and (2) an `int` literal integer value member. The following array

contains the symbol table generated by interpreting the first three lines of the sample `HAL` program.

a	0
b	0
c	0
⋮	⋮

The stack should be implemented using as an *array-based stack* defined as a C++ class, where each element contains only literal integer values. Values are pushed onto the stack and popped off the stack according to the execution model. The following array contains the stack generated by interpreting the first seven lines of the sample `HAL` program (this assumes the value read from the keyboard was `10`).

⋮
10
0

Your C++ classes should be implementations of the algorithms given on the CS210 Algorithms web page. Do not waste your time implementing any algorithms that you don't actually need.

The program `usefulCodeMaybe.cpp` from the directory `/home/venus/hilder/cs210/assignment4/datafiles` should give you some idea as to how your program array might be used. Note that this program is not "the answer".

PART 2

For programming problems, the Results are worth 70%. So, for this problem, the Results are worth 70 marks out of the 100 marks available. Demonstrate that your interpreter works by interpreting a `HAL` program that you have written yourself. Your `HAL` program will prompt the user to enter a test score from the keyboard and then read the test score from the keyboard. If the test score is not `-1`, your `HAL` program must add the test score to the total score and prompt for more input. If the test score is `-1`, your `HAL` program must stop prompting for input, print the total score and the number of test scores entered, calculate the average of the test scores, write the average to the screen, and terminate. Your demonstration should be captured in script files.

Use the following three sequences of test scores as input to your `HAL` program:

Test 1: `77 86 54 95 62 100 81 73 39 88 52 67 45`

Test 2: `155 201 318 184 279 126`

Test 3: `3  9  14  2  8  11  5  10`

WHAT TO SUBMIT

If you are working alone, submit to UR Courses: (1) all your source code files (i.e., *only* the `.cpp` and `.h` files) zipped into a single file called `cppandhfiles`, (2) a file called `halprogram` containing your `HAL` program, (3) a script file called `test1script` showing the compilation and execution of your program using the Test 1 sequence as input, (4) a script file called `test2script` showing the compilation and execution of your program using the Test 2 sequence as input, and (5) a script file called `test3script` showing the compilation and execution of your program using the Test 3 sequence as input.

If you are working with a partner, *one of the partners* should submit to UR Courses: (1) a file named `partners` that provides the names and student numbers of the partners and the relative contributions of the partners (should total 100%), (2) all your source code files (i.e., *only* the `.cpp` and `.h` files) zipped into a single file called `cppandhfiles`, (3) a file called `halprogram` containing your `HAL` program, (4) a script file called `test1script` showing the compilation and execution of your program using the Test 1 sequence as input, (5) a script file called `test2script` showing the compilation and execution of your program using the Test 2 sequence as input, and (6) a script file called `test3script` showing the compilation and execution of your program using the Test 3 sequence as input. *The other partner* should submit to UR Courses: (1) a file named `partners` that provides the names and student numbers of the partners and the relative contribution of the partners (should total 100%). Note that *both* partners need to submit the `partners` file.