CSE 305 Introduction to programming languages
Spring 2016
Homework 2: Language interpreter design, part I

Assigned: Thursday, February 18th, 2016 Due: on or before Monday, March 7th, 11:59 pm

Overview

The goal of this homework is to understand and build an interpreter in three languages (Python, SML, Java, 20 marks for each) for a small language. Your interpreter should read in an input file (`input.txt`) which contains lines of expressions, evaluate them and push the results onto a stack, then print the content of the stack to an output file (`output.txt`) when exit. This homework is the first of a multi-part homework, you will be implementing some of the features. More interesting features will be introduced in the following homework.

Example input and output

Some examples of the input file your interpreter takes in and the corresponding output file are shown below:

input.txt	output.txt
push 1 quit	1
push 5 neg push 10 push 20 add quit	30 -5
push 10 push 2 push 8 mul add push 3 sub quit	23
push 6 push 2 div mul quit	:error: 3

input.txt	output.txt
:true: push 7 push 8 :false: pop sub quit	-1 :true:
pop push 10 swap push 6 add pop push -20 mul push 5 swap quit	-120 5 :error: 10

Functionality

Your interpreter should read in expressions from the input file named "`input.txt`", maintain a stack when running, and output the content of the stack to an output file named "`output.txt`" when stops. It should be able to handle the following expressions for this homework:

1. push

```
push _num_
```

where `_num_` is an integer possibly with a '-' suggesting a negative value. Here, '-0' should be regarded as '0'. Entering this expression will simply push `_num_` onto the stack. For example,

```
push 5
push -0
```

0
5

If `_num_` is not an integer, only push the error literal (`:error:`) onto the stack instead of pushing `_num_`. For example,

```
push 5
push 2.5
```

:error:
5

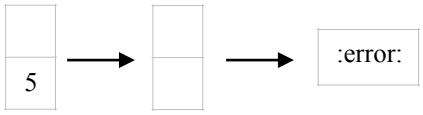
2. pop

```
pop
```

Remove the top value from the stack. If the stack is empty, an error literal (`:error:`) will be pushed onto the stack.

For example,

```
push 5
pop
pop
```

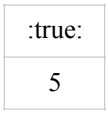


3. boolean

```
:true:
:false:
```

There are two kinds of boolean literals: `:true:` and `:false:`. Your interpreter should push the corresponding value onto the stack. For example,

```
push 5
:true:
```



4. error

```
:error:
```

Similar with boolean literals, entering error literal will push `:error:` onto the stack.

5. add

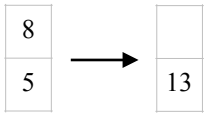
```
add
```

`add` refers to integer addition. Since this is a binary operator, it consumes the top two values in the stack, calculate sum and push the result back to the stack. If one of the following cases occurs, which means there is an error, any values popped out from the stack should be pushed back in the same order, then a value `:error:` should also be pushed onto the stack:

- not all top two values are integer numbers
- only one value in the stack
- stack is empty

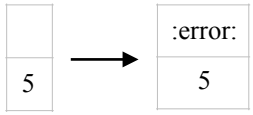
For example,

```
push 5
push 8
add
```



For another example, if there is only one number in the stack and we use `add`, an error will occur. Then 5 should be pushed back as well as `:error:`

```
push 5
add
```



6. sub

```
sub
```

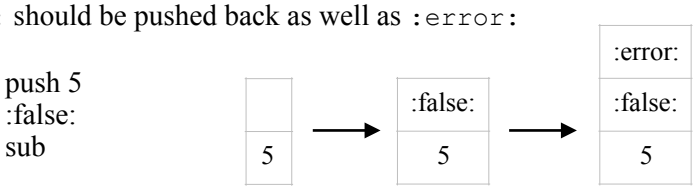
`sub` refers to integer subtraction. It is a binary operator and works in the following way:

- if top two elements in the stack are integer numbers, pop the top element(y) and the next element(x), subtract y from x, and push the result x-y back onto the stack
- if the top two elements in the stack are not all integer numbers, push them back in the same order and push `:error:` onto the stack

- if there is only one element in the stack, push it back and push `:error:` onto the stack
- if the stack is empty, push `:error:` onto the stack

For example,

For another example, if one of the top two values in the stack is not a numeric number when `sub` is used, an error will occur. Then 5 and `:false:` should be pushed back as well as `:error:`



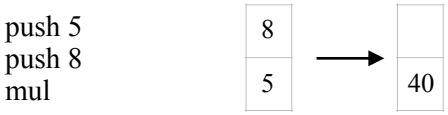
7. mul

`mul`

`mul` refers to integer multiplication. It is a binary operator and works in the following way:

- if top two elements in the stack are integer numbers, pop the top element(y) and the next element(x), multiply x by y, and push the result x*y back onto the stack
- if the top two elements in the stack are not all integer numbers, push them back in the same order and push `:error:` onto the stack
- if there is only one element in the stack, push it back and push `:error:` onto the stack
- if the stack is empty, push `:error:` onto the stack

For example,



For another example, if the stack empty when mul is used, an error will occur. Then :error: should be pushed onto the stack:

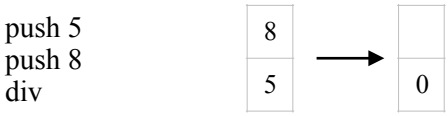
8. div

`div`

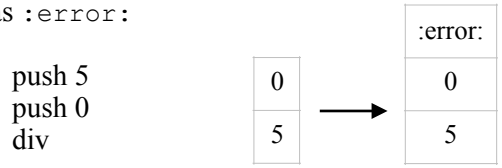
`div` refers to integer division. It is a binary operator and works in the following way:

- if top two elements in the stack are integer numbers, pop the top element(y) and the next element(x), divide x by y, and push the result x\y back onto the stack
- if top two elements in the stack are integer numbers but y equals to 0, push them back in the same order and push `:error:` onto the stack
- if the top two elements in the stack are not all integer numbers, push them back in the same order and push `:error:` onto the stack
- if there is only one element in the stack, push it back and push `:error:` onto the stack
- if the stack is empty, push `:error:` onto the stack

For example,



For another example, if the top element in the stack equals to 0, there will be an error if `div` is used. Then 5 and 0 should be pushed back as well as `:error:`



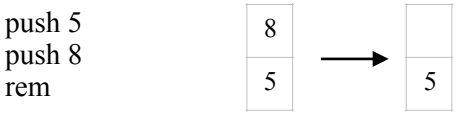
9. rem

`rem`

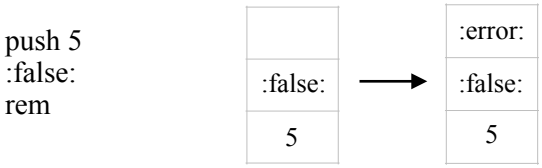
`rem` refers to the remainder of integer division. It is a binary operator and works in the following way:

- if top two elements in the stack are integer numbers, pop the top element(y) and the next element(x), calculate the remainder of x\y, and push the result back onto the stack
- if top two elements in the stack are integer numbers but y equals to 0, push them back in the same order and push `:error:` onto the stack
- if the top two elements in the stack are not all integer numbers, push them back and push `:error:` onto the stack
- if there is only one element in the stack, push it back and push `:error:` onto the stack
- if the stack is empty, push `:error:` onto the stack

For example,



For another example, if one of the top two elements in the stack is not an integer, an error will occur if `rem` is used. Then 5 and `:false:` should be pushed back as well as `:error:`



10. neg

`neg`

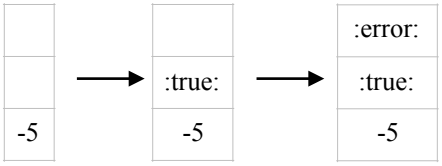
`neg` is to calculate the negation of an integer (negation of 0 should still be 0). It is unary therefore consumes only the top element from the stack, calculate its negation and push the result back. A value `:error:` will be pushed onto the stack if:

- the top element is not an integer, push the top element back and push `:error:`
- the stack is empty, push `:error:` onto the stack

For example,

For another example, if the top value is not an integer, when `neg` is used, it should be pushed back as well as `:error:`

```
push 5
neg
:true:
neg
```



11. swap

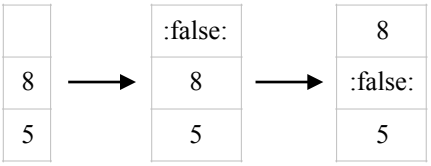
`swap`

`swap` interchanges the top two elements in the stack, meaning that the first element becomes the second and the second becomes the first. A value `:error:` will be pushed onto the stack if:

- there is only one element in the stack, push the element back and push `:error:`
- the stack is empty, push `:error:` onto the stack

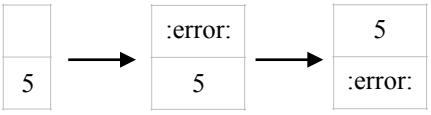
For example,

```
push 5
push 8
:false:
swap
```



For another example, if there is only one element in the stack when `swap` is used, an error will occur and `:error:` should be pushed onto the stack. Now we have two elements in the stack (5 and `:error:`), therefore the second swap will interchange the two elements:

```
push 5
swap
swap
```



12. quit

`quit`

`quit` causes the interpreter to stop. Then the whole stack should be printed out to an output file, named as "`output.txt`".

You can make the following assumptions:

- Expressions given in the input file are in correct formats. For example, there will not be expressions like "`push`", "`3`" or "`add 5`"
- No multiple operators in the same line in the input file. For example, there will not be "`pop pop swap`", instead it will be given as

  ```
  pop
  pop
  swap
  ```

- There will always be a "`quit`" in the input file to exit your interpreter and output the stack

Step by step examples

If your interpreter reads in expressions from “input.txt”, states of the stack after each operation are shown below:

First, push 10 onto the stack:

input.txt
push 10 push 15 push 30 sub :true: swap add pop neg quit

10

Similarly, push 15 and 30 onto the stack:

30
15
10

`sub` will pop the top two values from the stack, calculate 15-30 = -15, and push -15 back:

-15
10

Then push the boolean literal `:true:` onto the stack:

:true:
-15
10

`swap` consumes the top two values, interchanges them and pushes them back:

-15
:true:
10

`add` will pop the top two values out, which are -15 and `:true:`, then calculate their sum. Here, `:true:` is not a numeric value therefore push both of them back in the same order as well as an error literal `:error:`

:error:
-15
:true:
10

`pop` is to remove the top value from the stack, resulting in:

-15
:true:
10

Then after calculating the negation of -15, which is 15, and pushing it back, `quit` will terminate the interpreter and write the following values in the stack to "`output.txt`":

15
:true:
10

Now, please go back to the example inputs and outputs given before and make sure you understand how to get those results.

What to submit

Modification 02/19/2016

Define a function named 'hw2', which takes in two strings as input arguments(first one is the name of an input file and the second is the name of an output file). Function signatures of 'hw2' will be the same as they are specified in homework 1 (please see instructions in 'HW1_Modified.pdf'). **Do not include them in your submission**.

Create a folder `UBITName_HW2` which contains three sub folders: `Python`, `SML` and `Java`. The '`Python`' folder should contain '`hw2.py`', '`SML`' folder should contain '`hw2.sml`', and '`Java`' folder should contain '`hw2.java`'. There is no need for you to submit any input or output files in your folders.

Compress the `UBITName_HW2` folder to either '`UBITName_HW2.zip`' or '`UBITName_HW2.tar`', and submit it on Timberlake using the command `submit_cse305 your_file_name`.

Auto-grader will be used to grade your submission, so please follow the naming conventions mentioned above.

No late submissions will be accepted (unless you decide to use your "free days"). So please start early.

Modification 02/24/2016