Computer Science 320 (Spring, 2020)
Principles of Programming Languages

Programming Assignment 4: Interpreter (Part 1)

Out: Friday, March 20, 2020
Due: Friday, April 3, 2020, by 11:59 pm

In this assignment, and the two assignments following it, you will build an interpreter for a small, OCaml-like, stack-based bytecode programming language. Your interpreter will be implemented in OCaml. You will submit a file named `interpreter.ml` which contains a function, `interpreter`, with the following type signature:

```
val interpreter : string -> string -> unit
```

If your program does not match the type signature, it will not compile on Gradescope and you will receive 0 points. You may, however, have helper functions defined outside the function `interpreter`; the grader is only explicitly concerned with the type of `interpreter`.

Late submissions will not be accepted and will be given a score of 0. Test cases sample will be provided on Piazza for you to test your code locally. These will not be exhaustive, so you are highly encouraged to write your own tests to check your interpreter against all the functionality described in this document.

1 Preliminaries

This section contains background material for this programming assignment and the two following it. Read it carefully, and make sure you understand it, before you start on your assignment.

1.1 Functionality

Given the following function header:

```
let interpreter (input : string) (output : string ) : unit = ...
```

`input` file name and `output` file name will be passed in as strings that represent paths to files just like in the first programming assignment (Pangram). Your function should write the contents of the final stack which your interpreter produces to the file specified by `output`. In the five examples below, the input file is read from top to bottom (as shown in the left column of each of the five

boxes) and then each command is executed by your interpreter in the order it was read. It is very useful to read in all of the commands into a list prior to executing them, separating input from the actual interpretation of the commands. The input file can be arbitrarily long.

input	stack
PushI 1 Quit	1

input	stack
PushI 6 PushI 2 Div Mul Quit	<error> 3

input	stack
PushI 5 Neg PushI 10 PushI 20 Add Quit	 30 -5

input	stack
PushB <true> PushI 7 PushI 8 PushB <false> Pop Sub Quit	 1 <true>

input	stack
PushI 10 PushI 2 PushI 8 Mul Add PushI 3 Sub Quit	 -23

1.2 Grammar

The following is a context-free grammar for the bytecode language you will be implementing.[1] Terminal symbols are identified by `monospace font`, and nonterminal symbols are identified by *italic font*. Anything enclosed in square brackets, '[' and ']', denotes an optional character (zero or one occurrence). The form '(set_1 | set_2 | $\cdots$ | set_n)' means a choice of one character from any one of the n sets. A set enclosed in braces, '{' and '}', means zero or more occurrences.

- The set of decimal digits is {0,1,2,3,4,5,6,7,8,9}. Using the conventions of context-free grammars, we define the nonterminal symbol *digit* by the rule:

$$\textit{digit} ::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$$

- The set of all characters in the English alphabet (lowercase and uppercase) is $\{a, b, \ldots, A, B, \ldots\}$. The corresponding nonterminal symbol that generates all the English characters is defined by the following rule:

$$\textit{letter} ::= \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z}$$

- We call 'simple ASCII' The set of ASCII characters from which we omit '\' (back slash) and '"' (quotation marks). We denote by '*simpleASCII*' the nonterminal symbol that generates all simple-ASCII characters. Note that this necessarily implies that escape sequences will not need to be handled in your code.[2]

1.2.1 Constants

const ::= *int* | *bool* | *error* | *string* | *name* | *unit*

[1] A *context-free grammar* is the same as what is called a *BNF grammar* in the lecture slides.

[2] There are 128 ASCII characters, of which 95 are printable; the latter include the decimal digits $\{0, 1, \ldots, 9\}$ and the English characters $\{a, b, \ldots, z, A, B, \ldots, Z\}$. Hence, the nonterminal symbol *simpleASCII* generates all the characters generated by *digit* and *letter*, and many other.

int ::= [−] *digit* { *digit* }

bool ::= `<true>` | `<false>`

error ::= `<error>`

unit ::= `<unit>`

string ::= "{ *simpleASCII* }"

name ::= {_} *letter* {*letter* | *digit* | _}

1.2.2 Programs

prog ::= *com* {*com*} `Quit`

com ::= `PushI` *int* | `PushB` *bool* | `PushS` *string* | `PushN` *name* | `Push <unit>` |
`Add` | `Sub` | `Mul` | `Div` | `Rem` | `Neg` | `Swap` | `Pop` | `Concat` |
`And` | `Or` | `Not` | `LessThan` | `Equal` | `If` | `Bind` |
`Begin` *com* {*com*} `End` | *funBind com* {*com*} [`Return`] `FunEnd` | `Call` | `Quit`

funBind ::= (`Fun` | `InOutFun`) $name_1$ $name_2$

2 Basic Computations

Your interpreter should be able to handle the following commands:

2.1 Push

2.1.1 pushing Integers to the Stack

PushI *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,

input	stack
PushI 5	0
PushI -0	5

If *num* is not an integer, only Push the error literal (`<error>`) onto the stack instead of pushing *num*.

2.1.2 pushing Strings to the Stack

PushS *string*

where *string* is a string literal consisting of a sequence of characters enclosed in double quotation marks, as in "this is a string". Executing this command would Push the string onto the stack:

input	stack
PushS "deadpool" PushS "batman" PushS "this is a string"	this a string batman deadpool

Spaces are preserved in the string, i.e. any preceding or trailing whitespace must be kept inside the string that is Pushed to the stack:

input	stack
PushS " deadp ool " PushS "this is a string "	this␣is␣a␣string␣␣ ␣deadp␣ool␣

You can assume that the string value would always be legal and not contain quotations or escape sequences within the string itself, i.e. neither double quotes nor backslashes will appear inside a string.

2.1.3 pushing Names to the Stack

PushN *name*

where *name* consists of a sequence of letters, digits, and underscores, starting with a letter or underscore.

1. example

input	→	stack	→	stack
PushN a PushI 13		a		13 a

2. example

input	→	stack	→	stack
PushN __name1__ PushI 3		__name1__		3 __name1__

To bind 'a' to the value 13 and __name1__ to the value 3, we will use the 'Bind' operation (which we will see in the second part of the interpreter, in Programming Assignment 5). You can assume that name will not contain any illegal tokens—no commas, quotation marks, etc. It will always be a sequence of letters, digits, and underscores, starting with a letter (uppercase or lowercase) or an underscore.

2.1.4 boolean

PushB *bool*

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

input	stack
PushI 5 PushB <true>	<true> 5

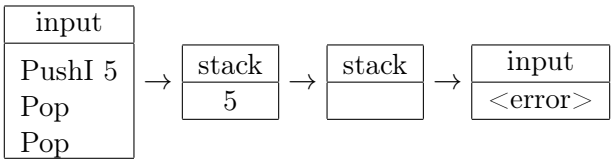
2.1.5 error and unit

Push <unit>

Similar with boolean literals, pushing a unit literal will Push `<unit>` onto the stack.

2.2 Pop

The command Pop removes the top value from the stack. If the stack is empty, an error literal (`<error>`) will be Pushed onto the stack. For example,



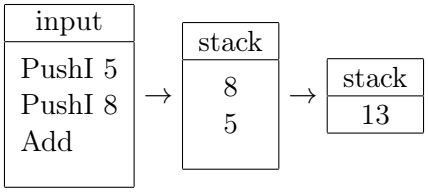
2.3 Add

The command Add refers to integer addition. Since this is a binary operator, it consumes the top two values in the stack, calculates the sum and Pushes 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:

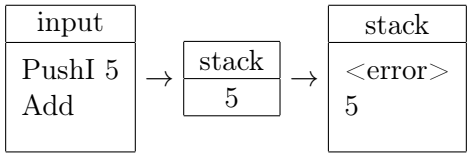
- not all top two values are integer numbers

- only one value in the stack
- stack is empty

for example, the following non-error case:



Alternately, 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>`

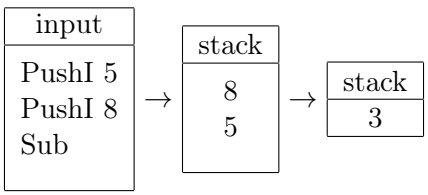


2.4 Sub

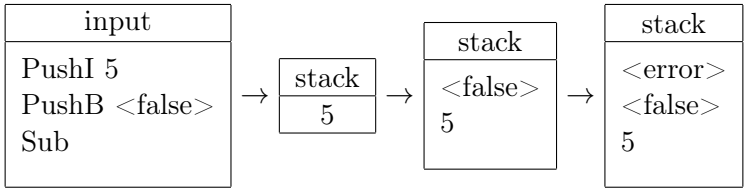
The command Sub refers to integer subtraction. It is a binary operator and works in the following way:

- if top two elements in the stack are integer numbers, pop the top element(y) and the next element(x), subtract x from y, and Push the result y-x 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, the following non-error case:



Alternately, 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>`

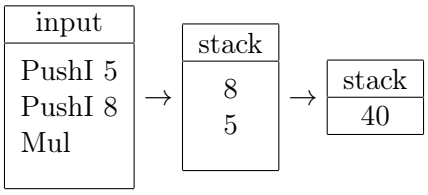


2.5 Mul

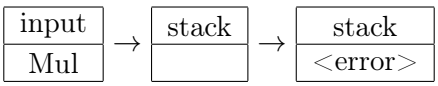
The command Mul refers to integer multiplication. It is a binary operator and works in the following way:

- 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, the following non-error case:



Alternately, if the stack empty when Mul is executed, an error will occur and `<error>` should be Pushed onto the stack:



2.6 Div

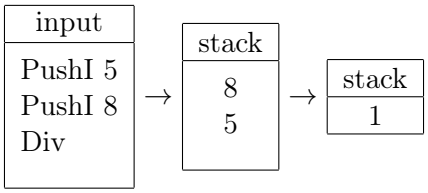
The command Div refers to integer division.[3] It is a binary operator and works in the following way:

- if top two elements in the stack are integer numbers, pop the top element(y) and the next element(x), divide y by x, and push the result $\frac{y}{x}$ back onto the stack
- if top two elements in the stack are integer numbers but x equals to 0, Push them back in the same order and Push `<error>` onto the stack

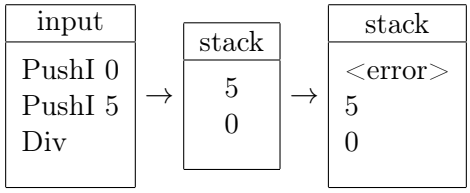
[3] Take "integer division" to behave exactly the way it behaves in OCaml

- 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, the following non-error case:



Alternately, if the second top element in the stack equals to 0, there will be an error if Div is executed. In such situations 0 and 5 should be Pushed back onto the stack as well as `<error>`



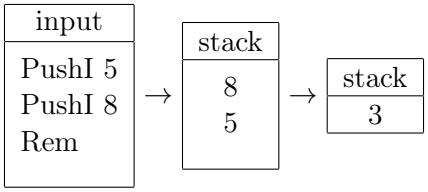
2.7 Rem

The command Rem refers to the remainder of integer division.[4] It is a binary operator and works in the following way:

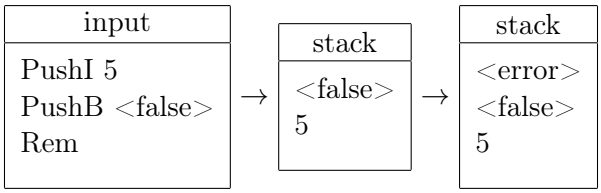
- if top two elements in the stack are integer numbers, pop the top element(y) and the next element(x), calculate the remainder of $\frac{y}{x}$, and Push the result back onto the stack
- if top two elements in the stack are integer numbers but x equals to 0, Push them back in the same order and Push `<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, the following non-error case:

[4] See footnote 3.



Alternately, if one of the top two elements in the stack is not an integer, an error will occur if Rem is executed. If this occurs the top two elements should be Pushed back onto the stack as well as `<error>`. For example:

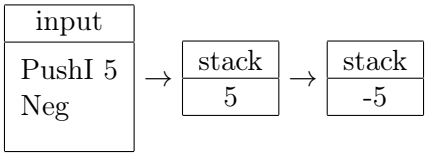


2.8 Neg

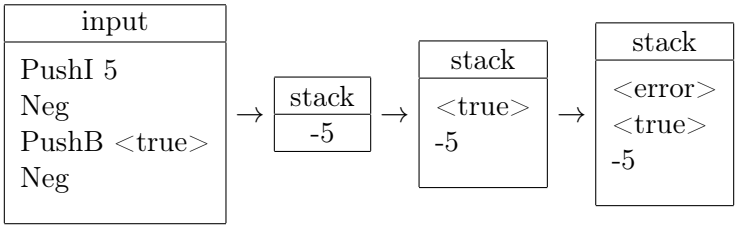
The command Neg is to calculate the negation of an integer (negation of 0 should still be 0). It is unary therefore consumes only the top element from the stack, calculate its negation and Push the result back. A value `<error>` will be Pushed onto the stack if:

- 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, the following non-error case:



Alternately, if the value on top of the stack is not an integer, when Neg is used, that value should be Pushed back onto the stack as well as `<error>`. For example:

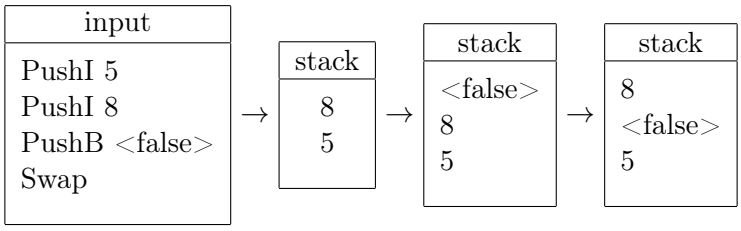


2.9 Swap

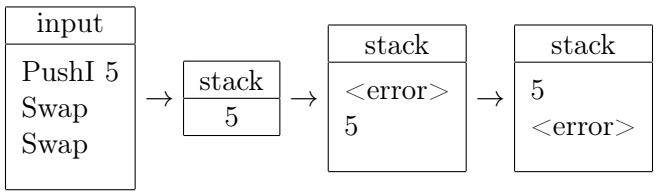
The command Swap interchanges the top two elements in the stack, meaning that the first element becomes the second and the second becomes the first. A value `<error>` will be Pushed onto the stack if:

- 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, the following non-error case:



Alternately, 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:



3 Frequently Asked Questions

1. Q: What are the contents of test case X?

   A: We purposefully withhold some test cases to encourage you to write your own test cases and reason about your code. You cannot test *every* possible input into the program for correctness. We will provide high-level overviews of the test cases, but beyond that we expect you to figure out the functionalities that are not checked with the tests we provide. But you can (and should) run the examples shown in this document! They're useful on their own, and can act as a springboard to other test cases.

2. Q: Why does my program run locally but fail on Gradescope?

   A: Check the following:

   - Ensure that your program matches the types and function header defined in section 2 on page 1.
   - Make sure that any testing code is either removed or commented out. If your program calls interpreter with input "input.txt", you will likely throw an exception and get no points.
   - *Do not submit testing code.*
   - `stdout` and `stderr` streams are not graded. Your program must write to the output file specified by *outputFile* for you to receive points.
   - *Close your input and output files.*
   - Core and any other external libraries are not available.
   - Gradescope only supports 4.04, so any features added after are unsupported.

3. Q: Why doesn't Gradescope give useful feedback?

   A: Gradescope is strictly a grading tool to tell you how many test cases you passed and your total score. Test and debug your program locally before submitting to Gradescope. The only worthwhile feedback Gradescope gives is whether or not your program compiled properly.

4. Q: Are there any runtime complexity requirements?

   A: Although having a reasonable runtime and space complexity is important, the only official requirement is that your program runs the test suite in less than three minutes.

5. Q: Is my final score the highest score I received of all my submissions?

   A: No. Your final score is only your most recent submission.

6. Q: What can I do if an old submission received a better grade than my most recent submission?

   A: You can always download any of your previous submissions. If the deadline is approaching, we suggest resubmitting your highest-scoring submission before Gradescope locks.