CS 320: Language Interpreter Design

Part 1 Due: November 11th 11:59pm EST
Part 2 Due: November 21st 11:59pm EST

1 Overview

The project is broken down into three parts. Each part is worth 100 points.

You will submit a file named `interpreter.ml` which contains a function, `interpreter`, with the following type signature:

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

2 Functionality

The function will take a program as an `input` string, consisting of the program lines, and will take in the output file path as the second parameter, and will return a `unit`, as your evaluated output will be logged into a file.

Because the function evaluates a stack-based programming language, a stack is used internally throughout the interpreter to keep track of intermediate evaluation results. Note that the stack will not be checked, but rather what is written in the output file — the stack and the output file are two different things.

The autograder will check your output file's content, and more information on the output file will be described below.

```
1
```

Example 3

```
Push 1
Push 4
Push "candy"
```

should result in the stack

```
"candy"
4
1
```

and a file with nothing inside (this will be checked)

3.3.2 Push

All kinds of *const* are pushed to the stack in the same way. Resolve the constant to the appropriate value and add it to the stack.

The program

```
Push 12
Push "abc"
Push -6
Push ""
Quit
```

should result in the following stack and the following contents in the output file:

```
""
-6
"abc"
12
```

3.3.3 Pop

The command Pop removes the top element from the stack. If the stack contains less than 1 element, terminate evaluation with error.

Example 1

```
Push 33
Push ""
Push "a"
Pop
Quit
```

should result in the following stack and the following contents in the output file:

```
""
33
```

Example 2

3 Part 1: Basic Computation
Due Date: November 11th 11:59pm EST

3.1 Grammar

For part 1 you will need to support the following grammar

3.1.1 Constants

<digit> ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

<int> ::= [−] <digit> {<digit>}

<char> ::= `a` | `b` | `c` | ... | `y` | `z` | `A` | `B` | `C` | ... | `Y` | `Z` (aka the alphabet)

<string> ::= `"`{<char>}`"`

<const> ::= <int> | <string>

3.1.2 Programs

<com> ::= `Quit` | `Push` <const> | `Pop` | `Add` | `Sub` | `Mul` | `Div` | `Swap` | `Neg` | `Concat`

<prog> ::= <com> <prog> | <com>

3.2 Errors

In part 1, when an error occurs during interpretation, evaluation must stop immediately and output the exact log `"Error"` into the provided output file (see `3.3.1` for more information about the output file).

3.3 Commands

Your interpreter should be able to handle the following commands:

3.3.1 Quit

This command causes the interpreter to stop and *does not have to be* on the last line of the program. When called, the created stack should be written out to an output file that is specified as the second argument to the main function, and no commands after the Quit call get evaluated.
Example 1

```
Push 1
Quit
```

should result in the following stack and the following contents in the output file:

```
1
```

Example 2

```
Push 1
Quit
Push 4
Quit
Push "candy"
```

should result in the following stack and the following contents in the output file:

```
Push 5
Pop
Pop
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

3.3.4 Add

Add consumes the top 2 values in the stack, and pushes their sum to the stack. If there are fewer than 2 values on the stack, terminate with error. If not all of the top 2 values on the stack are not integers, terminate with error.

Example 1

```
Push 5
Push 7
Add
Push 3
Add
Quit
```

should result in the following stack and the following contents in the output file:

```
15
```

Example 2

```
Push "hello"
Add
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

Example 3

```
Push "bottle"
Push "water"
Add
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

3.3.5 Sub

Sub consumes the top 2 values on the stack, and pushes the difference between the top value and the second top value to the stack.

If there are fewer than 2 values on the stack, terminate with error. If the top 2 values on the stack are not integers, terminate with error.

Example 1

```
Push "test"
Push 1
Push 10
Sub
Quit
```

should result in the following stack and the following contents in the output file:

```
9
"test"
```

Example 2

```
Push "test"
Push 1
Push 3
Push 4
Push 10
Sub
Quit
```

should result in the following stack and the following contents in the output file:

```
6
3
1
"test"
```

Example 3

```
Push "choco"
Push 10
Sub
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

3.3.6 Mul

Mul consumes the top 2 values in the stack, and pushes their product to the stack.

If there are fewer than 2 values on the stack, terminate with error. If the top 2 values on the stack are not integers, terminate with error.

Example 1

```
Push 5
Push 7
Mul
Quit
```

should result in the following stack and the following contents in the output file:

```
35
```

Example 2

```
Push 2
Push "laddoo"
Mul
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

Example 3

```
Push "coffee"
Push "cream"
Mul
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

3.3.7 Div

Div consumes the top 2 values on the stack, and pushes the quotient between the top value and the second top value onto the stack.

If there are fewer than 2 values on the stack, terminate with error. If the second value is 0, terminate with error. If the top 2 values on the stack are not integers, terminate with error.

Example 1

```
Push 2
Push 10
Div
Quit
```

should result in the following stack and the following contents in the output file:

```
5
```

Example 2

```
Push "samosa"
Push 10
Div
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

Example 3

```
Push 5
Push 5
Sub
Push 10
Div
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

3.3.8 Swap

Swap takes the top 2 elements in the stack and swaps their order. If there are less than 2 items in the stack, terminate with error

Example 1

```
Push "hello"
Push "world"
Swap
Quit
```

should result in the following stack and the following contents in the output file:

```
"hello"
"world"
```

Example 2

```
Push "320"
Push "cas"
Push "cs"
Swap
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

Example 3

```
Push 0
Swap
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

3.3.9 Neg

Neg would negate the top element on the stack. If the top element in the stack is not an integer, terminate with error. If the stack is empty, terminate with error.

Example 1

```
Push 2
Push 10
Neg
Quit
```

should result in the following stack and the following contents in the output file:

```
-10
2
```

Example 2

```
Push 5
Push "kenny"
Neg
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

Example 4

```
Neg
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

3.3.10 Concat

Concat consumes the top 2 values on the stack, and pushes the concatenation between the top value and the second top value onto the stack.

If there are fewer than 2 values in the stack, terminate with error. If the top 2 elements in the stack are not of *string* type, then terminate with error.

Example 1

```
Push "lemon"
Push "laddoo"
Concat
Quit
```

should result in the following stack and the following contents in the output file:

```
"laddoolemon"
```

Example 2

```
Push 3
Push "peanut"
Concat
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

Example 3

```
Push "chocolate chip"
Push "cookie"
Concat
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

since "chocolate chip" is an invalid string.

4 Part 2: More Computation, Definitions, and Conditionals

4.1 Grammar

For part 2 the grammar is extended in the following way

4.1.1 Constants

<char> ::= <uchar> | <lchar>

<uchar> ::= `A` | `B` | `C` | . . . | `Y` | `Z`

<lchar> ::= `a` | `b` | `c` | . . . | `y` | `z`

<name> ::= <lchar> {<char> | _ | <digit>}

<const> ::= . . . | <name>

4.1.2 Programs

<com> ::= . . . | `And` | `Or` | `Not` | `Equal` | `Lte`
| `Local` <name> | `Global` <name>
| `Begin` <prog> `End`
| `IfThen` <prog> `Else` <prog> `End`

4.1.3 Booleans

We interpret the integers 1 and 0 to also stand for the booleans `true` and `false`, respectively.

4.1.4 Environment

An environment is used to track bindings from `name`s to values. Using a `name` with `Push` looks up its value in the environment and pushes the value onto the stack. `Local` binds have priority over `Global` if a `name` is defined in both. If a new scope is defined with `Begin/End`, a new local environment should be created for it, but it can still read values from outer scopes' local environments.

Names in the global environment will always be visible, but a local environment is lexically restricted to inside `Begin`/`End`.

4.2 Commands

4.2.1 And

`And` consumes the two 2 values in the stack and pushes their conjunction to the stack. If there are fewer then 2 values on the stack, terminate with error. If the 2 top values in the stack are not booleans, terminate with error.

Example 1

```
Push 1
Push 0
And
Quit
```

should result in the following stack and the following contents in the output file:

```
0
```

Example 2

```
Push 1
Push 1
And
Quit
```

should result in the following stack and the following contents in the output file:

```
1
```

Example 3

```
Push 1
And
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

Example 4

```
Push 3
Push 1
And
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

4.2.2 Or

Or consumes the top 2 values in the stack and pushes their disjunction to the stack. If there are fewer than 2 values on the stack, terminate with error. If the 2 top values in the stack are not booleans, terminate with error.

Example 1

```
Push 1
Push 0
Or
Quit
```

should result in the following stack and the following contents in the output file:

```
1
```

Example 2

```
Push 0
Push 0
Or
Quit
```

should result in the following stack and the following contents in the output file:

```
0
```

Example 3

```
Push 1
Or
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

Example 4

```
Push 3
Push 1
Or
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

4.2.3 Not

`Not` consumes the top value of the stack and pushes its negation to the stack. If the stack is empty, terminate with error. If the top value on the stack is not a boolean, terminate with error.

Example 1

```
Push 1
Push 0
Not
Quit
```

should result in the following stack and the following contents in the output file:

```
1
1
```

Example 2

```
Push 1
Push 1
Not
Quit
```

should result in the following stack and the following contents in the output file:

```
0
1
```

Example 3

```
Push 3
Not
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

Example 4

```
Not
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

4.2.4 Equal

`Equal` consumes the top 2 values in the stack and pushes true to the stack if they are equal integers and false if they are not equal integers. If there are fewer than 2 values on the stack, terminate with error. If the 2 top values in the stack are not integers, terminate with error.

Example 1

```
Push 5
Push 5
Equal
Quit
```

should result in the following stack and the following contents in the output file:

```
1
```

Example 2

```
Push 1
Push 1
Push 1
Add
Equal
Quit
```

should result in the following stack and the following contents in the output file:

```
0
```

Example 3

```
Push 0
Push 1
Push 1
Sub
Equal
Quit
```

should result in the following stack and the following contents in the output file:

```
1
```

Example 4

```
Push "abc"
Push 1
Equal
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

Example 5

```
Push 1
Equal
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

4.2.5 Lte

`Lte` consumes the top 2 integer values in the stack and pushes true on the stack if the top value is less than or equal to the second top value. If there are fewer than 2 values on the stack, terminate with error. If the 2 top values in the stack are not integers, terminate with error.

Example 1

```
Push 1
Push 1
Lte
Quit
```

should result in the following stack and the following contents in the output file:

```
1
```

Example 2

```
Push 1
Push 2
Lte
Quit
```

should result in the following stack and the following contents in the output file:

```
0
```

Example 3

```
Push 2
Push 1
Lte
Quit
```

should result in the following stack and the following contents in the output file:

```
1
```

Example 4

```
Push "abc"
Push 1
Lte
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

Example 5

```
Push 1
Lte
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

4.2.6 Local

Local comes with a `name` argument and consumes the top element of the stack, which can be any value, and relates them in the current local environment. If the `name` is already assigned a value in the current local environment, overwrite it. `Push <name>` should push the value paired with that name in the current environment on to the stack.

If the stack is empty, terminate with error. If a name that is not currently in the environment is used with `Push`, terminate with error.

Example 1

```
Push 3
Local x
Push x
Local y
Push x
Push y
Quit
```

should result in the following stack and the following contents in the output file:

```
3
3
```

Example 2

```
Push 3
Local x
Quit
```

should result in an empty stack and nothing written to provided output file.

Example 3

```
Push 2
Local x
Push x
Push 3
Local x
Push x
Add
Push x
Quit
```

should result in the following stack and the following contents in the output file:

```
3
5
```

Example 4

```
Local x
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

since the stack is empty, and thus there is no `val` to assign to x.

Example 5

```
Push x
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

since x is not bound in the current environment.

4.2.7 Global

Global comes with a `name` argument and consumes the top element of the stack, which can be any value, and relates them in the global environment until the end of the program. If the `name` is already assigned a value in the global environment, overwrite it. `Push <name>` should push the value paired with that name in the environment on to the stack.

If the stack is empty, terminate with error. If a name that is not currently in the environment is used with `Push`, terminate with error.

Example 1

```
Push 3
Global x
Push x
Quit
```

should result in the following stack and the following contents in the output file:

```
3
```

Example 2

```
Push 3
Global x
Quit
```

should result in an empty stack and nothing written to provided output file.

Example 3

```
Push 2
Global x
Push x
Push 3
Global x
Push x
Add
Push x
Quit
```

should result in the following stack and the following contents in the output file:

```
3
5
```

Example 4

```
Push 2
Local x
Push x
Push 1
Add
Global x
Push x
Quit
```

should result in the following stack and the following contents in the output file:

```
2
```

since locally-defined variables have priority over globally-defined variables.

Example 5

```
Global x
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

since the stack is empty, and thus there is no `val` to assign to x.

Example 6

```
Push x
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

since x is not bound in the current environment.

4.2.8 Begin/End

A sequence of commands in a `Begin/End` block will be executed on a new empty stack with a copy of the current binding scope. When the commands finish, the top value from the stack will be pushed to the outer stack, and new local bindings made from within the block disregarded. Glocal bindings made from within the block are valid for the rest of the program.

Example 1

```
Push 1
Push 2
Begin
    Push 3
    Push 7
    Push 4
End
Push 5
Push 6
Quit
```

should result in the following stack and the following contents in the output file:

```
6
5
4
2
1
```

Example 2

```
Push 3
Begin
    Pop 1
    Push 7
End
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

since the `Pop` command is executed with an empty inner stack.

Example 3

```
Push 55
Local x
Push x
Begin
    Push 3
    Push 5
    Local x
    Push 7
    Push x
End
Push x
Quit
```

should result in the following stack and the following contents in the output file:

```
55
5
55
```

Example 4

```
Push 55
Local x
Push x
Begin
    Push 3
    Push 5
    Global x
    Push 7
    Push x
End
Push x
Quit
```

should result in the following stack and the following contents in the output file:

```
55
55
55
```

Example 5

```
Push 55
Global x
Push x
Begin
```

```
    Push 3
    Push 5
    Global x
    Push 7
    Push x
End
Push x
Quit
```

should result in the following stack and the following contents in the output file:

```
5
5
55
```

Example 6

```
Begin
End
Quit
```

should result in the following stack and the following contents in the output file:

```
"Error"
```

as the stack from the inner scope was empty upon ending, meaning that nothing could be pushed to the outer scope's stack.

Example 7

```
Push 1
Local x
Push 2
Local y
Begin
    Push 20
    Local x
    Push x
    Push y
    Add
End
Push x
Add
Quit
```

should result in the following stack and the following contents in the output file:

```
23
```

4.2.9 IfThen/Else

The `IfThen/Else` command will consume the top element of the stack. If that element is true it will execute the commands in the then branch, otherwise if false it will execute the commands in the else branch. In both cases, the remaining stack is used directly for executing the commands in corresponding branch. The resulting stack after evaluating the correct branch is used to evaluate the rest of the program.

If stack is empty, terminate with error. If the top value on the stack is not a boolean, terminate with error.

Example 1

```
Push 10
Push 1
IfThen
    Push 5
    Add
Else
    Push 5
    Sub
End
Quit
```

should result in the following stack and the following contents in the output file:

```
15
```

Example 2

```
Push 10
Push 0
IfThen
    Push 5
    Add
Else
    Push 5
    Sub
End
Quit
```

should result in the following stack and the following contents in the output file:

```
-5
```

Example 3

```
Push 10
Local x
Push 0
IfThen
    Push 5
    Add
Else
    Push x
    Push 234
    Local x
    Push x
End
Push x
Quit
```

should result in the following stack and the following contents in the output file:

```
234
234
10
```