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

Programming Assignment 6: Interpreter (Part 3)

Out: Friday, April 17, 2020
Due: Friday, May 1, 2020, by 11:59 pm

This assignment continues the work you did for Assignment 4 and Assignment 5, which were Part 1 and Part 2 of an intperpreter for a small, OCaml-like, stack-based bytecode programming language. Once more, 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

Identical to Section 1 in Assignment 5 (which included a few comments that had been omitted from Section 1 in Assignment 4).

2 Basic Computations

Identical to Section 2 in Assignment 5 (which is the same as Section 2 in Assignment 4).

3 Variables and Scope

Identical to Section 3 in Assignment 5.

4 Functions

4.1 Function declaration and Call

Fun *name1 name2*

Denotes a function declaration, i.e. the start of a function called *name1*, which has one formal parameter *name2*. The expressions that follow comprise the function body. The function body is terminated with a special keyword FunEnd:

FunEnd

denotes the end of a function body.

PushN *funName*
Push *arg*
Call

denotes applying the function *funName* to the actual parameter *arg*. Do note that *Push arg* can leverage any form of the Push command, i.e. *PushI, PushB, PushN, PushS, Push*. When Call is evaluated, it will apply the function *funName* to *arg* and Pop both *funName* and *arg* from the stack. *arg* can either be a name (this includes function names), an integer, a string, a boolean, `<unit>`, or `<error>`.

1. The environment for the closure will be a copy of the current environment.
2. To compute the code for the function, you should copy all the expressions in order starting with the first expressions after the function declaration up to, but not including, the FunEnd.
3. In the current environment you should create a binding between the function name and its closure.

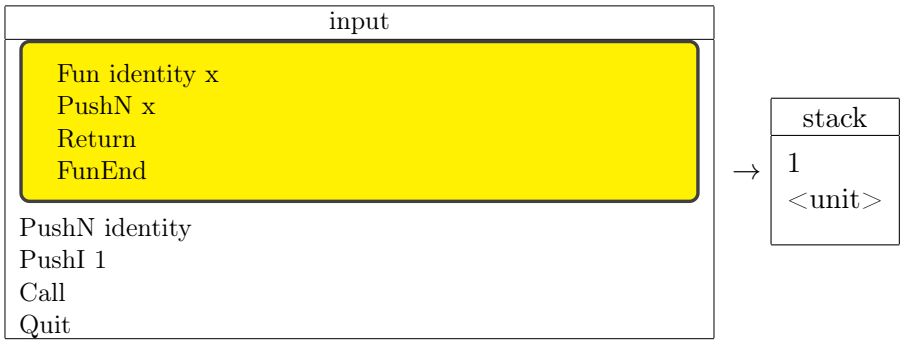
When a function is called, you should first check to see if there is a binding in the current environment, which maps *funName* to a closure. If one does not exist, Push `<error>` onto the stack. You should then check to see if the current environment contains a binding for *arg* if it is a name instead of a value. If it does not, then you should Push `<error>` onto the stack. If *arg* is an `<error>` you should Push `<error>` onto the stack.

4.2 Return

Functions can return values by using a Return expression.

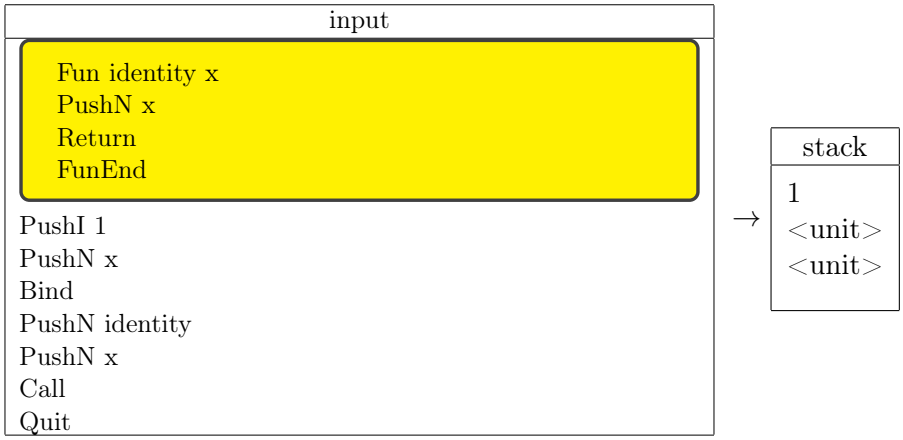
4.3 Examples

4.3.1 Example 1



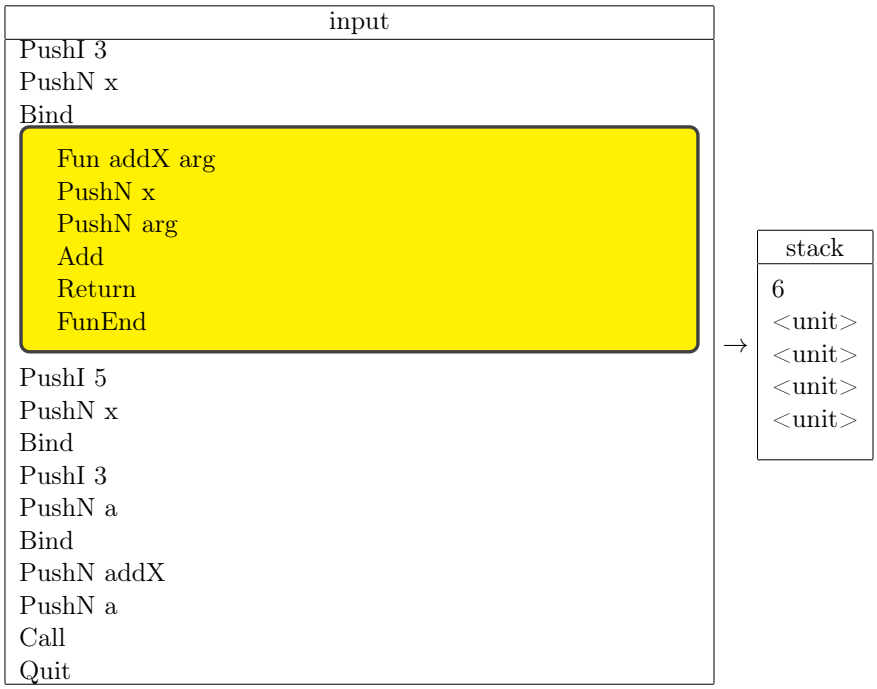
1 → return value of calling identity and passing in x as an argument
<unit> → result of declaring identity

4.3.2 Example 2



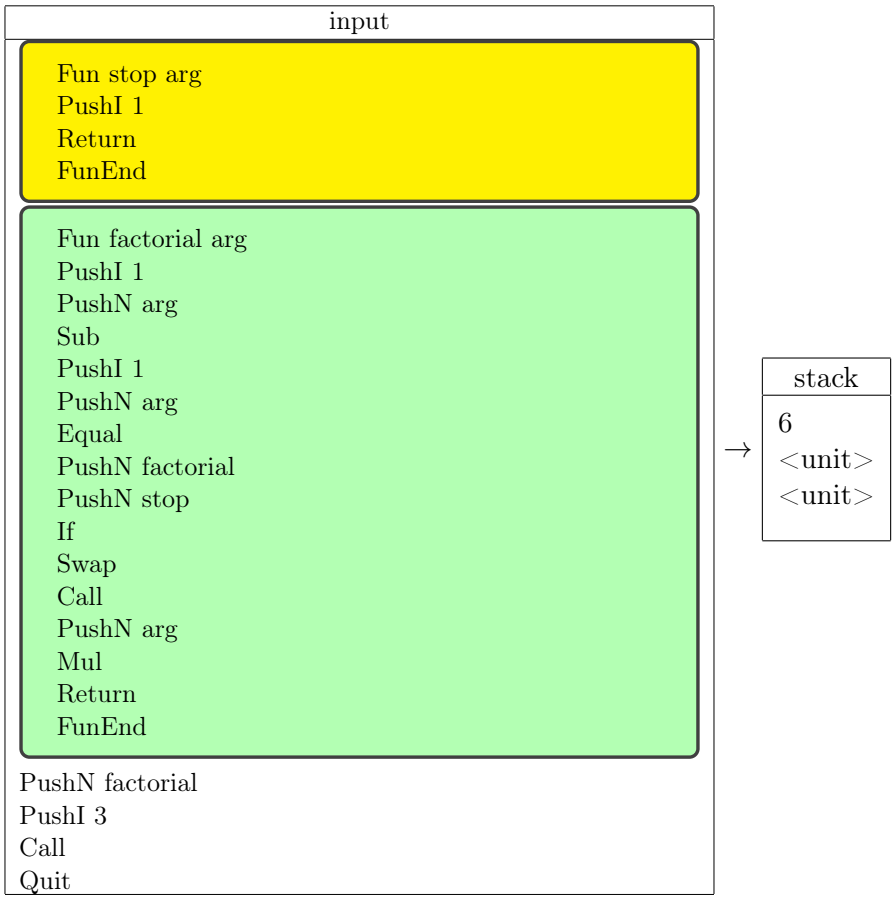
1 → return value of calling identity and passing in x as an argument
<unit> → result of binding x
<unit> → result of declaring identity

4.3.3 Example 3



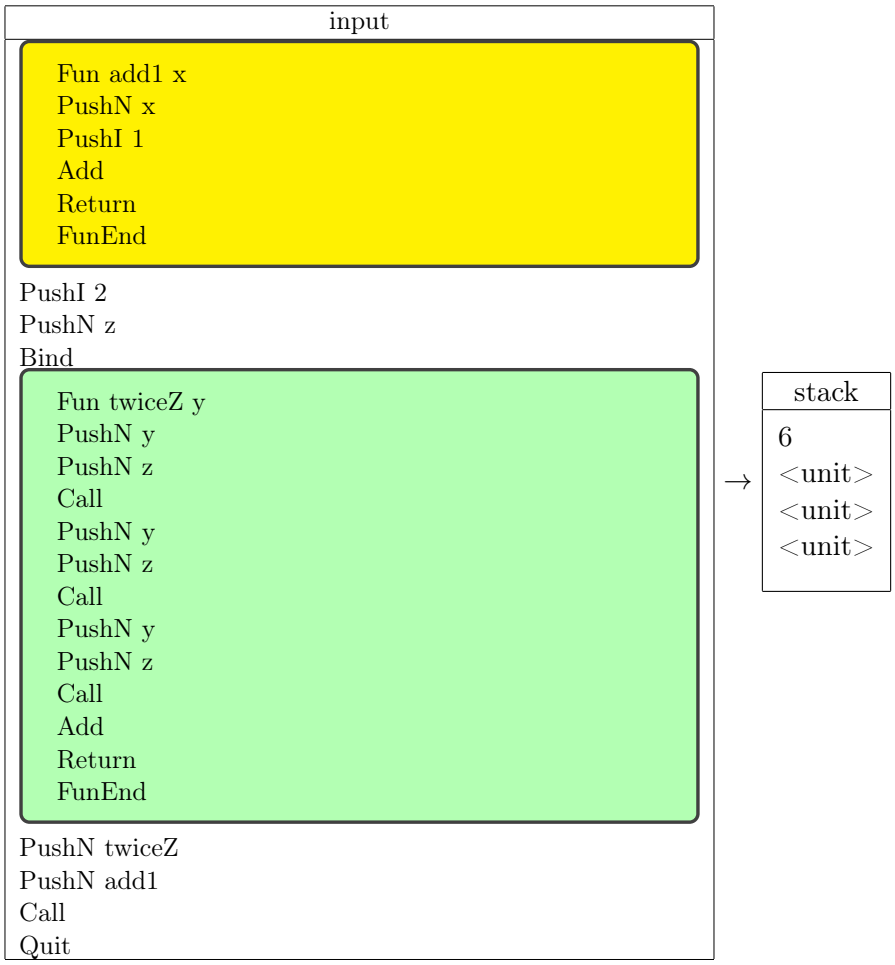
6 → result of function call
<unit> → result of third binding
<unit> → result of second binding
<unit> → result of function declaration
<unit> → result of first binding

4.3.4 Example 4



6 → value returned from factorial
<unit> → declaration of factorial
<unit> → declaration of stop

4.3.5 Example 5

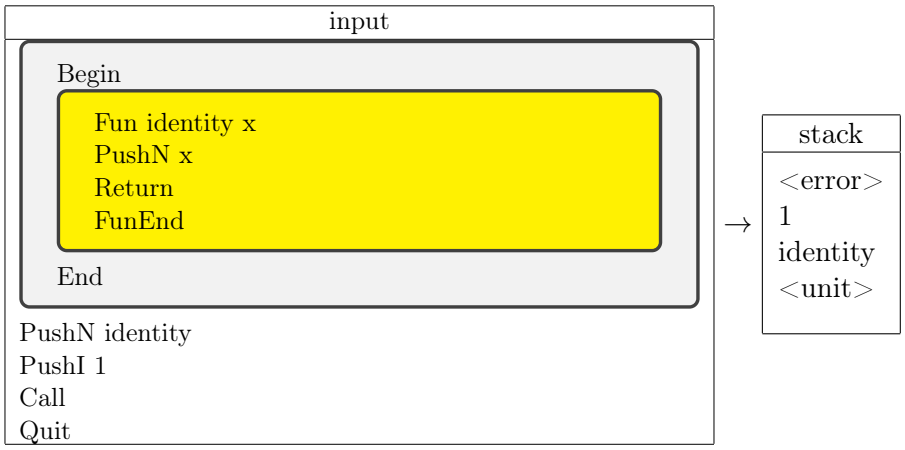


6 → return of calling twiceZ and passing add1 as an argument
<unit> → declaration of twiceZ
<unit> → binding of z
<unit> → declaration of the add1 function

4.4 Functions and Begin

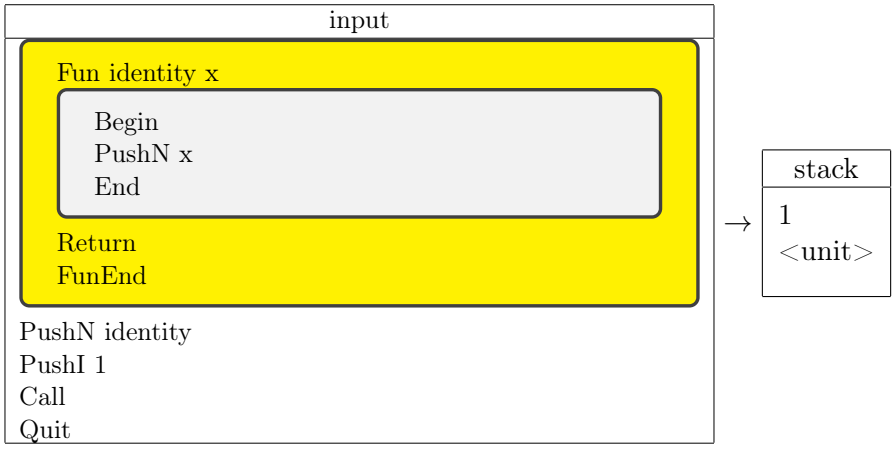
Functions can be declared inside a Begin expression. Much like the lifetime of a variable binding, the binding of a function obeys the same rules. Since Begin introduces a stack of environments, the closure should also take this into account.

4.4.1 Example 1



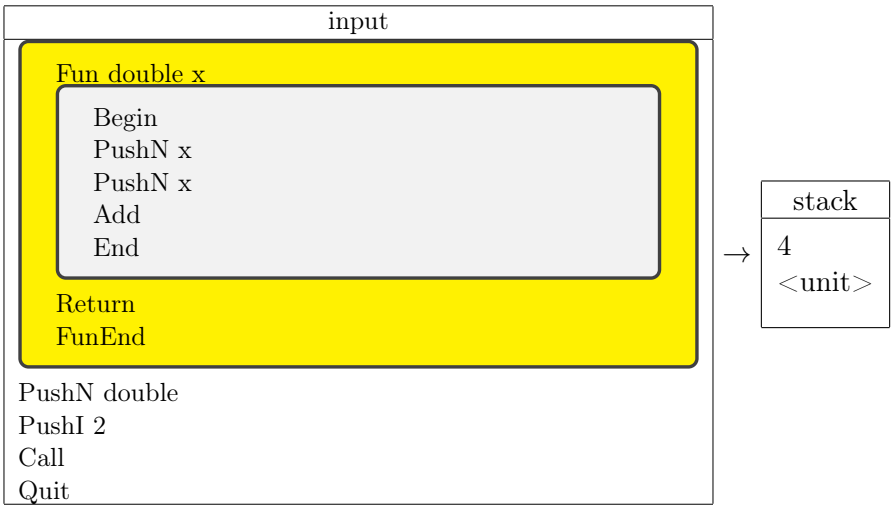
<error> → error since identity is not bound in the environment
1 → Push of 1
identity → Push of identity
<unit> → result of declaring identity, this is the result of the Begin expression

4.4.2 Example 2



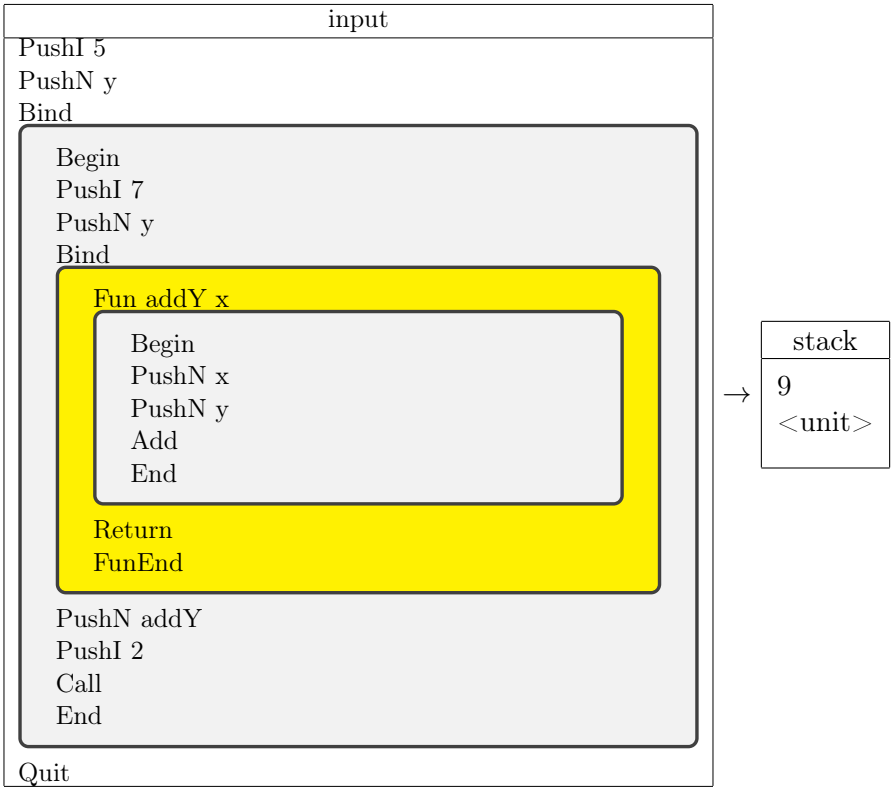
1 → return value of calling identity and passing in x as an argument
<unit> → result of declaring identity

4.4.3 Example 3



4 → return value of calling identity and passing in x as an argument
<unit> → result of declaring identity

4.4.4 Example 4



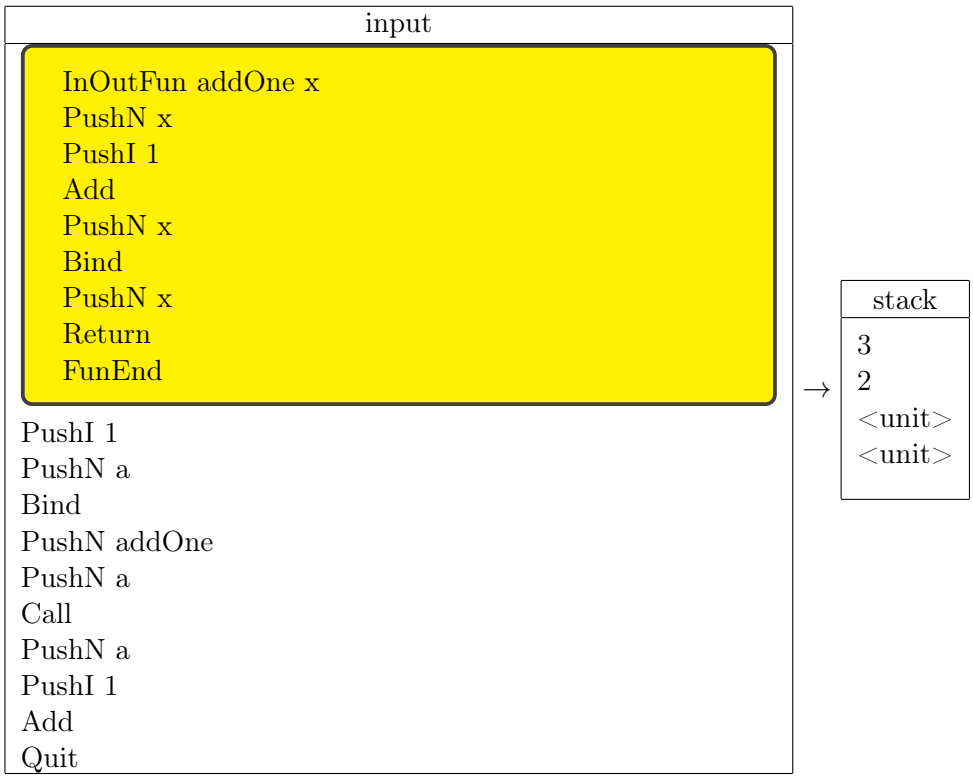
9 → return value of calling identity and passing in 2 as an argument

<unit> → result of binding y to 5

4.5 In/Out Functions

Our language will also support in/out parameters for specially denoted functions. Instead of using the Fun keyword, functions that have in/out parameters are declared using the InOutFun keyword.

In/out functions should have a similar implementation to regular functions. To this implementation you should add an additional operation when the function returns. The returned value will be bound to the actual parameter in the environment at the Call site.



3 → result of Add (note a is bound to two)
2 → return value of calling addOne and passing in x as an argument
<unit> → result of binding a
<unit> → result of declaring addOne

4.6 First-Class Functions

This language treats functions like any other value. They can be used as arguments to functions, and can be returned from functions.

4.6.1 Example 1: Curried adder

```
input
Fun makeAdder x
    Fun adder y
    PushN x
    PushN y
    Add
    Return
    FunEnd
PushN adder
Return
FunEnd
PushN add3
PushN makeAdder
PushI 3
Call
Swap
Bind
PushN add3
PushI 5
Call
Quit
```

→

stack
8
<unit>
<unit>

8 → Evaluated from calling the generated function add3 with argument 5
<unit> → The result of binding the generated function to the name add3
<unit> → The result of declaring the function makeAdder

Step by step (after declaring makeAdder, pushing add3, pushing 3, and pushing makeAdder):

stack
3
makeAdder
add3
<unit>

Call →

stack
⟨*CLOSURE*⟩
add3
<unit>

Swap →

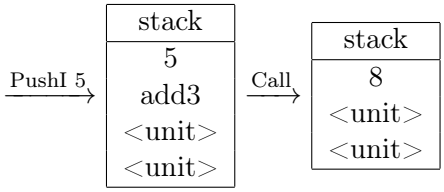
stack
add3
⟨*CLOSURE*⟩
<unit>

Bind →

stack
<unit>
<unit>

PushN add3 →

stack
add3
<unit>
<unit>



If a function is returned from another function, it need not be bound to a name in the environment it is returned in. For example:

input
Fun identity x
PushN x
Return
FunEnd
Fun _catExcl y
PushS "!"
PushN y
Concat
Return
FunEnd
PushN identity
PushN _catExcl
Call
PushS "Dunder Mifflin"
Call
Quit

→

stack
Dunder Mifflin!
<unit>
<unit>

Dunder Mifflin! → Computed from calling the *closure* returned by the identity function applied to concatExcl with the argument "Dunder Mifflin".
<unit> → The result of declaring the function _catExcl.
<unit> → The result of declaring the identity function.

Here is a closer look at how the stack develops through this program. Note that function closures will never be on the stack when the program finishes execution.

Step by step examples

1. If your interpreter reads in expressions from *inputFile*, states of the stack after each operation are shown below:

input
PushI 10
PushI 15
PushI 30
Sub
PushB <true>
Swap
Add
Pop
Neg
Quit

First, Push 10 onto the stack:

stack
10

Similarly, Push 15 and 30 onto the stack:

stack
30
15
10

Sub will pop the top two values from the stack, calculate 30 - 15 = 15, and Push 15 back:

stack
15
10

Then Push the boolean literal <true> onto the stack:

stack
<true>
15
10

Swap consumes the top two values, interchanges them and Pushes them back:

stack
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>

stack
<error> 15 <true> 10

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

stack
15 <true> 10

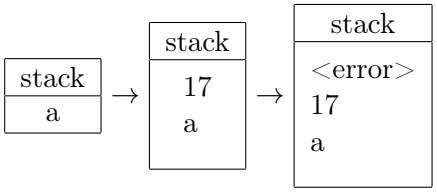
Then after calculating the negation of 15, which is -15, and pushing it back, Quit will terminate the interpreter and write the following values in the stack to *outputFile*:

stack
-15 <true> 10

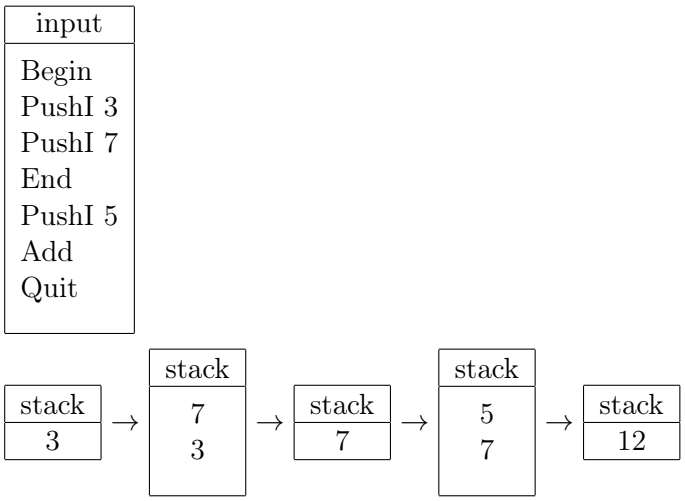
Now, go back to the example inputs and outputs given before and make sure you understand how to get those results.

2. More Examples of Bind and Begin...End:

input
PushN a PushI 17 Add



The error is because we are trying to perform an addition on an unbound variable "a".

3. 

Explanation :

PushI 3
PushI 7

Pushes 3 and 7 on top of the stack. When you encounter the "end", the last stack frame is saved (which is why the value of 7 is retained on the stack), then 5 is Pushed onto the stack and the values are added.

5 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.