CS 320: Language Interpreter Design

Part 1 Due: November 11th 11:59pm EST
Part 2 Due: November 21st 11:59pm EST
Part 3 Due: December 8th 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.

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:

```
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

```
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
```

5 Part 3: Union, Tuple, and Mutually Recursive Function.

5.1 Grammar

For part 3 the grammar is extended in the following way

5.1.1 Programs

com ::= ...
| `InjL`
| `InjR`
| `CaseLeft` <prog> `Right` <prog> `End`
| `Tuple` <int>
| `Get` <int>
| `Fun` <name> <name> <prog> {`Mut` <name> <name> <prog>} `End`
| `Call`
| `Return`

5.1.2 Unions and Tuples

Unions are values which capture the concept of "this" *or* "that". In this language, we will call them *Left* and *Right*. Tuples, on the other hand, capture the concept "this" *and* "that". In this language, like OCaml, they will be shown as a comma separated sequence.

5.1.3 Closures

We add closures, which are commands paired with a local environment for those commands to be executed in (function values). The paired environment is used to resolve names mentioned in the commands of the closure. For closures of mutually declared, recursive functions, all of the mutually declared functions are inside the closure, but there is an *active* function which represents the function to be used when the closure is called. Consider the following example:

```
Fun isOdd x
prog1
Mut isEven x
prog2
End
Push isOdd
Push isEven
Quit
```

Two different closures are pushed to the stack. The resulting stack and output is:

```
Clo (isEven x)
Clo (isOdd x)
```

For the first, we say that the active function for this closure is `isEven`. For second, we say that the active function for this closure is `isOdd`.

Another example:

```
Fun f1 x
prog1
Mut f2 x
prog2
```

```
Mut f3 x
prog3
End
Push f1
Push f2
Push f3
Quit
```

Three different closures are pushed to the stack. The resulting stack and output is:

```
Clo (f3 x)
Clo (f2 x)
Clo (f1 x)
```

For the first, we say that the active function for this closure is `f3`. For the second, we say that the active function for this closure is `f2`. For the third, we say that the active function for this closure is `f1`. The order of non-active elements in the closure does not matter.

5.2 Commands

5.2.1 InjL/InjR

InjL (respectively, InjR) consumes the top value of the stack and pushes a union value to the stack. If the stack is empty, terminate with error.

Example 1

```
Push 5
InjL
Quit
```

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

```
Left 5
```

Example 2

```
Fun foo x
End
Push foo
InjR
Quit
```

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

```
Right Clo (foo x)
```

Example 3

```
Push "hello"
InjL
InjR
Quit
```

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

```
Right Left "hello"
```

Example 4

```
InjL
Quit
```

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

```
"Error"
```

5.2.2 CaseLeft/Right

CaseLeft/Right consumes the top element of the stack. If the element is a left union, it will execute the commands in the first branch with the contents of the left pushed to the stack; otherwise, if the element is a right, it will execute the commands in the second branch with the contents of the right pushed to the stack. 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 the stack is empty, terminate with error. If the top value on the stack is not a union, terminate with error.

Example 1

```
Push 5
InjL
CaseLeft
Right
  Add
End
Quit
```

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

```
5
```

Example 2

```
Push 5
InjR
CaseLeft
Right
  Add
End
Quit
```

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

```
"Error"
```

Example 3

```
Push 5
InjL
CaseLeft
  Push 1
  Add
```

```
Right
  Push "Bob"
  Concat
End
Quit
```

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

```
6
```

Example 4

```
Push "hello"
InjR
CaseLeft
  Push 1
  Add
Right
  Push "Bob"
  Concat
End
Quit
```

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

```
"Bobhello"
```

5.2.3 Tuple

Tuple followed by an integer n consumes the top n values from the stack, places them in a tuple, and pushes the tuple to the stack. The 1st, 2nd, ..., nth values from the top of the stack will be put in the n-tuple in the order: (nth, ..., 2nd, 1st).

If there are fewer than n values on the stack, terminate with error. If n is a negative number, terminate with error.

Example 1

```
Push 1
Push "two"
Push 3
Tuple 3
Quit
```

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

```
(1, "two", 3)
```

Example 2

```
Fun bar x
End
Push 20
Push bar
Push 5
Tuple 2
Quit
```

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

```
(Clo (bar x), 5)
20
```

Example 3

```
Push 3
Tuple 2
Quit
```

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

```
"Error"
```

Example 4

```
Push "age"
Push 5
Tuple 2
Quit
```

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

```
("age", 5)
```

Example 5

```
Push 9
Local x
Push x
Push 4
Tuple 2
Quit
```

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

```
(9, 4)
```

5.2.4 Get

Get followed by an integer n takes (*not* consume) the top values from the stack and pushes the nth element of the tuple to the stack. Tuples are zero-indexed, so the first element is 0, the second is 1, etc.

If the stack is empty, terminate with error. If the top value on the stack is not a tuple, terminate with error. If n is out of bounds of the tuple, terminate with error.

Example 1

```
Push 1
Push 2
Tuple 2
Get 0
Quit
```

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

```
1
(1, 2)
```

Example 2

```
Push 1
Push 2
Tuple 2
Get 1
Quit
```

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

```
2
(1, 2)
```

Example 3

```
Push 1
Push 2
Tuple 2
Get 3
Quit
```

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

```
"Error"
```

Example 4

```
Push "there"
Push "hi"
Tuple 2
Get 0
Swap
Get 1
Swap
Pop
Concat
Quit
```

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

```
"hithere"
```

5.2.5 Mutually Recursive Function declarations

Functions are declared with the fun command

Fun *fname arg*
  prog
{ Mut *fname arg*
  prog }
End

Here, *fname* is the name of the function and *arg* is the name of the parameter to the function. *prog* are the commands that get executed when the function is called.

After a function is defined with the Fun command, a new closure is formed and bound to the functions name in the local environment. Each *fname*, *arg*, and *prog* in the mutually recursive functions with all local bindings in the current environment will be added to the new closure.

Example 1

```
Fun foo my_arg
End
Push foo
Quit
```

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

```
Clo (foo my_arg)
```

5.2.6 Call

Call consumes a closure and an argument value from the top of the stack. It then executes the active code inside the closure with a new stack, current global environment, and the closure's local environment extended with the following binds:

- the functions' formal parameter to the argument value received by Call
- all mutually declared function names to their corresponding closures

The active *prog* contained within the closure is executed using this newly formed environment and a fresh stack. As with Begin/End, all global bindings made within the called closure are valid after the closure returns and also within all of its recursive calls. But all the local bindings and stack made within the called closure are not valid after the closure returns and not valid within recursive calls.

Call then pushes the returned value (if the call terminates) from the closure onto the stack.

Example 1

```
Fun f x
  ...
End
Push f
Call
```

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

```
"Error"
```

5.2.7 Return

Return consumes the top value of the stack and returns it as the result of a closure.

If the stack is empty, terminate with error. If not inside a closure, terminate with an error. If the end of a function is reached with no return, terminate with an error.

Example 1

```
Fun f1 x
    Push x
    Return
Mut f2 x
```

```
    Push x
    Push 2
    Mul
    Local x
    Push x
    Push f1
    Call
    Return
Mut f3 x
    Push x
    Push 1
    Add
    Local x
    Push x
    Push f2
    Call
    Return
End
Push 3
Push f3
Call
Quit
```

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

```
8
```

Explanation of example 1:

We use strikethrough (~~Quit~~) to indicate a command has yet to run.

Step 1:

```
Fun f1 x
...
Mut f2 x
...
Mut f3 x
...
End
Push 3
Push f3
~~Call~~ (~~Call~~ means the Call command has not been executed yet)
~~Quit~~
```

The current stack before the Call command:

```
Clo (f3 x)
3
```

Step 2:

```
Fun f1 x
...
```

```
Mut f2 x
...
Mut f3 x
...
End
Push 3
Push f3
Call
Quit
```

Call consumes the two elements on the stack and executes *f3* with an empty stack, the global environment, and a new local environment containing *f1*, *f2*, *f3*, and x bound to the respective closures and 3.

Step 3 (executed *f3*):

```
Fun f1 x
...
Mut f2 x
...
Mut f3 x
Push x
Push 1
Add
Local x
Push x
Push f2
Call
Return
End
Push 3
Push f3
Call
Quit
```

The current stack after executing *f3* but before Call command:

```
Clo (f2 x)
4
```

Call will do the same process as above, but this time x will be bound to 4 inside *f2*.

Step 4 (call to *f2* in *f3* and execute *f2*):

```
Fun f1 x
...
Mut f2 x
Push x
Push 2
Mul
Local x
Push x
Push f1
Call
Return
```

```
Mut f3 x
Push x
Push 1
Add
Local x
Push x
Push f2
Call
Return
End
Push f3
Push 3
Call
Quit
```

The current stack after executing *f2* but before Call command:

```
Clo (f1 x)
8
```

Call will do the same process as above, but this time x will be bound to 8 inside *f1*.

Step 5 (call to *f1* in *f2*, execute *f1*, and return the value):

```
Fun f1 x
Push x
Return
Mut f2 x
Push x
Push 2
Mul
Local x
Push x
Push f1
Call
Return
Mut f3 x
Push x
Push 1
Add
Local x
Push x
Push f2
Call
Return
End
Push 3
Push f3
Call
Quit
```

The current stack when executing inside *f1* is just 8 and the Return command terminates the execution, giving back the value on the top of the stack, which is simply 8. Execution will then return to *f2* and the Return command there will run, returning the value on top of the stack, which is the 8 gotten from the execution of

Call. Execution will then return to *f3* and the Return command will run, returning the value on top of the stack, which is the 8 gotten from the execution of Call. Finally, Quit will be run on the top-level. And should result in the following stack and the following contents in the output file:

```
   8
```

Example 2

```
Fun f1 x
    Push y
    Return
Mut f2 x
    Push x
    Push 2
    Mul
    Local x
    Push x
    Push f1
    Call
    Return
Mut f3 y
    Push y
    Push 1
    Add
    Local x
    Push x
    Push f2
    Call
    Return
End
Push 3
Push f3
Call
Quit
```

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

```
    "Error"
```

Example 3

```
Fun regular x
    Push 11
    Push x
    Tuple 2
    Return
End
Push 22
Push regular
Call
Quit
```

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

```
    (11, 22)
```

Example 4

```
Fun odd x
    Push x
    Push 2
    Mul
    Local x
    Push x
    Push 46
    Equal
    IfThen
        Push x
        Return
    Else
        Push x
        Push even
        Call
        Return
    End
Mut even x
    Push 1
    Push x
    Add
    Local x
    Push x
    Push odd
    Call
    Return
End
Push 5
Push odd
Call
Quit
```

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

```
    46
```

Example 5

```
Fun numOfStepsToOne x
    Push numOfSteps
    Push 1
    Add
    Global numOfSteps
    Push x
    Push 1
    Add
    Local x
    Push x
    Push divideByTwo
```

```
    Call
    Return
Mut divideByTwo x
    Push numOfSteps
    Push 1
    Add
    Global numOfSteps
    Push 2
    Push x
    Div
    Local x
    Push x
    Push 1
    Equal
    IfThen
        Push numOfSteps
        Return
    Else
        Push x
        Push numOfStepsToOne
        Call
        Return
    End
End
Push 0
Global numOfSteps
Push 5
Push numOfStepsToOne
Call
Quit
```

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

```
    6
```

Example 6

```
Fun snack laddoo
    Push laddoo
    Push 100
    Mul
    Local cal
    Push cal
    Push calories
    Call
    Return
Mut calories cal
    Push 9
    Push cal
    Div
    Local g
    Push g
    Push sugar
```

```
    Call
    Return
Mut sugar g
    Push g
    Return
End
Push 9
Push snack
Call
Quit
```

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

```
    100
```

Example 7

```
Fun length ls
    Push ls
    CaseLeft
        Push 0
        Return
    Right
        Get 0
        Swap
        Pop
        Push length
        Call
        Push 1
        Add
        Return
    End
End

Push 0
InjL
Local nil

Fun cons ls
  Fun a_ x
     Push ls
     Push x
     Tuple 2
     InjR
     Return
  End
  Push a_
  Return
End

Push "this"
Push "isa"
Push "list"
Push nil
Push cons
Call
Call
Push cons
Call
Call
Push cons
Call
Call

Push length
Call
Quit
```

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

```
3
```