CSE 305: Language Interpreter Design

Due: April 18 2018, at 11:59 pm

1 Overview

The goal of this homework is to understand and build an interpreter in two languages (you may choose between Python and Java and you must use SML) for a small SML like, stack based, bytecode language. The homework is broke down into three parts. Part 1 is defined in Section 3, Part 2 is defined in Section 4, and Part 3 is defined in Section 5. Each part is worth 100 points, 50 for each language. You should spend roughly two weeks for each part. Test cases for each part will be provided in Autolab. Put your answers for Python or Java and SML in files named, respectively:

1. interpreter.py
2. interpreter.java
3. interpreter.sml

These files should contain a function, or static method in Java, called interpreter that takes two strings (`interpreter(input, ouput)`). You can submit multiple files as a tar or zip archive through Autolab. You will submit one solution for each separate part. Each part is graded individually. You may submit your solution for Part 3 for Parts 1 and 2. All parts are due at the same time, however, a suggested due date is given for Parts 1 and 2 to help you pace yourself throughout the semester. Late submissions will not be accepted and will be given a score of 0. Test cases will also be provided on Piazza for you to test your code locally.

2 Functionality

Your interpreter function (or static method) should take in two arguments, the file you are reading from (input) and the file name of your output file (output): `interpreter(input, ouput)`. Input and output will be passed in as strings that represent paths to files just like in your first homework assignment. Your function should write to the output file the contents of the final stack your interpreter produces. In the examples below the input file is read from top to bottom and each command is executed by your interpreter in the order it was read. You may find it useful to read in all of the commands into a list or other data structure prior to executing them. The input file can be arbitrarily long.

input	stack
push 1	1
quit	

input	stack
push 6	
push 2	
div	:error:
mul	3
quit	

input	stack
push 5	
neg	
push 10	30
push 20	-5
add	
quit	

input	stack
push :true:	
push 7	
push 8	
push :false:	-1
pop	:true:
sub	
quit	

input	stack
push 10	
push 2	
push 8	
mul	
add	23
push 3	
sub	
quit	

3 Part 1: Basic Computation Suggested Due Date: 3/09/2018

Your interpreter should be able to handle the following commands:

3.1 push

3.1.1 Pushing Integers to the Stack

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,

input	stack
push 5	0
push -0	5

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

input	stack
push 5	:error:
push 2.5	:error:
push -x	5

3.1.2 Pushing Strings to the Stack

push *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
push "deadpool"	batman
push "batman"	deadpool

You can assume that the string value would always be legal and not contain quotations within the string itself, i.e double quotes will not appear inside a string.

3.2 Pushing Names to the Stack

push *name*

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

1. example

input
push a
push 13

→

stack
a

→

stack
13
a

2. example

input
push name1
push 3

→

stack
name1

→

stack
3
name1

To bind 'a' to the value 13 and name1 to the value 3, we will use 'bind' operation which we will see later (Section 4.7) You can assume that name will not contain any illegal tokens no commas, quotation marks etc. It will always be a sequence of letters and digits starting with a letter.

3.3 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,

input
push 5
pop
pop

→

stack
5

→

stack

→

input
:error:

3.4 boolean

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

input	stack
push 5 push :true:	:true: 5

3.5 error

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

3.6 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,

input
push 5
push 8
add

→

stack
8
5

→

stack
13

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:

input
push 5
add

→

stack
5

→

stack
:error:
5

3.7 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 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,

input
push 5
push 8
sub

→

stack
8
5

→

stack
-3

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:

input
push 5
push :false:
sub

→

stack
5

→

stack
:false:
5

→

stack
:error:
:false:
5

3.8 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:

input		stack		stack
push 5	→	8	→	40
push 8		5		
mul				

If the stack empty when mul is executed, an error will occur and :error: should be pushed onto the stack:

input		stack		stack
mul	→		→	:error:

3.9 div

The command 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 $\frac{x}{y}$ 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:

input		stack		stack
push 5	→	8	→	0
push 8		5		
div				

If the top element in the stack equals to 0, there will be an error if div is executed. In such situations 5 and 0 should be pushed back onto the stack as well as :error:

input		stack		stack
push 5	→	0	→	:error:
push 0		5		0
div				5

3.10 rem

The command 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 $\frac{x}{y}$, 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,

input
push 5
push 8
rem

→

stack
8
5

→

stack
5

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 to elements should be pushed back onto the stack as well as :error:. For example:

input
push 5
push :false:
rem

→

stack
:false:
5

→

stack
:error:
:false:
5

3.11 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:

input
push 5
neg

→

stack
5

→

stack
-5

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:

input
push 5
neg
push :true:
neg

→

stack
-5

→

stack
:true:
-5

→

stack
:error:
:true:
-5

3.12 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:

input		stack		stack		stack
push 5	→	8	→	:false:	→	8
push 8		5		8		:false:
push :false:				5		5
swap						

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:

input		stack		stack		stack
push 5	→	5	→	:error:	→	5
swap				5		:error:
swap						

3.13 quit

The command quit causes the interpreter to stop. Then the whole stack should be printed out to an output file that is specified as the second argument to the interpret function.

4 Part 2: Variables and Scope Suggested Due Date: 3/28/2018

In part 2 of the interpreter you will be expanding the types of computation you will be able to perform, adding support for immutable variables, and structures for expressing scope.

4.1 cat

The cat command computes the concatenation of the top two elements in the stack and pushes the result onto the stack. The top two values of the stack—x and y—are popped off and the result is the string y concatenated with x.
: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
- if either of the top two elements are not strings, push the elements back onto the stack, and then push :error:
  - NB: Recall that names and strings are different.

Example:

input		stack		stack		stack
push ”hello ”	→	hello	→	world!	→	hello world!
push ”world!”				hello		
cat						

Note that strings can contain spaces, punctuation marks, and other special characters. You may assume that strings only contain ASCII characters and have no escape sequences, eg \n and \t.

input		stack		stack		stack
push ”Michael”	→	Michael	→	Scott	→	:error:
push Scott				Michael		Scott
cat						Michael

4.2 and

The command and performs the logical conjunction of the top two elements in the stack and pushes the result (a single value) onto the stack.
: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
- if either of the top two elements arent Boolean, push back the elements and push :error:

For example:

input		stack		stack		stack
push :true:	→	:true:	→	:false:	→	:false:
push :false:				:true:		
and						

Consider another example:

input		stack		stack
push :true:	→	:true:	→	:error:
and				:true:

4.3 or

The command or performs the logical disjunction of the top two elements in the stack and pushes the result (a single value)onto the stack.
: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
- if either of the top two elements arent Boolean, push back the elements and push :error:

For example:

input		stack		stack		stack
push :true:	→	:true:	→	:false:	→	:true:
push :false:				:true:		
or						

Consider another example:

input		stack		stack		stack
push :false:	→	:false:	→	khaleesi	→	:error:
push ”khaleesi”				:false:		khaleesi
or						:false:

4.4 not

The command not performs the logical negation of the top element in the stack and pushes the result (a single value)onto the stack. Since the operator is unary, it only consumes the top value from the stack. The :error: value will be pushed onto the stack if:

- the stack is empty, push :error: onto the stack
- if the top element isnt Boolean, push back the element and push :error:

For example:

input		stack		stack
push :true:	→	:true:	→	:false:
not				

Consider another example:

input		stack		stack
push 3	→	3	→	:error:
not				3

4.5 equal

The command equal refers to numeric equality (so you are not supporting string comparisons). This operator consumes the top two values on the stack and pushes the result(a single boolean value) onto the stack. The :error: value 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
- if either of the top two elements are not integers, push back the elements and push :error:

For example:

input		stack		stack		stack
push 7	→	7	→	7	→	:true:
push 7				7		
equal						

Consider another example:

input		stack		stack		stack
push 8	→	8	→	:error:	→	:error:
push 9.5				8		:error:
equal						8

4.6 lessThan

The command lessThan refers to numeric less than ordering. This operator consumes the top two values on the stack and pushes the result(a single Boolean value) onto the stack. The :error: value 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
- if either of the top two elements arent integers, push back the elements and push :error:

For example:

input
push 7
push 8
lessThan

→

stack
7

→

stack
8
7

→

stack
:true:

4.7 bind

The bind command binds a name to a value. It is evaluated by popping two values from the stack. The second value popped must be a name (see section on push for details on what constitutes a 'name'). The name is bound to the value (the first thing popped off the stack). The value can be any of the following:

- An integer
- A string
- Boolean
- :unit:
- The value of a name that has been previously bound

The name value binding is stored in an environment data structure. The result of a bind operation is :unit: which is pushed onto the stack. :error: will be pushed onto the stack if:

- If we are trying to bind an identifier to an unbound identifier, in which case all elements popped must be pushed back before pushing :error: onto the stack.
- the stack is empty, push :error: onto the stack.

4.7.1 Example 1

input
push a
push 3
bind

→

stack
a

→

stack
3
a

→

stack
:unit:

4.7.2 Example 2

input
push sum1
push 7
bind
push sum2
push 5
bind

→

stack
sum1

→

stack
7
sum1

→

stack
:unit:

→

stack
sum2
:unit:

→

stack
5
sum2
:unit:

→

stack
:unit:
:unit:

You can use bindings to hold values which could be later retrieved and used by functionalities you already implemented. For instance in the example below, an addition on a + name1 in example1, would add 13 + 3 and push the result 16 onto the stack.

4.7.3 Example 3

input
push a
push 13
bind
push name1
push 3
bind
push a
push name1
add

→

stack
a

→

stack
13
a

→

stack
:unit:

→

stack
name1
:unit:

→

stack
3
name1
:unit:

→

stack
:unit:
:unit:

→

stack
a
:unit:
:unit:

→

stack
name1
a
:unit:
:unit:

→

stack
16
:unit:
:unit:

While performing operations, if a name has no binding, push :error: onto the stack, in which case all elements popped must be pushed back before pushing :error: onto the stack. Bindings can be overwritten, for instance:

input
push a
push 9
bind
push a
push 10
bind

Here, the second bind updates the value of 'a' to 10.

Common Questions

(a) What values can _name_ be bound to?

name can be bound to integers, Boolean, string, :unit: and also previously bound values. For example,

1)

input
push a push :true: bind

would bind a to :true:

2)

input
push a 7.5 bind

would result in bind producing an :error: because a CANNOT be bound to :error:

3)

input
push b let push a push 7 bind end

would bind a to 7 and b to :unit:

4)

input
push b push 8 bind push a push b bind

would bind b to 8 and would bind a to the VALUE OF b which is 8.

5)

input
push b push a bind

would result in an :error: because you are trying to bind b to an unbound variable a.

(b) How can we bind identifiers to previously bound values?

input
push a
push 7
bind
push b
push a
bind

The first bind binds the value of a to 7. The second bind statement would result in the name b getting bound to the VALUE of a which is 7. This is how we can bind identifiers to previously bound values. Note that we are not binding b to a we are binding it to the VALUE of a.

(c) Can we have something like this:

input
push a
push 15
push a

Yes. In this case 'a' is not bound to any value yet. And the stack contains:

stack
a
15
a

If we had :

input
push a
push 15
bind
push a

The stack would be :

stack
a
:unit:

(d) Can we push the same _name_ twice to the stack? For instance , what would be the result of the following:

input
push a
push a
quit

This would result in the following stack output:

stack
a
a

Yes, you can push the same _name_ twice to the stack. Consider binding it this way :

input
push a
push a
push 2
bind

This would result in
:unit: → as a result of binding a to 2
a → as a result of pushing the first a to the stack

(e) Output of the following code:

input
push a
push 9
bind
push a
push 10
bind

This would result in the following stack output:

would result in
:unit: → as a result of second bind
:unit: → as a result of first bind

4.8 if

The if command pops three values off the stack; x,y and z. The third value popped (z, in this case) must always be a Boolean. If z is :true:, executing the if command will push y back onto the stack, and if z is :false:, executing the if will push x back onto the stack.
:error: will be pushed onto the stack if:

- the third value is not Boolean, all elements (x,y, and z) should be pushed back onto the stack before pushing :error: onto the stack.
- the stack is empty, push :error: onto the stack
- there are less than 3 values on the stack in which case all elements popped must be pushed back before pushing :error: onto the stack.

For example:

input
push :true:
push 9
push 8
if

→

stack
:true:

→

stack
9
:true:

→

stack
8
9
:true:

→

stack
9

Common Questions

(a) What values can 'if' take?

The result of executing a 'if' can be an integer or Boolean or string or :error: or :unit:

For instance,

a)

input
push :true:
push oracle
push jive
if

the result of if would be oracle

b)

input
push :false:
let
push a
push 8
bind
end
push 8.9
if

the result of if would be :error:

(b) What is the result of executing the following:

input
push a
push 5
bind
pop
push :false:
push 4
push a
if

The stack would have a. Although the value of a is bound to 5, we only resolve the name to the value if we need to perform computation. (For 'if', the only value needed for computation is Boolean.)

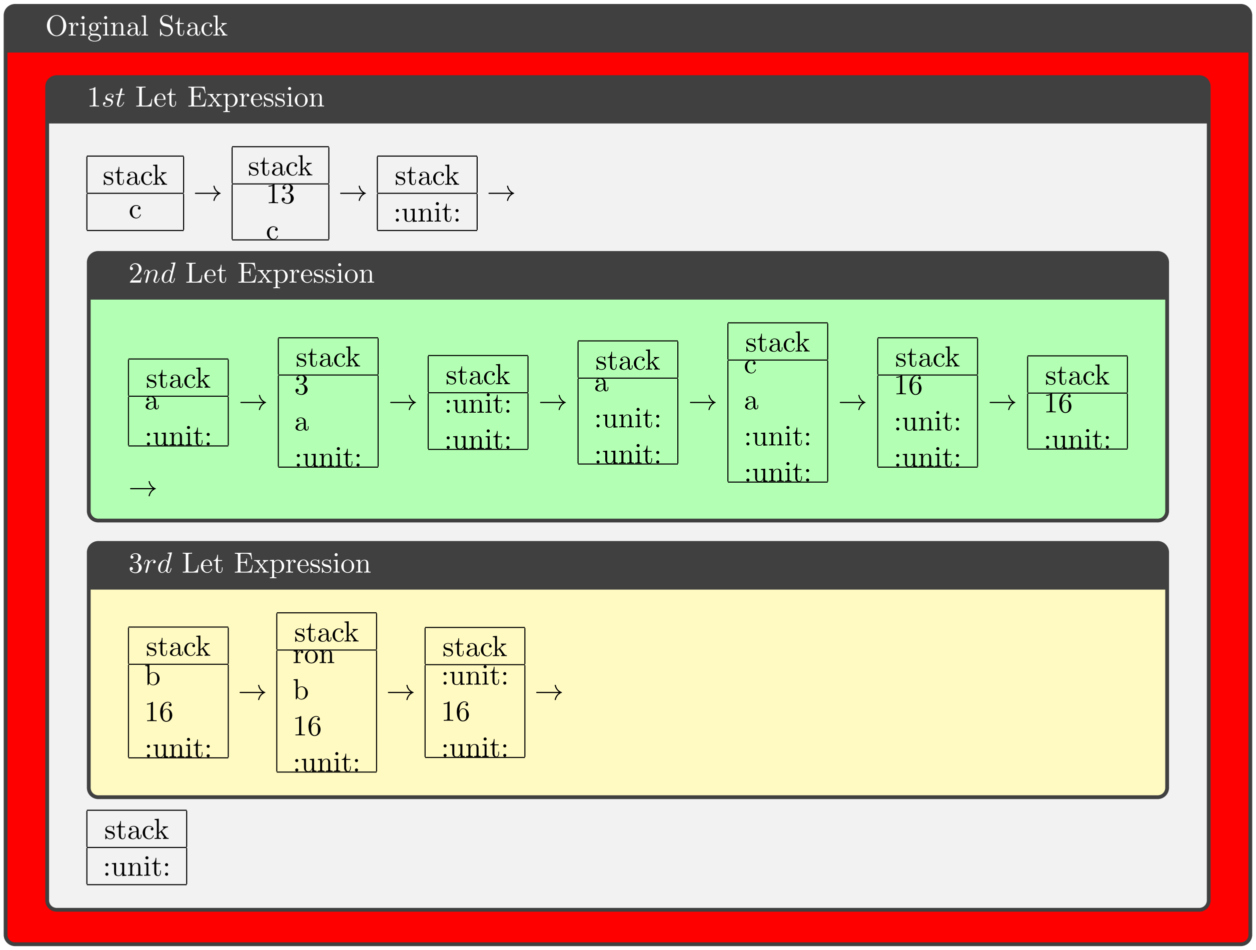
4.9 let...end

let...end limits the scope of variables. let marks the beginning of a new environment which is basically a sequence of bindings. The result of the let..end is the last stack frame of the let. Let..end can contain any number of operations but it will always result in a stack frame that is strictly larger than the stack prior to the let.

Trying to access an element that is not in scope of the let..end block would push :error: on the stack. let..end blocks can also be nested.

For example,

input
let
push c
push 13
bind
let
push a
push 3
bind
push a
push c
add
end
let
push b
push "ron"
bind
end
end



Common Questions

(a) What would be the output of running the following :

input
push 1
let
push 2
push 3
push 4
end
push 5

This would result in the stack :

stack
5
4
1

Explanation : After the let..end is executed the last frame is returned which is hy we have 4 on the stack.

(b) What would be the result of executing the following :

input
let push a1 push 7.2 bind end quit

7.2 cant be pushed to the stack and a1 cannot be bound to :error: so, the result would be :error:

(c) What would be the output of running the following:

input
let push 3 end let push b swap bind end

The stack would result in :unit:
(3 is a value not a binding and hence is not limited to the scope of the first let..end) We will NOT be testing code like this since this violates the assumption that let..end is monotonically increasing. So we do NOT expect your code to handle such cases.

(d) What would be the output of running the following code:

input
let push 3 push 10 end add quit

The stack output would be

stack
:error: 10

In the above example, the first let statement creates an empty environment (environment 1), then the name c is bound to 13. The result of this bind is a :unit: on the stack and a name value pair in the environment. The second let statement creates a second empty environment.

Name a is bound here. To add a and c, these names are first looked up for their values in the current environment. If the value isn't found in the current environment, it is searched in the outer environment. Here, c is found from environment 1. The sum is pushed to the stack. A third environment is created with one binding 'b'.The second last end is to end the scope of environment 3 and the last end statement is to end the scope of environment 1. You can assume that the stack is left with at least 1 item after the execution of any let..end block.

5 Part 3: Functions Due Date: 4/18/2018

5.1 Functions

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. Note, name1 and name2 can be any valid name, but will never be any of the keywords in our language (e.g. add, push, pop, fun, funEnd, etc.). Also the function name and argument name cannot be the same.

funEnd

Denotes the end of a function body

push funName
push arg
call

Denotes applying the function funName to the actual parameter arg. 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, boolean, or :unit:. :error: is pushed on the stack if either funName and arg are not bound in the current environment or if funName is not bound to a closure in the current environment. :error: is also pushed if the stack size is less than 2 when evaluating call.

When the interpreter encounters a function declaration expression it should being construction a closure. A closure will consist of (1) an environment, (2) the code for the function (the expressions between the function declaration and funEnd), and (3) the name of the formal parameter. :unit: should be pushed to the stack once the function declaration is evaluated and the closure created and bound to the function name in the environment.

1. The environment for the closure will be a copy of the current environment. (Challenge: if you would like to optimize your closure representation you do not need the entire environment, just the bindings of the variables used inside the function that are not defined inside the function and are not the formal parameter).

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 created 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 exists 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.

If both funName and arg have appropriate bindings, or arg is a valid value, then the call to the function can proceed. To do this push the environment stored in the closure onto the stack. To this environment add a binding between the formal parameter (extracted from the closure) and the value of the actual parameter (arg). Note that if arg is a name, then it will have a binding in the environment at the point of the call (i.e. the environment before you pushed the environment stored in the closure). You should then save the current stack and create a new stack that will be used for the execution of the function (note: you may want to implement the stack as a stack of stacks to handled nested function calls and recursion, much like implementing the environment as a stack of maps). Next retrieve the code for the function and begin executing the expressions. The function completes once the last expression in code for the function is executed. When this happens you should restore the environment to the environment that existed prior to the function call (Hint: if you are implementing your environment as a stack of local environments, this will entail popping of the top environment.). The stack should also be restored to what the stack was at the point of the call (hint: if you implemented your stack as a stack of stacks, this only requires popping of the top stack to restore the stack to what it was prior to the call). Once the environment has been restored, execution should resume with the expression that follows the call.

return

Functions can return values by using a return expression. Since functions themselves are values (a closure), this means functions can take other functions as arguments and can return functions. When a return expression is evaluated, the function stops execution. When this happens you should restore the environment to the environment that existed prior to the function call, just like if the function completed by execution the last expression in the function's code. The stack should also be restored to what the stack was at the point of the call. Additionally you should push the last stack frame the function pushed onto the restored stack (the stack at the point of the call).
Please note that background color and indentation is used only to improve readability. Closure would consist of code within colored background.

5.1.1 Example 1

input		stack
fun identity x push x return funEnd push identity push 1 call quit	→	1 :unit:

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

5.1.2 Example 2

input
fun identity x push x return funEnd
push identity push 1.2 call quit

→

stack
:error:
:error:
identity
:unit:

:error: → error as a result of calling a function with error as the actual parameter
:error: → result of pushing 1.2
identity → push of identity
:unit: → result of declaring identity

5.1.3 Example 3

input
fun identity x push x return funEnd
push x push 1 bind push identity push x call quit

→

stack
1
:unit:
:unit:

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

5.1.4 Example 4

input
push x
push 3
bind
fun addX arg
push x
push arg
add
return
funEnd
push x
push 5
bind
push a
push 3
bind
push addX
push a
call
quit

→

stack
6
:unit:
:unit:
:unit:
:unit:

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

5.1.5 Example 5

input
fun stop arg push 1 return funEnd
fun factorial arg push arg push 1 sub push 1 push arg equal push stop push factorial if swap call push arg mul return funEnd
push factorial push 3 call quit

→

stack
6
:unit:
:unit:

6 → value returned from factorial
:unit: → declaration of factorial
:unit: → declaration of stop

5.1.6 Example 6

input

```
fun add1 x
push x
push 1
add
return
funEnd

push z
push 2
bind

fun twiceZ y
push y
push z
call
push y
push z
call
push y
push z
call
add
return
funEnd

push twiceZ
push add1
call
quit
```

→

stack
6
:unit:
:unit:
:unit:

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

5.2 Functions and Let

Functions can be declared inside a Let expression. Much like the lifetime of a variable binding, the binding of a function obeys the same rules. Since Let introduces a stack of environments, the closure should also take this into account. The easiest way to implement this is for the closure to store the stack of environments present at the declaration of the function. (note: you can create a more optimal implementation by only storing the bindings of the free variables you for the function to do this you would look up each free variable in the current environment and add a binding from the free variable to the value in the environment stored in the closure)
(please note background color is used only to improve readability):

5.2.1 Example 1

input

let
  fun identity x
  push x
  return
  funEnd
end
push identity
push 1
call
quit

→

stack
:error:
1
identity
:unit:

: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 Let expression

5.2.2 Example 2

input

fun identity x
  let
  push x
  end
return
funEnd
push identity
push 1
call
quit

→

stack
1
:unit:

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

5.2.3 Example 3

input
fun double x let push x push x add end return funEnd push double push 2 call quit

→

stack
4
:unit:

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

5.2.4 Example 4

input
push y push 5 bind let push y push 7 bind fun addY x let push x push y add end return funEnd push addY push 2 call end quit

→

stack
9
:unit:

9 → return value of calling identity and passing in 2 as an argument
:unit: → result of binding y to 5

5.3 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 behave just like regular functions and all the rules defined for functions apply. In addition, when an in/out function returns, the value bound to the formal parameter is bound to the actual parameter in the environment after the call.

In/out functions should have a similar implementation to regular functions. To this implementation you should add an additional operation when the function returns. In addition to restoring the environment at the call site, the return will do a look up of formal parameter in the environment for the function. This value will be bound to the actual parameter in the environment at the call site.

input		stack
inOutFun addOne x push x push x push 1 add bind push x return funEnd push a push 1 bind push addOne push a call push a push 1 add quit	→	3 2 :unit: :unit:

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

5.4 First-Class Functions

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

input
fun makeAdder x
fun adder y push x push y add return funEnd
push adder return funEnd push makeAdder push 3 call push add3 swap bind push add3 push 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

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 push x return funEnd push identity push identity call push "Dunder Mifflin" call quit

→

stack
Dunder Mifflin
:unit:

Dunder Mifflin → Evaluated from calling the identity function after it was returned from a previous function call and passing in the string "Dunder Mifflin"
:unit: → The result of declaring the identity function

1. 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
2. You can assume that all test cases will have a quit statement at the end.
3. You can assume that your interpreter function will only be called ONCE per execution of your program

Step by step examples

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

input
push 10 push 15 push 30 sub push :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 15-30 = -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 output.txt:

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 let...end:

input
push a
push 17
add

stack		stack		stack
a	→	17	→	:error:
		a		17
				a

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

3.

input
let
push a1
push 7.2
bind
end

stack		stack		stack		stack
a1	→	:error:	→	:error:	→	:error:
		a1		:error:		
				a1		

4.

input
let
push 3
push 7
end
push 5
add
quit

stack		stack		stack		stack		stack
3	→	7	→	7	→	5	→	12
		3				7		

Explanation :

Push 3
Push 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.

You may ask - But isn't the 7 local to the let..end? 7 is not a binding it is just a value. The local scopes are only for bindings.