CSE 305 Introduction to programming languages
Spring 2016
Homework 4: Language interpreter design, part III

Assigned: April 17th , 2016 Due: on or before Friday May 6th 11:59 pm

Overview

The goal of this homework is to understand and build an interpreter in three languages (Python, SML, Java, 40 marks for each) for a small language. Your interpreter should read in an input file (`input.txt`) which contains lines of expressions, evaluate them and push the results onto a stack, then print the content of the stack to an output file (`output.txt`) when exit. In the description below, added functionality of HW4, above that of HW2 and HW3, is highlighted in yellow.

Example input and output

Some examples of the input file your interpreter takes in and the corresponding output file are shown below:

input.txt	output.txt
push 1 quit	1
push 5 Neg push 10 push 20 Add Quit	30 -5
push 10 push 2 push 8 mul add push 3 sub quit	23
push 6 push 2 div mul quit	:error: 3

input.txt	output.txt
:true: push 7 push 8 :false: pop sub quit	-1 :true:
pop push 10 swap push 6 add pop push -20 mul push 5 swap quit	-120 5 :error: 10

Functionality

Your interpreter should read in expressions from the input file named "`input.txt`", maintain a stack when running, and output the content of the stack to an output file named "`output.txt`" when stops. It should be able to handle the following expressions for this homework:

1. push

`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,

push 5
push -0

0
5

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

push 5
push 2.5

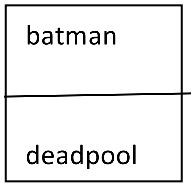
:error:
5

Pushing strings to stack:

push _string_literal_

where _string_literal_ consists of a sequence of characters enclosed in double quotation marks, as in “this is a string”. Entering this expression, would push the string onto the stack for example,

push “deadpool”
push “batman”



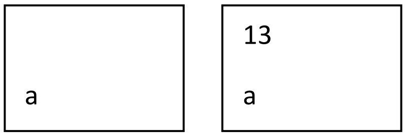
You can assume that the string value would always be legal i.e double quotes will not appear inside a string.

Pushing names to stack:

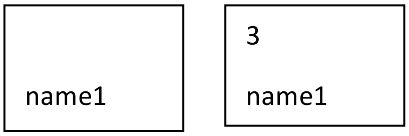
push _name_

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

push a
push 13



push name1
push 3



To bind ‘a’ to the value 13 and name1 to the value 3, we will use ‘bind’ operation which we will see later (section 18)
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.

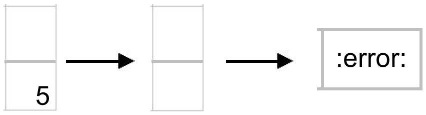
2. pop

```
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,

```
push 5
pop
pop
```

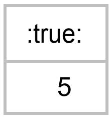


3. boolean

```
:true:
:false:
```

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

```
push 5
:true:
```



4. error

```
:error:
```

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

5. add

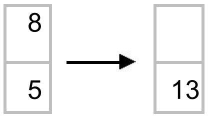
```
add
```

`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

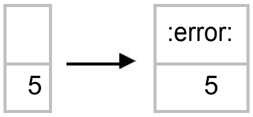
For example,

```
push 5
push 8
add
```



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:`

```
push 5
add
```



6. sub

```
sub
```

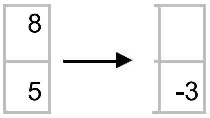
`sub` refers to integer subtraction. It is a binary operator and works in the following way:

- 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

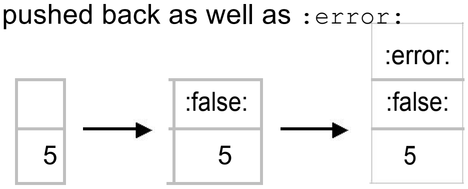
For example,

push 5
push 8
sub



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:`

push 5
:false:
sub



7. mul

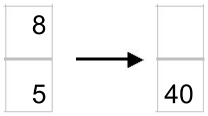
`mul`

`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

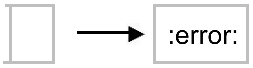
For example,

push 5
push 8
mul



For another example, if the stack empty when mul is used, an error will occur. Then :error: should be pushed onto the stack:

mul



8. div

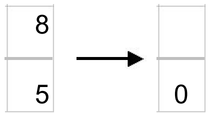
`div`

`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 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

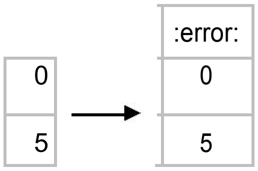
For example,

```
push 5
push 8
div
```



For another example, if the top element in the stack equals to 0, there will be an error if `div` is used. Then 5 and 0 should be pushed back as well as `:error:`

```
push 5
push 0
div
```



9. rem

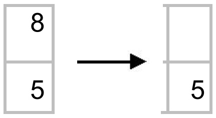
```
rem
```

`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 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

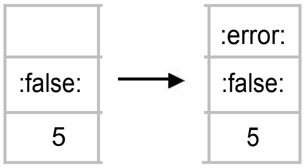
For example,

```
push 5
push 8
rem
```



For another example, if one of the top two elements in the stack is not an integer, an error will occur if `rem` is used. Then 5 and `:false:` should be pushed back as well as `:error:`

```
push 5
:false:
rem
```



10. neg

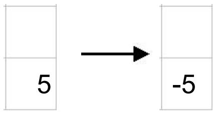
```
neg
```

`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

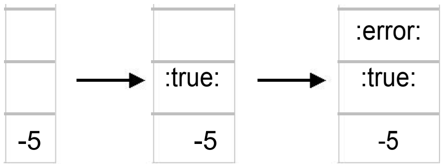
For example,

```
push 5
neg
```



For another example, if the top value is not an integer, when `neg` is used, it should be pushed back as well as `:error:`

```
push 5
neg
:true:
neg
```



11. swap

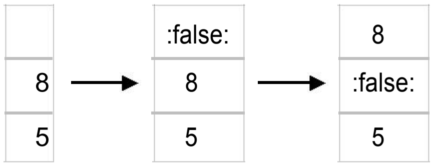
`swap`

`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

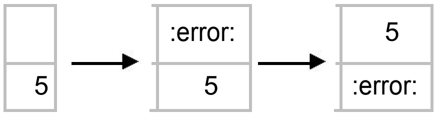
For example,

```
push 5
push 8
:false:
swap
```



For another example, 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:

```
push 5
swap
swap
```



12. quit

`quit`

`quit` causes the interpreter to stop. Then the whole stack should be printed out to an output file, named as “`output.txt`”.

13. and

`and`

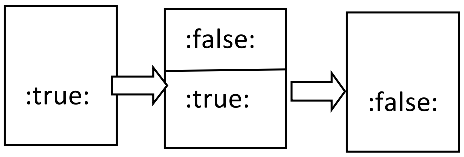
`and` performs the logical conjunction of the top two elements in the stack and pushes the result (a single value) onto the stack.

`: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 aren’t Boolean, push back the elements and push `:error:`

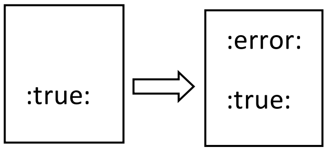
For example,

```
:true:
:false:
and
```



Another example,

```
:true:
and
```



14. or

`or`

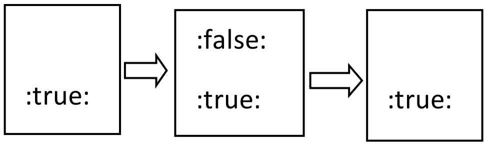
`or` performs the logical disjunction of the top two elements in the stack and pushes the result (a single value)onto the stack.

`:error:` will be pushed onto the stack if:

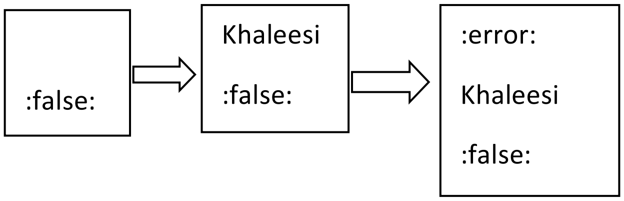
- there is only one element in the stack, push the element back and push `:error:`
- the stack is empty, push `:error:` onto the stack
- if either of the top two elements aren't Boolean, push back the elements and push `:error:`

For example,

```
:true:
:false:
or
```



```
:false:
push "Khaleesi"
or
```



15. not

`not`

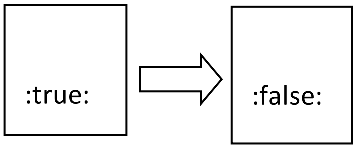
`not` performs the logical negation of the top element in the stack and pushes the result (a single value)onto the stack. Since the operator is unary, it only consumes the top value from the stack.

`:error:` will be pushed onto the stack if:

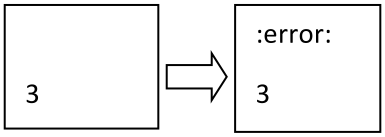
- the stack is empty, push `:error:` onto the stack
- if the top element isn't Boolean, push back the element and push `:error:`

For example,

:true:
not



push 3
not



16. equal

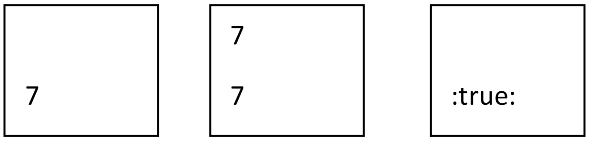
`equal`

`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.

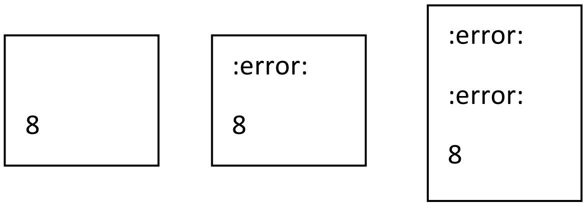
`:error:` will be pushed onto the stack if:
- there is only one element in the stack, push the element back and push `:error:`
- the stack is empty, push `:error:` onto the stack
- if either of the top two elements aren't integers, push back the elements and push `:error:`

For example,

push 7
push 7
equal



push 8
push 9.5
equal



17. lessThan

`lessThan`

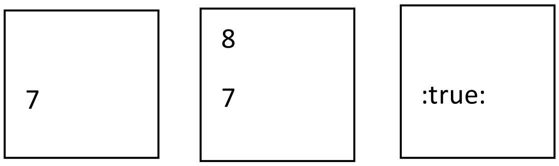
`lessThan` refers to numeric less than ordering. This operator consumes the top two values on the stack and pushes the result(a single Boolean value) onto the stack.

`: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 aren't integers, push back the elements and push `:error:`

For example,

push 7
push 8
lessThan



18.bind

`bind`

`bind` 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.

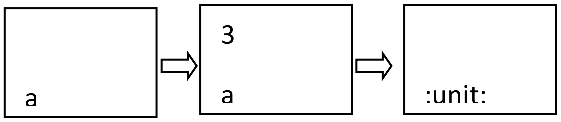
`:error:` will be pushed onto the stack if:

Modification 3/26/16

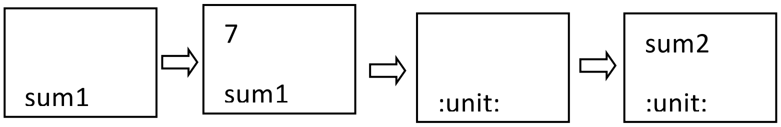
- ~~the name is already bound in the current environment~~
- If we are trying to bind an identifier to an unbound identifier.
- the stack is empty, push `:error:` onto the stack
  in which case all elements popped must be pushed back before pushing :error: onto the stack.

For example :

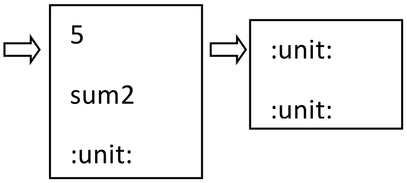
push a
push 3
bind



push sum1
push 7

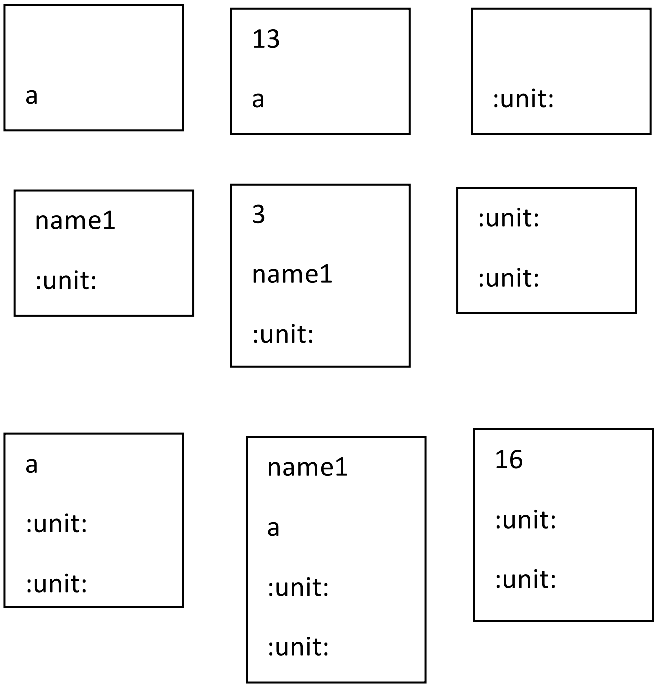


```
bind
push sum2
push 5
bind
```



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.

```
push a
push 13
bind
push name1
push 3
bind
push a
push name1
add
```



While performing operations, if a name has no binding, push :error: onto the stack.

Bindings can be overwritten, for instance:

```
push a
push 9
bind
push a
push 10
bind
```

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

19.if

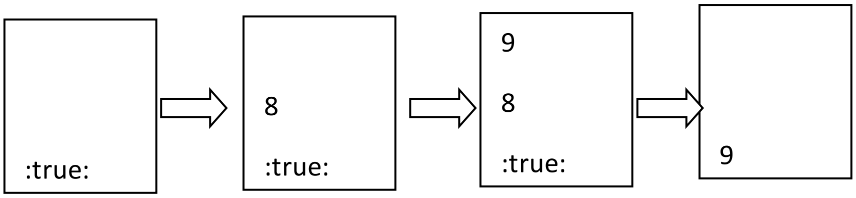
```
if
```

`if` 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:, if has the value of x, and if z is :false:, if has the value y.

`:error:` will be pushed onto the stack if:

- the third value is not Boolean.
- 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.

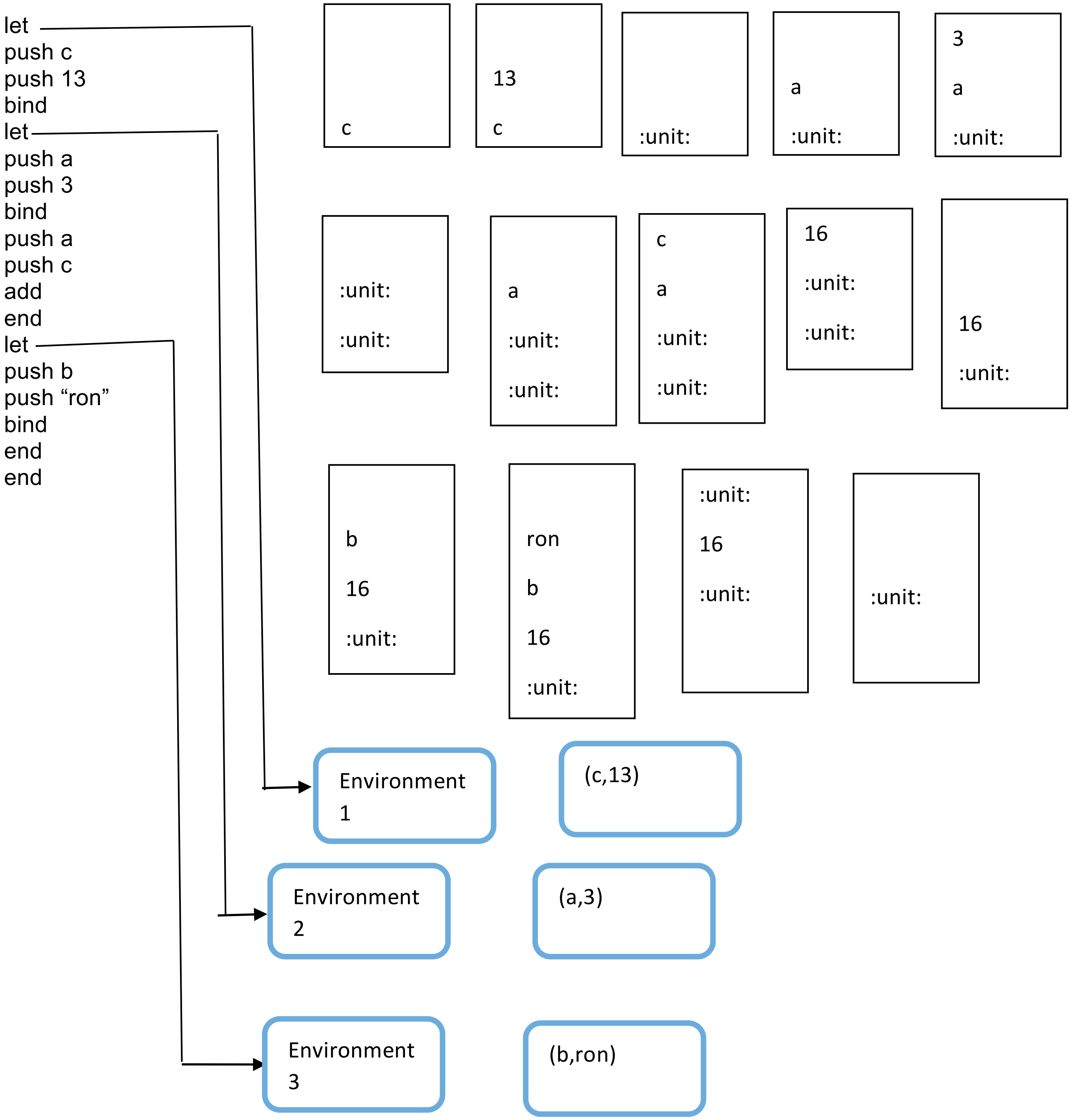
```
:true:
push 8
push 9
if
```



20. 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.



```
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
```

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.

21. 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 arg
push funName
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 expressions. When a return expressions 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). If the last frame contains a name, the value bound to that name should be returned.

Example 1 (please note indentation is used only to improve readability):

```
fun identity x  ← closure would consist of indented code, an empty env, and x
        push x
        return
funEnd
push 1
push identity
call
quit
```

Final stack would be:

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

Example 2 (please note indentation is used only to improve readability):

```
fun identity x  ← closure would consist of indented code, an empty env, and x
        push x
        return
funEnd
```

```
push 1.2
push identity
call
quit
```

Final stack would be:

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

Example 3 (please note indentation is used only to improve readability):

```
fun identity x  ← closure would consist of indented code, an empty env, and x
          push x
          return
funEnd
push x
push 1
bind ← environment would consist of (x->1)
push x
push identity
call
quit
```

Final stack would be:

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

Example 4 (please note indentation is used only to improve readability):

```
push x
push 3
bind
fun addX arg    ← closure would consist of indented code, the env {x->3}, and arg
          push x
          push arg
          add  ← adds x and arg
          return ← returns the top frame
funEnd  ← environment would consist of (x->3, addX -> closure)
push x
push 5
bind ← environment would consist of (x->5, addX -> closure)
push a
push 3
bind ← environment would consist of (x->5, addX -> closure, a-> 3)
push a
push addX
call ← the environment for the call would be (x->3, arg->3), this is created by taking the environment
from the closure we get by looking up addX in the current environment and adding a binding from
arg1 to the value stored in a in the current environment
```

```
quit
```

Final stack would be:

```
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
```

Example 5 (please note indentation is used only to improve readability):

```
fun stop arg
        push 1
        return  ← just returns 1
funEnd
(the closure for stop should be:  ("push 1 return",  {} empty environment, arg)
fun factorial arg
        push arg
        push 1
        sub   ← subtracts 1 from arg, this will be the argument passed to the call
        push 1
        push arg
        equal ← checks if arg is equal to 1
        push factorial
        push stop
        if  ← pushed either the function stop or factorial
        call  ← calls one of the functions and pushes the return value
        push arg  ← pushes arg
        mul ← multiplies arg with the return value
        return
funEnd
(the closure for factorial should be: ( all the code, {stop -> closure for stop}, arg)
push 3
push factorial
call
quit
```

Final stack would be:

```
6 ← value returned from factorial
:unit: ← declaration of factorial
:unit: ← declaration of factorial
```

Stack during the first call to factorial, step by step:

```
:arg:

1
:arg:

2  (after sub)

1
```

```
2

arg
1
2


:false:  (after equal)
2

factorial
:false:
2

stop
factorial
:false:
2

:factorial:  (after if)
2

2  (after call to factorial)

arg
2

6 (after mul)
```

Example 6 (please note indentation is used only to improve readability):

```
fun add1 x
        push x
        push 1
        add
        return
funEnd
push z
push 2
bind
fun twiceZ y
        push z
        push y
        call
        push z
        push y
        call
        add
        return
funEnd
push add1
push twiceZ ← the formal parameter y will be bound to the closure of add1
call
quit
```

Final stack would be:

```
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
```

22. 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)

Example 1 (please note indentation is used only to improve readability):

```
let
fun identity x  ← closure would consist of indented code, an empty env, and x
        push x
        return
funEnd
end
push 1
push identity
call
quit
```

Final stack would be:

```
:error: ← error since identity is not bound in the environment
identity ← push of identity
1 ← push of 1
:unit: ← result of declaring identity, this is the result of the Let expression
```

Example 2 (please note indentation is used only to improve readability):

```
fun identity x  ← closure would consist of indented code, an empty env, and x
        let
        push x
        end
        return
funEnd
push 1
push identity
call
quit
```

Final stack would be:

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

Example 3 (please note indentation is used only to improve readability):

```
fun double x  ← closure would consist of indented code, an empty env, and x
        let
        push x
        push x
        add
        end
        return
funEnd
push 2
push double
call
quit
```

Final stack would be:

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

Example 4 (please note indentation is used only to improve readability):

```
push y
push 5
bind
let
push y
push 7
bind
fun addY x  ← closure would consist of indented code, {y->7}::{y->5}, and x
        let
        push x
        push y
        add
        end
        return
funEnd
push 2
push addY
call
end
quit
```

Final stack would be:

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

23. 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.

Example 1 (please note indentation is used only to improve readability):

```
inOutFun addOne x  ← closure would consist of indented code, an empty env, and x
          push x
          push x
          push 1
          add
          bind  ← binds x to x+1
          push x
          return
funEnd
push a
push 1
bind ← environment would consist of (a->1)
push a
push addOne
call
push a ← environment would consist of (a->2)
push 1
add
quit
```

Final stack would be:
3 ← result of add (note a is bound to two)
2 ← return value of calling addOne and passing in a as an argument
:unit: ← result of binding a
:unit: ← result of declaring addOne

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

You can still assume that all test cases will have a quit statement at the end.

You can assume that your hw4 function will only be called ONCE per execution of your program

Step by step examples

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

```
input.txt
push 10
push 15
push 30
sub
:true:
swap
add
pop
neg
quit
```

First, push 10 onto the stack:

10

Similarly, push 15 and 30 onto the stack:

30
15
10

`sub` will pop the top two values from the stack, calculate 15-30 = -15, and push -15 back:

-15
10

Then push the boolean literal `:true:` onto the stack:

:true:
-15
10

`swap` consumes the top two values, interchanges them and pushes them back:

-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:`

:error:
-15
:true:
10

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

-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`":

15
:true:
10

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

More Examples of bind and let..end:

push a
push 17
add

		:error:
	17	17
a	a	a

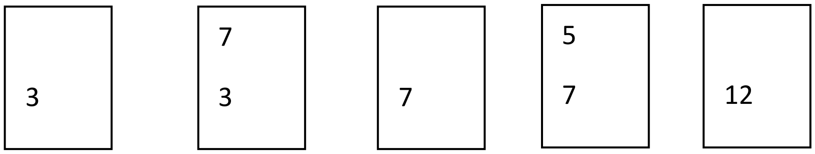
The error is because we are trying to perform an addition on an unbound variable "a".

let
push a1
push 7.2
bind
end

		:error:	
	:error:	:error:	
a1	a1	a1	:error:

Additional examples for let..end and frequently asked questions:

```
let
push 3
push 7
end
push 5
add
quit
```



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.

FAQs from Piazza:

1. What values can _name_ be bound to?

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

   ```
   push a
   :true:
   bind
   ```

   would bind a to :true:

   ```
   push a
   push 7.5
   bind
   ```

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

```
push b
let
push a
push 7
bind
end
bind
```

would bind a to 7 and b to :unit:

```
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.

```
push b
push a
bind
```

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

2. 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,

```
:true:
push “oracle”
push “jive”
if
```

the result of if would be “jive”

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

the result of if would be :unit:

3. What is the result of executing the following:

```
push a
push 5
bind
pop
:true:
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. What would be the result of executing the following :

```
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:

5. How can we bind identifiers to previously bound values?

```
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.

6. What would be the output of running the following :

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

   This would result in the 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.

7. Can we have something like this:

   ```
   push a
   push 15
   push a
   ```

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

   ```
   a
   15
   a
   ```

   If we had :

   ```
   push a
   push 15
   bind
   push a
   ```

   The stack would be :

   ```
   a
   :unit:
   ```

8. What would be the output of running the following:

   ```
   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.

9. What would be the output of running the following code:

```
let
push 3
push 10
end
add
quit
```

The stack output would be

```
:error:
10
```

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

```
push a
push a
quit
```

This would result in the following stack output:

```
a
a
```

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

```
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

11. Output of the following code:

```
push a
push 9
bind
```

```
push a
push 10
bind

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

IMPORTANT PLEASE READ:

What to submit

Define a function named 'hw4', which takes in two strings as input arguments (first one is the name of an input file and the second is the name of an output file). Function signatures of 'hw4' will be the same as they are specified in homework 1 **(please see instructions in 'HW1_Modified.pdf').**

Create a folder `UBITName_HW4` which contains three sub folders: `Python`, `SML` and `Java`. The '`Python`' folder should contain **ONLY** '`hw4.py`', '`SML`' folder should contain **ONLY** '`hw4.sml`', and '`Java`' folder should contain **ONLY** '`hw4.java`'. **Please DO NOT submit any input output files or test scripts.**

Compress the `UBITName_HW4` folder to either '`UBITName_HW4.zip`' or '`UBITName_HW4.tar`', and submit it on Timberlake using the command `submit_cse305 your_file_name`.

Auto-grader will be used to grade your submission, so please follow the exact naming conventions mentioned above. If your code fails the auto grader due to incorrect naming conventions, you will receive a 0.

No late submissions will be accepted (unless you decide to use your "free days"). So please start early.

BEFORE YOU SUBMIT :

- **Did you name your function hw4() – for the SML, Java and Python part?**

- **Did you remove the function call to hw4() from the Python and SML part?**
- **Did you remove the Tester file/ class file/ input output files/ any extra files from the Python, Java and SML sub folders?**
- **Is your home folder called yourUBITName_HW4 ?**
- **Does it have ONLY three sub folders in it called Java, Python and SML?**
- **Does the Java folder have JUST ONE FILE called hw4.java?**
- **Does the SML folder have JUST ONE FILE called hw4.sml?**
- **Does the Python folder have JUST ONE FILE called hw4.py?**
- **Did you zip/ tar your home folder? (NO tar.gz or any other forms of compression please)**
- **Before you submit, did you unzip/untar it and double check to make sure you aren’t making an empty file submission?**