CS 320: Language Interpreter Design

Part 1 Due: October 29, at 11:59pm
Part 2 Due: November 12, at 11:59pm
Part 3 Due: November 26, at 11:59pm

1 Overview

The goal of this project is to understand and build an interpreter for a small, OCaml-like, stack-based bytecode language. You will be implementing this interpreter in OCaml, like the previous assignments. The project is broken down into three parts. Part 1 is defined in Section 4, Part 2 is defined in Section 5, and Part 3 is defined in Section 6. Each part is worth 80 points.

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

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

If your program does not match the type signature, it will not compile on Gradescope and you will receive 0 points. You may, however, have helper functions defined outside of `interpreter`—the grader is only explicitly concerned with the type of `interpreter`.

You must submit a solution for each part and each part is graded individually. Late submissions will not be accepted and will be given a score of 0. Test cases/sample will also be provided on Piazza for you to test your code locally. These will not be exhaustive, so you are highly encouraged to write your own tests to check your interpreter against all the functionality described in this document.

2 Functionality

Given the following function header:

```
let interpreter (input : string) (output : string ) : unit = ...
```

`input` file name and `output` file name will be passed in as strings that represent paths to files just like in the Pangram assignment. Your function should write to the contents of the final stack your interpreter produces to the file specified by `output`. In the examples below, the input file is read from top to bottom and then each command is executed by your interpreter in the order it was read. It is incredibly useful to read in all of the commands into a list prior to executing them, separating input from the actual interpretation of the commands. The input file can be arbitrarily long. You may find the library function ***String.split_on_char*** to be useful for separating a string into a string list.

input	stack
PushI 1 Quit	1

input	stack
PushI 6 PushI 2 Div Mul Quit	<error> 3

input	stack
PushI 5 Neg PushI 10 PushI 20 Add Quit	30 -5

input	stack
PushB <true> PushI 7 PushI 8 PushB <false> Pop Sub Quit	-1 <true>

input	stack
PushI 10 PushI 2 PushI 8 Mul Add PushI 3 Sub Quit	23

3 Grammar

The following is a context free grammar for the bytecode language you will be implementing. Terminal symbols are identified by `monospace font`, and nonterminal symbols are identified by *italic font*. Anything enclosed in [brackets] denotes an optional character (zero or one occurrences). The form '(set_1 | set_2 | set_n)' means a choice of one character from any one of the n sets. A set enclosed in {braces means zero or more occurrences}.

The set *digit* is the set of digits {0,1,2,3,4,5,6,7,8,9}, *letter* is the set of all characters in the English alphabet (lowercase and uppercase), and *ASCII* is the ASCII character set. The set *simpleASCII* is *ASCII* without quotation marks and the backslash character. Do note that this necessarily implies that escape sequences will not need to be handled in your code.

3.1 Constants

const ::= *int* | *bool* | *error* | *string* | *name* | *unit*

int ::= [−] *digit* { *digit* }

bool ::= `<true>` | `<false>`

error ::= `<error>`

unit ::= `<unit>`

string ::= "*simpleASCII* { simpleASCII }"

simpleASCII ::= *ASCII* \ {'\', '"'}

name ::= {_} *letter* {*letter* | *digit* | _}

3.2 Programs

prog ::= *com* {*com*} `Quit`

com ::= `PushI` *const* | `PushB` *const* | `PushS` *const* | `PushN` *const* | `Push` *const* | `Add` | `Sub` | `Mul` | `Div` | `Rem` | `Neg` | `Swap` | `Pop` | `Concat` | `And` | `Or` | `Not` | `LessThan` | `Equal` | `If` | `Bind`
|
`Begin` *com* {*com*} `End` | *funBind com* {*com*} [`Return`] `FunEnd` | `Call`

funBind ::= (`Fun` | `InOutFun`) $name_1$ $name_2$

4 Part 1: Basic Computation
Due Date: October 27, at 11:59pm

Your interpreter should be able to handle the following commands:

4.1 Push

4.1.1 pushing Integers to the Stack

PushI *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
PushI 5	0
PushI -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
PushI 5	<error>
PushI 2.5	<error>
PushI x	<error>

4.1.2 pushing Strings to the Stack

PushS *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
PushS "deadpool"	this a string
PushS "batman"	batman
PushS "this is a string"	deadpool

Spaces are preserved in the string, i.e. any preceding or trailing whitespace must be kept inside the string that is Pushed to the stack:

input	stack
PushS " deadp ool "	this␣is␣a␣string␣␣
PushS "this is a string "	␣deadp␣ool␣

You can assume that the string value would always be legal and not contain quotations or escape sequences within the string itself, i.e. neither double quotes nor backslashes will appear inside a string.

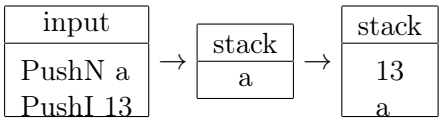
If *string* is not string, only Push the error literal (`<error>`) onto the stack instead of pushing *string*.

4.2 pushing Names to the Stack

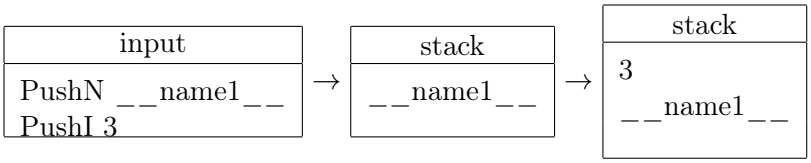
PushN *name*

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

1. example



2. example



If *name* does not conform to previously mentioned specifications, only Push the error literal (`<error>`) onto the stack instead of pushing *name*.

To bind 'a' to the value 13 and `__name1__` to the value 3, we will use the 'Bind' operation which we will see later (Section 5.7) You can assume that name will not contain any illegal tokens—no commas, quotation marks, etc. It will always be a sequence of letters, digits, and underscores, starting with a letter (uppercase or lowercase) or an underscore.

4.3 boolean

PushB *bool*

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

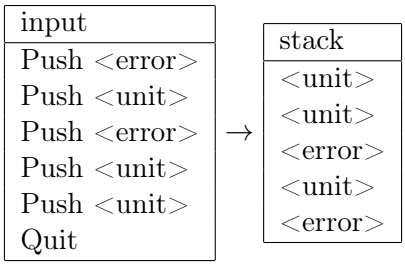
input	stack
PushI 5 PushB <true>	<true> 5

If *bool* is not a boolean, only Push the error literal (`<error>`) onto the stack instead of pushing *bool*.

4.4 error and unit

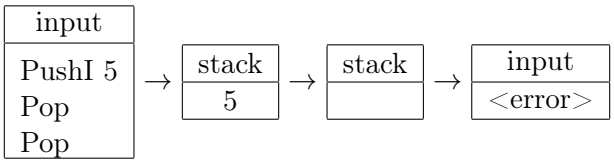
Push <error>
Push <unit>

Similar with boolean literals, pushing an error literal or unit literal will Push `<error>` or `<unit>` onto the stack, respectively. If anything except these two literals are passed to Push (without the suffix), Push `<error>` onto the stack.



4.5 Pop

The command Pop removes the top value from the stack. If the stack is empty, an error literal (`<error>`) will be Pushed onto the stack. For example,

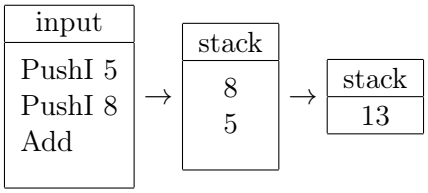


4.6 Add

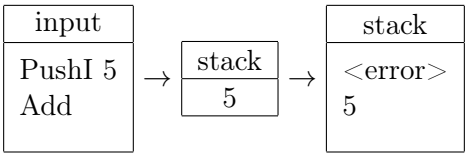
The command Add refers to integer addition. Since this is a binary operator, it consumes the top two values in the stack, calculates the sum and Pushes the result back to the stack. If one of the following cases occurs, which means there is an error, any values popped out from the stack should be Pushed 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, the following non-error case:



Alternately, 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>`

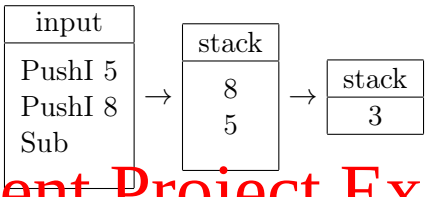


4.7 Sub

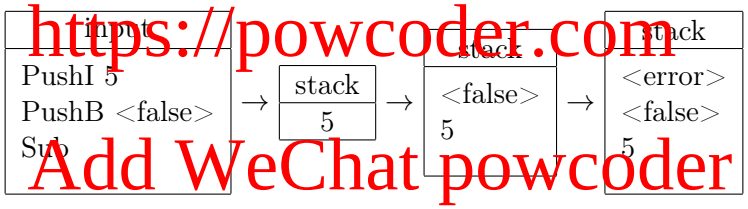
The command Sub refers to integer subtraction. It is a binary operator and works in the following way:

- if top two elements in the stack are integer numbers, pop the top element(y) and the next element(x), subtract x from y, and Push the result y-x 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, the following non-error case:



Alternately, 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>`

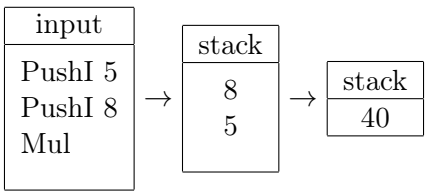


4.8 Mul

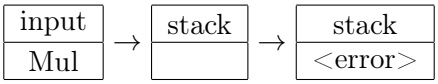
The command Mul refers to integer multiplication. It is a binary operator and works in the following way:

- 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, the following non-error case:



Alternately, if the stack empty when Mul is executed, an error will occur and `<error>` should be Pushed onto the stack:

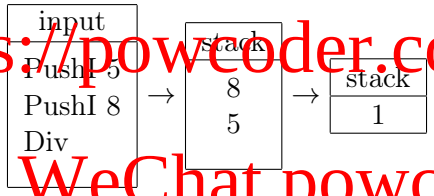


4.9 Div

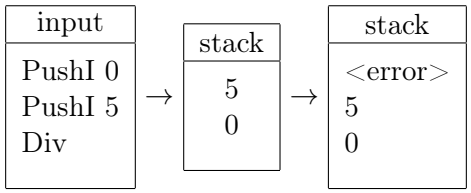
The command Div refers to integer division. It is a binary operator and works in the following way:

- if top two elements in the stack are integer numbers, pop the top element(y) and the next element(x), divide y by x, and push the result $\frac{y}{x}$ back onto the stack
- if top two elements in the stack are integer numbers but x 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, the following non-error case:



Alternately, if the second top element in the stack equals to 0, there will be an error if Div is executed. In such situations 0 and 5 should be Pushed back onto the stack as well as `<error>`



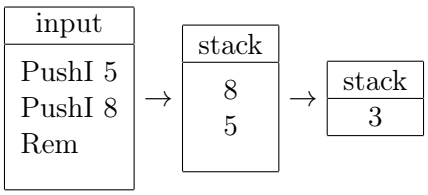
4.10 Rem

The command Rem refers to the remainder of integer division. It is a binary operator and works in the following way:

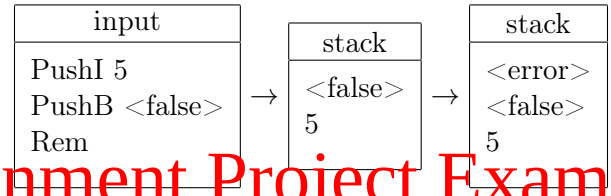
- if top two elements in the stack are integer numbers, pop the top element(y) and the next element(x), calculate the remainder of $\frac{y}{x}$, and Push the result back onto the stack
- if top two elements in the stack are integer numbers but x equals to 0, Push them back in the same order and Push `<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, the following non-error case:



Alternately, 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 two elements should be Pushed back onto the stack as well as `<error>`. For example:

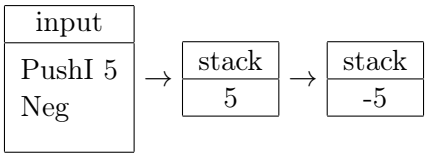


4.11 Neg

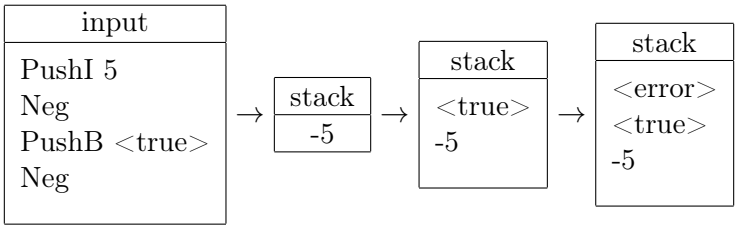
The command Neg is to calculate the negation of an integer (negation of 0 should still be 0). It is unary therefore consumes only the top element from the stack, calculate its negation and Push the result back. A value `<error>` will be Pushed onto the stack if:

- 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, the following non-error case:



Alternately, 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:

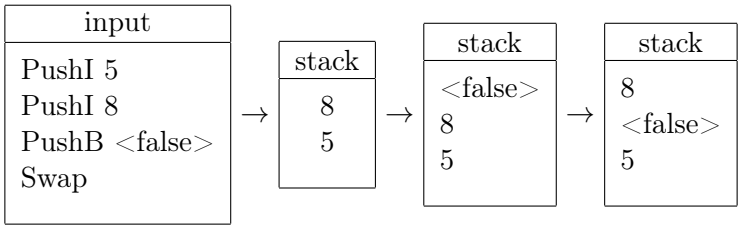


4.12 Swap

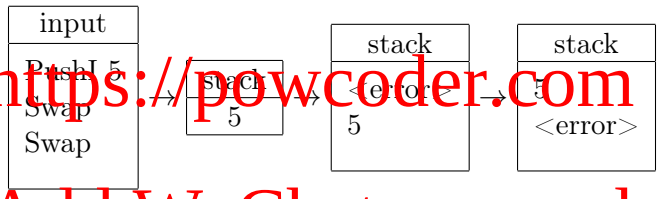
The command Swap interchanges the top two elements in the stack, meaning that the first element becomes the second and the second becomes the first. A value `<error>` will be Pushed onto the stack if:

- 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, the following non-error case:



Alternately, 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:



4.13 Quit

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

5 Part 2: Variables and Scope
Due date: November 10, at 11:59pm

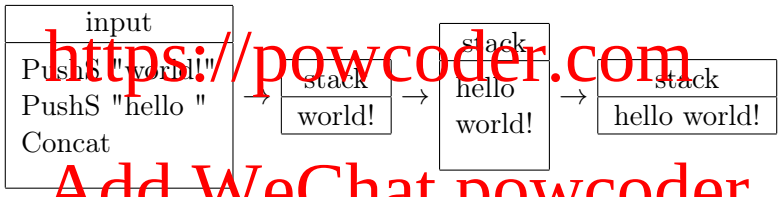
In part 2 of the interpreter you will be expanding the types of computation you will be able to perform, adding support for immutable variables and structures for expressing scope.

5.1 Concat

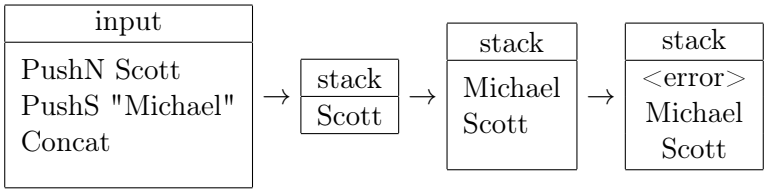
The Concat command computes the concatenation of the top two elements in the stack and Pushes the result onto the stack. The top two values of the stack — x and y — are popped off and the result is the string x concatenated onto y.
`<error>` will be Pushed onto the stack if:

- 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>`
  - Hint: Recall that names and strings are different.

For example:



Consider another example:



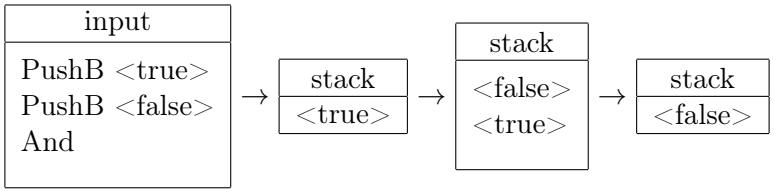
Note that strings can contain spaces, punctuation marks, and other special characters. You may assume that strings only contain ASCII characters and have no escape sequences, e.g. \n and \t.

5.2 And

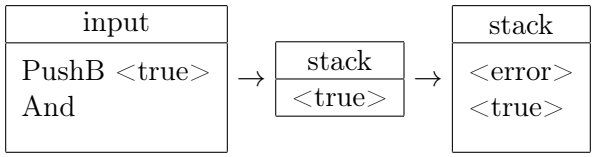
The command And performs the logical conjunction of the top two elements in the stack and Pushes the result (a single value) onto the stack.
`<error>` will be Pushed onto the stack if:

- 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 are not booleans, Push back the elements and Push `<error>`

For example:



Consider another example:



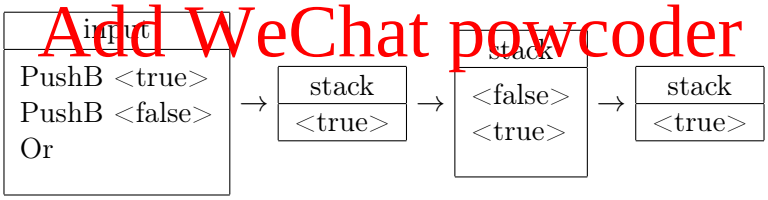
5.3 Or

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:

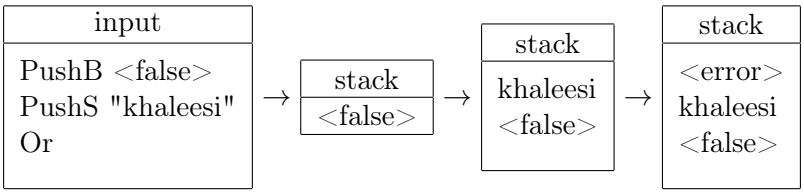
- there is only one element in the stack, Push the element back and Push `<error>`
- the stack is empty, Push `<error>` onto the stack

- if either of the top two elements are not Booleans, Push back the elements and Push `<error>`

For example:



Consider another example:

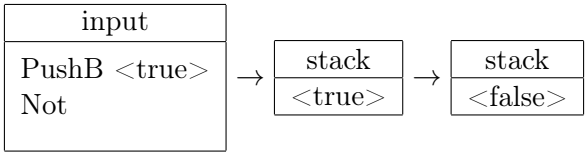


5.4 Not

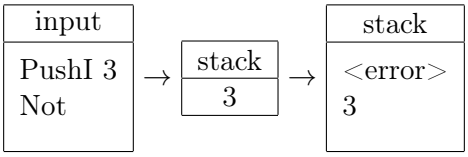
The command Not performs the logical negation of the top element in the stack and Pushes the result (a single value) onto the stack. Since the operator is unary, it only consumes the top value from the stack. The `<error>` value will be Pushed onto the stack if:

- the stack is empty, Push `<error>` onto the stack
- if the top element is not a boolean, Push back the element and Push `<error>`

For example:



Consider another example:

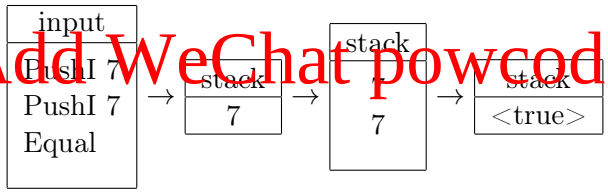


5.5 Equal

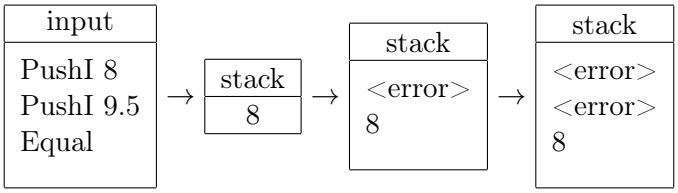
The command Equal refers to numeric equality (so you are not supporting string comparisons). This operator consumes the top two values on the stack and Pushes the result (a single boolean value) onto the stack. The `<error>` value will be Pushed onto the stack if:

- 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:



Consider another example:

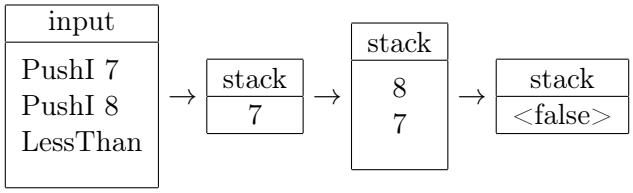


5.6 LessThan

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 aren't integers, Push back the elements and Push `<error>`

For example:



5.7 Bind

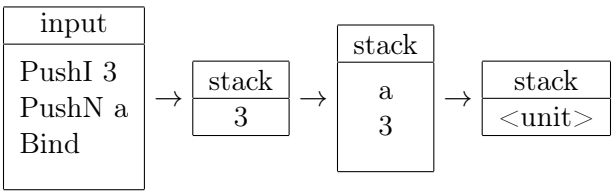
The Bind command binds a name to a value. It is evaluated by popping two values from the stack. The first value popped must be a name (see section 4.2 for details on what constitutes a 'name'). The name is bound to the value (the second thing popped off the stack). The value can be any of the following:

- An integer
- A string
- A boolean
- `<unit>`
- The *value* of a name that has been previously bound

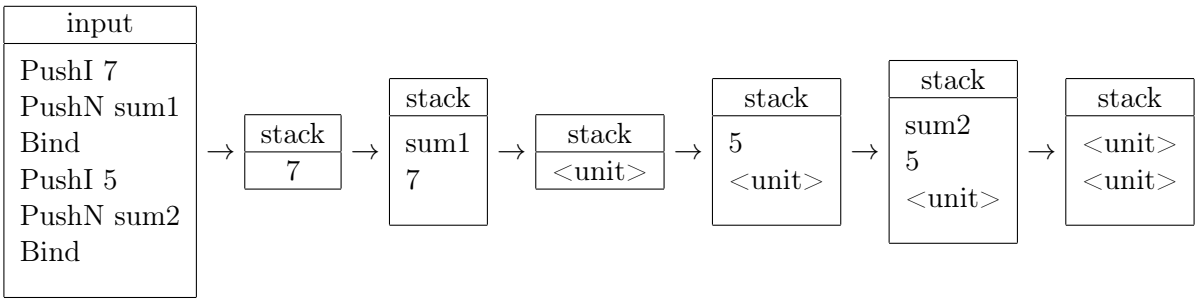
The name value binding is stored in an environment data structure. The result of a Bind operation is `<unit>` which is Pushed onto the stack. The value `<error>` will be Pushed onto the stack if:

- 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.

5.7.1 Example 1



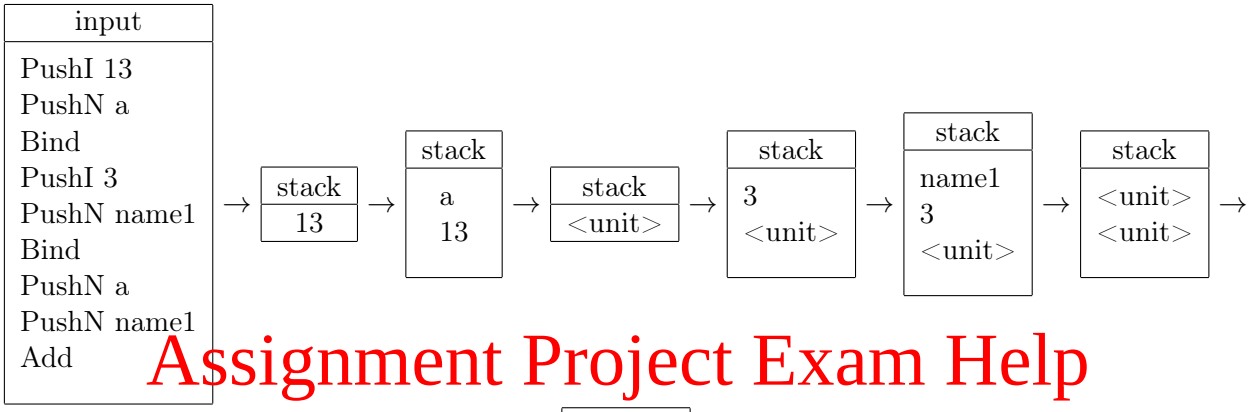
5.7.2 Example 2



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* and *name1* would add 13 + 3 and Push the result 16 onto the stack.

This, in effect, allows names to be in place of proper constants in all the operations we've seen so far. Take for example, when you encounter a name in an Add operation, you should retrieve the value the name is bound to, if any. Then if the value the name is bound to has the proper type, you can perform the operation.

5.7.3 Example 3



Notice how we can substitute a constant for a bound name and the commands work as we expect. The idea is that when we encounter names in a command, we resolve the name to the value it's bound to, and then use that value in the operation.

5.8 Example 4

input
PushI 5
PushN a
Bind
Pop
PushI 3
PushN a
Add
PushS "str"
PushN b
Bind
Pop
PushI 10
PushN b
Sub
Quit

→

stack
<error>
b
10
8

You can see that the Add operation completes, because a is bound to an integer (5, specifically). The Sub operation fails because b is bound to a string, and thus does not type-check. While performing operations, if a name has no binding or it evaluates to an improper type, Push `<error>` onto the stack, in which case all elements popped must be Pushed back before pushing `<error>` onto the stack.

5.9 Example 5

Bindings can be overwritten for instance:

input
PushI 9
PushN a
Bind
PushI 10
PushN a
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, booleans, strings, `<unit>` and also previously bound values. For example,

1)

input
PushB <true> PushN a Bind

would bind a to `<true>`

2)

input
PushI 7.5 PushN a Bind

would result in Bind producing an `<error>` because a CANNOT be bound to `<error>`

3)

input
Begin PushI 7 PushN a Bind End PushN b Bind

would bind a to 7 and b to `<unit>`

4)

input
PushI 8 PushN b Bind PushN b PushN a Bind

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

5)

input
PushN b PushN 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
PushI 7 PushN a Bind PushN a PushN b 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
PushI 15 PushN a PushN a

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

stack
a a 15

If we had:

input
PushI 15 PushN a Bind PushN 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
PushN a PushN 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
PushI 2 PushN a PushN a Bind

This would result in
`<error>` $\rightarrow$ as we cannot bind a unbound name a to a name a
$a \rightarrow$ as a result of pushing the second a to the stack
$a \rightarrow$ as a result of pushing the first a to the stack
$2 \rightarrow$ as a result of pushing the first 2 to the stack

(e) Output of the following code:

input
PushI 9 PushN a Bind PushI 10 PushN a Bind

This would result in the following stack output:

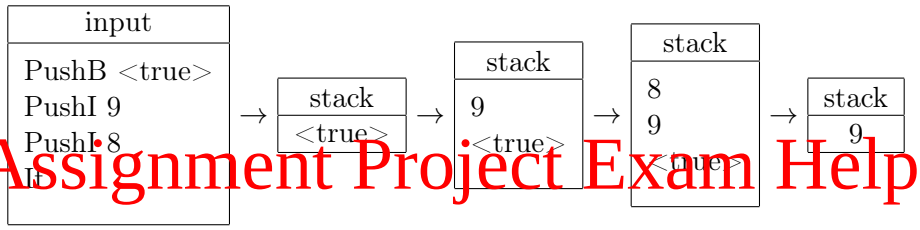
would result in
`<unit>` $\rightarrow$ as a result of second Bind
`<unit>` $\rightarrow$ as a result of first Bind

5.10 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 a 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:



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,

1)

input
PushB <true> PushS "oracle" PushS "jive" If

the result of If would be oracle

2)

input
PushB <false> PushI 8.9 Begin PushI 8 PushN a Bind End If

the result of If would be `<unit>`

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

input
PushI 5 PushN a Bind Pop PushB <true> PushN a PushI 4 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 a boolean.)

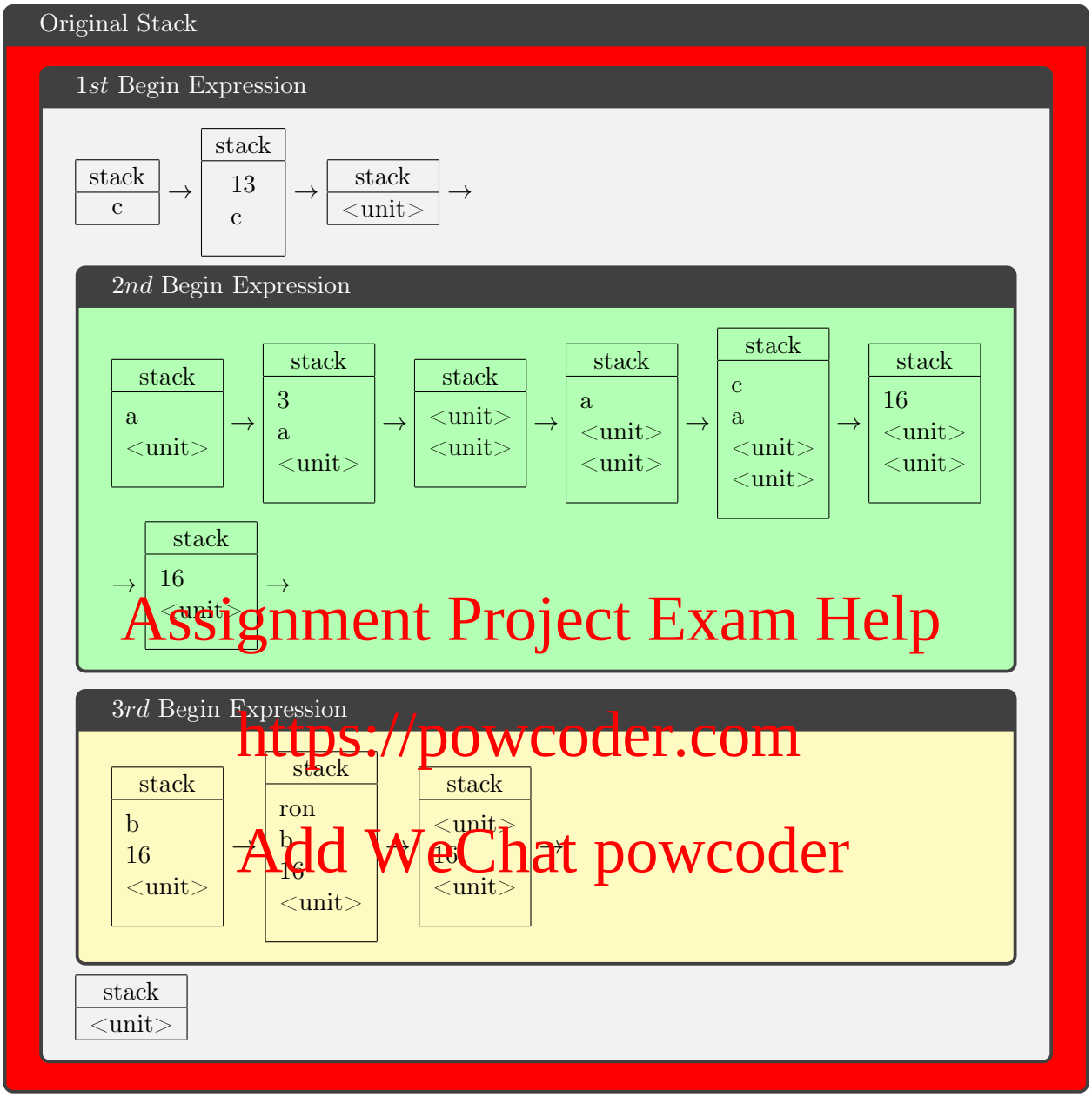
5.11 Begin...End

Begin...End limits the scope of variables. "Begin" marks the beginning of a new environment—which is basically a sequence of bindings. The result of the Begin...End is the last stack frame of the Begin. Begin...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 Begin.

Trying to access an element that is not in scope of the Begin...End block would Push `<error>` on the stack. Begin...End blocks can also be nested.

For example,

input
Begin PushI 13 PushN c Bind Begin PushI 3 PushN a Bind PushN a PushN c Add End Begin PushS "ron" PushN b Bind End End



In the above example, the first Begin 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 Begin 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 Begin...End block.

Common Questions

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

input
PushI 1 Begin PushI 2 PushI 3 PushI 4 End PushI 5

This would result in the stack:

stack
5 4 1

Explanation: After the Begin...End is executed the last frame is returned—which is why we have 4 on the stack.

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

input
Begin PushI 7.2 PushN a1 Bind End Quit

7.2 can't 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 code:

input
Begin PushI 3 PushI 10 End Add Quit

The stack output would be:

stack
<error> 10

6 Part 3: Functions Due date: November 24, at 11:59pm

6.1 Function declaration and Call

Fun *name1 name2*

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

PushN *funName*
Push *arg*
Call

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

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. The value `<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 create a binding between the function name and its closure.

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

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 [1] and the value of the actual

[1] you will extract the formal parameter from the closure

parameter (i.e. the argument). Note that if *arg* is a name, then it must have a binding in the environment at the point of the Call [2]. You should then save the current stack and create a new stack that will be used for the execution of the function [3]. 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 [4]. The stack should also be restored to what the stack was at the point of the Call [5]. Once the environment has been restored, execution should resume with the expression that follows the Call.

6.2 Return

Functions can return values by using a Return expression. Since functions themselves are values (a closure), 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 executing 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.

6.3 Examples

6.3.1 Example 1

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

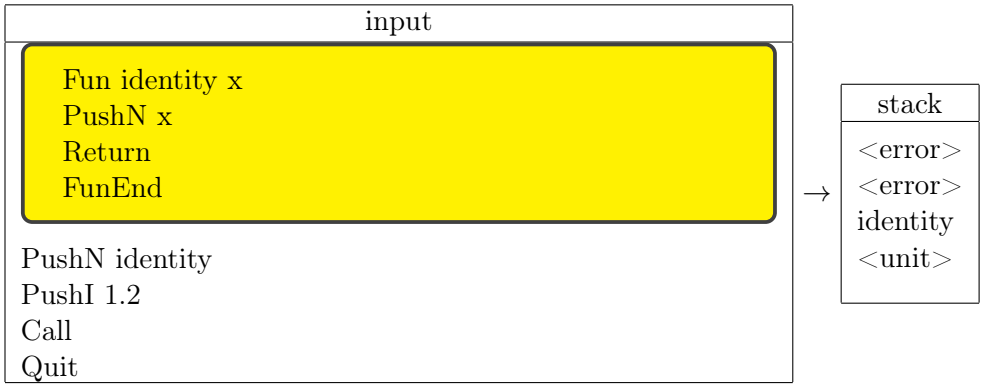
[2] this is the current environment before you Push the closure's environment

[3] hint: 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

[4] hint: if you are implementing your environment as a stack of local environments, this will entail popping off the top environment

[5] hint: if you implemented your stack as a stack of stacks, this only requires popping off the top stack to restore the stack to what it was prior to the Call

6.3.2 Example 2

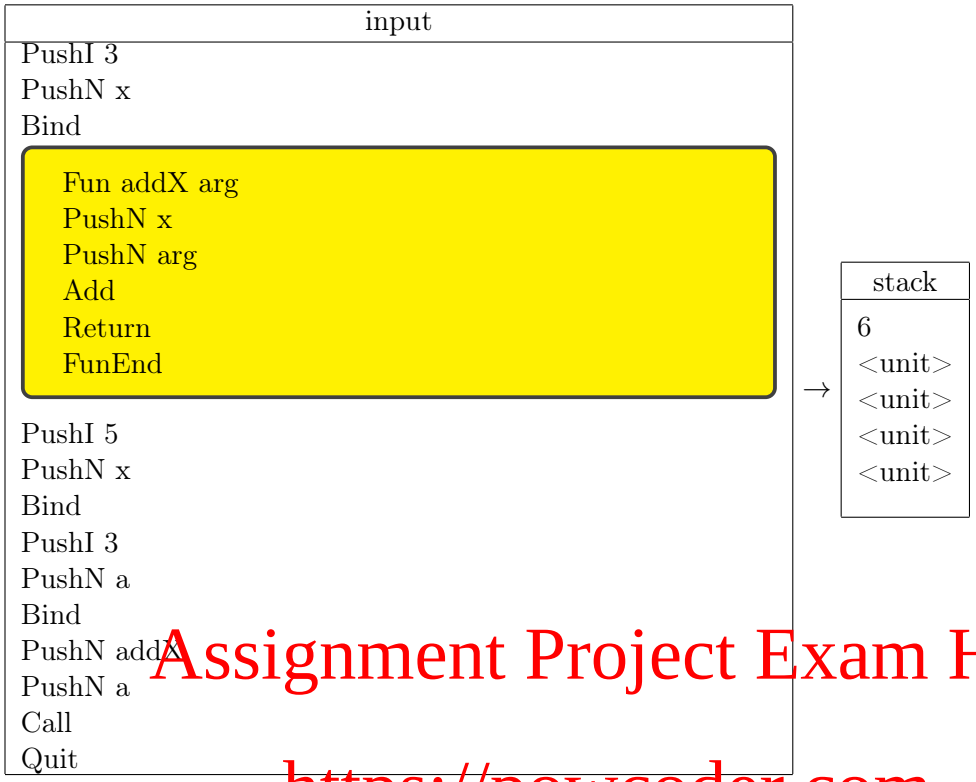


<error> → error as a result of calling a function with error as the actual parameter
<error> → result of pushing 1.2
identity → Push of identity
<unit> → result of declaring identity

6.3.3 Example 3

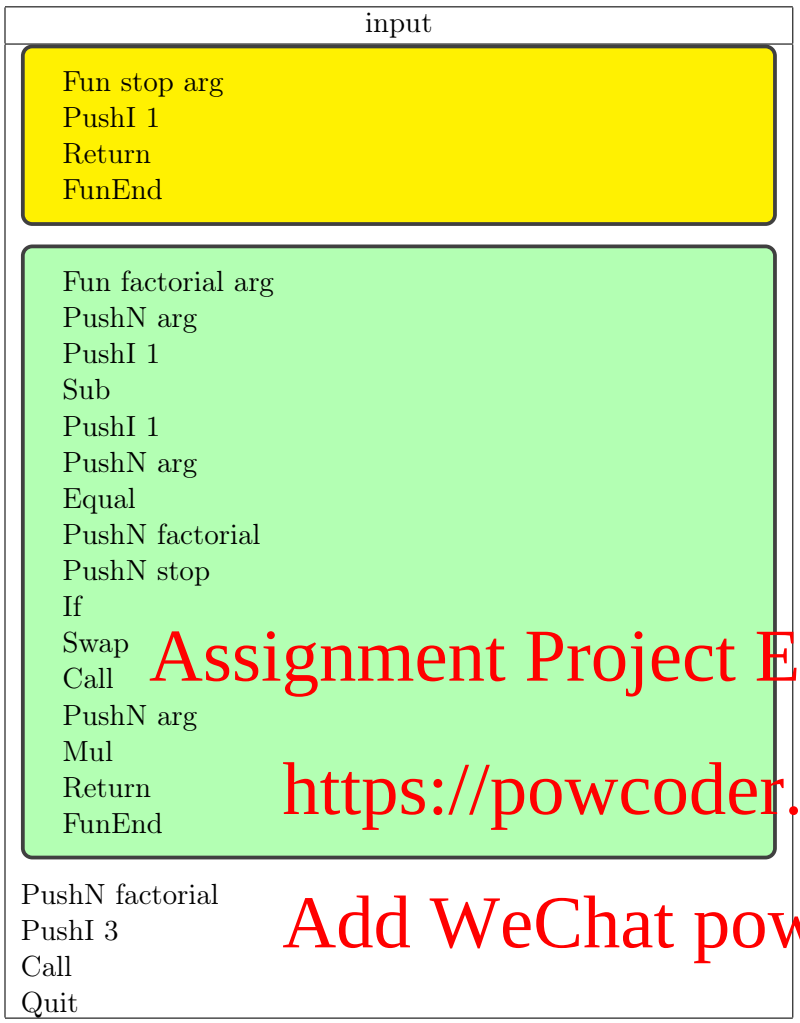
1 → return value of calling identity and passing in x as an argument
<unit> → result of binding x
<unit> → result of declaring identity

6.3.4 Example 4



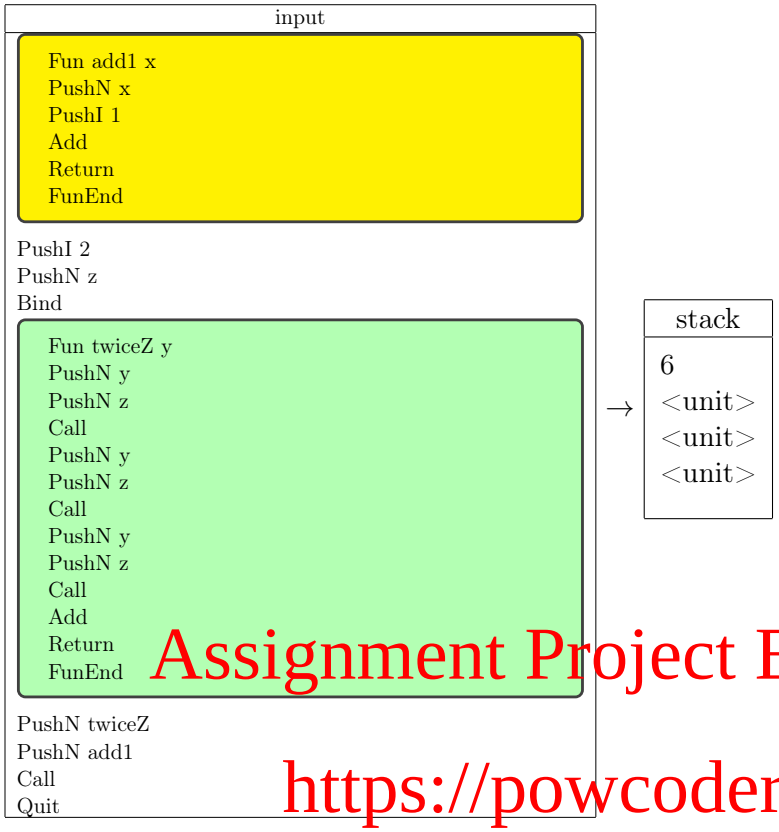
6 → result of function call
<unit> → result of third binding
<unit> → result of second binding
<unit> → result of function declaration
<unit> → result of first binding

6.3.5 Example 5



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

6.3.6 Example 6



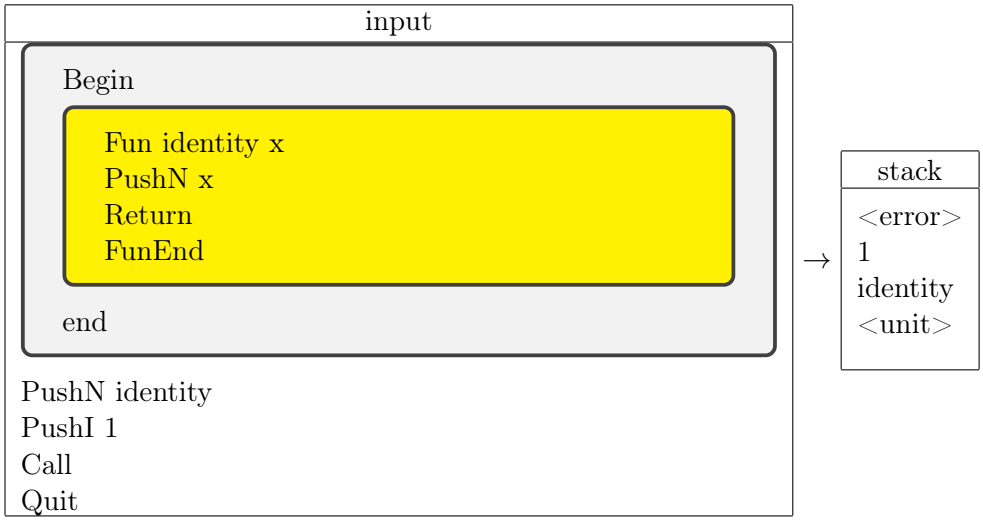
6 → return of calling twiceZ and passing add1 as an argument
<unit> → declaration of twiceZ
<unit> → binding of z

<unit> → declaration of the add1 function

6.4 Functions and Begin

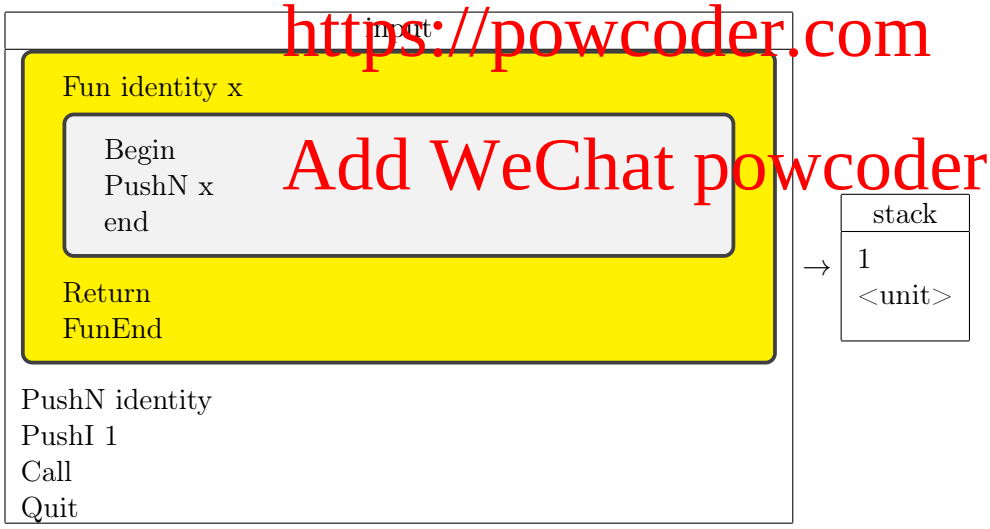
Functions can be declared inside a Begin expression. Much like the lifetime of a variable binding, the binding of a function obeys the same rules. Since Begin introduces a stack of environments, the closure should also take this into account. The easiest way to implement this is for the closure to store 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 used in 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):

6.4.1 Example 1



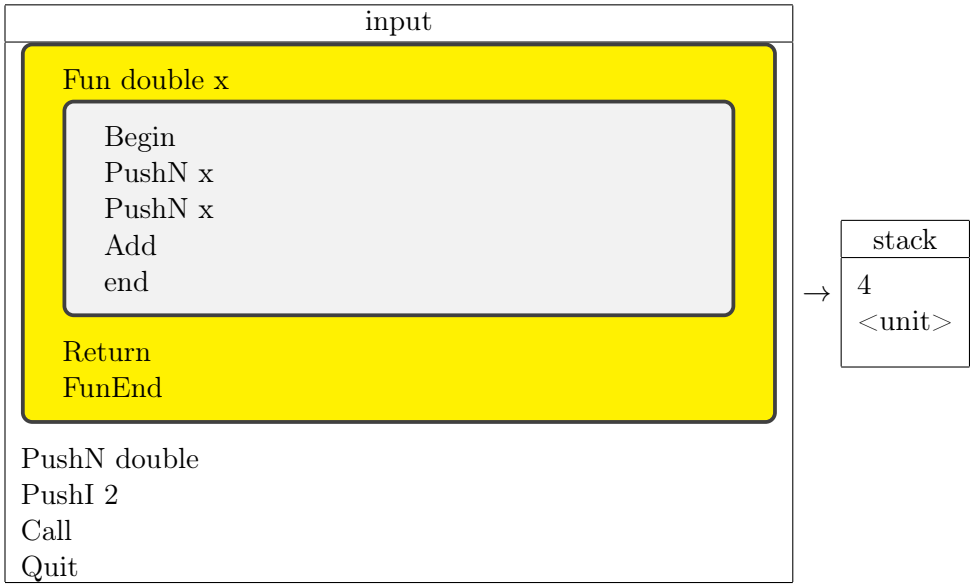
<error> → error since identity is not bound in the environment
1 → Push of 1
identity → Push of identity
<unit> → result of declaring identity, this is the result of the Begin expression

6.4.2 Example 2



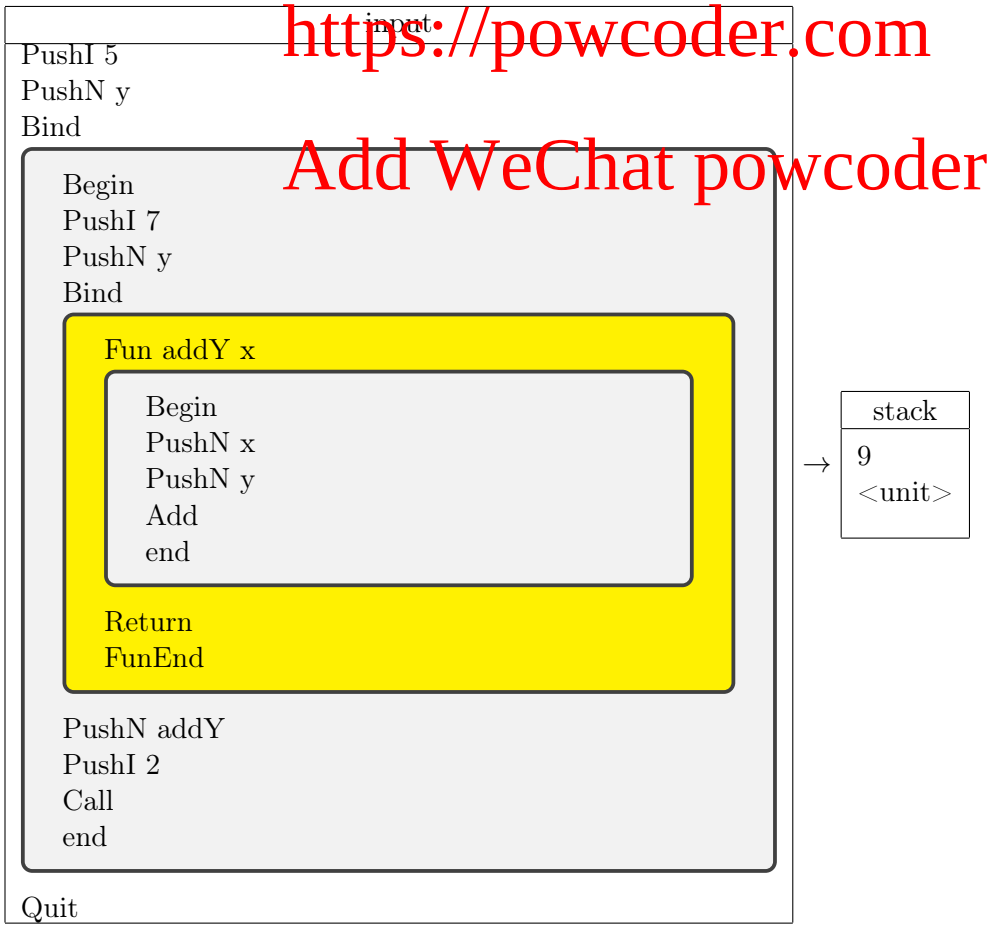
1 → return value of calling identity and passing in x as an argument
<unit> → result of declaring identity

6.4.3 Example 3



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

6.4.4 Example 4

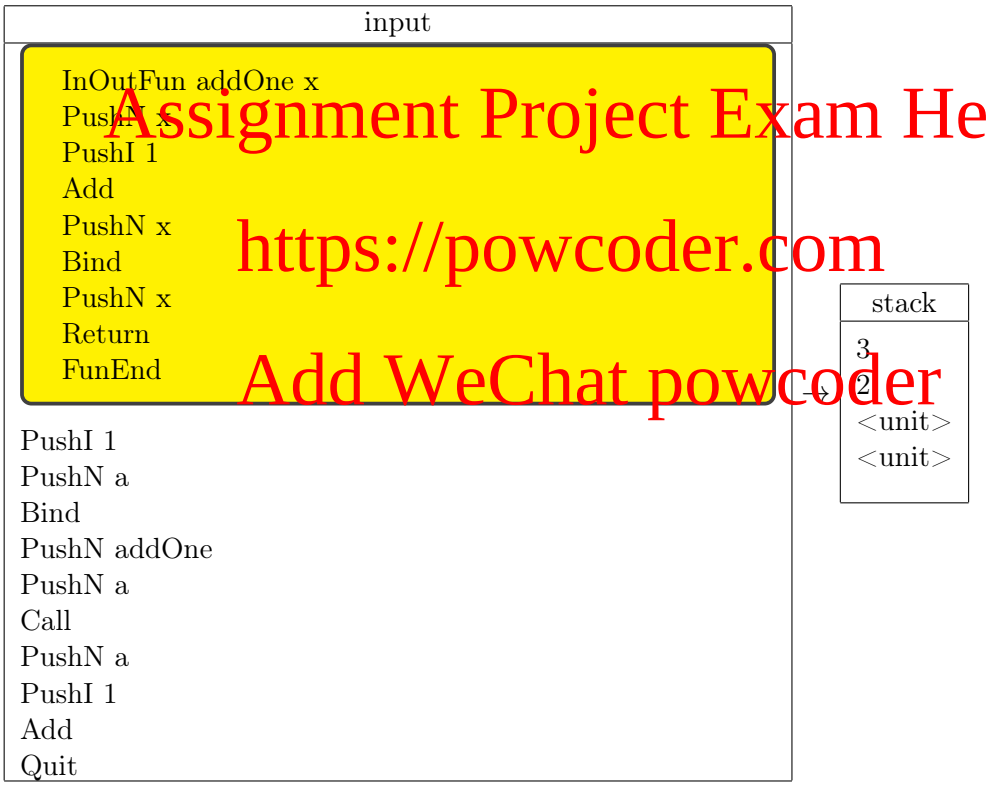


9 → return value of calling identity and passing in 2 as an argument
<unit> → result of binding y to 5

6.5 In/Out Functions

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

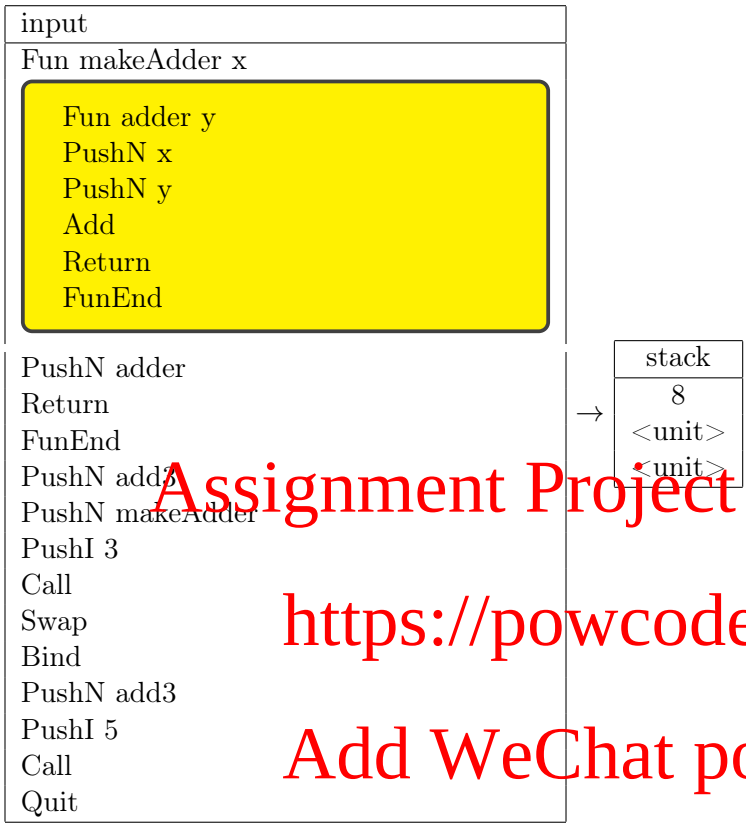


3 → result of Add (note a is bound to two)
2 → return value of calling addOne and passing in x as an argument
<unit> → result of binding a
<unit> → result of declaring addOne

6.6 First-Class Functions

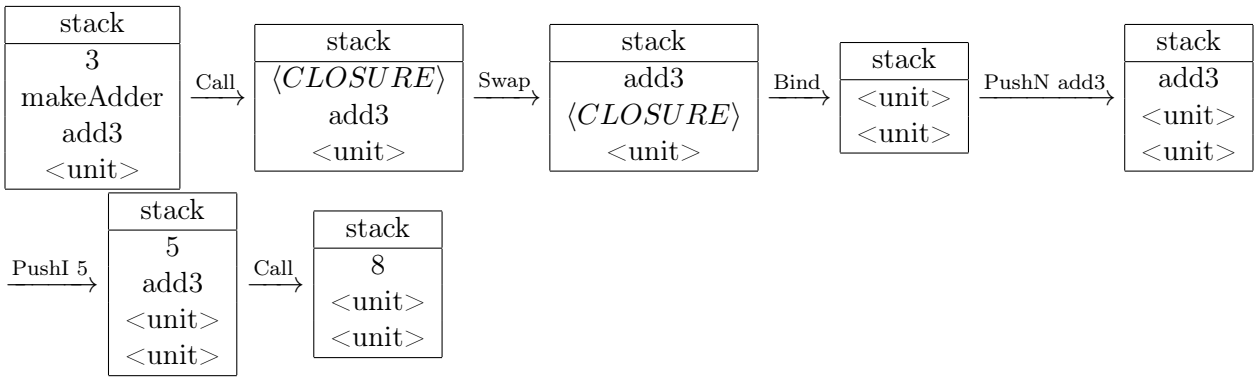
This language treats functions like any other value. They can be used as arguments to functions, and can be returned from functions.

6.6.1 Example 1: Curried adder



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

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



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

it is returned in. For example:

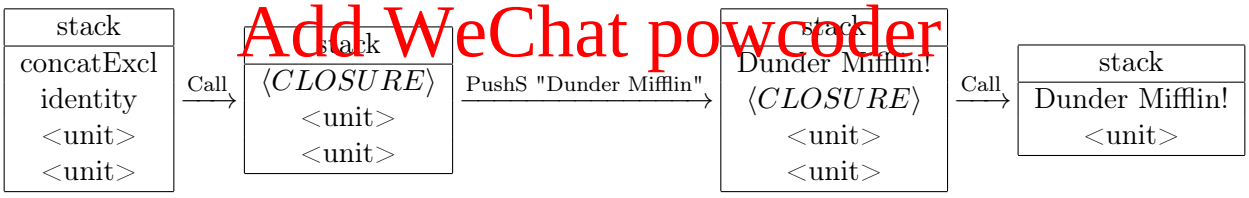
input
Fun identity x
PushN x
Return
FunEnd
Fun _catExcl y
PushS "!"
PushN y
Concat
Return
FunEnd
PushN identity
PushN _catExcl
Call
PushS "Dunder Mifflin"
Call
Quit

→

stack
Dunder Mifflin!
<unit>
<unit>

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

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



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

   - No function closures will be left on the stack.
   - All `Begin` commands will have a matching `End`.
   - There will always be at least one value inside the final stack.

2. You can assume that all test cases will have a Quit statement at the end to exit your interpreter and output the stack, and that "Quit" will never appear mid-program.

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 *inputFile*, states of the stack after each operation are shown below:

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

First, Push 10 onto the stack:

stack
10

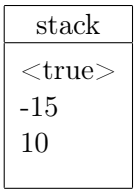
Similarly, Push 15 and 30 onto the stack:

stack
30 15 10

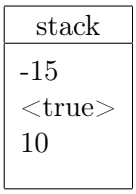
Sub will pop the top two values from the stack, calculate 15 - 30 = -15, and Push -15 back:

stack
-15 10

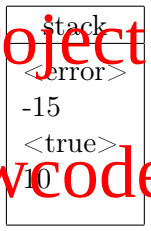
Then Push the boolean literal <true> onto the stack:



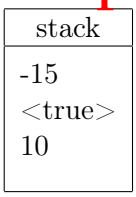
Swap consumes the top two values, interchanges them and Pushes them back:



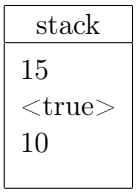
Add will pop the top two values out, which are -15 and <true>, then calculate their sum. Here, <true> is not a numeric value therefore Push both of them back in the same order as well as an error literal <error>



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



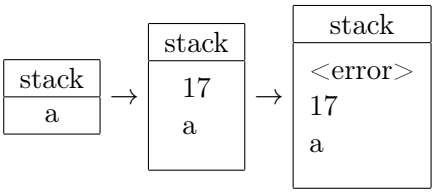
Then after calculating the negation of -15, which is 15, and pushing it back, Quit will terminate the interpreter and write the following values in the stack to *outputFile*:



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

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

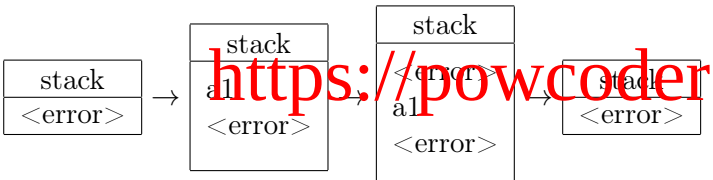
input
PushN a PushI 17 Add



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

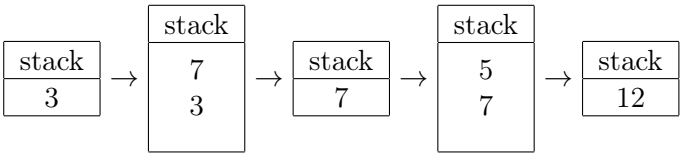
3.

input
Begin PushI 7.2 PushN a1 Bind End



4.

input
Begin PushI 3 PushI 7 End PushI 5 Add Quit



Explanation :

PushI 3
PushI 7

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

7 Frequently Asked Questions

1. Q: What are the contents of test case X?

   A: We purposefully withhold some test cases to encourage you to write your own test cases and reason about your code. You cannot test *every* possible input into the program for correctness. We will provide high-level overviews of the test cases, but beyond that we expect you to figure out the functionalities that are not checked with the tests we provide. But you can (and should) run the examples shown in this document! They're useful on their own, and can act as a springboard to other test cases.

2. Q: Why does my program run locally but fail on Gradescope?

   A: Check the following:

   - Ensure that your program matches the types and function header defined in section 2 on page 1.
   - Make sure that any testing code is either removed or commented out. If your program calls interpreter with input "input.txt", you will likely throw an exception and get no points.
   - *Do not submit testing code.*
   - `stdout` and `stderr` streams are not graded. Your program must write to the output file specified by *outputFile* for you to receive points.
   - *Close your input and output files.*
   - Core and any other external libraries are not available.
   - Gradescope only supports 4.04, so any features added after are unsupported.

3. Q: Why doesn't Gradescope give useful feedback?

   A: Gradescope is strictly a grading tool to tell you how many test cases you passed and your total score. Test and debug your program locally before submitting to Gradescope. The only worthwhile feedback Gradescope gives is whether or not your program compiled properly.

4. Q: Are there any runtime complexity requirements?

   A: Although having a reasonable runtime and space complexity is important, the only official requirement is that your program runs the test suite in less than three minutes.

5. Q: Is my final score the highest score I received of all my submissions?

   A: No. Your final score is only your most recent submission.

6. Q: What can I do if an old submission received a better grade than my most recent submission?

   A: You can always download any of your previous submissions. If the deadline is approaching, we suggest resubmitting your highest-scoring submission before Gradescope locks.