CS 320: Language Interpreter Design

Part 1 Due: November 15, at 11:59pm
Part 2 Due: November 24, at 11:59pm
Part 3 Due: December 6, at 11:59pm

1 Overview

The goal of this project is to understand and build an interpreter for a small 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 100 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 = ...
```

the `input` file name and `output` file name will be passed in as strings that represent paths to files. Your function should read the program to execute from the file specified by `input`, and write 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.

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 ::= *coms*

com ::= `Push` *const* | `Swap` | `Pop` |
`Add` | `Sub` | `Mul` | `Div` | `Rem` | `Neg` |
`And` | `Or` | `Not` |
`Lte` | `Lt` | `Gte` | `Gt` | `Eq` |
`Cat` |
`Bnd` |
`Begin` *coms* `End` |
`If` *coms* `Then` *coms* `Else` *coms* `EndIf` |
`Fun` $name_1$ $name_2$ *coms* `EndFun` |
`Call` | `Return` |
`Try` *coms* `With` *coms* `EndTry` |
`Quit`

coms ::= *com* {*com*}

4 Part 1: Basic Computation
Due Date: November 15, at 11:59pm

Your interpreter should be able to handle the following commands:

4.1 Push

4.1.1 pushing Integers to the Stack

Push *num*

where *num* is an integer, possibly with a '-' suggesting a negative value. Here '-0' should be regarded as '0'. Entering this expression will simply Push *num* onto the stack. For example,

input	stack
Push 5	0
Push -0	5

4.1.2 pushing Strings to the Stack

Push *string*

where *string* is a string literal consisting of a sequence of characters enclosed in double quotation marks, as in "this is a string". Executing this command would Push the string onto the stack:

input	stack
Push "deadpool"	this a string
Push "batman"	batman
Push "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
Push " deadp ool "	this␣is␣a␣string␣␣
Push "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.

4.2 pushing Names to the Stack

Push *name*

where *name* consists of a sequence of characters as specified by the grammar.

1. example

input
Push a
Push 13

→

stack
a

→

stack
13
a

2. example

input		stack		stack
Push __name1__ Push 3	→	__name1__	→	3 __name1__

To bind ‘a’ to the value 13 and __name1__ to the value 3, we will use the ‘Bnd’ 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

Push *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
Push 5 Push <true>	<true> 5

4.4 error and unit

Push <error>
Push <unit>

Pushing an error literal or unit literal will Push `<error>` or `<unit>` onto the stack, respectively.

input		stack
Push <error> Push <unit> Push <error> Push <unit> Push <unit> Quit	→	<unit> <unit> <error> <unit> <error>

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,

input		stack		stack		input
Push 5 Pop Pop	→	5	→		→	<error>

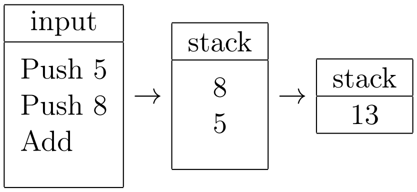
4.6 Add

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:

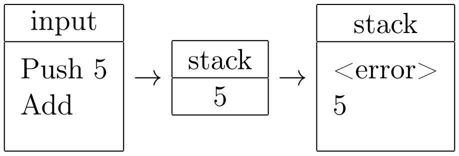
- the two top values in the stack are not integer numbers

- only one value is in the stack
- the stack is empty

for example, the following is a non-error case:



Alternately, if there is only one number in the stack and we use Add, an error will occur, as illustrated in the next example. In this case, 5 should be Pushed back as well as `<error>`

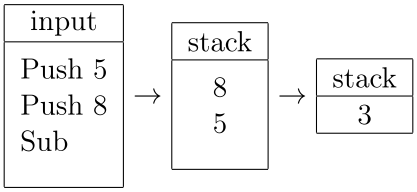


4.7 Sub

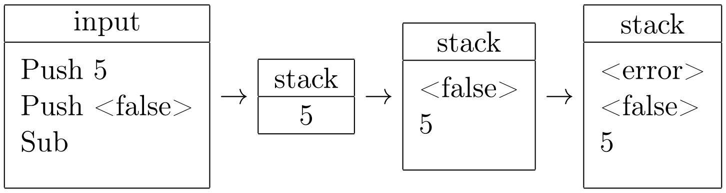
The command Sub refers to integer subtraction. It is a binary operator and works in the following way:

- if the two top 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 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 is a non-error case:



Alternately, if one of the two top values in the stack is not an integer number when Sub is used, an error will occur. For example, when executing the program below the number 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 the two top 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 two top elements in the stack are not integer, 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 is a non-error case:

input		stack		stack
Push 5 Push 8 Mul	→	8 5	→	40

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

input		stack		stack
Mul	→		→	<error>

4.9 Div

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

- if the 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 the 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 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 is a non-error case:

input		stack		stack
Push 5 Push 8 Div	→	8 5	→	1

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

input		stack		stack
Push 0 Push 5 Div	→	5 0	→	<error> 5 0

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 the two top 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 the two top 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 two top elements in the stack are not 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 is a non-error case:

input		stack		stack
Push 5	→	8	→	3
Push 8		5		
Rem				

Alternately, if one of the top two elements in the stack is not an integer, an error will occur if Rem is executed, as illustrated in the next example. If this occurs the top two elements should be Pushed back onto the stack as well as `<error>`. For example:

input		stack		stack
Push 5	→	<false>	→	<error>
Push <false>		5		<false>
Rem				5

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 is a non-error case:

input		stack		stack
Push 5	→	5	→	-5
Neg				

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:

input		stack		stack		stack
Push 5 Neg Push <true> Neg	→	-5	→	<true> -5	→	<error> <true> -5

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 is a non-error case:

input		stack		stack		stack
Push 5 Push 8 Push <false> Swap	→	8 5	→	<false> 8 5	→	8 <false> 5

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, as shown in the following example. Notice that after the first Swap fails, we have two elements in the stack (5 and `<error>`), therefore the second Swap will interchange the two elements:

input		stack		stack		stack
Push 5 Swap Swap	→	5	→	<error> 5	→	5 <error>

4.13 Quit

The command Quit causes the interpreter to stop. Then the whole stack should be printed to the output file that is specified as the second argument to the interpreter function. If no Quit command is encountered during the execution of the program, the whole stack should be printed out to the output file once the program finishes execution.

For Example:

input		stack
Push 1 Push 2 Quit Push 3 Push 4	→	2 1

5 Part 2: Variables and Scope
Due date: November 24, at 11:59pm

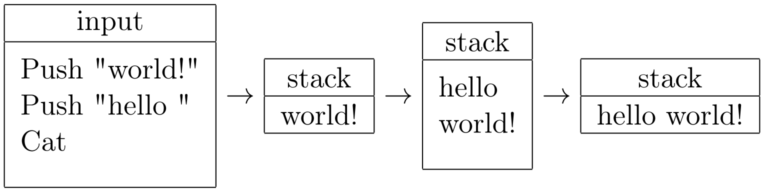
In part 2 of the interpreter you will be expanding the types of computation you will be able to perform, adding support for immutable variables and structures for expressing scope.

5.1 Cat

The Cat command computes the concatenation of the top two elements in the stack and Pushes the result onto the stack. The top two values of the stack — x and y — are popped off and the result is the string 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:



input

Push "world!"
Push "hello "
Cat

→

stack

world!

→

stack

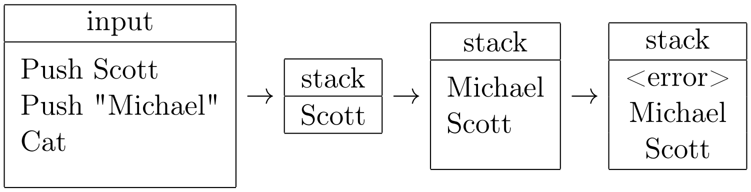
hello
world!

→

stack

hello world!

Consider another example:



input

Push Scott
Push "Michael"
Cat

→

stack

Scott

→

stack

Michael
Scott

→

stack

<error>
Michael
Scott

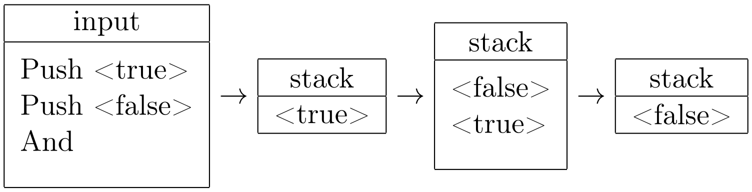
Note that strings can contain spaces, punctuation marks, and other special characters. You may assume that strings only contain ASCII characters and have no escape sequences, e.g. \n and \t.

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:

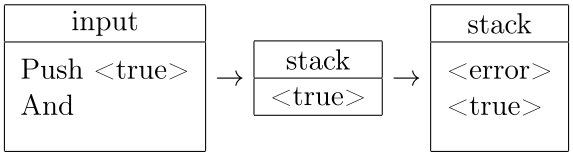


input

Push <true>
Push <false>
And

→ stack <true>
→ stack <false> <true>
→ stack <false>

Consider another example:



input

Push <true>
And

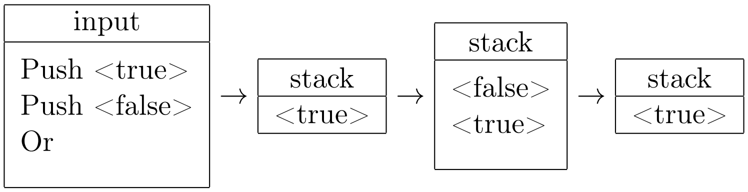
→ stack <true>
→ stack <error> <true>

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:

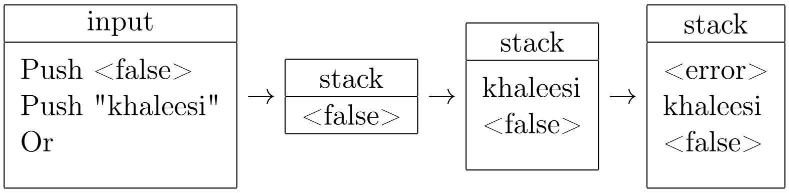


input

Push <true>
Push <false>
Or

→ stack <true>
→ stack <false> <true>
→ stack <true>

Consider another example:



input

Push <false>
Push "khaleesi"
Or

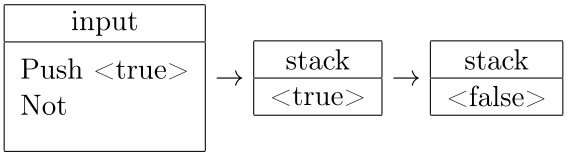
→ stack <false>
→ stack khaleesi <false>
→ stack <error> khaleesi <false>

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:

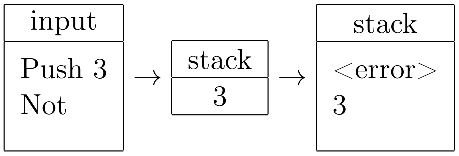


input

Push <true>
Not

→ stack <true>
→ stack <false>

Consider another example:

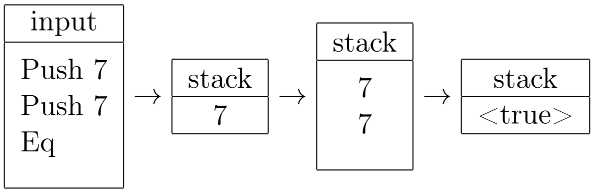


5.5 Eq

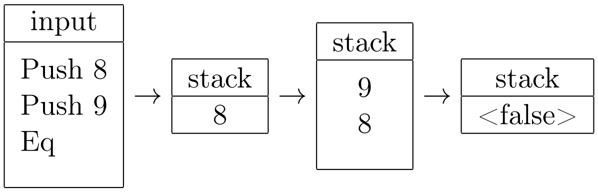
The command Eq refers to numeric equality (so you are not supporting string comparisons). This operator consumes the top two values on the stack and Pushes the result (a single boolean value) onto the stack. The <error> value will be Pushed onto the stack if:

- 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 Lte, Lt, Gte, Gt

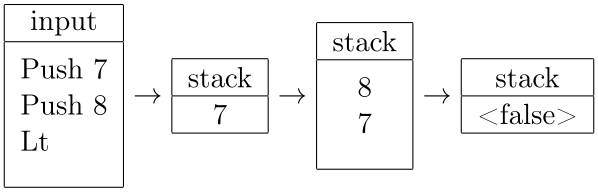
The command Lt refers to numeric < ordering. This operator consumes the top two values on the stack and Pushes the result (a single boolean value) onto the stack.

The <error> value will be pushed onto the stack if:

- there are less then 2 element on the stack
- if either of the top two elements aren't integers, push back the elements and push <error>

The commands Lte, Gte, Gt correspond to $\leq, \geq, >$ ordering respectively. They behave exactly the same as Lt apart from the ordering.

For example:



Another example:

input		stack		stack
Push 7	→	7	→	<error>
Gt				7

5.7 Bnd

The Bnd 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 Bnd 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

input		stack		stack		stack
Push 3	→	3	→	a	→	<unit>
Push a				3		
Bnd						

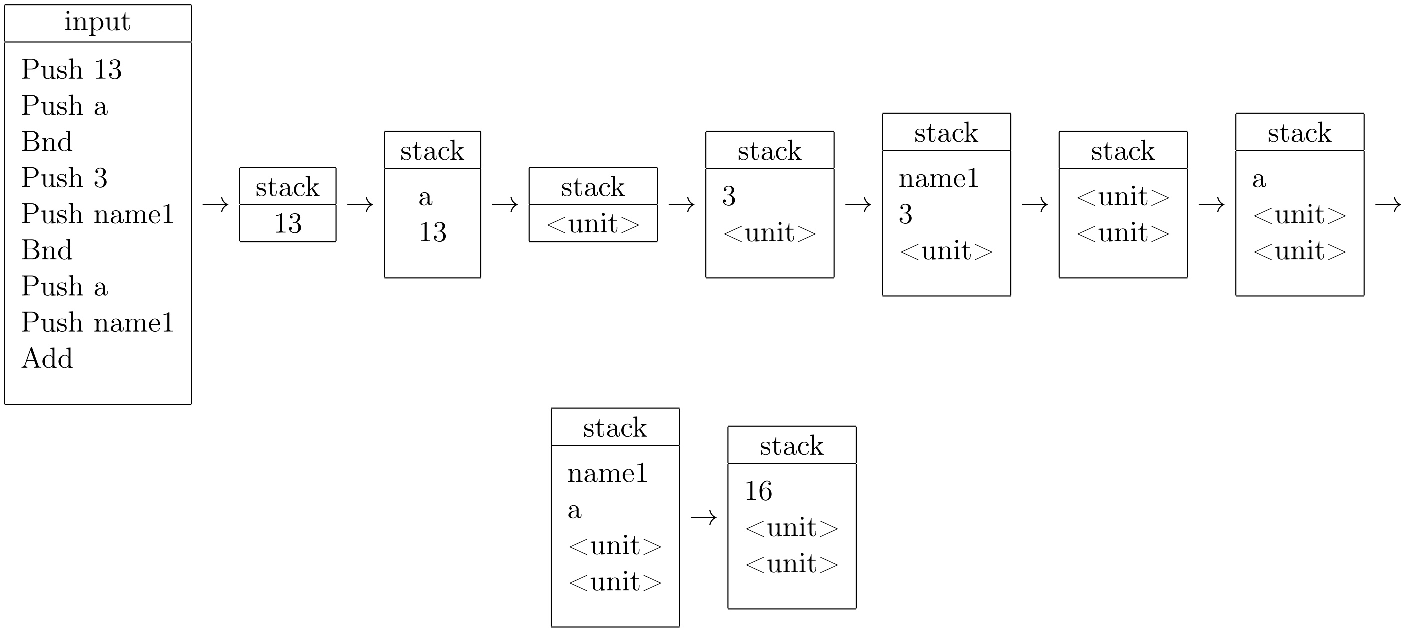
5.7.2 Example 2

input		stack		stack		stack		stack		stack		stack
Push 7	→	7	→	sum1	→	<unit>	→	5	→	sum2	→	<unit>
Push sum1				7				<unit>		5		<unit>
Bnd										<unit>		
Push 5												
Push sum2												
Bnd												

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



```
input
Push 13
Push a
Bnd
Push 3
Push name1
Bnd
Push a
Push name1
Add
```

```
stack
13
```

```
stack
a
13
```

```
stack
<unit>
```

```
stack
3
<unit>
```

```
stack
name1
3
<unit>
```

```
stack
<unit>
<unit>
```

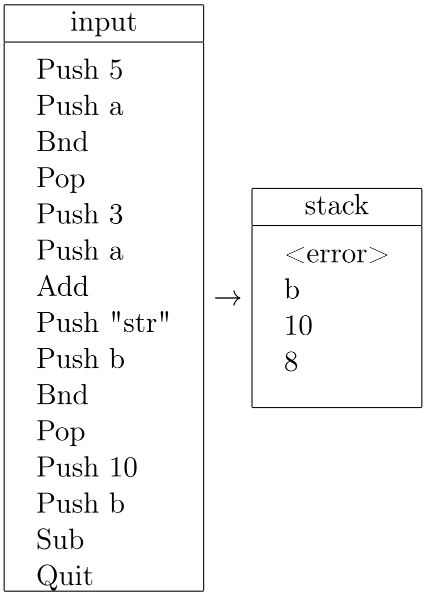
```
stack
a
<unit>
<unit>
```

```
stack
name1
a
<unit>
<unit>
```

```
stack
16
<unit>
<unit>
```

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
Push 5
Push a
Bnd
Pop
Push 3
Push a
Add
Push "str"
Push b
Bnd
Pop
Push 10
Push 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
Push 9 Push a Bnd Push 10 Push a Bnd

Here, the second Bnd 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
Push <true> Push a Bnd

would bind *a* to `<true>`

2)

input
Push 7 Push a Bnd

would bind *a* to 7

3)

input
Begin Push 7 Push a Bnd End Push b Bnd

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

4)

input
Push 8 Push b Bnd Push b Push a Bnd

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

5)

input
Push b
Push a
Bnd

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

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

input
Push 7
Push a
Bnd
Push a
Push b
Bnd

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

(c) Can we have something like this?

input
Push 15
Push a
Push 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
Push 15
Push a
Bnd
Push a

The stack would be:

stack
a
<unit>

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

input
Push a
Push a
Quit

This would result in the following stack output:

stack
a
a

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

input
Push 2
Push a
Push a
Bnd

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

(e) Output of the following code:

input
Push 9
Push a
Bnd
Push 10
Push a
Bnd

This would result in the following stack output:

would result in
`<unit>` → as a result of second Bnd
`<unit>` → as a result of first Bnd

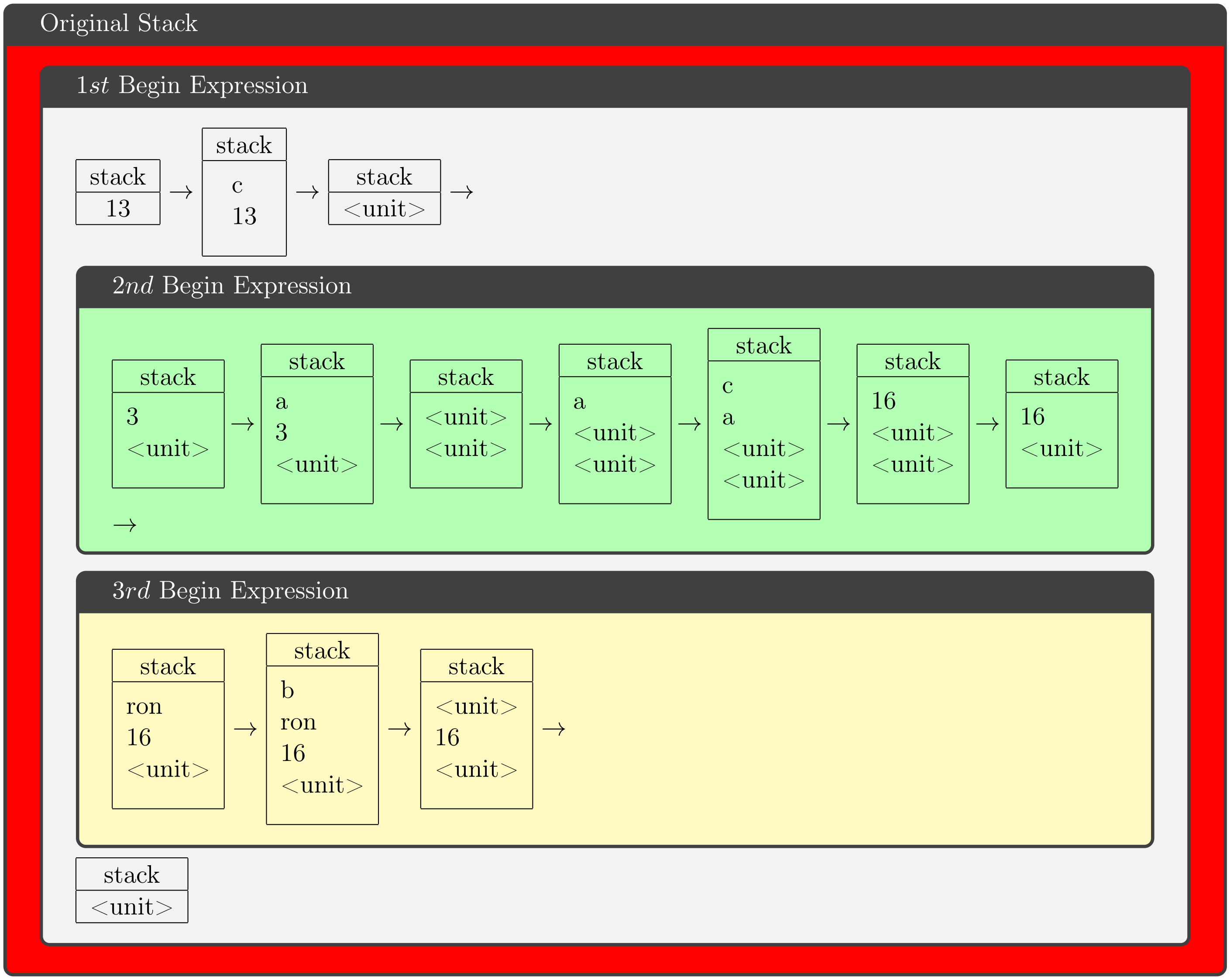
5.10 Begin...End

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
Push 13
Push c
Bnd
Begin
Push 3
Push a
Bnd
Push a
Push c
Add
End
Begin
Push "ron"
Push b
Bnd
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 Bnd 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
Push 1
Begin
Push 2
Push 3
Push 4
End
Push 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
Push a
Bnd
End
Quit
```

The name a cannot be bound, so `<error>` is pushed onto the stack. The `BeginEnd` command finished execution with `<error>` as the topmost element on the stack, so the final state of the stack is `<error>`.

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

```
input
Begin
Push 3
Push 10
End
Add
Quit
```

The stack output would be:

```
stack
<error>
10
```

5.11 If Then Else

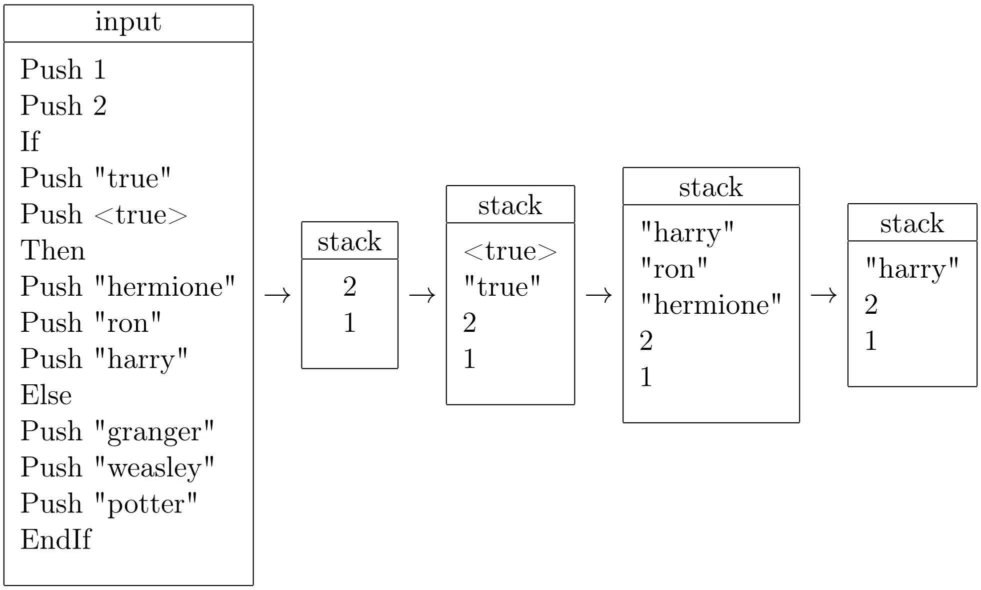
The `IfThenElse` command introduces 3 sets of commands: *test* commands, *true* commands, and *false* commands. (`If` *test* `Then` *true* `Else` *false* `Then`).

First, a *test* environment is formed and the *test* commands are executed in this environment. When these commands finish executing, the top most element on the stack is checked. The *test* environment is exited, and the stack is restored to the state before the *test* commands were executed.

Suppose this element evaluates to `<true>`. A new environment is formed and the *true* commands are executed within this new environment. Once these commands have finished executing, the top most element on the stack is kept whilst the rest of the stack is restored to the state before the `IfThenElse` command was performed.

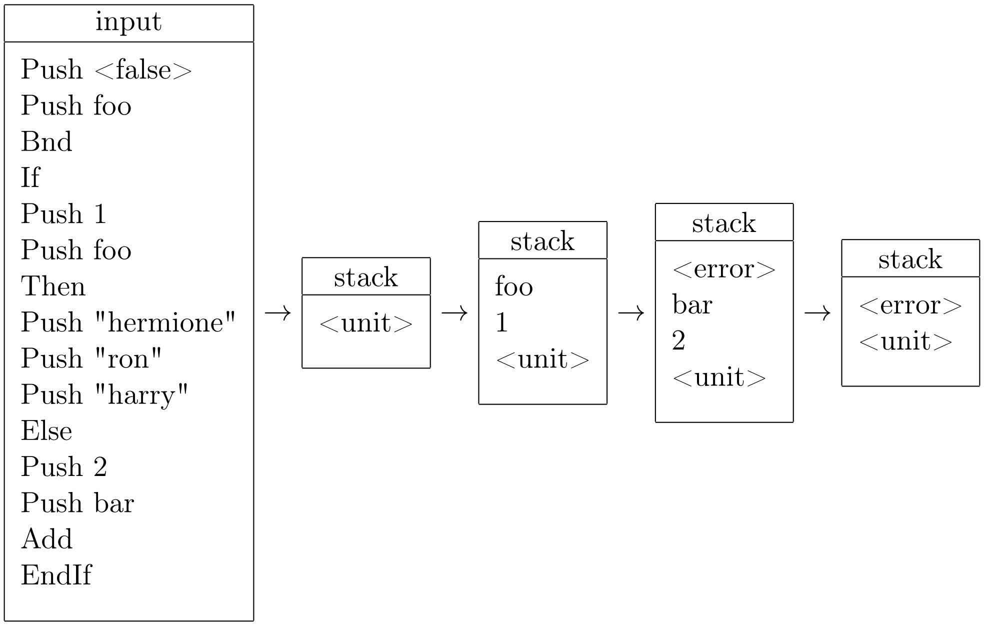
Suppose this element evaluates to `<false>`. A new environment is formed and the *false* commands are executed within this new environment. Once these commands have finished executing, the top most element on the stack is kept whilst the rest of the stack is restored to the state before the `IfThenElse` command was performed.

Suppose this value does not evaluate to a boolean, an `<error>` should be pushed onto the stack.
For example:



input		stack		stack		stack		stack
Push 1 Push 2 If Push "true" Push <true> Then Push "hermione" Push "ron" Push "harry" Else Push "granger" Push "weasley" Push "potter" EndIf	→	2 1	→	<true> "true" 2 1	→	"harry" "ron" "hermione" 2 1	→	"harry" 2 1

In this example, the first and second stack shows the state of the stack before and after executing the *test* commands. The top most element evaluates to `<true>`, so the *test* scope is exited, the stack is restored, and the *true* commands begin executing. The third stack shows the state of the stack after executing the *true* commands. The fourth stack shows that the top most element (`"harry"`), is kept whilst the rest of the stack is restored to the state before the `IfThenElse` command was executed.
Another example:



```
input
Push <false>
Push foo
Bnd
If
Push 1
Push foo
Then
Push "hermione"
Push "ron"
Push "harry"
Else
Push 2
Push bar
Add
EndIf
```

```
stack
<unit>
```

→

```
stack
foo
1
<unit>
```

→

```
stack
<error>
bar
2
<unit>
```

→

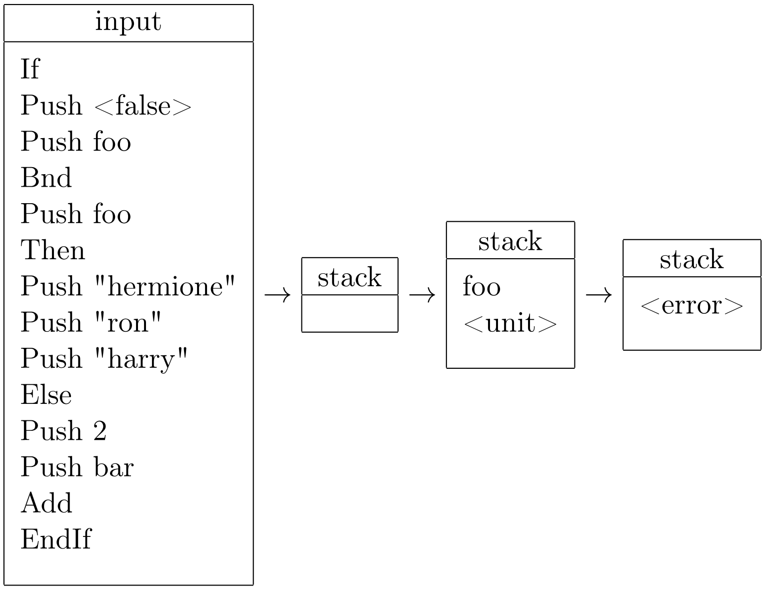
```
stack
<error>
<unit>
```

In this example, the first stack shows the state of the stack before entering the `IfThenElse` command. The name `foo` is bound to `<false>`, pushing `<unit>` onto the stack.

The test commands are executed, 1 and `foo` are pushed onto the stack. Since `foo` evaluates to `<false>`, the false branch executes. 2 and `bar` are pushed onto the stack, but since `bar` is unbound, an `<error>` is pushed onto the stack because `Add` cannot execute properly.

Now that `<error>` is the top most element on the stack, it is kept whilst the rest of the stack is restored to the state before the `IfThenElse` command. This gives us the last stack figure.

Another example:



```
input
If
Push <false>
Push foo
Bnd
Push foo
Then
Push "hermione"
Push "ron"
Push "harry"
Else
Push 2
Push bar
Add
EndIf
```

```
stack

```

→

```
stack
foo
<unit>
```

→

```
stack
<error>
```

In the second stack diagram, the name `foo` although bound to `<false>` within the test environment, is not bound in the outer environment, so it cannot resolve to a boolean term. So `<error>` is pushed onto the stack, indicating that the whole `IfThenElse` command has failed.

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

6.1 Function declarations

A function declaration command (will be refered to as `Fun` commands) is of the form

```
Fun fname arg
    coms
  EndFun
```

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

Functions in our language are closures. This means that when a function is defined, a snapshot of the current environment is taken and stored along with the actual definition of the function. A closure can be thought of as a triple (*arg, coms, env*), where *arg* is the parameter of the function, *coms* are the commands to be executed by the function, and *env* is the state of the environment at the time the closure was formed. Closures have the property that once formed, modification to variable bindings in the global enviroment will not affect variable bindings within the closure's local environment.

When a `Fun` command is executed, a closure is immediately formed using the *arg*, `coms`, and the current *env*. Next, the closure is bound to *fname* in the enviroment, and `<unit>` is pushed onto the stack (similar to `Bnd` for values).

6.2 Call

In order to call a function, its name *fname* should be pushed onto the stack. Next, a value x is pushed onto the stack. The `Call` command is then executed. The environment is queried for *fname*, if *fname* is unbound or bound to a non-closure value, `<error>` will be pushed onto the stack. Suppose *fname* is bound to a closure, the argument, commands and environment stored within the closure are extracted. The value of x (might need resolving if x is a bound name within the global environment) will now be bound to *arg* within environment *env*. A subtle detail to keep in mind is that *fname* should also be bound to the closure within *env* in order to facilitate recusive functions.

Now the entries *fname* and x in the stack are popped, and *coms* will begin executing under the updated *env* and stack state. Once *coms* have finished executing, the top most element of the stack is kept whilst the rest of the stack is restored (to the state after popping *fname* and x). The environment is restored to the state before the `Call` command (*env* is exited).

6.3 Return

Sometimes it is useful to return from a function early. The `Return` command immediately stops the execution of a function and returns the top most element of the stack. If the top most element of the stack is a name, it should be resolved in the current environment before being returned, this is different from returning due to execution completion of function commands which do not resolve returned names.

6.4 Examples

6.4.1 Example 1

```
input
  Fun identity x
  Push x
  Return
  EndFun
Push identity
Push 1
Call
Quit
```

→

```
stack
1
<unit>
```

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

6.4.2 Example 2

```
input
  Fun identity x
  Push x
  Return
  EndFun
Push identity
Call
Quit
```

→

```
stack
<error>
identity
<unit>
```

<error> → error as a result of calling a function without an argument
identity → Push of identity
<unit> → result of declaring identity

6.4.3 Example 3

```
input
  Fun identity x
  Push x
  Return
  EndFun
Push 1
Push x
Bnd
Push identity
Push x
Call
Quit
```

→

```
stack
1
<unit>
<unit>
```

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

6.4.4 Example 4

```
input
Push 3
Push x
Bnd
    Fun addX arg
    Push x
    Push arg
    Add
    Return
    EndFun
Push 5
Push x
Bnd
Push 3
Push a
Bnd
Push addX
Push a
Call
Quit
```

→

```
stack
6
<unit>
<unit>
<unit>
<unit>
```

6 → result of function call

<unit> → result of third binding

<unit> → result of second binding

<unit> → result of function declaration

<unit> → result of first binding

6.4.5 Example 5

```
input
    Fun factorial n
    If
    Push n
    Push 0
    Lt
    Then
    Push fact
    Push 1
    Push n
    Sub
    Call
    Push n
    Mul
    Else
    Push 1
    EndIf
    EndFun
Push factorial
Push 10
Call
Quit
```

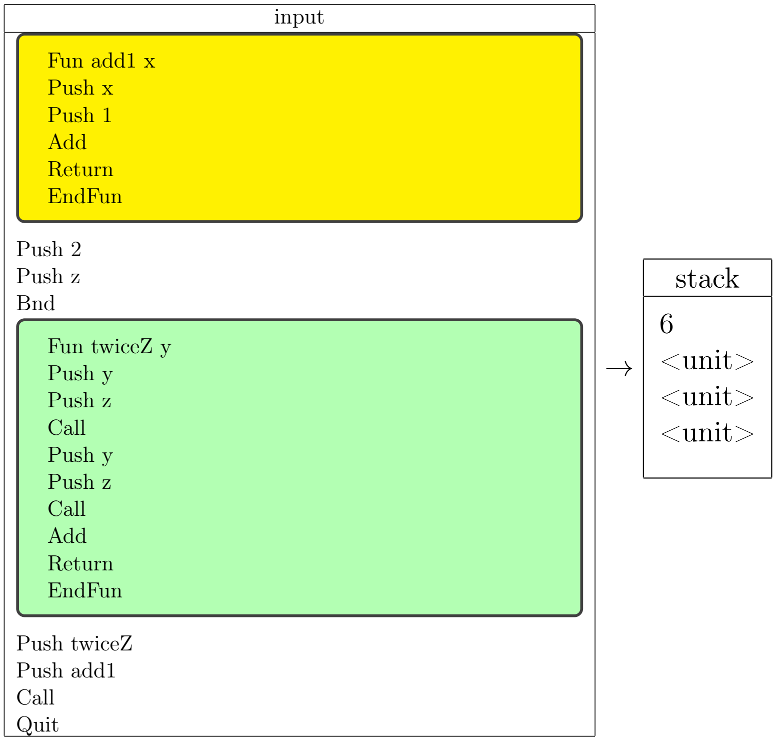
→

```
stack
120
<unit>
<unit>
```

120 → value returned from factorial

<unit> → declaration of factorial

6.4.6 Example 6



input

```
Fun add1 x
Push x
Push 1
Add
Return
EndFun
Push 2
Push z
Bnd
Fun twiceZ y
Push y
Push z
Call
Push y
Push z
Call
Add
Return
EndFun
Push twiceZ
Push add1
Call
Quit
```

→

stack

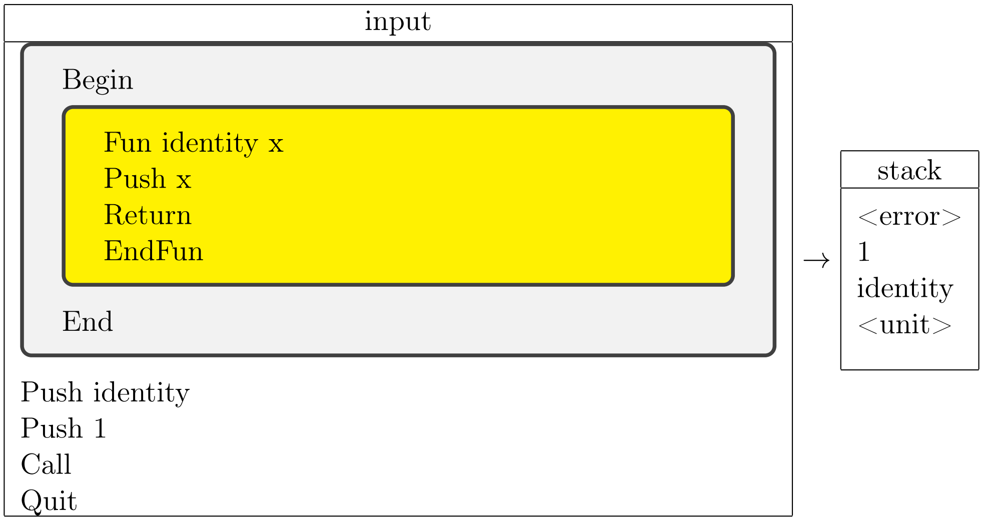
```
6
<unit>
<unit>
<unit>
```

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

6.5 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):

6.5.1 Example 1



input

```
Begin
    Fun identity x
    Push x
    Return
    EndFun
End
Push identity
Push 1
Call
Quit
```

→

stack

```
<error>
1
identity
<unit>
```

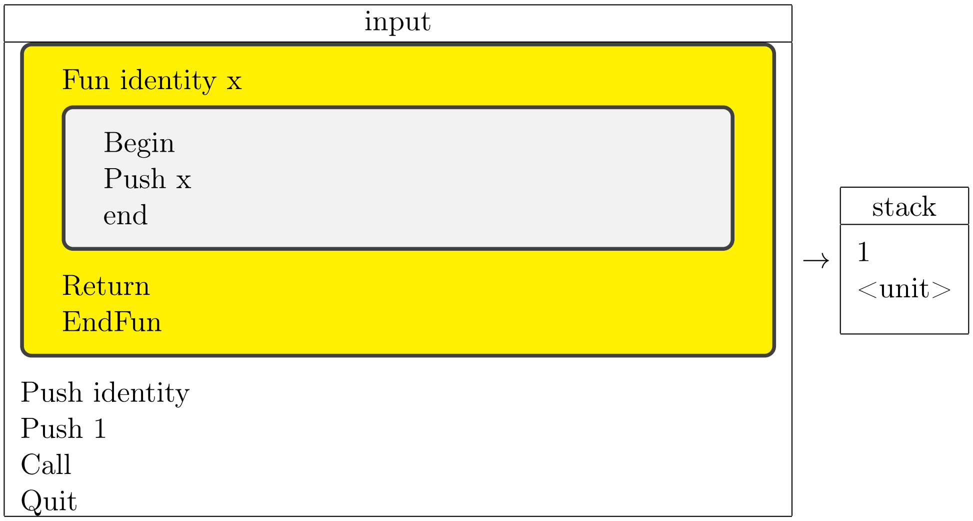
<error> → error since identity is not bound in the environment

1 → Push of 1

identity → Push of identity

<unit> → result of declaring identity, this is the result of the Begin expression

6.5.2 Example 2



input

```
Fun identity x
    Begin
    Push x
    end
Return
EndFun
Push identity
Push 1
Call
Quit
```

→

stack

```
1
<unit>
```

1 → return value of calling identity and passing in x as an argument

<unit> → result of declaring identity

6.5.3 Example 3

```
input
Fun double x
    Begin
    Push x
    Push x
    Add
    End
Return
EndFun
Push double
Push 2
Call
Quit
```

→

stack
4
<unit>

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

6.5.4 Example 4

```
input
Push 5
Push y
Bnd
    Begin
    Push 7
    Push y
    Bnd
        Fun addY x
            Begin
            Push x
            Push y
            Add
            End
        Return
        EndFun
    Push addY
    Push 2
    Call
    End
Quit
```

→

stack
9
<unit>

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

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

input

```
Fun makeAdder x
    Fun adder y
    Push x
    Push y
    Add
    Return
    EndFun
Push adder
Return
EndFun
Push add3
Push makeAdder
Push 3
Call
Swap
Bnd
Push add3
Push 5
Call
Quit
```

→

stack
8
<unit>
<unit>

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

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

stack
3
makeAdder
add3
<unit>

$\xrightarrow{\text{Call}}$

stack
$\langle CLOSURE \rangle$
add3
<unit>

$\xrightarrow{\text{Swap}}$

stack
add3
$\langle CLOSURE \rangle$
<unit>

$\xrightarrow{\text{Bnd}}$

stack
<unit>
<unit>

$\xrightarrow{\text{Push add3}}$

stack
add3
<unit>
<unit>

$\xrightarrow{\text{Push 5}}$

stack
5
add3
<unit>
<unit>

$\xrightarrow{\text{Call}}$

stack
8
<unit>
<unit>

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

input
Fun identity x
Push x
Return
EndFun
Fun _catExcl y
Push "!"
Push y
Cat
Return
EndFun
Push identity
Push _catExcl
Call
Push "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.

stack
concatExcl
identity
<unit>
<unit>

$\xrightarrow{\text{Call}}$

stack
$\langle CLOSURE \rangle$
<unit>
<unit>

$\xrightarrow{\text{Push "Dunder Mifflin"}}$

stack
Dunder Mifflin!
$\langle CLOSURE \rangle$
<unit>
<unit>

$\xrightarrow{\text{Call}}$

stack
Dunder Mifflin!
<unit>

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
Push 10
Push 15
Push 30
Sub
Push <true>
Swap
Add
Pop
Neg
Quit

First, Push 10 onto the stack:

stack
10

Similarly, Push 15 and 30 onto the stack:

stack
30
15
10

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

stack
15
10

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

stack
<true>
15
10

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

stack
15
<true>
10

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

stack
<error> 15 <true> 10

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

stack
15 <true> 10

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

stack
-15 <true> 10

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

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

input
Push a Push 17 Add

stack
a

→

stack
17 a

→

stack
<error> 17 a

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

3.

input
Begin Push 7 Push a Bnd End

stack	→	stack	→	stack	→	stack
7		a 7		<unit>		<unit>

4.

input
Begin Push 3 Push 7 End Push 5 Add Quit

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

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.

6.7 Error Handling with TryWith command

Programming languages often have mechanisims to handle errors in a graceful manner. A common approach is by catching exceptions. Within a special designated block of code, if an error was produced during execution, this block of code stops executing and a handler block starts excution instead. For our language, we have the command `TryWith` for handing runtime errors.

`TryWith` is of the form `Try` $coms_1$ `With` $coms_2$ `EndTry`. Here $coms_1$ denotes a block of commands that may produce an error. $coms_1$ is executed in a new environment, if an error is produced, the execution stops, the stack state is restored to the state before the `TryWith` command executes, and $\texttt{coms}_2$ begins executing.

If $\texttt{coms}_1$ execute successfully, the top most element on the stack is kept whilst the rest of the stack and environment is restored to the state before the `TryWith` command.

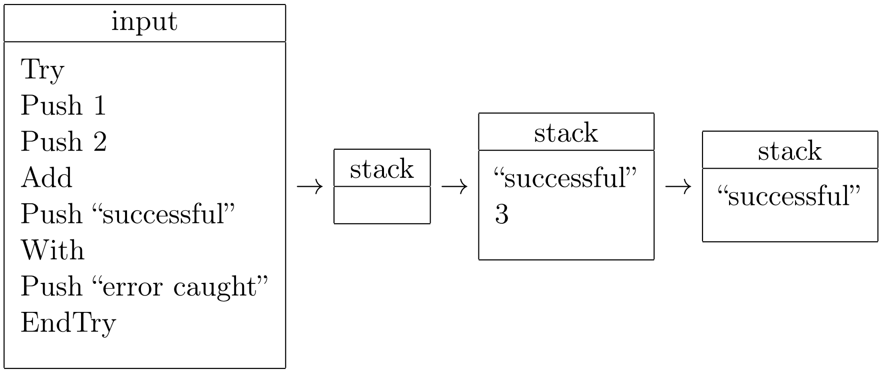
If $\texttt{coms}_2$ execute successfully, the top most element on the stack is kept whilst the rest of the stack and environment is restored to the state before the `TryWith` command. A subtle detail to keep in mind is that `TryWith` commands can be nested, an error executing $\texttt{coms}_2$ could triger the error handling of an outer `TryWith`.

For Example:

input	→	stack	→	stack	→	stack	→	stack
Try Push “1” Push 1 Add Push “successful” With Push “error caught” EndTry				1 “1”		“error caught”		“error caught”

In the second stack figure, the string “1” and the integer 1 are pushed onto the stack. At this point, the `Add` command cannot add a string to an integer. Since the runtime error occurs within a `Try` block, the execution of this block is immediately stopped. The stack is restored to the state before the `TryWith` command is executed, and the `With` block starts executing. Within the `With` block, the string “error caught” is pushed onto the stack, this gives us the third stack figure showing the state of the stack after executing the `With` block successfully. The top most element of the stack is kept while the rest of the stack is restored to the state before the `TryWith` command is executed.

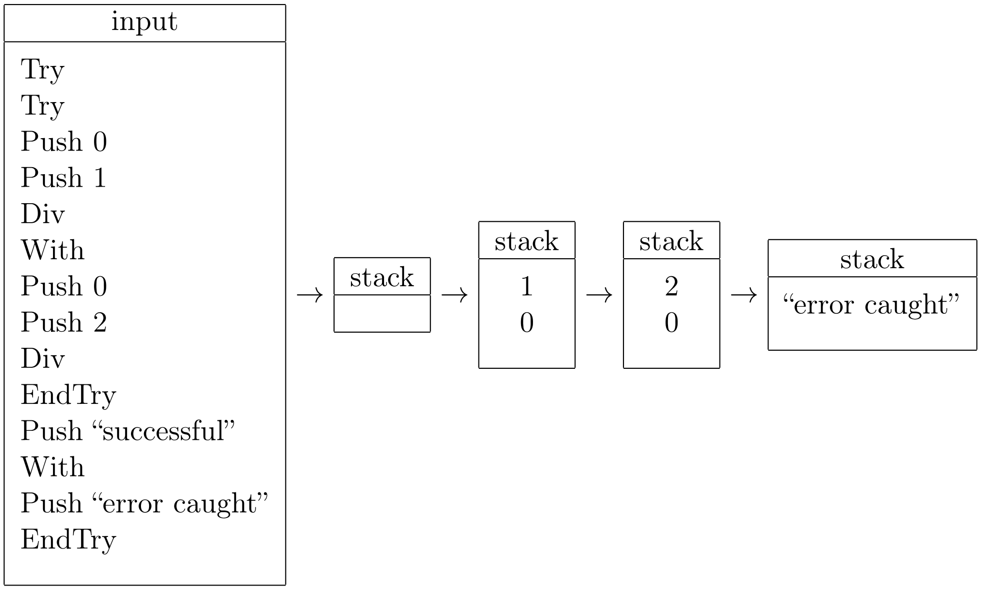
Another Example:



```
Try
Push 1
Push 2
Add
Push “successful”
With
Push “error caught”
EndTry
```

In the `Try` block, 1 and 2 are pushed onto the stack and added together successfully, the string “successful” is then pushed onto the stack, giving us the second stack figure. Since no errors are encountered during the execution of the `Try` block, the topmost element of the resulting stack is kept whilst the rest of the stack is restored to the state before the `TryWith` was executed.

Another Example:



```
Try
Try
Push 0
Push 1
Div
With
Push 0
Push 2
Div
EndTry
Push “successful”
With
Push “error caught”
EndTry
```

The second stack figure corresponds to the stack state of the inner `Try` block. Division by 0 incurs a runtime exception that gets caught by the inner `With` block. The third stack figure correpsonds to the stack state of the inner `With` block, but division by 0 incurs another runtime error, which get caught by the outer `With` block. The string “error caught” is now finally pushed onto the stack. The topmost element of the stack is kept whilst the rest of the stack is restored to the state before the `TryWith` block is executed.

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.