CS 3323: Language Interpreter Project

Due: November 21, 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 and is worth 100 points, Part 2 is defined in Section 5 and is worth 150 points, and Part 3 is defined in Section 6 and is worth 150 points. All parts are due by November 21st at 11:59pm, but we recommend you keep to the following time schedule for each part:

1. Part 1 by October 15th 11:59pm (100 points)
2. Part 2 by November 2nd 11:59pm (150 points)
3. Part 3 by November 21st 11:59pm (150 points) ***this is the* ***last day*** *to submit***

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 pass the tests 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. Each part is graded individually, so you may wish, for example, to submit your Part 3 solution for parts 1 and 2. Late submissions will not be accepted and will be given a score of 0. Test cases will also be provided on Canvas 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` and `output` will be passed in as strings that represent paths to files just like in the Pangram assignment. Your function should write any output 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
push 1 quit	1

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

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

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

input	stack
push 10 push 2 push 8 mult add push 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 ::= `push` *const* | `add` | `sub` | `mult` | `div` | `rem` | `sign` | `swap` | `pop` | `cat` | `and` | `or` | `not` | `lessThan` | `equal` | `if` | `bind` | `let` *com* {*com*} `end` | *funBind* *com* {*com*} [`return`] `funEnd` | `call`

funBind ::= (`fun` | `inOutFun`) $name_1$ $name_2$

4 Part 1: Basic Computation
Suggested Completion Date: October 15, 2025, at 11:59 pm

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

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

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

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 letters, digits, and underscores, starting with a letter or underscore.

1. example

input
push a push 13

→

stack
a

→

stack
13 a

2. example

input
push __name1__ push 3

→

stack
__name1__

→

stack
3 __name1__

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 'bind' operation which we will see later (Section **??**) You can assume that a 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:

Similar with boolean literals, pushing an error literal or unit literal will push `:error:` or `:unit:` onto the stack, respectively.

input
push :error: push :unit: push :error: push :unit: push :unit: quit

→

stack
: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:

- not all top two values are integer numbers
- only one value in the stack
- stack is empty

for example, the following non-error case:

input		stack		stack
push 5 push 8 add	→	8 5	→	13

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:`

input		stack		stack
push 5 add	→	5	→	:error: 5

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 y from x, and push the result x-y back onto the stack
- if the top two elements in the stack are not all integer numbers, push them back in the same order and push `:error:` onto the stack
- if there is only one element in the stack, push it back and push `:error:` onto the stack
- if the stack is empty, push `:error:` onto the stack

for example, the following non-error case:

input		stack		stack
push 5 push 8 sub	→	8 5	→	-3

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:`

input		stack		stack		stack
push 5 push :false: sub	→	5	→	:false: 5	→	:error: :false: 5

4.8 mult

The command mult 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:

input		stack		stack
push 5 push 8 mult	→	8 5	→	40

Alternately, if the stack empty when mult 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 top two elements in the stack are integer numbers, pop the top element(y) and the next element(x), divide x by y, and push the result $\frac{x}{y}$ back onto the stack

- if top two elements in the stack are integer numbers but y equals to 0, push them back in the same order and push `:error:` onto the stack
- if the top two elements in the stack are not both 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:

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

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

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

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 $\frac{x}{y}$, and push the result back onto the stack
- if top two elements in the stack are integer numbers but y equals to 0, push them back in the same order and push `:error:` onto the stack
- if the top two elements in the stack are not all integer numbers, push them back and push `:error:` onto the stack
- if there is only one element in the stack, push it back and push `:error:` onto the stack
- if the stack is empty, push `:error:` onto the stack

For example, the following non-error case:

input		stack		stack
push 5	→	8	→	5
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. 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 push :false: rem	→	:false: 5	→	:error: :false: 5

4.11 sign

The command sign 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:

input		stack		stack
push 5 sign	→	5	→	-5

Alternately, if the value on top of the stack is not an integer, when sign is used, that value should be pushed back onto the stack as well as `:error:`. For example:

input		stack		stack		stack		stack
push 5 sign push :true: sign	→	5	→	-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 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. Now we have two elements in the stack (5 and `:error:`), therefore the second swap will interchange the two elements:

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

4.13 toString

The command toString removes the top element of the stack and converts it to a string. If the stack is empty, the error value is pushed instead. Conversions of values to strings should be done as follows:

- **Integers** shall be rendered as decimal values with no leading zeros. Negative integers should be prefixed with a '-'.
- **Booleans** shall be rendered as the values ":true" and ":false:".
- **Error** values shall be rendered as ":error:"
- **Unit** values shall be rendered as ":unit:"
- **String** values should be left unchanged (i.e. surrounding punctuation may not be added)
- **Names** shall be converted to the corresponding string, contents unchanged.
- **Closures** (once introduced in Part 3) shall be rendered as ":fun:". For now you do not need to handle implementing this, though plan to refactor later.

4.14 println

The println command pops a value of type Integer, Boolean, Error, Unit, String, Name, or Closure (part 3) off the top of the stack and writes it, followed by a newline, to the output file that is specified as the second argument to the interpreter function. In the case that the top element is not a String, you will need to invoke toString. As such, toString is implicitly invoked by println. If the stack is empty, an :error: shall be pushed.

4.15 quit

The command quit causes the interpreter to stop. All commands following this one must be ignored.

5 Part 2: Variables and Scope
Suggested Completion Date: November 2nd, at 11:59 pm

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 y concatenated onto x.
`:error:` will be pushed onto the stack if:

- there is only one element in the stack, push the element back and push `:error:`
- the stack is empty, push `:error:` onto the stack
- if either of the top two elements are not strings, push the elements back onto the stack, and then push `:error:`
  - Hint: Recall that names and strings are different.

For example:

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

Consider another example:

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

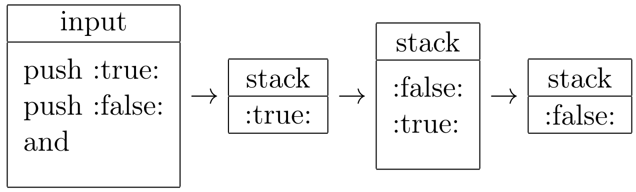
Note that strings can contain spaces, punctuation marks, and other special characters. You may assume that strings only contain ASCII characters and have no escape sequences, e.g. \n and \t.

5.2 and

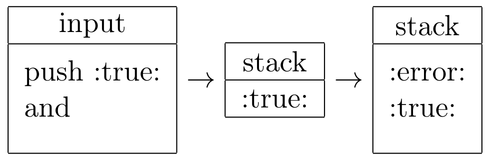
The command and performs the logical conjunction of the top two elements in the stack and pushes the result (a single value) onto the stack.
`:error:` will be pushed onto the stack if:

- 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:

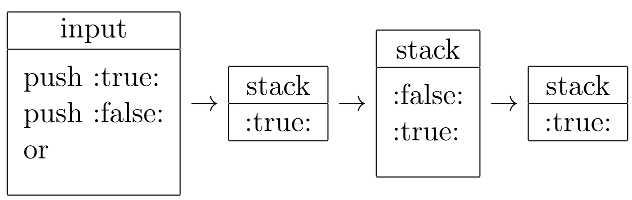


5.3 or

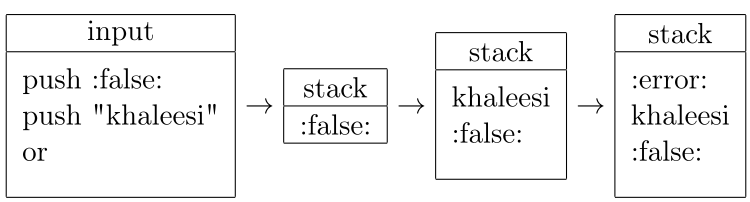
The command or performs the logical disjunction of the top two elements in the stack and pushes the result (a single value) onto the stack.
`:error:` will be pushed onto the stack if:

- 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:

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

Consider another example:

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

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:

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

Consider another example:

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

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:

input		stack		stack		stack
push 7 push 8 lessThan	→	7	→	8 7	→	:true:

5.7 assign

The assign command defines a name value association. It is evaluated by popping two values from the stack. The second value popped must be a name (see section 4.2 for details on what constitutes a 'name'). The name is assigned to the value (the first 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 an assign 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 a push 3 assign	→	a	→	3 a	→	:unit:

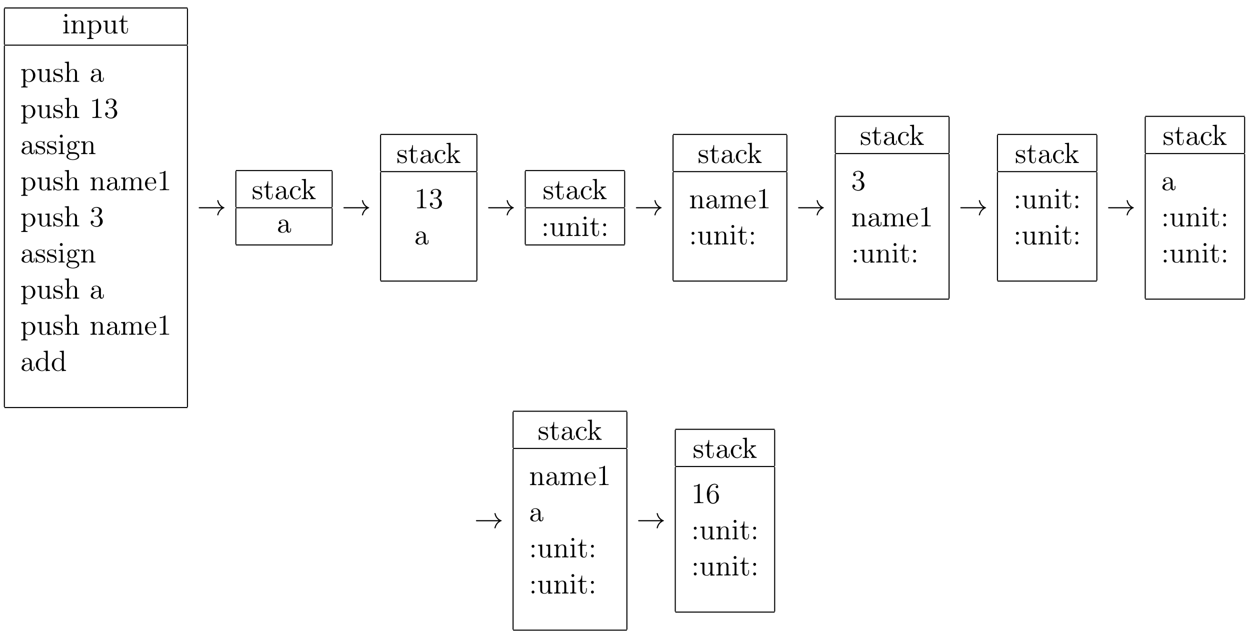
5.7.2 Example 2

input		stack		stack		stack		stack		stack		stack
push sum1 push 7 assign push sum2 push 5 assign	→	sum1	→	7 sum1	→	:unit:	→	sum2 :unit:	→	5 sum2 :unit:	→	:unit: :unit:

You can use these name value 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		stack
push a push 5 assign pop push a push 3 add push b push "str" assign pop push 10 push b sub quit	→	: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 a push 9 assign push a push 10 assign

Here, the second assign updates the value bound to 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 a push :true: assign

would assign a to `:true:`

2)

input
push a push 7.5 assign

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

3)

input
push b let push a push 7 assign end assign

would assign a to 7 and b to `:unit:`

4)

input
push b push 8 assign push a push b assign

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

5)

input
push b push a assign

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

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

input
push a push 7 assign push b push a assign

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

(c) Can we have something like this?

input
push a push 15 push a

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

stack
a 15 a

If we had:

input
push a push 15 assign 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 assigning it this way:

input
push a push a push 2 assign

This would result in
`:unit:` $\rightarrow$ as a result of assigning a to 2
a $\rightarrow$ as a result of pushing the first a to the stack

(e) Output of the following code:

input
push a push 9 assign push a push 10 assign

This would result in the following stack output:

would result in
`:unit:` $\rightarrow$ as a result of second assign
`:unit:` $\rightarrow$ as a result of first assign

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 x back onto the stack, and if z is `:false:`, executing the if will push y 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:

input
push :true: push 9 push 8 if

→

stack
:true:

→

stack
9 :true:

→

stack
8 9 :true:

→

stack
8

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
push :true: push "oracle" push "jive" if

the result of if would be jive

2)

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

the result of if would be `:unit:`

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

input
push a push 5 assign pop push :true: push 4 push a if

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

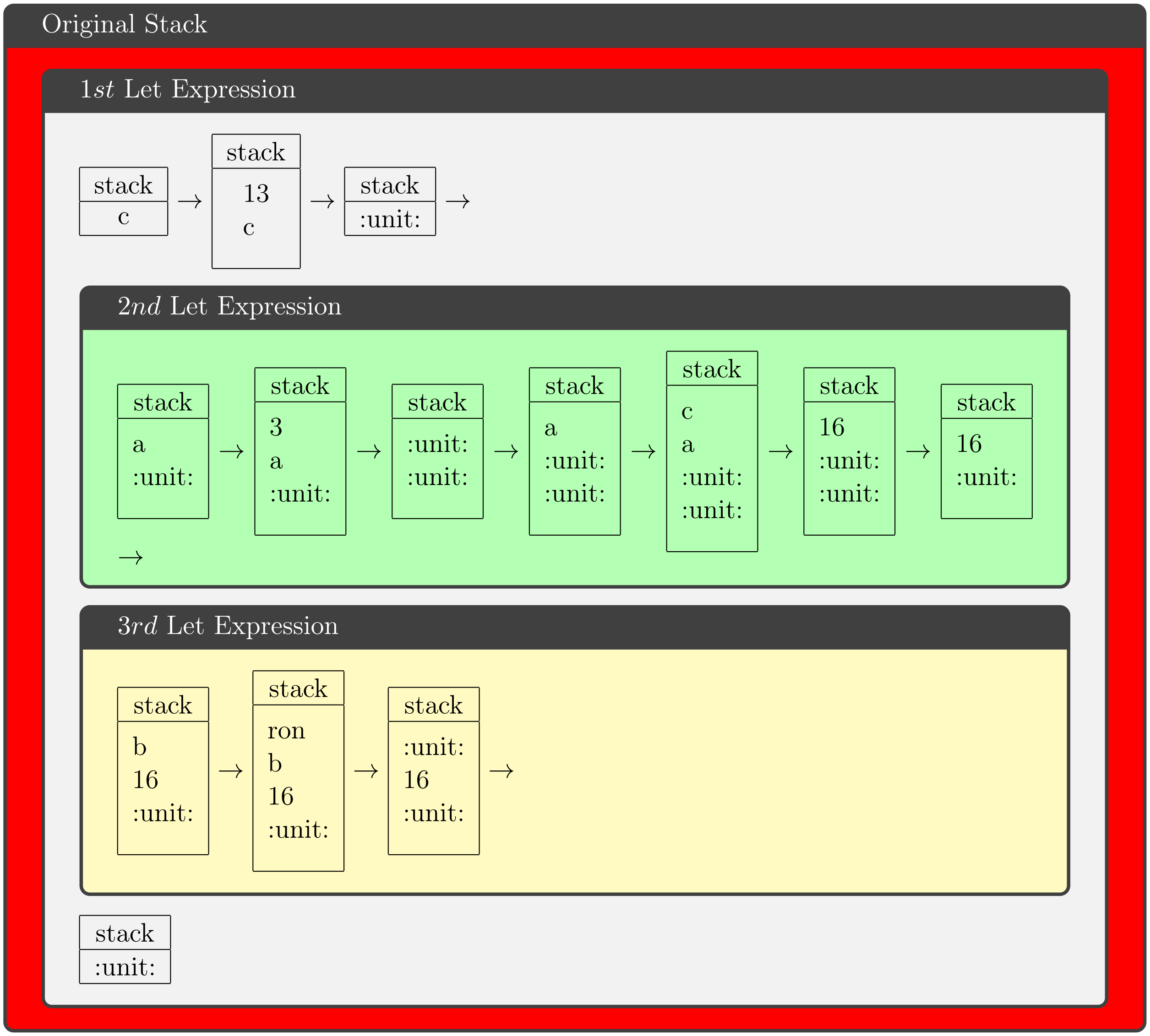
5.11 let...end

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

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

For example,

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



In the above example, the first let statement creates an empty environment (environment 1), then the name c is bound to 13. The result of this assign is a `:unit:` on the stack and a name value pair in the environment. The second let statement creates a second empty environment. Name a is bound here. To add a and c, these names are first looked up for their values in the current environment. If the value isn't found in the current environment, it is searched in the outer environment. Here, c is found from environment 1. The sum is pushed to the stack. A third environment is created with one assignment 'b'. The second last end is to end the scope of environment 3 and the last end statement is to end the scope of environment 1. You can assume that the stack is left with at least 1 item after the execution of any let...end block.

Common Questions

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

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

This would result in the stack:

stack
5 4 1

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

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

input
let push a1 push 7.2 assign 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
let push 3 push 10 end add quit

The stack output would be:

stack
:error: 10

6 Part 3: Functions Due: November 21st , at 11:59 pm

6.1 Adding an additional type

At this point in the project, we have only handled integers as type int. We now want to add an additional constant to our type definition for a floating point number. Handling this means needing to update most of the operations we have implemented thus far, including those in part 1 and part 2. Consider the following constant definitions:

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

int ::= [−] *digit* { *integer digit* }

float ::= [−] *floating point digit* { *floating digit* }

bool ::= `:true:` | `:false:`

error ::= `:error:`

unit ::= `:unit:`

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

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

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

Note that we now have a distinction between floating point and integer mathematical values. We will need to handle each of these cases separately. You do not need to redefine integer at all. Pay close attention to the test files for part 3.

6.2 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.

push *funName*
push *arg*
call

Denotes applying the function *funName* to the actual parameter *arg*. When call is evaluated, it will apply the function *funName* to *arg* and pop both *funName* and *arg* from the stack. *arg* can either be a name (this includes function names), an integer, a string, 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 assignments 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 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.

6.3 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.

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

[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

6.4 Examples

6.4.1 Example 1

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

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

6.4.2 Example 2

input		stack
fun identity x push x return funEnd push identity push 1.2 call quit	→	:error: :error: identity :unit:

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

6.4.3 Example 3

input		stack
fun identity x push x return funEnd push x push 1 assign push identity push x call quit	→	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		stack
push x push 3 assign fun addX arg push x push arg add return funEnd push x push 5 assign push a push 3 assign push addX push a call quit	→	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 stop arg
push 1
return
funEnd

fun factorial arg
push arg
push 1
sub
push 1
push arg
equal
push factorial
push stop
if
swap
call
push arg
mult
return
funEnd

push factorial
push 3
call
quit
```

→

stack
6
:unit:
:unit:

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

6.4.6 Example 6

input

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

push z
push 2
assign

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

push twiceZ
push add1
call
quit
```

→

stack

```
6
:unit:
:unit:
:unit:
```

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

6.5 Functions and Let

Functions can be declared inside a let expression. Much like the lifetime of a variable binding, the binding of a function obeys the same rules. Since let introduces a stack of environments, the closure should also take this into account. The easiest way to implement this is for the closure to store the stack of environments present at the declaration of the function. (Note: you can create a more optimal implementation by only storing the bindings of the free variables 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

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

→

stack

```
:error:
1
identity
:unit:
```

:error: → error since identity is not bound in the environment
1 → push of 1
identity → push of identity
:unit: → result of declaring identity, this is the result of the let expression

6.5.2 Example 2

input

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

→

stack

```
1
:unit:
```

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

6.5.3 Example 3

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

→

stack
4 :unit:

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

6.5.4 Example 4

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

→

stack
9 :unit:

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

6.6 In/Out Functions

Our language will also support in/out parameters for specially denoted functions. Instead of using the fun keyword, functions that have in/out parameters are declared using the inOutFun keyword. In/out functions behave just like regular functions and all the rules defined for functions apply. In addition, when an in/out function returns, the value bound to the formal parameter is bound to the actual parameter in the environment after the call.

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

input

```
inOutFun addOne x
push x
push x
push 1
add
assign
push x
return
funEnd
```

```
push a
push 1
assign
push addOne
push a
call
push a
push 1
add
quit
```

$\rightarrow$

stack
3
2
:unit:
:unit:

3 $\rightarrow$ result of add (note a is bound to two)
2 $\rightarrow$ return value of calling addOne and passing in x as an argument
:unit: $\rightarrow$ result of binding a
:unit: $\rightarrow$ result of declaring addOne

6.7 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.

6.7.1 Example 1: Curried adder

input

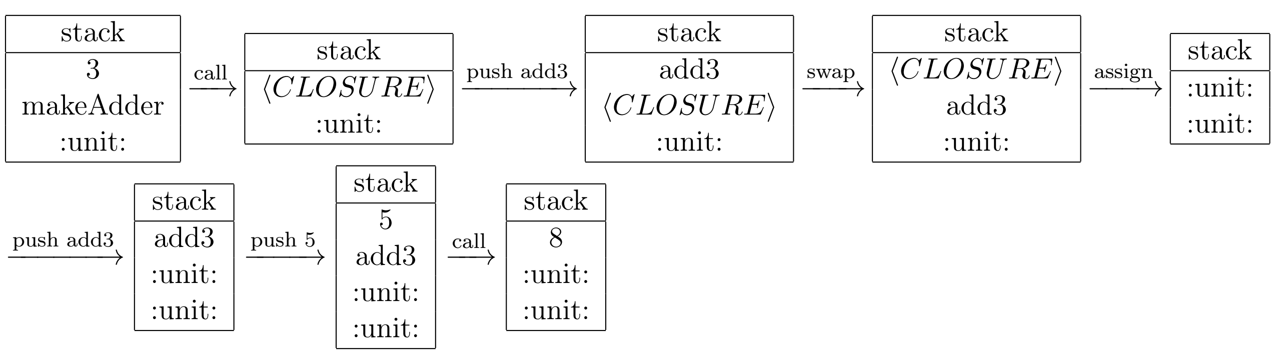
fun makeAdder x
    fun adder y
    push x
    push y
    add
    return
    funEnd
push adder
return
funEnd
push makeAdder
push 3
call
push add3
swap
assign
push add3
push 5
call
quit

→

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 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
push x
return
funEnd
fun _catExcl y
push y
push "!"
cat
return
funEnd
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 multtiple 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 `let` commands will have a matching `end`.
2. You can assume that all test cases will have a quit statement at the end to exit your interpreter, 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 sign quit

First, push 10 onto the stack:

stack
10

Similarly, push 15 and 30 onto the stack:

stack
30 15 10

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

stack
-15 10

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

stack
:true: -15 10

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

stack
-15 :true: 10

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

stack
:error: -15 :true: 10

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

stack
-15 :true: 10

Then after calculating the negation of -15, which is 15, and pushing it back, quit will terminate the interpreter and write the following values in the stack to *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 assign and let...end:

input
push a push 17 add

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

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

3.

input
let push a1 push 7.2 assign end

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

4.

input
let 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.

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: 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.

3. Q: Will I receive intermediate grades for parts 1 and 2?

   A: Yes, any submissions for Part 1 made by the due date will be graded at some point prior to the due date for part 2. Any submissions for Part 2 made by it's due date will be graded at some point prior to the due date for part 3. Note that you may submit after a grade is received. The only grades that will count are the ones posted after November 21st.

4. Q: When will the tests for parts 2 and 3 be posted?

   A: At the release of the project, the tests for parts 2 and 3 may not be available. They will become available closer to the suggested start of their parts (after the current part is due.)