Computer Science 320 (Spring, 2020)
Principles of Programming Languages

Programming Assignment 5: Interpreter (Part 2)

Out: Friday, April 3, 2020
Due: Friday, April 17, 2020, by 11:59 pm

This assignment continues the work you did for Assignment 4, which was Part 1 of an intperpreter for a small, OCaml-like, stack-based bytecode programming language. Once more, your interpreter will be implemented in OCaml. 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 the function `interpreter`; the grader is only explicitly concerned with the type of `interpreter`.

Late submissions will not be accepted and will be given a score of 0. Test cases sample will 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.

1 Preliminaries

Identical to Section 1 in Assignment 4, with two additional comments:

- From Piazza note @209: There is a typographical error in the figure at the top of page 2, right before Section 1.2, where the second input should be:

  ```
  PushI 2
  PushI 6
  Div
  Mul
  Quit
  ```

  i.e., `PushI 2` is now ahead of `PushI 6`. In Assignment 4, `PushI 6` was ahead of `PushI 2`.

- From Piazza note @198: Assume a more restricted grammar, whereby each command is followed by a new line. For example, even though the grammar can generate the following line:

```
PushI 1 PushI 2 PushI 2 Quit
```

We will always format our tests so that the preceding line will read as follows:

```
PushI 1
PushI 2
PushI 2
Quit
```

Moreover, assume that strings will never contain line breaks. For example, even though the grammar allows the following:

```
PushS "a
b
c"
Quit
```

it will never be one of our test cases. Instead, the preceding will be given as:

```
PushS "abc"
Quit
```

2 Basic Computations

Identical to Section 2 in Assignment 4.

3 Variables and Scope

In this assignment, 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.

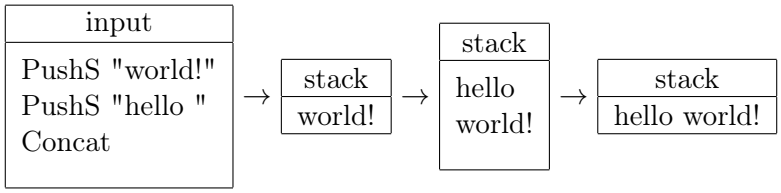
3.1 Concat

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

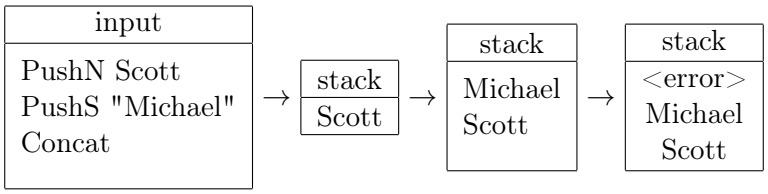
- there is only one element in the stack, Push the element back and Push `<error>`
- the stack is empty, Push `<error>` onto the stack
- if either of the top two elements are not strings, Push the elements back onto the stack, and then Push `<error>`

– Hint: Recall that names and strings are different.

For example:



Consider another example:

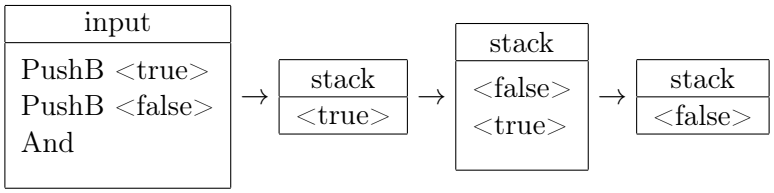


3.2 And

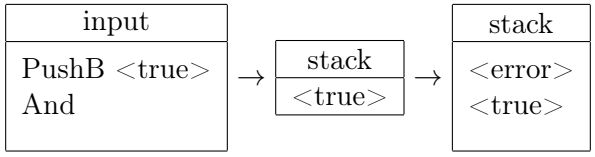
The command And performs the logical conjunction of the top two elements in the stack and Pushes the result (a single value) onto the stack.
`<error>` will be Pushed onto the stack if:

- 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:

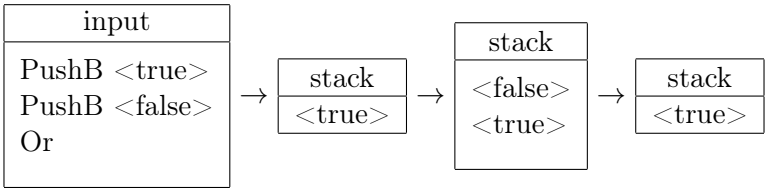


3.3 Or

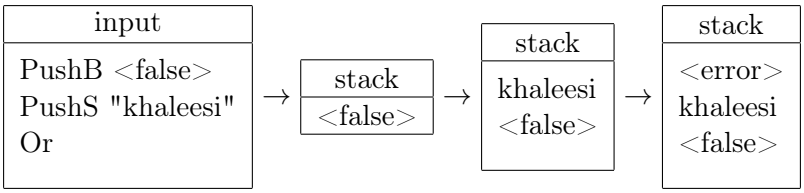
The command Or performs the logical disjunction of the top two elements in the stack and Pushes the result (a single value) onto the stack.
`<error>` will be Pushed onto the stack if:

- 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:

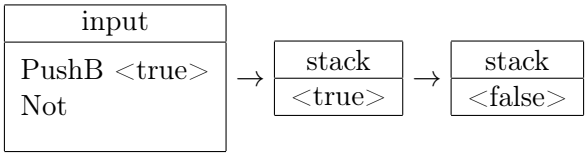


3.4 Not

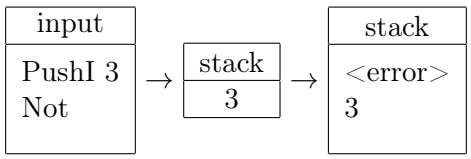
The command Not performs the logical negation of the top element in the stack and Pushes the result (a single value) onto the stack. Since the operator is unary, it only consumes the top value from the stack. The `<error>` value will be Pushed onto the stack if:

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

For example:



Consider another example:

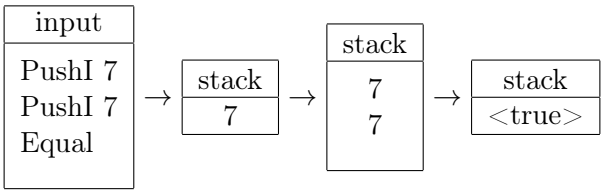


3.5 Equal

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:

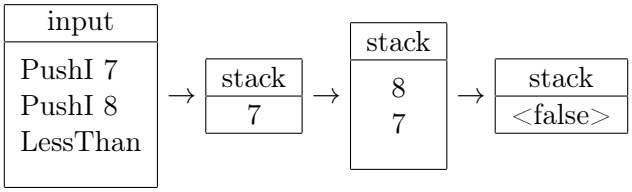


3.6 LessThan

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:



3.7 Bind

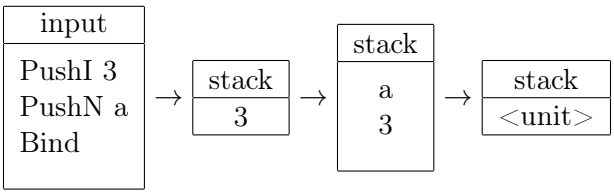
The Bind command binds a name to a value. It is evaluated by popping two values from the stack. The first value popped must be a name. The name is bound to the value, the second thing popped off the stack. The value can be any of the following:

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

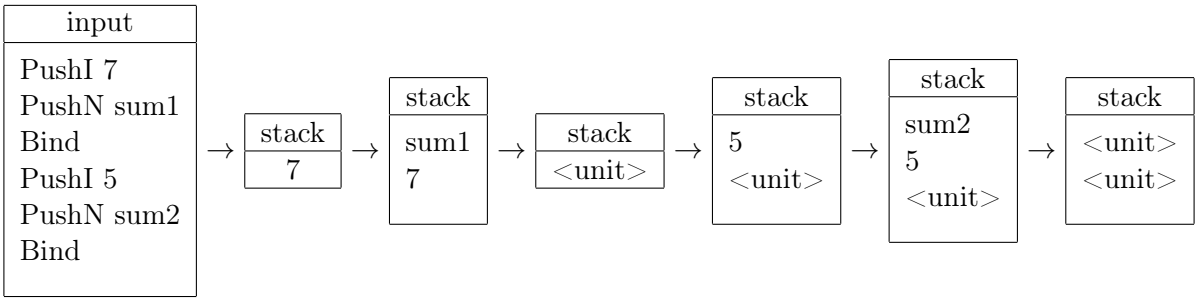
The result of a Bind operation is `<unit>` which is Pushed onto the stack. The value `<error>` will be Pushed onto the stack if:

- we are trying to bind an identifier to an unbound identifier, in which case all elements popped must be Pushed back before pushing `<error>` onto the stack.
- the stack is empty, Push `<error>` onto the stack.

3.7.1 Example 1

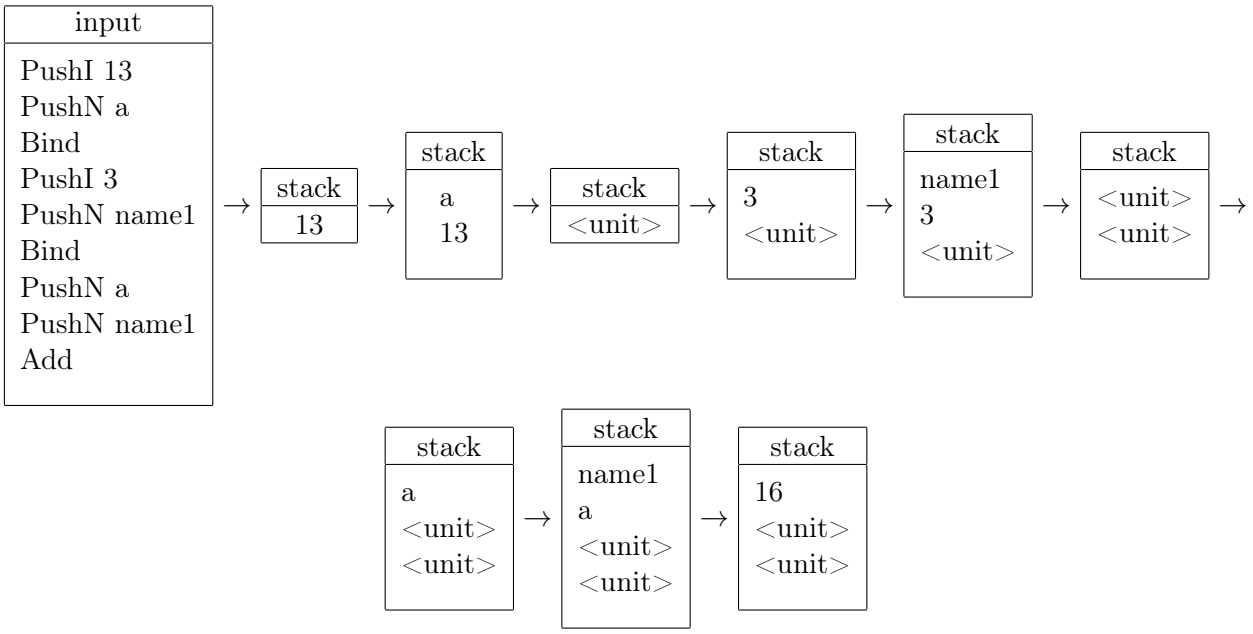


3.7.2 Example 2



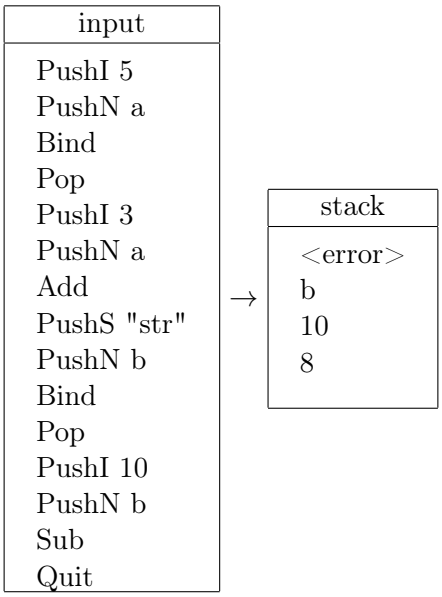
You can use bindings to hold values which could be later retrieved and used by functionalities you already implemented. For instance, in the example below, an addition on *a* and *name1* would add 13 + 3 and Push the result 16 onto the stack.

3.7.3 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.

3.7.4 Example 4



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.

3.7.5 Example 5

Bindings can be overwritten, for instance:

input
PushI 9 PushN a Bind PushI 10 PushN a Bind

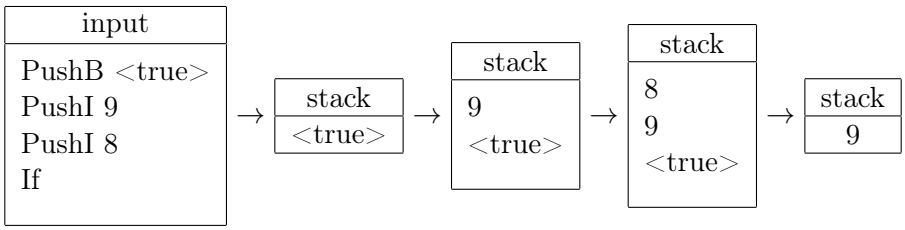
Here, the second Bind updates the value of a to 10.

3.8 If

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

- the third value is not a boolean, all elements (x, y, and z) should be Pushed back onto the stack before pushing `<error>` onto the stack.
- there are fewer than 3 values on the stack, in which case all elements popped must be pushed back before pushing `<error>` onto the stack.

For example:



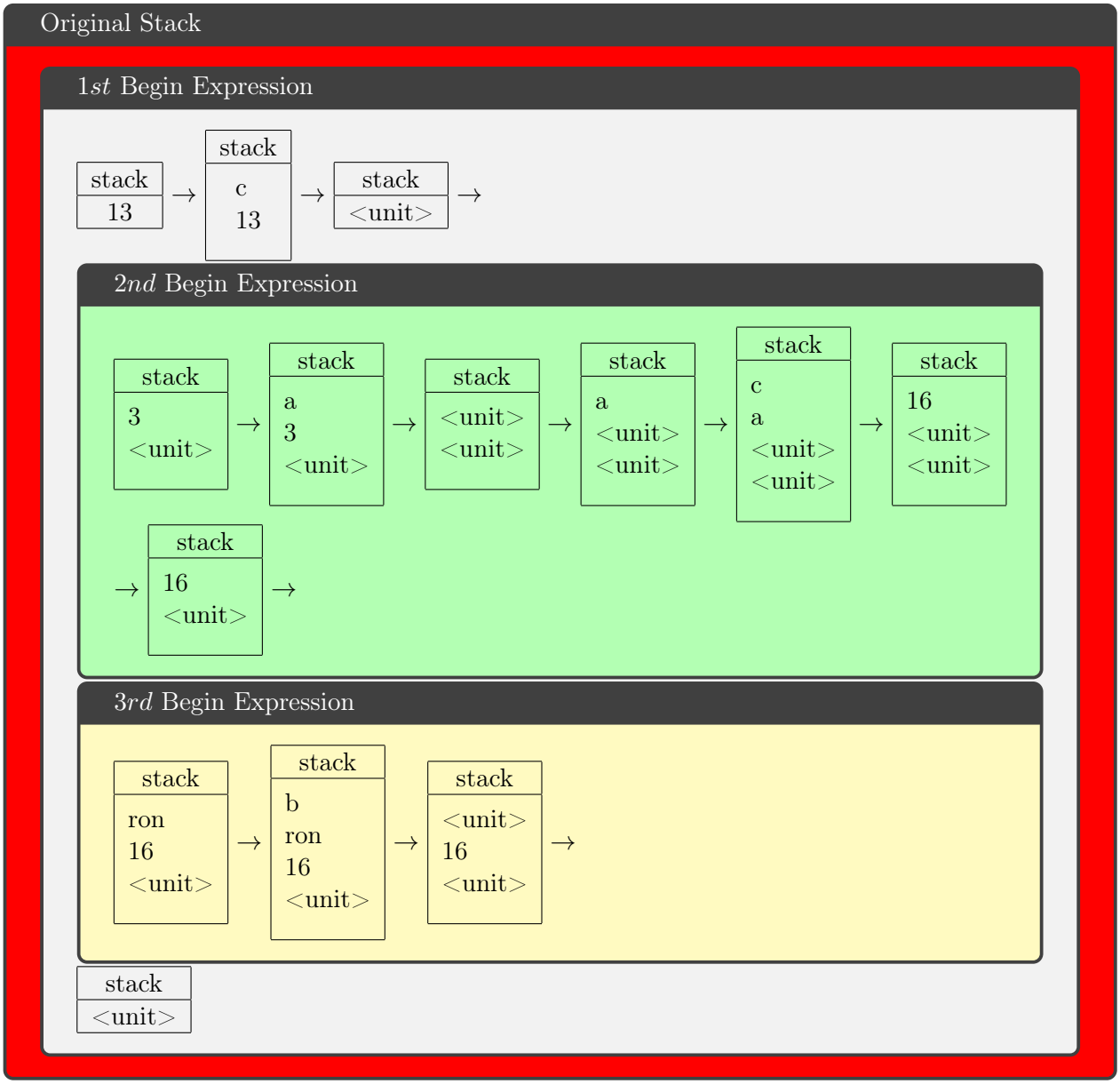
3.9 Begin...End

`Begin...End` blocks limit the scope of names by beginning a new environment for local name bindings in a new empty stack. Evaluating `Begin...End` blocks will result in the top symbol of the stack that

results from evaluating the nested commands. Local name bindings inside a `Begin...End` block are not in scope outside of that block. As implied by the grammar, `Begin...End` blocks can be nested.

For example,

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



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

Common Questions

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

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

This would result in the stack:

stack
5 4 1

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

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

input
Begin PushI "joe" PushN a1 Bind End Quit

"joe" can't be Pushed to the stack ("joe" is not an integer) and a1 cannot be bound to `<error>` so, the result would be `<error>`.

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

input
Begin PushI 3 PushI 10 End Add Quit

The stack output would be:

stack
<error> 10

4 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.