CS 320: Language Interpreter Design

Part 1 Due: April 7th 11:59pm EST
Part 2 Due: April 18th 11:59pm EST
Part 3 Due: April 28th 11:59pm EST

1 Overview

The project is broken down into three parts. Each part is worth 100 points.

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

```
interpreter :  string -> string list
```

2 Functionality

the function will take a program as an `input` string, and will return list of strings "logged" by the program. A stack is used internally to keep track intermediate evaluation results. Only the string log will be tested during grading, the stack will not be tested.

3 Part 1: Basic Computation Due Date: April 7th, 11:59pm EST

3.1 Grammar

For part 1 you will need to support the following grammar

3.1.1 Constants

digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

int ::= [−] *digit* {*digit*}

bool ::= `True` | `False`

const ::= *int* | *bool* | `()`

3.1.2 Programs

prog ::= *coms*

com ::= `Push` *const* | `Pop` *int* | `Trace` *int*
  | `Add` *int* | `Sub` *int* | `Mul` *int* | `Div` *int*

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

3.1.3 Values

val ::= *int* | *bool* | `()`

3.2 Errors

In part 1, when an error occurs during interpretation, evaluation must stop immediately and output the exact log `["Error"]`.

3.3 Commands

Your interpreter should be able to handle the following commands:

3.3.1 Push

Push *const*

All kinds of *const* are pushed to the stack in the same way. Resolve the constant to the appropriate value and add it to the stack.

The program

```
Push 9
Push True
Push False
Push ()
```

should result in the stack

```
()
False
True
9
```

3.3.2 Pop

The command Pop n removes the top n values from the stack. If n is negative or the stack contains less than n values, terminate evaluation with error.

Example 1

```
Push True
Push False
Push ()
Pop 1
```

should result in the stack

```
False
True
```

Example 2

```
Push True
Push False
Push ()
Pop 2
```

should result in the stack

```
True
```

Example 3

```
Push True
Push False
Push ()
Pop 4
```

should terminate with the following list of string.

```
["Error"]
```

3.3.3 Trace

The Trace n command consumes the top n values on the stack and adds their string representations to the output list. New entries to the log should be added to the head position.

If n is negative or the stack contains less than n values, terminate with error. For a Trace n command where $n > 1$, the produced log should be equivalent to n executions of Trace 1 command.

Example 1

```
Push ()
Push 5
Push 1
Push 2
Trace 1
Trace 1
```

should result in the stack

```
5
()
```

and the output log

```
["1"; "2"]
```

Example 2

```
Push ()
Push 5
Push 1
Push 2
Trace 2
```

should result in the stack

```
5
()
```

and the output log

```
["1"; "2"]
```

Example 3

```
Push ()
Push 5
Push 1
Push 2
Trace 2
Trace 2
```

should result in an empty stack and the output log

```
["()"; "5"; "1"; "2"]
```

3.3.4 Add

Add n consumes the top n values in the stack, and pushes their sum to the stack.

If n is negative or there are fewer than n values on the stack, terminate with error. If the top n values on the stack are not integers, terminate with error. If n is zero, push 0 onto the stack without consuming anything on the stack.

Example 1

```
Push 5
Push 7
Add 2
Push 3
Add 2
```

should result in the stack

```
15
```

Example 2

```
Push 5
Add 0
```

should result in the stack

```
0
5
```

Example 3

```
Push 5
Push 4
Add 1
```

should result in the stack

```
4
5
```

3.3.5 Sub

Sub n consumes the top n values on the stack, and pushes the difference between the top value and the sum of next $n - 1$ values to the stack.

If n is negative or there are fewer than n values on the stack, terminate with error. If the top n values on the stack are not integers, terminate with error. If n is zero, push 0 onto the stack without consuming anything on the stack.

Example 1

```
Push ()
Push 1
Push 10
Sub 2
```

should result in the stack

```
9
()
```

Example 2

```
Push ()
Push 1
Push 3
Push 4
Push 10
Sub 3
```

should result in the stack

```
3
1
()
```

Example 3

```
Push ()
Push 1
Push 10
Sub 0
```

should result in the stack

```
0
10
1
()
```

Example 4

```
Push ()
Push 1
Push 10
Sub 1
```

should result in the stack

```
10
1
()
```

3.3.6 Mul

Mul n consumes the top n values in the stack, and pushes their product to the stack.

If n is negative or there are fewer than n values on the stack, terminate with error. If the top n values on the stack are not integers, terminate with error. If n is zero, push 1 onto the stack without consuming anything on the stack.

Example 1

```
Push 5
Push 7
Mul 2
```

should result in the stack

```
35
```

Example 2

```
Push 2
Push 5
Push 7
Mul 3
```

should result in the stack

```
70
```

Example 3

```
Push 2
Push 5
Push 7
Mul 1
```

should result in the stack

```
7
5
2
```

Example 4

```
Push 2
Push 5
Push 7
Mul 0
```

should result in the stack

```
1
7
5
2
```

3.3.7 Div

Div n consumes the top n values on the stack, and pushes the quotient between the top value and the product of the next $n-1$ values to the stack.

If n is negative or there are fewer than n values on the stack, terminate with error. If the product of the next $n-1$ values are 0, terminate with error. If the top n values on the stack are not integers, terminate with error. If n is zero, push 1 onto the stack without consuming anything on the stack.

Example 1

```
Push 2
Push 10
Div 2
```

should result in the stack

```
5
```

Example 2

```
Push 0
Push 5
Push 10
Div 3
```

will terminate with output `["Error"]`.

Example 3

```
Push 2
Push 5
Push 10
Div 3
```

should result in the stack

```
1
```

Example 4

```
Push 2
Push 5
Push 10
Div 0
```

should result in the stack

```
1
10
5
2
```

Example 5

```
Push 2
Push 5
Push 10
Div 1
```

should result in the stack

```
10
5
2
```

4 Part 2: More Computation, Definitions and Scope Due date: April 18th 11:59pm EST

4.1 Grammar

For part 2 the grammar is extended in the following way

4.1.1 Constants

letter ::= `a`...`z` | `A`...`Z`

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

const ::= ... | *name*

4.1.2 Programs

com ::= ... | `And` | `Or` | `Not`
| `Equal`
| `Lte`
| `Local`
| `Global`
| `Lookup`
| `Begin` *coms* `End`
| `If` *coms* `Else` *coms* `End`

4.1.3 Values

val ::= ... | *name*

4.1.4 Environment

An environment is used to track bindings from names to values. Names in the enviroment can be resolved to their bound values through the `Lookup` command.

env ::= $\{name \mapsto val\}$

4.2 Commands

4.2.1 And

And consumes the top two values in the stack, and pushes their conjunction to the stack.

If there are fewer then 2 values on the stack, terminate with error. If the two top values in the stack are not booleans, terminate with error.

Example 1

```
Push True
Push False
And
```

should result in the stack

```
False
```

Example 2

```
Push True
Push True
And
```

should result in the stack

```
True
```

4.2.2 Or

Or consumes the top two values in the stack, and pushes their disjunction to the stack.

If there are fewer then 2 values on the stack, terminate with error. If the two top values in the stack are not booleans, terminate with error.

Example 1

```
Push True
Push False
Or
```

should result in the stack

```
True
```

Example 2

```
Push False
Push False
Or
```

should result in the stack

```
False
```

4.2.3 Not

Not consumes the top value of the stack, and pushes it's negation to the stack.

If the stack is empty, terminate with error. If the top value on the stack is not an boolean,terminate with error.

Example 1

```
Push True
Push False
Or
```

should result in the stack

```
True
True
```

Example 2

```
Push True
Push True
Or
```

should result in the stack

```
False
True
```

4.2.4 Equal

Equal consumes the top two values in the stack, and pushes True to the stack if they are equal integers and False if they are not equal integers.

If there are fewer then 2 values on the stack, terminate with error. If the two top values in the stack are not integers, terminate with error.

Example 1

```
Push 1
Push 1
Equal
```

should result in the stack

```
True
```

Example 2

```
Push 1
Push 1
Push 1
Add 2
Equal
```

should result in the stack

```
False
```

Example 3

```
Push 0
Push 1
Push 1
Sub 2
Equal
```

should result in the stack

```
True
```

4.2.5 Lte

Lte consumes the top two integer values on the stack, and pushes True on the stack if the top value is less than or equal to the bottom value

If there are fewer then 2 values on the stack, terminate with error. If the two top values in the stack are not integers, terminate with error.

Example 1

```
Push 1
Push 1
Lte
```

should result in the stack

```
True
```

Example 2

```
Push 1
Push 2
Lte
```

should result in the stack

```
False
```

Example 3

```
Push 2
Push 1
Lte
```

should result in the stack

```
True
```

4.2.6 Local

Local consumes a name and a value from the top of the stack, and binds the name with that value until the end of the current scope. A `()` is then pushed onto the stack. These name/value bindings will come into full effect in Lookup command.

If there are fewer then 2 values on the stack, terminate with error. If the top value in the stack is not a name, terminate with error.

Example 1

```
Push 3
Push x
Local
```

will result in x being locally bound to 3 and () being on the stack.

Example 2

```
Push 3
Push x
Local
Push 2
Push x
Local
```

will result in x being locally bound to 3 then being locally bound to 2. The stack is empty at the end of execution. Multiple bindings will be explain in the Lookup command section.

Example 3

```
Push 3
Push y
Local
Push y
Push x
Local
```

will result in x being locally bound to the name y, and an empty stack.

4.2.7 Global

Global consumes a name and a value from the top of the stack, and binds the name with that value until the end of the program. A `()` is then pushed onto the stack. These name/value bindings will come into full effect in Lookup command.

If there are fewer then 2 values on the stack, terminate with error. If the top value in the stack is not a name, terminate with error.

Example 1

```
Push 3
Push x
Global
```

will result in x being globally bound to 3 and () being on the stack.

Example 2

```
Push 3
Push x
Global
Push 2
Push x
Global
```

will result in x being globally bound to 3 then being globally bound to 2. The stack is empty at the end of execution. Multiple bindings will be explain in the Lookup command section.

Example 3

```
Push 3
Push y
Global
Push y
Push x
Global
```

will result in x being globally bound to the name y, and an empty stack.

4.2.8 Lookup

Lookup consumes a name from the top of the stack and puts its bound value on top of the stack. If a name was bound both locally and globally, prefer the local binding over global. If multiple bindings for a name exists (either locally or globally), prefer the latest binding.

If the stack is empty, terminate with error. If the top value on the stack is not a name, terminate with error. If the name was not bound in the current scope, terminate with error.

Example 1

```
Push 3
Push x
Local
Push x
Lookup
```

should result in stack

```
3
()
```

Example 2

```
Push 3
Push x
Local
Push 6
Push x
Local
Push x
Lookup
```

should result in stack

```
6
()
()
```

Example 3

```
Push 3
Push x
Local
Push 6
Push x
Global
Push x
Lookup
```

should result in stack

```
3
()
()
```

Example 4

```
Push 3
Push x
Local
Push 6
Push y
Global
Push y
Lookup
```

should result in stack

```
6
()
()
```

4.2.9 Begin...End

A sequence of commands in a BeginEnd block will be executed on a new empty stack with a copy of the current binding scope. When the commands finish, the top value from the stack will be pushed to the outer stack, and new local bindings made from within the block disregarded. Global bindings made from within the block are valid for the rest of the program.

Example 1

```
Push 1
Push 2
Begin
  Push 3
    Push 7
  Push 4
End
Push 5
Push 6
```

will result in stack

```
6
5
4
2
1
```

Example 2

```
Push 3
Begin
  Pop 1
  Push 7
End
```

will terminate with error since the Pop command is executed with an empty inner stack.

Example 3

```
Push 55
Push x
Local
Begin
  Push 5
  Push x
  Local
End
Push x
Lookup
```

will result in the stack following stack because the local binding of x to 5 made within BeginEnd is discarded after execution leaves the scope of the BeginEnd block. Hence the binding of x reverts back to 55.

```
55
()
()
```

Example 3

```
Push 55
Push x
Local
Begin
  Push 5
  Push x
  Global
End
Push x
Lookup
```

will result in the following stack because the local binding of x to 55 is preferred over the global binding made within the BeginEnd block. So although Example 3 and Example 2 have the same ending stack state, the reason they do so are different.

```
55
()
()
```

Example 4

```
Push 55
Push x
Global
Begin
  Push 5
  Push x
  Global
End
Push x
Lookup
```

will result in the follow stack because the global binding of x to 5 made from within the BeginEnd block remains valid even after execution leaves the block.

```
5
()
()
```

4.2.10 IfElse

The IfElse command will consume the top element of the stack. If that element is True it will execute the commands in the first branch, if False it will execute the commands in the else branch. In both cases, the remaining stack is used directly for executing the commands in corresponding branch. Bindings both local and global made from within each branch are valid after executing the branch. The resulting stack after evaluating the correct branch is used to evaluate the rest of the program.

If stack is empty, terminate with error. If the top value on the stack is not a boolean, terminate with error.

Example 1

```
Push 10
Push True
If
  Push 5
  Add 2
Else
  Push 5
  Sub 2
End
```

will result in the stack

```
15
```

Example 2

```
Push 10
Push False
If
  Push 5
  Add 2
Else
  Push 5
  Sub 2
End
```

will result in the stack

```
-5
```

Example 3

```
Push 10
Push False
If
  Push 5
  Add 2
Else
  Push 234
  Push x
  Local
End
Push x
Lookup
```

will result in the following stack because the local binding of x to 234 made from within the False branch is valid after the execution of the branch. The changes made to the stack from within the False branch are also valid after executing the branch, so we see () as the second value on the stack.

```
234
()
10
```

5 Part 3: Functions and errors Due date: April 28th 11:59 AM EST

5.1 Grammar

5.1.1 Programs

com ::= ... | `Fun` *name name coms* `End`
| `Call`
| `Try` *coms* `End`
| `Switch` {`Case` *int cmds*} `End`

5.1.2 Values

Value are extended with closures, which are essentially commands paired with an environment. The environment is used to resolve names mentioned in the commands of the closure.

val ::= ... | `Clo` *env name name coms*

5.2 Error Handling

In this section, commands resulting in errors do not terminate evalution if wrapped within a TryEnd block. More detailed information is given in the TryEnd section. We will refer to errors encountered during evaluation as "raising an error" since errors do not necessarily terminate evaluation altogether.

5.3 Remarks on Examples

More examples beyond those shown in this document will be given as a .zip file.

5.4 Commands

5.4.1 Function declarations

A functions are declared with the fun command

```
Fun fname arg
  coms
End
```

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.

After a function is defined with the Fun command, a closure is formed with all local bindings in the current environment. The closure is then locally bound in the current environment to *fname*.

5.4.2 Call

The Call command tries to consume an argument value and a closure from the top of the stack. It then extends the environment contained within the closure with the following bindings.

- all current global bindings
- the function's formal parameter to the argument value
- the function's name to the closure itself

The commands contained within the closure are then executed using this newly formed environment and a fresh stack. Once the commands have finshed executing, take the topmost value of the final resulting stack (obtained from closure call) and push it onto the current stack (stack of the closure caller). All global bindings made within the called closure are valid after the closure returns and also within all of its recursive calls if the closure was defined recursively.

If an argument value and a closure are not found on top of the stack when the Call command is executed, raise an error. If the final stack produced by evaluating the closure is empty, raise an error.

Example 1

```
Fun f x
  Push x
  Lookup
  Trace 1
  Push ()
End
Push 10
Push f
Call
```

will result in log `[‘‘10’’]` and the stack

```
()
```

Example 2

```
Push 1
Push x
Local
Fun f z
  Push x
  Lookup
End
Push 3
Push x
Local
Push ()
Push f
Lookup
Call
```

will result in the following stack

```
1
()
()
```

Example 3

```
Fun fact x
  Push 0
  Push x
  Lookup
  Lte
  If
    Push 1
  Else
    Push 1
    Push x
    Lookup
    Sub 2
    Push fact
    Lookup
    Call
    Push x
    Lookup
    Mul 2
  End
End
Push 10
Push fact
Lookup
Call
```

will result in the following stack

```
3628800
```

Example 4

```
Push 10
Push x
Global
Fun loop n
  Push 0
  Push n
  Lookup
  Lte
  If
    Push ()
  Else
    Push 10
    Push x
    Lookup
    Add 2
    Push x
    Global
    Push 1
    Push n
    Lookup
    Sub 2
    Push loop
    Lookup
    Call
  End
End
Push 10
Push loop
Lookup
Call
Push x
Lookup
```

will result in the following stack

```
110
()
()
```

5.4.3 Try

The Try command will attempt to evaluate the commands in it with an empty stack and the current enviroment. If no errors are raised during evaluation, take the topmost value of the resulting stack and push it onto the current stack. If an error was raised during execution of the inner commands, evaluate the rest of the program with no modifications to the current stack. In both cases, global bindings made by the inner commands (up to the point of error) are valid when executing the rest of the program.

If the Try command successfully evaluated its inner commands but obtained an empty resulting stack, raise an error.

Example 1

```
Try
  Push 1
End
Trace 1
```

will result in the following stack

```
1
```

Example 2

```
Push 5
Try
  Push 0
  Push 10
  Div 2
End
```

will result in the following stack

```
5
```

Example 3

```
Push 10
Push x
Global
Try
  Push 20
  Push x
  Global
  Pop 10
  Push 30
  Push x
  Global
End
Push x
Lookup
```

will result in the following stack

```
20
()
```

5.4.4 Switch

The Switch command will consume an integer value from the top of the stack. Switch will then execute the commands contained in the first Case with matching integer label. These commands are executed using the current environment and remaining stack. Bindings both local and global made from within each Case are valid after executing the Case. The resulting stack after evaluating the correct Case is used to evaluate the rest of the program.

If the Switch command cannot find an intger value on top of the stack raise an error. If an integer can be found on top of the stack but non of the Case labels match it, raise an error.
Example 1

```
Push 1
Switch
Case 1
  Push 4
  Trace 1
Case 2
  Push 5
  Trace 1
Case 3
  Push 6
  Trace 1
End
```

will result in the log `[‘‘4’’]` and an empty stack.

Example 2

```
Push 2
Switch
Case 1
  Push 4
  Trace 1
Case 2
  Push 20
  Switch
  Case 10
    Push 5
    Trace 1
  Case 20
    Push 6
    Trace 1
  End
Case 3
  Push 7
  Trace 1
End
```

will result in the log `[‘‘6’’]` and an empty stack.

6 Full Grammar

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}.

6.1 Constants

digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

letter ::= `a...z` | `A...Z`

int ::= [−] *digit* {*digit*}

bool ::= `True` | `False`

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

const ::= *int* | *bool* | *name* | `()`

6.2 Programs

prog ::= *coms*

com ::= `Push` *const* | `Pop` *int* | `Trace` *int*
| `Add` *int* | `Sub` *int* | `Mul` *int* | `Div` *int*
| `Equal` | `Lte` | `And` | `Or` | `Not`
| `Local` | `Global` | `Lookup`
| `Begin` *coms* `End`
| `If` *coms* `Else` *coms* `End`
| `Fun` *name name coms* `End`
| `Call`
| `Try` *coms* `End`
| `Switch` {`Case` *int cmds*} `End`

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

6.3 Values

env ::= {*name* ↦ *val*}

val ::= *int* | *bool* | `()` | *name* | `Clo` *env name name coms*

int values can be as imprecise as machine integers.