CS 320: Stack Language Interpreter

Part 1 Due: 12 November 2021, 11:59pm.
Part 2 Due: 3 December 2021, 11:59pm.

1 Overview

The project is broken down into two 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 * (string list)
```

1.1 Functionality

The function will take a program as an `input` string, and will return the topmost element of the final stack state and a list of strings of "Traced" values. We will refer to this list of strings as the log.

1.2 Successful Execution

When the interpreter function successfully parses and executes the entirety of its input, report the string representation of the topmost value of its final stack. Also report the log generated by the Trace command.

1.3 Error Reporting

It is possible for certain commands to encounter errors during execution. When an error is encountered, the interpreter stops interpreting any remaining commands. Return the following as the result of the `interpreter` function when these error occur.

```
("Error", [])
```

2 Part 1: Basic Computation
Due Date: 12 November 2021, 11:59pm.

2.1 Grammar

For part 1 you will need to support the following grammar

2.1.1 Constants

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

nat ::= *digit* { *digit* }

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

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

const ::= *nat* | *name* | `()`

2.1.2 Programs

prog ::= *coms*

com ::= `Push` *const* | `Trace`
| `Add` | `Sub` | `Mul` | `Div`
| `If` *coms* `Else` *coms* `End`

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

2.1.3 Values

val ::= *nat* | *name* | *unit*

2.2 Commands

Your interpreter should be able to handle the following commands:

2.2.1 Push

Push *const*

All *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 8
Push ()
Push 1
```

should result in the stack

```
1
()
8
9
```

2.2.2 Trace

The Trace command consumes the top value of the stack and adds its string representation to the output list. A unit is then pushed onto the stack. Attempting to perform Trace on an empty stack results in an error.

For example,

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

should result in the stack

```
()
()
5
()
```

and

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

generated as the log.

2.2.3 Add

Add consumes the top two values in the stack, and pushes their sum to the stack. If there are fewer then 2 values on the stack, terminate and report error. If two top values in the stack are not integers, terminate and report error.

```
Push ()
Push 5
Push 7
Add
Push 3
Add
```

should result in the stack

```
15
()
```

2.2.4 Sub

Sub consumes the top two values in the stack, and pushes their difference to the stack. If there are fewer then 2 values on the stack, terminate and report error. If two top values in the stack are not integers, terminate and report error.

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

should result in the stack

```
9
()
```

2.2.5 Mul

Mul consumes the top two values in the stack, and pushes their multiplication to the stack. If there are fewer then 2 values on the stack, terminate and report error. If two top values in the stack are not integers, terminate and report error.

```
Push 5
Push 7
Mul
```

should result in the stack

```
35
```

2.2.6 Div

Div consumes the top two values in the stack, and pushes their quotient to the stack. If there are less then 2 values on the stack, terminate and report error. If two top values in the stack are not integers, terminate and report error. If the divisor is 0, terminate and report error.

```
Push 10
Push 2
Div
```

should result in the stack

```
5
```

The following program will report an error.

```
Push 10
Push 0
Div
```

will cause the interpreter function to terminate with output

```
("Error", [])
```

2.2.7 If...Else...End

The `IfElseEnd` command will consume the topmost element of the stack. If that element is greater than 0 it will execute the commands in the first branch, otherwise it will execute the commands in the second branch. If stack is empty, terminate and report error. If the top value on the stack is not an integer, terminate and report error.

Example 1:

```
Push 1;
If
    Push 123
Else
    Push 456
End;
Push ()
```

will result in the stack

```
()
123
```

Example 2:

```
Push 0
If
    Push 123
Else
    Push 12
    If
        Push 456
    Else
        Push 789
    End
End
Push ()
```

will result in the stack

```
()
456
```

Example 3:

```
Push ()
If
    Push 123
Else
    Push 456
End
Push ()
```

will cause the interpreter function to terminate with output

```
("Error", [])
```

2.3 Examples

Example 1:

```
Push 1
Push 2
Push 3
Trace
Trace
Trace
```

with result returned by the interpreter function as follows

```
("()", ["1"; "2"; "3"])
```

Example 2:

```
Push 1
Trace
Trace
Push 2
Push 3
Add
```

with result returned by the interpreter function as follows

```
("5", ["1", "()"])
```

Example 2:

```
Push 1
Trace
Trace
Push 3
Add
```

with result returned by the interpreter function as follows

```
("Error", [])
```

Example 3:

```
Push 23
If
    Push 6
    If
        Push 2
        Push 0
        Div
    Else
        Push 0
    End
Else
    Push ()
End
```

with result returned by the interpreter function as follows

```
("Error", [])
```

3 Part 2: More Computation
Due date: 3 December 2021, 11:59pm.

3.1 Grammar

For part 2, the grammar of part1 is extended in the following way

3.1.1 Programs

com ::= ...
| `Let`
| `Lookup`
| `Begin` *coms* `End`
| `Fun` *name name coms* `End`
| `Call`

3.1.2 Environments

An environment is a data structure that associates *names* and *values*. We will refer to environments as *env*.

3.1.3 Values

The values from part 1 is extended with *closures*. A *closure* associates the commands of a function with the environment it was created in. When the `Trace` command is called on a *closure*, the string `"<fun>"` should be put into the log.

val ::= ... | `Fun` *name name coms env* `End`

3.2 Commands

3.2.1 Let

`Let` consumes first a value, then a name from the top of the stack, and associates the name with that value in the current environment. If there are fewer then 2 values on the stack, terminate and report error. If the second value in the stack is not a name, terminate and report error.

Example 1:

```
Push x
Push 3
Let
Push y
Push 4
Let
```

Will result in x being bound to 3, y bound to 4 and an empty stack.

Example 2:

```
Push x
Push 34
Let;
Push x
Push 2
Let
```

Will result in x being bound to 2, and an empty stack.

Example 3:

```
Push y;
Push 3;
Let;
Push x;
Push y;
Let;
```

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

3.2.2 Lookup

`Lookup` consumes a name from the top of the stack and puts its associated value on top of the stack. If the stack is empty, terminate and report error. If the top value on the stack is not a name, terminate and report error. If the name is not bound in the current environment, terminate and report error.

Example 1:

```
Push x
Push 3
Let
Push x
Lookup
```

will result in a stack only containing 3.

Example 2:

```
Push x
Push y
Let
Push x
Lookup
```

will result in a stack only containing the name y.
Example 3:

```
Push x
Push y
Let
Push y
Lookup
```

will result in an error, because y is unbound in the current environment.

3.2.3 Begin...End

A sequence of commands in a `BeginEnd` block will be executed on a fresh stack with a copy of the current binding environment. When the commands finish, the top value from final inner stack will be pushed onto the outer stack. If stack is empty, terminate and report error. Bindings created by the inner commands are disregarded. The current log should be continued inside the `BeginEnd` block and any Traces should be kept.

Example 1:

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

will result in a stack with

```
6
5
4
2
1
```

Example 2:

```
Push 3
Begin
    Push 7
    Add
End
```

Will result in an error, because `Add` cannot find 2 integers on the stack.

Example 3:

```
Begin
    Push x
    Push 7
    Let
End
```

Will result in an error, because after the inner commands of `BeginEnd` finishes, the final stack is empty.

Example 4:

```
Begin
    Push x
    Push 7
    Let
    Push ()
End
Push x
Lookup
```

Will result in an error, because the binding created for `x` within `BeginEnd` is not valid outside of `BeginEnd`.

Example 5:

```
Push x
Push 3
Let
Begin
```

```
    Push x
    Lookup
    Trace
    Push x
    Push 2
    Let
    Push x
    Lookup
    Trace
End
Push x
Lookup
Trace
```

will result in log `["3"; "2"; "3"]`

3.2.4 Function Declarations

Functions are declared with the `Fun` command.

`Fun` *fname arg*
coms
`End`

Here, *fname* is the name of the function, *arg* is the name of the parameter to the function, and *coms* are the commands that are executed when the function is called. After a function is declare with the `Fun` command, a closure is formed using the current environment and the closure is bound in the env to *fname*. The closure value is only put in the env, not on the stack.

3.2.5 Call

The `Call` command tries to consume first an argument value and second a closure from the top of the stack. The closure's env is extended with the closure value being bound to *fname* and the argument value being bound to *arg*. Then the commands in the closure body are executed using a fresh stack and the new env.

When the commands of the closure finish, the top element of its final stack is pushed to the calling stack. If there are fewer then 2 values on the stack when `Call` is executed, terminate and report error. If second value on the stack is not a closure, terminate and report error. If after the closure has finished executing and its stack is empty, terminate and report error.

Example 1:

```
Fun f x
    Push x
    Lookup
    Trace
    Push 1
End
Push f
Lookup
Push 35
Call
```

Will result in "35" being logged and 1 on the stack.

Functions use lexical scope: names in the body of the function are captured from the environment when the function is defined.

Example 2:

```
Push x
Push 1
Let
Fun f z
    Push x
    Lookup
    Push x
    Push 2
    Let
End
Push x
Push 3
Let
Push f
Lookup
Push 4
Call
```

Will result in a stack containing only 1 and x bound to 3 in the environment.

Functions can refer to themselves (recursion).

Example 3:

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

will result in a stack containing only (), with log ["10"; "9"; "8"; "7"; "6"; "5"; "4"; "3"; "2"; "1"].