CS 320: Language Interpreter Design

Part 1 Due: 4th April, at 11:59pm EST
Part 2 Due: 16th April, at 11:59pm EST
Part 3 Due: 25th April, at 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) * int
```

2 Functionality

the function will take a program as an `input` string, and will return list of strings "logged" by the program and an error code.

3 Part 1: Basic Computation Due Date: TODO, at 11:59pm

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`

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

int ::= [−] *digit* { *digit* }

bool ::= `<true>` | `<false>`

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

string ::= `"`{ ASCII \" }`"`

const ::= *int* | *bool* | *string* | *name* | `<unit>`

3.1.2 Programs

prog ::= *coms*

com ::= `Push` *const* | `Pop` | `Swap`
| `Log`
| `Add` | `Sub` | `Mul` | `Div` | `Rem` | `Neg`

coms ::= *com* `;` {*com* `;` }

3.1.3 Values

val ::= *int* | *bool* | *string* | *unit*

3.2 Error Codes

For part 1 you will need to support the following error codes

0	no error
1	type error
2	too few elements on stack
3	div by 0

For this part, throwing an exception results in the program exiting immediately with with the given error code.

3.3 Commands

Your interpreter should be able to handle the following commands:

3.3.1 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 "  a string ";
Push <true>;
Push <false>;
Push <unit>;
```

should result in the stack

```
<unit>
<false>
<true>
"  a string "
9
```

3.3.2 Pop

The command Pop removes the top value from the stack. If the stack is empty, throw an exception with error code 2.

For example,

```
Push <true>;
Push <false>;
Push <unit>;
Pop;
```

should result in the stack

```
<false>
<true>
```

and

```
Push <true>;
Push <false>;
Push <unit>;
Pop;
Pop;
Pop;
Pop;
```

should result in termination with error code 2.

3.3.3 Log

The Log command consumes the top value of the stack and adds its string representation to the output list.

If the stack is empty, throw an exception with error code 2.

For example,

```
Push <unit>;
Push 5;
Push 1;
Push 2;
Log;
Log;
```

should result in the stack

```
<unit>
5
```

and

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

returned in the output list

When logging a function value (from Part 3) use the string "<fun>".

For instance,

```
DefFun f x
    Push x;
End;
Push f; Ask; Log;
```

should result in "<fun>".

3.3.4 Swap

The command Swap interchanges the top two elements in the stack

If there are fewer then 2 values on the stack, throw an exception with error code 2.

```
Push <unit>;
Push 5;
Swap;
```

should result in the stack

```
<unit>
5
```

3.3.5 Add

Add consumes the top two values in the stack, and pushes their addition to the stack.

If there are fewer then 2 values on the stack, throw an exception with error code 2.

If two top values in the stack are not integers, throw an exception with error code 1.

```
Push <unit>;
Push 5;
Push 7;
Add;
Push 3;
Add;
```

should result in the stack

```
15
<unit>
```

3.3.6 Sub

Sub consumes the top two values in the stack, and pushes their subtraction to the stack.
If there are fewer then 2 values on the stack, throw an exception with error code 2.
If two top values in the stack are not integers, throw an exception with error code 1.

```
Push <unit>;
Push 1;
Push 10;
Sub;
```

should result in the stack

```
9
<unit>
```

3.3.7 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, throw an exception with error code 2.
If two top values in the stack are not integers, throw an exception with error code 1.

```
Push 5;
Push 7;
Mul;
```

should result in the stack

```
35
```

3.3.8 Div

Div consumes the top two values in the stack, and pushes their division to the stack.
If there are fewer then 2 values on the stack, throw an exception with error code 2.
If two top values in the stack are not integers, throw an exception with error code 1.
If the 2nd value of the stack is 0, throw an exception with error code 3.
For example,

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

should result in the stack

```
5
```

For example,

```
Push 0;
Push 10;
Div;
```

will throw error code 3.

3.3.9 Rem

Rem consumes the top two values in the stack, and pushes their mod to the stack. Rem mimicks OCaml's mod for dealing with negative behaviour.

If there are fewer then 2 values on the stack, throw an exception with error code 2.

If two top values in the stack are not integers, throw an exception with error code 1.

If the 2nd value of the stack is 0, throw an exception with error code 3.

```
Push 3;
Push 10;
Rem;
```

should result in the stack

```
1
```

3.3.10 Neg

Neg consumes the top value of the stack, x, and pushes -x to the stack.

If stack is empty, throw an exception with error code 2.

If the top value on the stack is not an integer, throw an exception with error code 1.

4 Part 2: More Computation, Definitions and Scope Due date: TODO, at 11:59pm

4.1 Grammar

For part 2 the grammar is extended in the following way

4.1.1 Constants

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

const ::= ... | *name*

4.1.2 Programs

com ::= ... | `Cat`
  | `And` | `Or` | `Not`
  | `Eq`
  | `Lte` | `Lt` | `Gte` | `Gt`
  | `Let`
  | `Ask`
  | `Begin` *coms* `End`
  | `If` *coms* `Else` *coms* `End`

4.1.3 Values

val ::= ... | *name*

4.2 Error Codes

...	
4	var not in scope

For this part, throwing an exception results in the program exiting immediately with with the given error code.

4.3 Commands

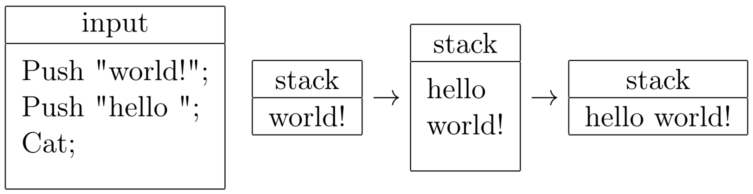
4.3.1 Cat

Cat consumes the top two values in the stack and if they are strings pushes a new string to the stack that appends the 2 strings together.

If there are fewer then 2 values on the stack, exit immediately with error code 2.

If the two top values in the stack are not strings, exit immediately with error code 1.

For example:



```
Push "world!";
Push "hello ";
Cat;
```

4.3.2 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, throw an exception with error code 2.

If the two top values in the stack are not booleans, throw an exception with error code 1.

For example:

input
Push <true>; Push <false>; And;

stack
<true>

→

stack
<false> <true>

→

stack
<false>

4.3.3 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, throw an exception with error code 2.

If the two top values in the stack are not booleans, throw an exception with error code 1.

For example:

input
Push <true>; Push <false>; Or;

stack
<true>

→

stack
<false> <true>

→

stack
<true>

4.3.4 Not

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

If the stack is empty, throw an exception with error code 2.

If the top value on the stack is not an boolean, throw an exception with error code 1.

For example:

input
Push <true>; Not;

stack
<true>

→

stack
<false>

4.3.5 Eq

Eq 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, throw an exception with error code 2.

If the two top values in the stack are not integers, throw an exception with error code 1.

For example:

input

Push 7;
Push 7;
Eq;

stack
7

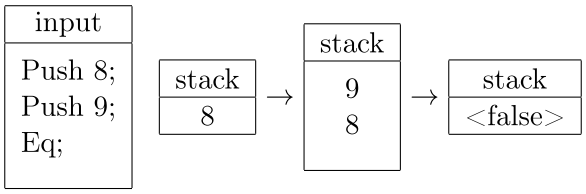
→

stack
7 7

→

stack
<true>

Consider another example:



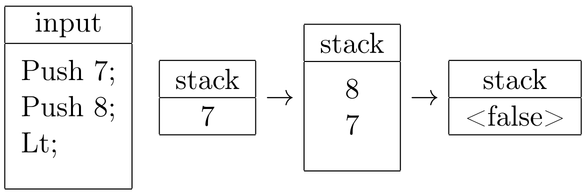
4.3.6 Lte, Lt, Gte, Gt

Lt consumes the top two values in the stack, and pushes true on the stack if the top value is less then the bottom value

If there are fewer then 2 values on the stack, throw an exception with error code 2.

If the two top values in the stack are not integers, throw an exception with error code 1.

The commands Lte, Gte, Gt behave similarly (corresponding to the standard $\leq, \geq, >$ ordering on integers). For example:



4.3.7 Let

Let consumes a name and a value from the top of the stack, and associates the name with that value until the end of the scope.

If there are fewer then 2 values on the stack, throw an exception with error code 2.

If the top value in the stack is not a name, throw an exception with error code 1.

For instance,

```
Push 3;
Push x;
Let;
Push "hello";
Push y;
Let;
```

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

```
Push 3;
Push x;
Let;
Push 2;
Push x;
Let;
```

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

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

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

4.3.8 Ask

Ask consumes a name from the top of the stack and returns the associated value.

If the stack is empty, throw an exception with error code 2.

If the top value on the stack is not a name, throw an exception with error code 1.

For instance,

```
Push 3;
Push x;
Let;
Push x;
Ask;
```

will result in a stack only containing 3.

4.3.9 Begin...End

A sequence of commands in a begin end block will be executed on a new empty stack with a copy of the current binding environment. When the commands finish, the top value from the stack will be pushed to the outer stack, and new bindings disregarded.

If stack is empty, throw an exception with error code 2.

```
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
```

For example,

```
Push 3;
Begin
    Pop;
    Push 7;
End;
```

Will exit with error code 2 since you cannot Pop an empty stack

and,

```
Begin
    Push 7;
    Pop;
End;
```

Will exit with error code 2 since the stack ends empty.

```
Push 3;
Push x;
Let;
Begin
    Push x;
    Ask;
    Log;
    Push 2;
    Push x;
    Let;
    Push x;
    Ask;
    Log;
    Push unit;
End;
Push x;
Ask;
Log;
```

will log [3,2,3]

4.3.10 If...Else...End

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.

If stack is empty, throw an exception with error code 2.

If the top value on the stack is not a Boolean, throw an exception with error code 1.

For example:

```
Push "before...";
Push <true>;
If
    Push "in the true branch";
Else
    Push "in the false branch";
End;
Push "...after";
```

will result in the stack

```
"...after"
"in the true branch"
"before..."
```

and

```
Push "before...";
Push <false>;
If
    Push "in the true branch";
Else
    Push "in the false branch";
End;
Push "...after";
```

will result in the stack

```
"...after"
"in the false branch"
"before..."
```

5 Part 3: Functions and errors — Due date: TODO, at 11:59pm

5.1 Grammar

5.1.1 Programs

com ::= ... | `DefFun` *name name coms* `End`
  | `Call`
  | `Throw`
  | `Try` *coms* `Catch` *coms* `End`

5.1.2 Values

env ::= {*name* , *val* ; }

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

5.2 Error Codes

...	
i	user defined errors

5.2.1 Commands

5.2.2 Function declarations

A functions are declared with the fun command

`DefFun` *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 DefFun command it is bound in the environment to fname.

5.2.3 Call

The `Call` command tries to consume an argument value and a function. Then it executes the commands in the function body in a fresh stack using the original environment with the function bound to fname and the value bound to the originally defined arg name, when the commands of the function are finished the top element is pushed to the calling stack.

If there are fewer then 2 values on the stack, throw an exception with error code 2.

If 2nd value on the stack is not a function, throw an exception with error code 1.

If after the function is finished running its stack is empty, throw an exception with error code 1.

For instance,

```
DefFun f x
    Push x; Ask; Log;
    Push 1;
End;
Push f; Ask;
Push "hi";
Call;
```

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

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

For instance,

```
Push 1; Push x; Let;
DefFun f z
    Push x; Ask;
    Push 2; Push x; Let;
End;
Push 3; Push x; Let;
Push f; Ask;
Push 4;
Call;
```

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

Functions can refer to themselves:

```
DefFun f x
    Push x; Ask;
    Push 0;
    Eq;
    If
        Push <unit>;
    Else
        Push f; Ask;
        Push x; Ask; Log;
        Push 1;
        Push x; Ask;
        Sub;
        Call;
    End;
End;
Push f; Ask;
Push 10;
Call;
```

will result in a stack containing only <unit>, and 10,9,8,7,6,5,4,3,2,1 will be logged.

5.2.4 Throw

The throw command tries to read an integer off of the top of the stack. Then immediately throws an exception of that error code.

If stack is empty, immediately throw error code 2.

If the top value on the stack is not an integer, immediately throw error code 1.

For example,

```
Begin
    Push "a"; Log;
    Begin
        Push "b"; Log;
        Begin
            Push "c"; Log;
            Push 42; Throw;
            Push "d"; Log;
        End;
        Push "e"; Log;
    End;
    Push "f"; Log;
End;
```

Will terminate with error code 42 and log "a","b","c"

5.2.5 TryCatch

Both user defined and built in errors can be recovered from with the TryCatch construct.

If no errors are thrown in the TryCatch construct then execution happens as normal, and the catch branch is ignored.

If an exception is thrown from a command executed in a try catch block the catch commands are run with the the original environment, and original stack with the error code pushed to the top.

For example,

```
Try
    Push "a";
Catch
    Push "b";
End;
Log;
```

Will log "a" and leave an empty stack

```
Push 1; Push x; Let;
Push "a";
Try
    Push 2; Push x; Let;
    Push "b";
    Push 42; throw;
    Push 2; Push x; Let;
    Push "c";
Catch
    Log;
End;
Push x; Ask;
```

will result in the stack 1,"a". and "42" will be logged.

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}.

ASCII is the ASCII character set.

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* | _ | ´}

string ::= `"`{ ASCII \\`"` }`"`

const ::= *int* | *bool* | *string* | *name* | `<unit>`

6.2 Programs

prog ::= *coms*

com ::= `Push` *const* | `Pop` | `Swap`
| `Log`
| `Add` | `Sub` | `Mul` | `Div` | `Rem` | `Neg`
| `Cat`
| `And` | `Or` | `Not`
| `Eq`
| `Lte` | `Lt` | `Gte` | `Gt`
| `Let`
| `Ask`
| `Begin` *coms* `End`
| `If` *coms* `Else` *coms* `End`
| `DefFun` *name* *name* *coms* `End`
| `Call`
| `Throw`
| `Try` *coms* `Catch` *coms* `End`

coms ::= *com* `;` {*com* `;` }

6.3 Values

env ::= {*name* , *val* ; }

val ::= *int* | *bool* | *string* | *unit* | *name* | `Fun` *env* *name* *name* *coms* `End`

int values can be as imprecise as machine integers.

7 Error Codes

0	no error
1	type error
2	too few elements on stack
3	div by 0
4	var not in scope
i	user defined errors