BOOTSTRAP

IT’S TORTOISES ALL THE WAY DOWN...

BOOTSTRAP

The goal of this boostrap is to help you start to write a virtual machine (VM) to run the code of your language.

This VM will be a stack machine. It's not the only way to implement a VM, but it's a popular one for its good tradeof between speed and simplicity.

Nothing in this boostrap is stricly mandatory. You are free to add, remove, amend every aspects of the system described in this document. Neverthless, it's a good starting point.

1 - Base types and simple operations

First, we have to define some types and data structures for your virtual machines.

Values

We have to define a **Value** data type which will describe the types of the objects of your VM (and ultimatly, your language).

✓ Declare a data structure which can hold:
  - A numerical type (Int)
  - A boolean type (Bool)

(we will add more soon...)

Builtins / Operators

We need builtin operations our VM will know how to perform on its stack. Here wee will start with the obvious arithmetic operations:

✓ Declare a data structure to represent:
  - Addition
  - Substraction
  - Multiplication
  - Division

(we will add more soon...)

Each of this operators will take N values at the top of the stack, and push its result instead.

✓ Example in pseudocode:

```
stack = [ 11, 31 , 5 ]
Apply Addition:
a <- Pop
stack = [ 31, 5 ]
b <- Pop
stack = [ 5 ]
Push (b+a)
stack = [ 42, 5 ]
```

Instructions:

Finaly we need a data structure to represente the instructions our VM suports.

✓ Declare a data structure to represent our instructions:
  - **Push** {Value} (or Load Immediate) : add a new value on top of the stack
  - **Call** {Op} : apply the builtin **Op** on the stack
  - **Ret** : take the value on the top of the stack and returns it (terminating the execution)

(guess what? We will add more soon!)

Types and Signatures

Now for convenience only, we can define a couple of helper types:

- ✓ A stack will be defined as a list of values
- ✓ A program will be defined as a list of instructions

```
type Stack = ???
type Insts = ???
```

First VM

With these building blocks, let's implement the first version of our VM, which will take a list of instructions, a stack, and will return the computed value.

Its signature should look like this:

```
exec :: Insts -> Stack -> Val
```

This first version should be able to take simple aritmetic programs and evaluate them:

```
Push 42
Ret
# exec => 42

Push 10
Push 52
Call Sub
Ret
# exec => 42
```

The examples above are presented in pseudo-code, you should represent them using your Haskell data structures and invoke your **exec** function from GHCI, unit tests or main

2 - Error handling

Now is a good time to change our VM so it can handle errors safely:

```
exec :: Insts -> Stack -> Either String Val
```

Example:

```
Push 10
Call Add
# exec => Error: Add need two arguments

Push 0
Push 42
Call Div
Ret
# exec => Error: division by 0
```

3 - Conditions

All of this is nice and good but to implement a real programming language we will need conditionals.

Our conditions will operate on bools, so before implementing them we need to add predicates to our list of builtins which will take values from the stack and leave boolean results on it:

Add Eq and Less to Operators

✓ Example:

```
Push 10
Push 10
Call Eq
Ret
# exec => True
Push 2
Push 5
Call Less
# exec => False
```

Add JumpIfFalse

Now we have everything we need to add a conditional jump in our code.
It takes a number of instruction as argument, which is the number of instructions to ignore in case of a jump. Otherwise the program continues with the next instruction.

✓ Example:

```
Push 10
Push 10
Call Eq
JumpIfFalse 2
Push 1
Ret
Push 2
Ret
# exec => 1
```

4 - Arguments

Our end goal being to execute whole functions, we need to represent their arguments. To do so we need a type to represent a list of argument and we need to add a new parameter to our exec function:

```
type Args = ???

[...]

exec :: Args -> Insts -> Stack -> Either String Val
```

PushArg

With this in place, we can now add a new instruction to pick an argument in the list of arguments and push it's value on the stack.

This instruction takes an int which is the index of the argument to push on the stack

✓ Example:

```
PushArg 0
Push 0
Call Less
JumpIfFalse 2
PushArg 0
Ret
PushArg 0
Push -1
Call Mul
Ret
# exec 42 => 42
# exec -42 => 42
```

5 - User defined functions

We want to be able to define user functions and call them from our code.
This will need some refactoring of our types.

We want to be able to manipulate functions in the same way we manipulate other kind of data (as it should be in a functional language). To do so our functions must be represented as values.

But we already have something akin to functions, it's our builtin operators. In order to keep everything consistent, we will first make our operators real values, and change how the Call instruction work to reflect this change: Call will no longer take an argument but pop the object to call from the stack.

Therefor the code previously written:

```
PushArg 0
Push -1
Call Mul
```

Will now be noted:

```
PushArg 0
Push -1
Push Mul
Call
```

After this refactoring, we can add a new type of value which will represent user defined functions. We will also have to modify Call so it knows how to handle both Op and Functions.

For now, user defined functions are just lists of instructions

We can now express something like this:

```
absCode =
  PushArg 0
  Push 0
  Push Less
  Call
  JumpIfFalse 2
  PushArg 0
  Ret
  PushArg 0
  Push -1
  Push Mul
  Call
  Ret

Push -42
Push absCode
Call
Ret
exec => 42
```

6 - Env and recursive functions

At this point, it's already possible to write recurcive functions, even if our functions can't call themselves, but it's very tricky.

To make recursive functions easier to write, we'll introduce yet another parameter to our exec function, to hold an environement.

Write a type to represent an environement

```
type Env = ???
```

Add a new instruction to push on the stack a value from the environment.

✓ Example:

```
env =
    fact =
        PushArg 0
        Push 1
        Push Eq
        Call
        JumpIfFalse 2
        Push 1
        Ret
        Push 1
        PushArg 0
        Push Sub
        Call
        PushEnv "fact"
        Call
        PushArg 0
        Push Mul
        Call
        Ret

Push 5
PushEnv "fact"
Call
Ret
exec => 120
```

6bis - A simpler environment

This is optional, but kepping in the runtime a mechanism to resolve a name to a value is not very sound.

You could change how your environment is represented and accessed in such a way that the resolution is done once and for all at compile time (or at loading time).

This is an optimisation, don't spend too much time on it.

Conclusion

At this point you should have a pretty usable Virtual Machine for your language. What you need to implement now is a **Compiler** function which will take your **Ast** tree representing a function code, and returns a data structure representing this function in the form expected by your **Virtual Machine**.

Also, now is a good time to go read 500 Lines or Less | A Python Interpreter Written in Python.

You will notice that some things are very similar, other are done differently. Keep in mind that they are implementing an interpreter for an imperative language. Are you implementing an imperative language?

v 2.1