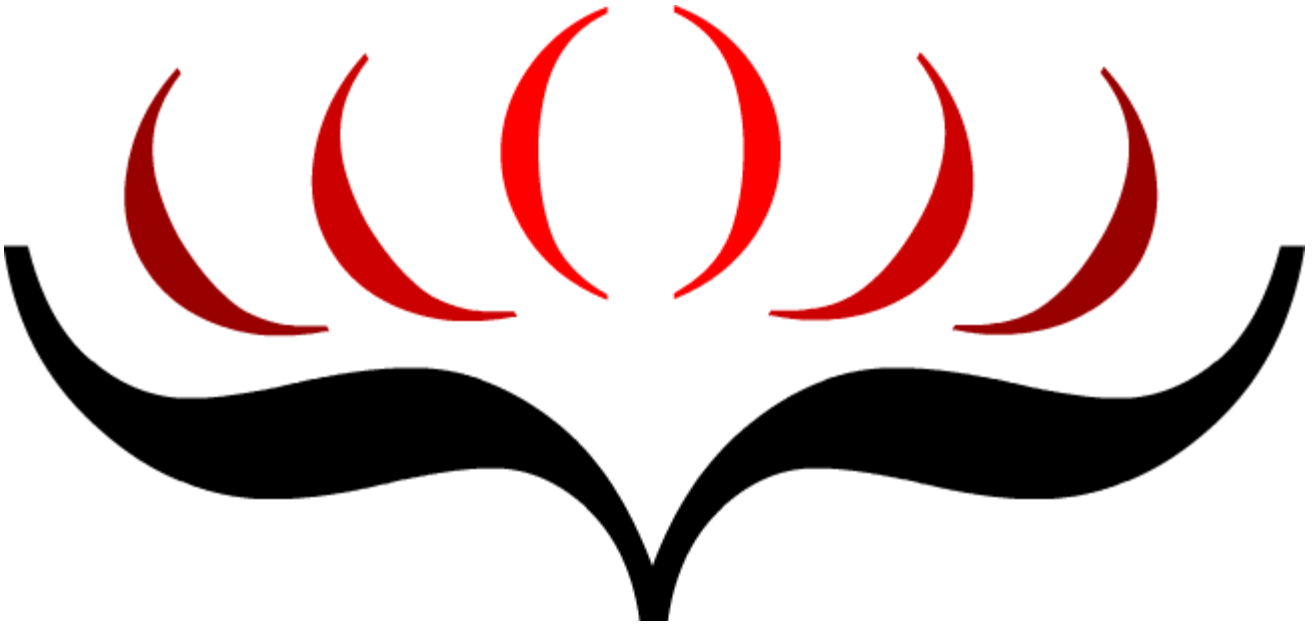
Brainfuck interpreter in Haskell



Vladimir Alekseichenko

Agenda

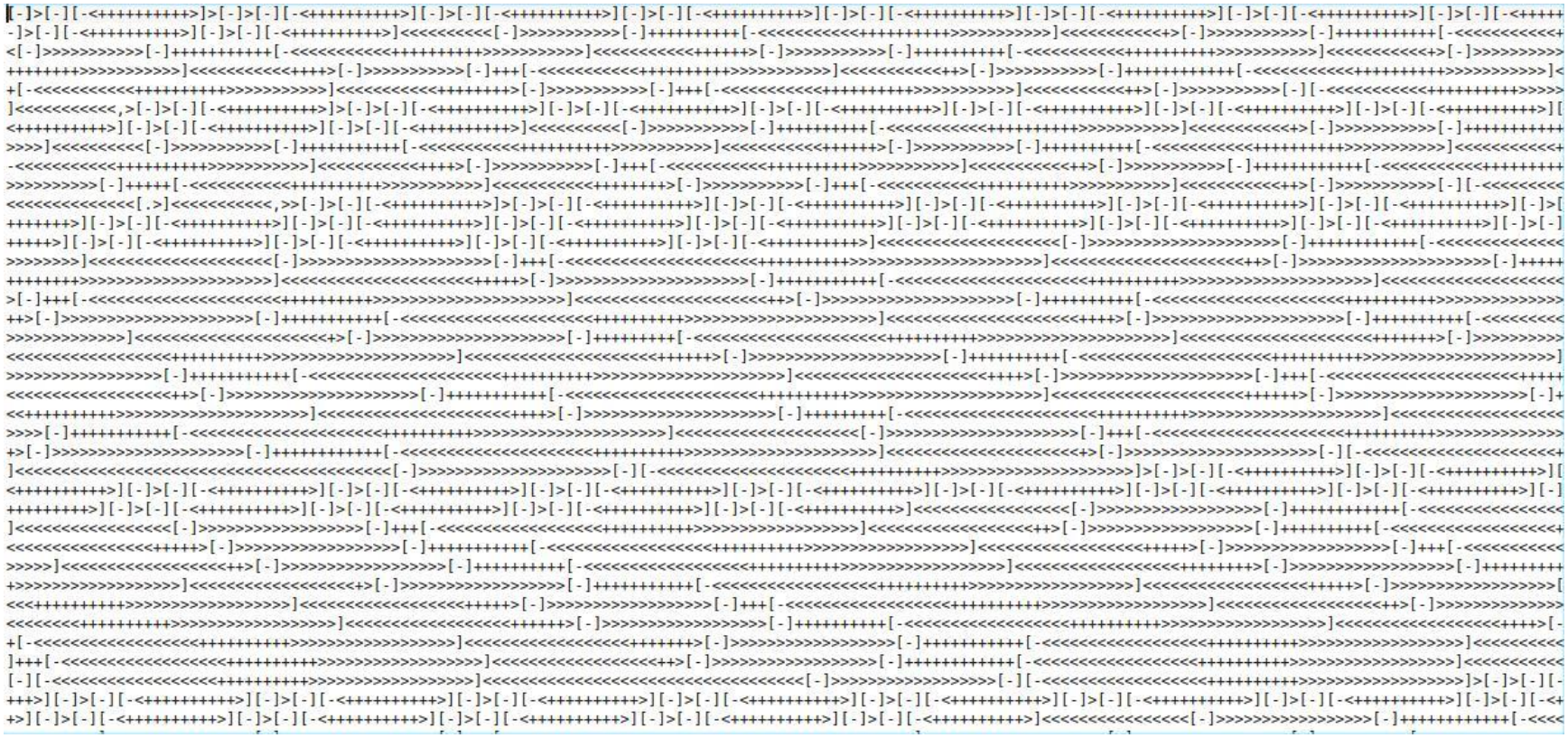
- Intro to brainfuck
- Explanation idea
- Implement of simple version
- Some advice - what you can do next

Brainfuck

It is an **esoteric** programming language noted for its **extreme minimalism**.

http://en.wikipedia.org/wiki/Brainfuck

Brainfuck

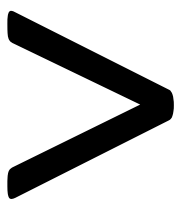


http://sourceforge.net/projects/brainfix/

BF

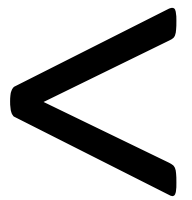
> < + - . , [ ] #

Commands (1)



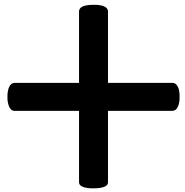
Step to the right

Commands (2)



Step to the left

Commands (3)



Increment the value

(increase by one)

Commands (4)

-

Decrement the value

(decrease by one)

Input/Output (1)

Write a sybmol

Input/Output (2)

,

Read a symbol

Control structures (1)

[

The begin loop

Control structures (2)

]

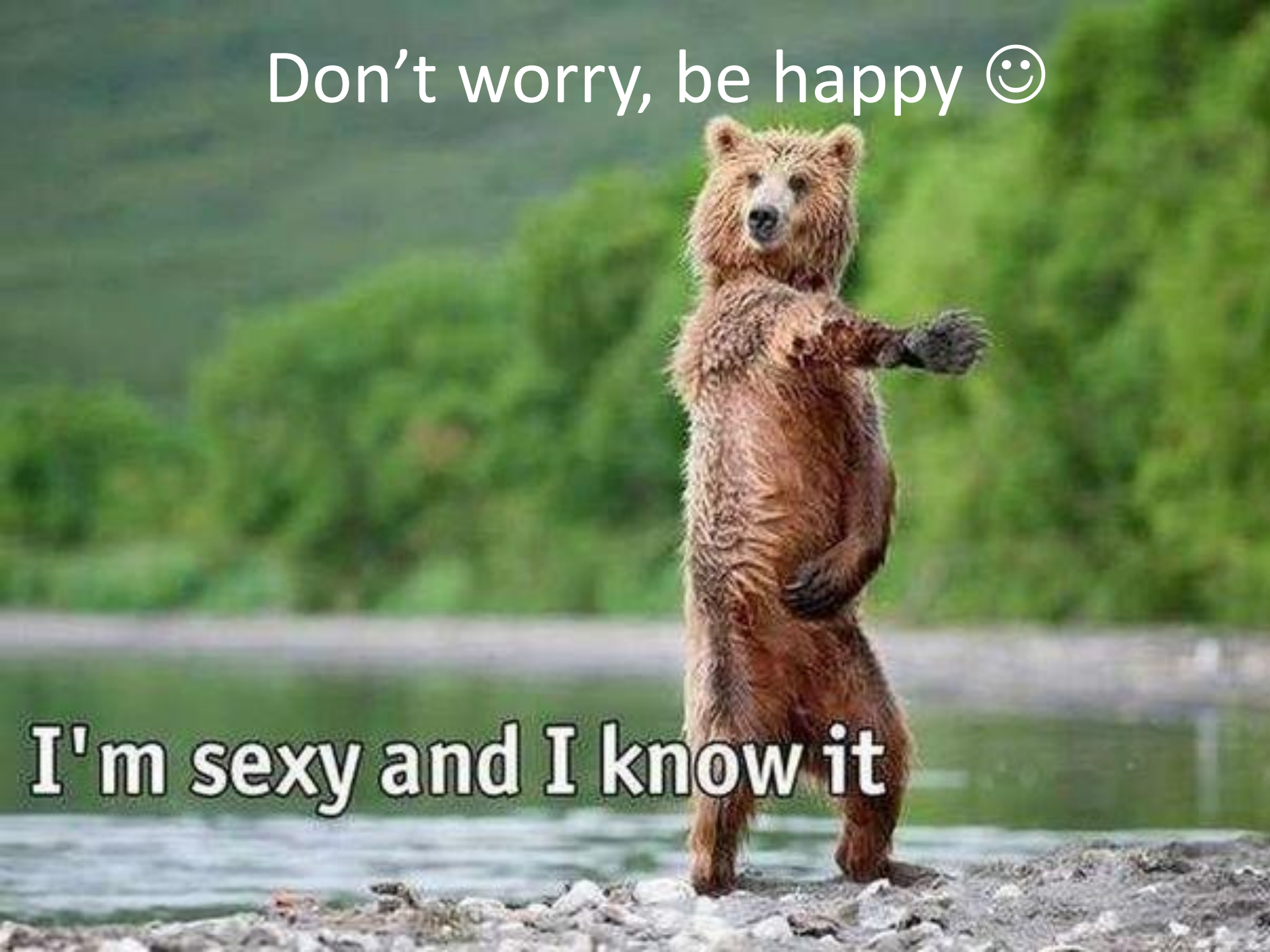
The end loop

Brainfuck in hardware

How does it implement?

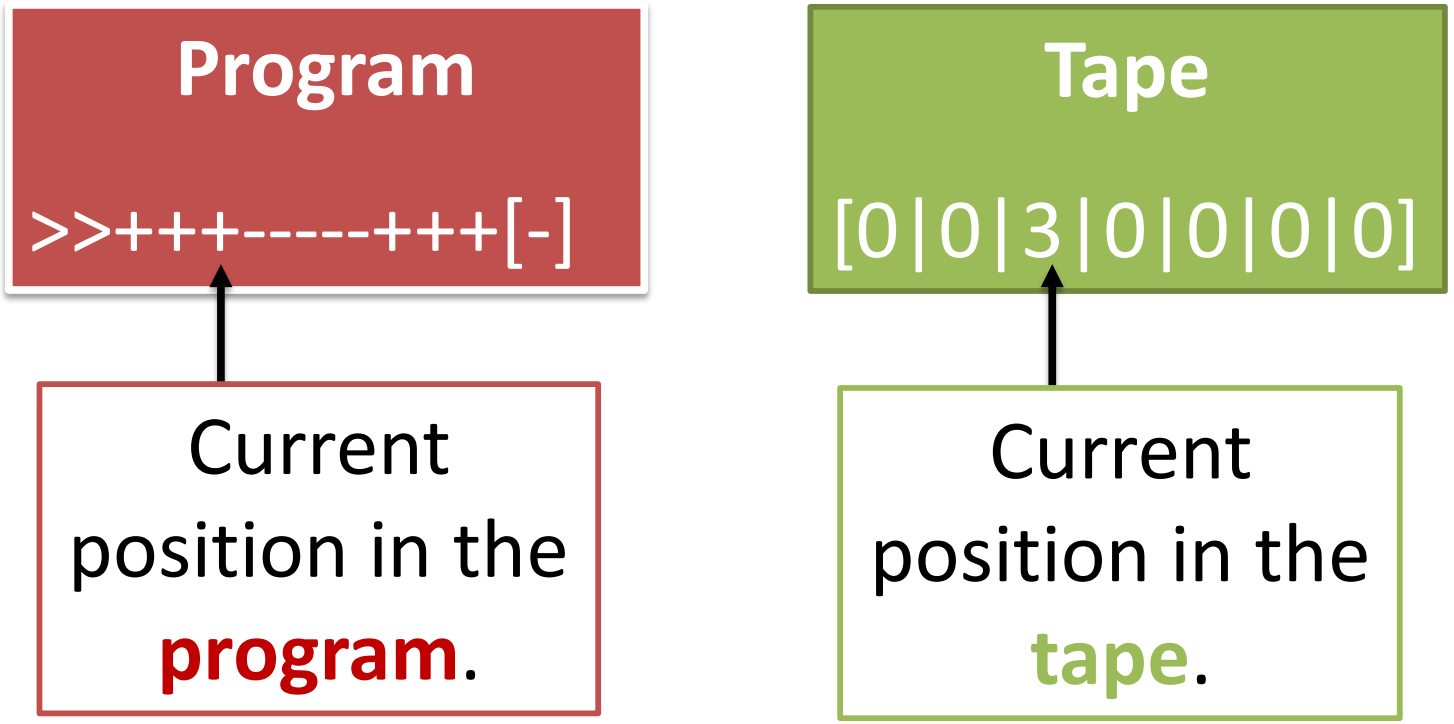
http://www.gigante.pl/obrazki-3-1-12

Don’t worry, be happy ☺



What we need to be happy?

Four things ☺.

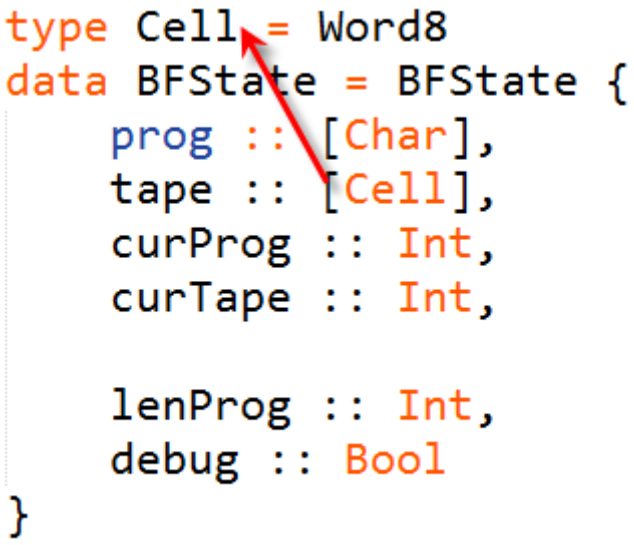


BFState

```
type Cell = Word8
data BFState = BFState {
    prog :: [Char],
    tape :: [Cell],
    curProg :: Int,
    curTape :: Int,

    lenProg :: Int,
    debug :: Bool
}
```

BFState


type Cell = Word8
data BFState = BFState {
prog :: [Char],
tape :: [Cell],
curProg :: Int,
curTape :: Int,
lenProg :: Int,
debug :: Bool
}

main :: *IO()*

```
main :: IO ()
main = do
    run $ initState 10 "++++++++++[>+++++++>++++++++++>+++>+<<<<-]>+\
          \+.>+.+++++++..+++.>++.<<+++++++++++++++.>.++\
          \+.------.--------.>+.>.<"
```

main :: *IO()*

```haskell
main :: IO ()
main = do
    run $ initState 10 "++++++++++[>+++++++>++++++++++>+++>+<<<<-]>+\
          \+.>+.+++++++..+++.>++.<<+++++++++++++++.>.++\
          \+.------.--------.>+.>.<"
```

main :: *IO()*

```
main :: IO ()
main = do
    run $ initState 10 "++++++++++[>+++++++>++++++++++>+++>+<<<<-]>+\
        \+.>+.+++++++..+++.>++.<<+++++++++++++++.>.++\
        \+.------.--------.>+.>.<"
```

initState :: *Int -> [Char] -> BFState*

```
initState :: Int -> [Char] -> BFState
initState capTape prog = BFState {
    prog=prog, tape=(take capTape $ repeat 0),
    curProg = 0, curTape = 0, lenProg = length(prog), debug = False}
```

repeat :: *a -> [a]*

repeat x is an infinite list, with x the value of every element.

take :: *Int -> [a] -> [a]*

take n, applied to a list xs, returns the prefix of xs of length n.

initState :: *Int -> [Char] -> BFState*

Initialize Tape

```
initState :: Int -> [Char] -> BFState
initState capTape prog = BFState {
    prog=prog, tape=(take capTape $ repeat 0),
    curProg = 0, curTape = 0, lenProg = length(prog), debug = False}
```

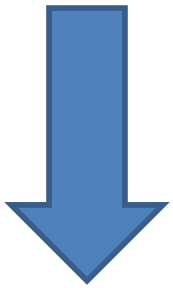
run :: *BFState -> IO ()*

```haskell
run :: BFState -> IO ()
run st = do
    if (debug st)
        then putStrLn $ show ((curTape st), (curProg st), (tape st))
        else return ()

    if isEnd st
        then return ()
        else step st >>= \st' -> run st'
```

It’s equivalent

step st >>= \st’ -> **run** st’



step st >>= **run**

run :: *BFState -> IO ()*

```
run :: BFState -> IO ()
run st = do
    if (debug st)
        then putStrLn $ show ((curTape st), (curProg st), (tape st))
        else return ()

    if isEnd st
        then return ()
        else step st >>= \st' -> run st'
```

Show info in the case if debugging is enabled

run :: *BFState -> IO ()*

```
run :: BFState -> IO ()
run st = do
    if (debug st)
        then putStrLn $ show ((curTape st), (curProg st), (tape st))
        else return ()

    if isEnd st
        then return ()
        else step st >>= \st' -> run st'
```

If this isn't the end, go make another step

step :: *BF State -> IO BFState*

```
step :: BFState -> IO BFState
step st = case elem of
    '+' -> return st {tape = chngVal st (+1), curProg = (curProg st) + 1}
    '-' -> return st {tape = chngVal st (subtract 1), curProg = (curProg st) + 1}
    '>' -> return st {curTape = (curTape st) + 1, curProg = (curProg st) + 1}
    '<' -> return st {curTape = (curTape st) - 1, curProg = (curProg st) + 1}
    '[' -> return st {curProg = (opnLoop st)}
    ']' -> return st {curProg = (clsLoop st)}
    '.' -> writeVal st
    ',' -> readVal st
    '#' -> return st {debug = True, curProg = (curProg st) + 1}

    _   -> return st {curProg = (curProg st) + 1}

    where elem = (prog st) !! (curProg st)
```

step :: *BF State -> IO BFState*

```
step :: BFState -> IO BFState
step st = case elem of
    '+' -> return st {tape = chngVal st (+1), curProg = (curProg st) + 1}
    '-' -> return st {tape = chngVal st (subtract 1), curProg = (curProg st) + 1}
    '>' -> return st {curTape = (curTape st) + 1, curProg = (curProg st) + 1}
    '<' -> return st {curTape = (curTape st) - 1, curProg = (curProg st) + 1}
    '[' -> return st {curProg = (opnLoop st)}
    ']' -> return st {curProg = (clsLoop st)}
    '.' -> writeVal st
    ',' -> readVal st
    '#' -> return st {debug = True, curProg = (curProg st) + 1}

    _   -> return st {curProg = (curProg st) + 1}

    where elem = (prog st) !! (curProg st)
```

step :: *BF State -> IO BFState*

```
step :: BFState -> IO BFState
step st = case elem of
    '+' -> return st {tape = chngVal st (+1), curProg = (curProg st) + 1}
    '-' -> return st {tape = chngVal st (subtract 1), curProg = (curProg st) + 1}
    '>' -> return st {curTape = (curTape st) + 1, curProg = (curProg st) + 1}
    '<' -> return st {curTape = (curTape st) - 1, curProg = (curProg st) + 1}
    '[' -> return st {curProg = (opnLoop st)}
    ']' -> return st {curProg = (clsLoop st)}
    '.' -> writeVal st
    ',' -> readVal st
    '#' -> return st {debug = True, curProg = (curProg st) + 1}

    _   -> return st {curProg = (curProg st) + 1}
    where elem = (prog st) !! (curProg st)
```

step +

```haskell
step :: BFState -> IO BFState
step st = case elem of
    '+' -> return st {tape = chngVal st (+1), curProg = (curProg st) + 1}
    '-' -> return st {tape = chngVal st (subtract 1), curProg = (curProg st) + 1}
    '>' -> return st {curTape = (curTape st) + 1, curProg = (curProg st) + 1}
    '<' -> return st {curTape = (curTape st) - 1, curProg = (curProg st) + 1}
    '[' -> return st {curProg = (opnLoop st)}
    ']' -> return st {curProg = (clsLoop st)}
    '.' -> writeVal st
    ',' -> readVal st
    '#' -> return st {debug = True, curProg = (curProg st) + 1}

    _   -> return st {curProg = (curProg st) + 1}

    where elem = (prog st) !! (curProg st)
```

step +

```
'+' ->
    return st {
        tape = chngVal st (+1),
        curProg = (curProg st) + 1}
```

chgVal and **putVal**

tape = **chngVal** st (+1)

```
chngVal :: BFState -> (Cell -> Cell) -> [Cell]
chngVal st f = putVal (tape st) (curTape st) (f $ getVal st)

putVal :: [Cell] -> Int -> Cell -> [Cell]
putVal tape idx newVal = befTape ++ [newVal] ++ (tail aftTape)
                         where (befTape, aftTape) = splitAt idx tape
```

splitAt

splitAt :: *Int -> [a] -> ([a], [a])*

splitAt n xs returns a tuple where first element is xs prefix of length n and second element is the remainder of the list

http://hackage.haskell.org/package/base-4.6.0.1/docs/Data-List.html

splitAt

```
splitAt 6 "Hello World!" == ("Hello ","World!")
splitAt 3 [1,2,3,4,5] == ([1,2,3],[4,5])
splitAt 1 [1,2,3] == ([1],[2,3])
splitAt 3 [1,2,3] == ([1,2,3],[])
splitAt 4 [1,2,3] == ([1,2,3],[])
splitAt 0 [1,2,3] == ([],[1,2,3])
splitAt (-1) [1,2,3] == ([],[1,2,3])
```

http://hackage.haskell.org/package/base-4.6.0.1/docs/Data-List.html

chgVal and **putVal**

tape = **chngVal** st (+1)

```
chngVal :: BFState -> (Cell -> Cell) -> [Cell]
chngVal st f = putVal (tape st) (curTape st) (f $ getVal st)

putVal :: [Cell] -> Int -> Cell -> [Cell]
putVal tape idx newVal = befTape ++ [newVal] ++ (tail aftTape)
                    where (befTape, aftTape) = splitAt idx tape
```

step -

```haskell
step :: BFState -> IO BFState
step st = case elem of
    '+' -> return st {tape = chngVal st (+1), curProg = (curProg st) + 1}
    '-' -> return st {tape = chngVal st (subtract 1), curProg = (curProg st) + 1}
    '>' -> return st {curTape = (curTape st) + 1, curProg = (curProg st) + 1}
    '<' -> return st {curTape = (curTape st) - 1, curProg = (curProg st) + 1}
    '[' -> return st {curProg = (opnLoop st)}
    ']' -> return st {curProg = (clsLoop st)}
    '.' -> writeVal st
    ',' -> readVal st
    '#' -> return st {debug = True, curProg = (curProg st) + 1}

    _   -> return st {curProg = (curProg st) + 1}

    where elem = (prog st) !! (curProg st)
```

step -

```
'_' ->
    return st {
        tape = chngVal st (subtract 1),
        curProg = (curProg st) + 1}
```

step >

```
step :: BFState -> IO BFState
step st = case elem of
    '+' -> return st {tape = chngVal st (+1), curProg = (curProg st) + 1}
    '-' -> return st {tape = chngVal st (subtract 1), curProg = (curProg st) + 1}
    '>' -> return st {curTape = (curTape st) + 1, curProg = (curProg st) + 1}
    '<' -> return st {curTape = (curTape st) - 1, curProg = (curProg st) + 1}
    '[' -> return st {curProg = (opnLoop st)}
    ']' -> return st {curProg = (clsLoop st)}
    '.' -> writeVal st
    ',' -> readVal st
    '#' -> return st {debug = True, curProg = (curProg st) + 1}

    _   -> return st {curProg = (curProg st) + 1}

    where elem = (prog st) !! (curProg st)
```

step >

```
'>' ->
    return st {
        curTape = (curTape st) + 1,
        curProg = (curProg st) + 1}
```

step <

```haskell
step :: BFState -> IO BFState
step st = case elem of
    '+' -> return st {tape = chngVal st (+1), curProg = (curProg st) + 1}
    '-' -> return st {tape = chngVal st (subtract 1), curProg = (curProg st) + 1}
    '>' -> return st {curTape = (curTape st) + 1, curProg = (curProg st) + 1}
    '<' -> return st {curTape = (curTape st) - 1, curProg = (curProg st) + 1}
    '[' -> return st {curProg = (opnLoop st)}
    ']' -> return st {curProg = (clsLoop st)}
    '.' -> writeVal st
    ',' -> readVal st
    '#' -> return st {debug = True, curProg = (curProg st) + 1}

    _   -> return st {curProg = (curProg st) + 1}

    where elem = (prog st) !! (curProg st)
```

step <

```
'<' ->
    return st {
        curTape = (curTape st) - 1,
        curProg = (curProg st) + 1}
```

step [

```
step :: BFState -> IO BFState
step st = case elem of
    '+' -> return st {tape = chngVal st (+1), curProg = (curProg st) + 1}
    '-' -> return st {tape = chngVal st (subtract 1), curProg = (curProg st) + 1}
    '>' -> return st {curTape = (curTape st) + 1, curProg = (curProg st) + 1}
    '<' -> return st {curTape = (curTape st) - 1, curProg = (curProg st) + 1}
    '[' -> return st {curProg = (opnLoop st)}
    ']' -> return st {curProg = (clsLoop st)}
    '.' -> writeVal st
    ',' -> readVal st
    '#' -> return st {debug = True, curProg = (curProg st) + 1}

    _   -> return st {curProg = (curProg st) + 1}

    where elem = (prog st) !! (curProg st)
```

step [

```
'[' ->
   return st {
      curProg = opnLoop st }
```

opnLoop :: *BFState -> Int*

```
opnLoop :: BFState -> Int
opnLoop st  = case (getVal st) of
    0 -> opnLoop' (prog st) ((curProg st)+1) 0
    _ -> (curProg st) + 1

opnLoop' prog idx lvl = case (prog !! idx) of
    '[' -> opnLoop' prog (idx+1) (lvl+1)
    ']' -> if lvl == 0 then idx + 1 else opnLoop' prog (idx+1) (lvl-1)
    _   -> opnLoop' prog (idx+1) lvl
```

opnLoop :: *BFState -> Int*

```
opnLoop :: BFState -> Int
opnLoop st  = case (getVal st) of
    0 -> opnLoop' (prog st) ((curProg st)+1) 0
    _ -> (curProg st) + 1

opnLoop' prog idx lvl = case (prog !! idx) of
    '[' -> opnLoop' prog (idx+1) (lvl+1)
    ']' -> if lvl == 0 then idx + 1 else opnLoop' prog (idx+1) (lvl-1)
    _   -> opnLoop' prog (idx+1) lvl
```

step]

```
step :: BFState -> IO BFState
step st = case elem of
    '+' -> return st {tape = chngVal st (+1), curProg = (curProg st) + 1}
    '-' -> return st {tape = chngVal st (subtract 1), curProg = (curProg st) + 1}
    '>' -> return st {curTape = (curTape st) + 1, curProg = (curProg st) + 1}
    '<' -> return st {curTape = (curTape st) - 1, curProg = (curProg st) + 1}
    '[' -> return st {curProg = (opnLoop st)}
    ']' -> return st {curProg = (clsLoop st)}
    '.' -> writeVal st
    ',' -> readVal st
    '#' -> return st {debug = True, curProg = (curProg st) + 1}

    _   -> return st {curProg = (curProg st) + 1}

    where elem = (prog st) !! (curProg st)
```

step]

```
']' ->
    return st {
        curProg = clsLoop st }
```

clsLoop :: *BFState -> Int*

```haskell
clsLoop :: BFState -> Int
clsLoop st = clsLoop' (prog st) ((curProg st)-1) 0

clsLoop' prog idx lvl = case (prog !! idx) of
    ']' -> clsLoop' prog (idx-1) (lvl+1)
    '[' -> if lvl == 0 then idx else clsLoop' prog (idx-1) (lvl-1)
    _   -> clsLoop' prog (idx-1) lvl
```

clsLoop :: *BFState -> Int*

```
clsLoop :: BFState -> Int
clsLoop st = clsLoop' (prog st) ((curProg st)-1) 0

clsLoop' prog idx lvl = case (prog !! idx) of
    ']' -> clsLoop' prog (idx-1) (lvl+1)
    '[' -> if lvl == 0 then idx else clsLoop' prog (idx-1) (lvl-1)
    _   -> clsLoop' prog (idx-1) lvl
```

step .

```haskell
step :: BFState -> IO BFState
step st = case elem of
    '+' -> return st {tape = chngVal st (+1), curProg = (curProg st) + 1}
    '-' -> return st {tape = chngVal st (subtract 1), curProg = (curProg st) + 1}
    '>' -> return st {curTape = (curTape st) + 1, curProg = (curProg st) + 1}
    '<' -> return st {curTape = (curTape st) - 1, curProg = (curProg st) + 1}
    '[' -> return st {curProg = (opnLoop st)}
    ']' -> return st {curProg = (clsLoop st)}
    '.' -> writeVal st
    ',' -> readVal st
    '#' -> return st {debug = True, curProg = (curProg st) + 1}

    _   -> return st {curProg = (curProg st) + 1}

    where elem = (prog st) !! (curProg st)
```

writeVal :: *BFState -> IO BFState*

```
writeVal :: BFState -> IO BFState
writeVal st =    (putChar . chr . fromEnum . getVal) st >>
                 return st {curProg = (curProg st) + 1}
```

step ,

```
step :: BFState -> IO BFState
step st = case elem of
    '+' -> return st {tape = chngVal st (+1), curProg = (curProg st) + 1}
    '-' -> return st {tape = chngVal st (subtract 1), curProg = (curProg st) + 1}
    '>' -> return st {curTape = (curTape st) + 1, curProg = (curProg st) + 1}
    '<' -> return st {curTape = (curTape st) - 1, curProg = (curProg st) + 1}
    '[' -> return st {curProg = (opnLoop st)}
    ']' -> return st {curProg = (clsLoop st)}
    '.' -> writeVal st
    ',' -> readVal st
    '#' -> return st {debug = True, curProg = (curProg st) + 1}

    _   -> return st {curProg = (curProg st) + 1}

    where elem = (prog st) !! (curProg st)
```

readVal :: *BFState -> IO BFState*

```haskell
readVal :: BFState -> IO BFState
readVal st = do
    c <- getChar
    let val = (fromIntegral . fromEnum) c
        in return st {tape = (putVal (tape st) (curTape st) val), curProg = (curProg st) + 1}
```

fromEnum :: *a -> Int*

Convert to Int from a.

fromIntegral :: *(Integral a, Num b) => a -> b*

General coercion from integral types.

step

```
step :: BFState -> IO BFState
step st = case elem of
    '+' -> return st {tape = chngVal st (+1), curProg = (curProg st) + 1}
    '-' -> return st {tape = chngVal st (subtract 1), curProg = (curProg st) + 1}
    '>' -> return st {curTape = (curTape st) + 1, curProg = (curProg st) + 1}
    '<' -> return st {curTape = (curTape st) - 1, curProg = (curProg st) + 1}
    '[' -> return st {curProg = (opnLoop st)}
    ']' -> return st {curProg = (clsLoop st)}
    '.' -> writeVal st
    ',' -> readVal st
    '#' -> return st {debug = True, curProg = (curProg st) + 1}

    _   -> return st {curProg = (curProg st) + 1}

    where elem = (prog st) !! (curProg st)
```

step

```
'#' ->
    return st {
        debug = True,
        curProg = (curProg st) + 1}
```

Is there any **library** that will help in **working** with **lists**?

Is there any **library** that will help in **working** with **lists**?

Yes! For example: **ListZipper**.

ListZipper

```
cabal update
cabal install cabal-install
```

```
cabal install ListZipper
```

ListZipper

- **fromList** :: *[a] -> Zipper a*
- **toList** :: *Zipper a -> [a]*
- **endp** :: *Zipper a -> Bool*
- **cursor** :: *Zipper a -> a*
- **right** :: *Zipper a -> Zipper a*
- **left** :: *Zipper a -> Zipper a*
- *…*

http://hackage.haskell.org/package/ListZipper-1.1.0.0/docs/Data-List-Zipper.html

Links

- *http://sabbatical-year.blogspot.com*
- *http://bonsaicode.wordpress.com/2010/05/14/programming-praxis-–-brainfuck-interpreter/*
- *http://lpaste.net/71690*
- *https://github.com/niklasb/haskell-brainfuck/blob/master/Brainfuck.hs*

Code

All examples of this presentation
(and even more) are available at

github.com/**slon1024/interpreter_brainfuck**