Contents

REPL Code

- A common programming style is the so-called REPL idiom (Read-Eval/Execute-Print-Loop)
- We can do this in Haskell using `putStr` and `putStrLn` for output and `getLine` for input

A simple dumb program that shouts back at you

```
shout
  = do putStr "Say something: "
       utterance <- getLine
       putStrLn("You said: "++map toUpper utterance)
       if null utterance then return () else shout
```

Slightly less dumb - it checks fro no utterance first

```
shout2
  = do putStr "Say something: "
       said <- getLine
```

```
    if null said
    then putStrLn "I CAN't HEAR YOU! I'M OFF!!"
    else putStrLn("You said: "++map toUpper utterance)
      shout2
```

R-Eval-PL Template

There is a common pattern to most RE(eval)PL programs

- Issue a prompt
- Get user input
- Evaluate user input
- Print result
- Look at result and decide either to continue or exit

We can capture this as the following code

```
revpl prompt eval print done
  = do putStr prompt
       userinp <- getLine
       let result = eval userinp
       print result
       if done result
         then return ()
         eval revpl prompt eval print done
```

We can now write

```
shout1
  = revpl "Say something: "
      (map toUpper) print1 null

print1 res = putStrLn ("You said: "++res)
```

R-Execute-PL Template

There is another common pattern to most RE(execute)PL programms

- Issue a prompt
- Get user input
- Parse input and perform requested action
- Look at outcome and decide either to continue or exit

We can capture this as the following code

```
rexpl prompt execute
  = do putStr prompt
       usercmd <- getLine
       done <- execute usercmd
       if done then return ()
         else rexpl prompt execute
```

We can now write

```
shout3
  = rexpl "Say something: " doshout3

doshout3 utt
  = if null ut
    then do putStrLn "I CAN'T HEAR YOU! I'M OFF!!"
            return True
    else do putStrLn ("You said: "++map toUpper utt)
            return False
```

These examples show how easy it is to "grow our own" control structures

Most REPLs Need State

- Consider implementing a single "Totting-up" program:
  - User enters numbers one at a time
  - There are added up, and a running total is displayed
  - An empty line terminates the process
- This cannot be implemented using `revpl` or `rexpl`
- Given that there is a state being updated (here the running total), it makes sense to view this as being a R-Execute-PL rather than Eval

Totting-Up REPL

- We need to initialise the running total

```
totup = detotting 0.0
```

- We then implement the REPL loop, passing the total (state) in as an argument

```haskell
dototting tot
  = do putStr ("["++show tot++"]\n:- ")
       numtxt <- getLine
       if null numtxt
         then putStrLn("\nTotal = "++show tot)
         else dototting (tot+read numtxt) -- state update!
```

- We use `read :: Read a => String -> a` here, which has numeric instances
- Again, we can build a HOF that abstracts this pattern

State REPL

- State REPL building

```haskell
srepl prompt done exist execute satte
  = do prompt state
       cmd <- getLine
       if done cmd
         then exit state
         else
           let state' = execute cmd state
           in srepl prompt done exist execute state'
```

- Haskell derives the following type, where `t` denotes the state type

```haskell
srepl :: (t -> IO a)          -- prompt
      -> (String -> Bool)     -- done
      -> (t -> IO b)          -- exit
      -> (String -> t -> t)   -- execute
      -> t                    -- state
      -> IO b
```

We can focus on the four key processing steps: prompting,

```haskell
totpr tot = putStr("["++show tot++"]\n:- ")
```

checking if done,

```haskell
null
```

exiting cleanly

```
totxit tot = putStrLn ("\nTotal = "++show tot)
```

and computing the next state

```
totexe cmd tot = tot+read cmd
```

We then invoke the REPL-generate with these and the starting state:

```
totup2 = srepl totpr null totxit totexe 0.0
```

REPL with putStr and getLine

- Building REPL code using `getLine :: IO String` is very convenient
- Unfortunately, keys such as delete or backspace are not handled properly (on Unix-based systems at least - it seems to work dine on Windows!)
- There are modules that help
  - Best is probably: `System.Console.Haskeline`. Careful: uses monad transformers
  - An alternative: `System.Console.Readline`. Interfaces to GNU readline, but has restricted portability

Real World Programming Requires I/O

- I/O has been problematic for (pure) functional languages
- In order to understand why I/O in Haskell is the way it is
  - We need to know what it meant by "pure"
  - We need to know how Haskell is implemented (a little)
  - We need to understand the key problem with I/O
- Summing up, we first need to understand how functional languages work

Functional Languages as Rewirte Systems

- We can view function/value definitions as rules describing how to transform (rewrite) an expression
  - If we have a definition like `myfun this_pattern = result_expression`
  - We then invoke the function in a call matching the above pattern: `myfun some_argument`

  - We expect to see the call replaced by the result, with appropriate substitutions: `result_expression [ some_argument / this_pattern]`
  - The notation `e[a|x]` is standard mathematical notation for "expression e where expression a is sibstituted for all (free) occurrences of `x`"

- This is formalised in the so-calld "Lambda-calsulus"

Definitions in Haskell

- One way to define a function called `myfun` is as a series of *declarations* in the form:

```
myfun pat11 pat12 ... pat1K = exp1
myfun pat21 pat22 ... pat2K = exp2
...
myfun patN1 patN1 ... patNK = expN
```

where each line has the same number of patterns (`pat`)

- Each pattern can be:
  - A constant value (numer, character, string, nullary data-constructor)
  - A variable (no variable can occur more than once in a pattern)
  - An expression built from patterns, and n-ary data constructs applied to n patterns

Pattern Examples

- Expect three arbitrary arguments, `myfun x y z`
- Illegal - if we want first two arguments to be the same then we need to use a conditional, `myfun x x z`
- First argument must be zero, second is arbitrary, and this is a non-empty like, `myfun 0 y (z:zs)`
- First argument must be zero, second is arbitrary, and this is a non-empty list, whose first element is character 'c', `myfun 0 y ('c':zs)`
- First argument must be zero, second is arbitrary, and this is a non-empty list, whose tail is a singleton, `myfun 0 y (z:[z'])`

Pattern Matching

We describe how pattern matching works and what it does by example:

- A constant pattern matches the specified value only
  - pattern 3 only matches the value 3
- A variable mtaches anything, and we get a binding of that variable to the value matched
  - Pattern `x` matches any value `v` and the result is that variable `x` is bound to value `v`
- A constructor pattern matches something of the same "shape" as well as matching the corresponding sub-components
  - Pattern `x:xs` matches a non-empty list, and binds `x` to the head of the value and `xs` to the tail value of the list

Summary:

- Pattern matching can *succeed* or *fail*
- If successful, a pattern match returns a (possibly empty) *binding*
- A binding is a mappin from (pattern) variables to values

Patterns	Values	Outcome
x (y:ys) 3	99 [] 3	Fail
x (y:ys) 3	99 [1, 2, 3] 3	x -> 99, y -> 1, ys -> [2, 3]
x (1:ys) 3	99 [1, 2, 3] 3	x -> 99, ys -> [2, 3]

Haskell Executing (Rewriting version)

- Haskell execution proceeds by reducing function applications until this is no longer possible
- An expression with no reducible applications is said to be in *normal form*
- Generally the form (a value) is taken as the result/meaning of the program
- Some (hard) theorem show that normal forms (if they exist) are unique