

Haskell Review

naming conventions

- Use PascalCase for data types, classes, : constructors
- camelCase for function names : variable names

Arrays

- ++ operators joins 2 lists $[10, 20] ++ [30, 40]$ while
- joins a value to the top of the list $10 : [20, 30] \rightarrow [10, 20, 30]$

Operators

- Comparison: $\neq$ (not equals) • \ lambda
- && (And) || (or) • a mod b infix
- >>, >=> (If you need data coming out of the left, use `getLine>>=`)

Let Bindings : unpacking

- let statement can be used to unpack list or tuples
return = (10.2, 75)
let (left, right) = return values

console output (Slide 26)

- putStrLn - puts a string par. to screen w /n at end
- print - converts non-string into string : prints
- printf - needs to be imported from `import Text.Printf (printf)`
putstr

console input

getLine

- (composition)
 $a(b(x))$ - composes 2 functions

23/03/

Process Console data

lines | break string into list of lines | "a\nb\nc" to ["a", "b", "c"]
words | break string into list of words | "hello world" to ["hello", "world"]
unlines |
unwords | combine words with space | ["hello", "world"] to "hello world"
read | parse string to value | "123" :: Int to 123
show | converts an Int/Double/List into a String
read "3.14" :: Double ^{or Var} || [Int]

Map

Converting : Casting (i.e. fromIntegral, floor, truncate Slide 30)
Example of Reading Lines from input : summing lines from input (Slide 36)

lists

map - applies a transformation for each element

fold - a list is folded to a single value

scanl

zip - zip [1,2,3,4] ["a", "b", "c", "d"] = [(1, "a"), (2, "b"), ...] (Slide 50)

concat - flattens a list of lists: [["10", "20"], ["30", "40"]] = ["10", "20", "30", "40"]

sum

take/drop - takes first 3 characters (Slide 57) / drops first ch.

head/tail

elem

filter

sort

scan

reverse

Arguments : Case statement

- use import System.Environment (getArgs, getProgName)

args <- getArgs
print args

OR main = getArgs >>= print

main = do!

src to lead file

functions

- order matters in case statements

args <- getArgs

case args of

[x] -> readFile x >>= print sumFile
- -> getProgName >>= print 115

print (sumFile)

gets program name

where : maybe (slide 41)

- define functions that are nested inside other functions to create "helpers"

main = print \$ myData where
myData = 2133_44

Prints: 21344

- maybe has 2 values

maybe = Nothing (x) (a x b or a, b)
Just (x) (there's data)

factors - #s that divide into that $\neq$ $\begin{cases} 4 \div 1 = 4 \\ 4 \div 2 = 2 \\ 4 \div 3 = 1.33 \end{cases}$ $\times$
 $4 \div 4 = 1$

1 + 5

1 %

map (function (parameter))

code snippets

get a line
: call
function

```
var line <- getLine
print $ processLine line
```

↑ function ↑ param

filter
using
lambda

```
main = print $ filter (>= 0) $ map (\x -> x+2) [1,2,3,4,5]
```

filter only expects funct. for 1st arg takes in 1 arg returns $x+2$

Map applies it to every element

let
unpacking

```
main :: IO ()
main = print $ callMyFunc (10, "string!!")
```

```
callMyFunc :: (Int, String) -> Int
callMyFunc x = let (left, right) = x in
  left
```

Pattern
Matching

```
MyDrop :: Int -> String -> String
MyDrop 0 s = s
MyDrop i (-:sx) = myDrop (i-1) sx
```

our int } if $i=0$, return s

Pattern matching (recursively pull off head until $i=0$)

Pattern matching
using guards

```
MyDrop i s@( _ : sx) =
  | i <= 0 = s
  | otherwise = MyDrop (i-1) sx
```

it pops off head : puts it inside (empty) : everything else goes in sx
takes s the entire 'list' of sx

if $i \leq 0$, then it's true (return list s), otherwise recurse

list
comprehension

```
[<value> | <extration> <- <generator/function>, <optional con>]
```

```
print [v | v <- init [1..10]] --> [1,2,3,4,5,6,7,8,9]
print [val * val | val <- [1..10], val > 5] --> [36,49,64,81,100]
```


Rust Review

PascalCase for data types, structures, : enums
snake_case for function names : variable names

Converting data console

- Since all console input is String, you must use
`Var.trim().parse<T>().expect("")`

Command line args

header, use `std::env::args`

```
let a: Vec<String> = args().collect() /n println!("{}", a)
```

Match statements

Ex:1 `match val {`
 `10 => println!("{}",`
 `x => println!("{}", "Not 10!");`
 `}`

Ex:2

Vectors

`let mut grade = Vec::new()`

header, `std::vec::Vec`, they can come with macros!

`Vec![]` - empty vector

`Vec![0u8; 32]` - vector with 32 unsi. all set to 0

Slices

`myVec[start..end]` creates slice of values from [start, end)

`myVec[..end]` vals between [0, end)

`myVec[..]` creates a slice between [0, len())

Slide 212 (functional rust)

Slide 213 (Anonymous lambda)

Signed primitives

`i8` ranges -128 to 127

`i16` ranges -32,768 to 32,767

`i32` ranges -2³¹ to 2³¹

`i64` ranges -9 quint to 9 quint

`isize` (architecture-dependent)

(i32 on 32 bit mach, i64 on 64 bit)

`u8` ranges 0 to 255

`u16` ranges 0 to 65,535

`u32` ranges 0 to 4 mil

`u64` ranges 0 to 18 quint

unsigned can only store positive #s

functions

25, 21

`trim()` - removes whitespaces

`parse()` - parses string to another data type

`push()` - Appends an element to the back of a collect. `Vec[1,2] . push(3) → [1,2,3]`

`iter()` - returns an iterator over the slice `[1,2,3]`

`read_to_string()` - reads all bytes until EOF

`split()` - ex: `split(',')` `"Mary is 10"` to `"[Mary, is, 10]"`

`sort()` - sorts a list `[1,2,3,0] → [-2,0,1,3]`

`File::create()` - opens a file in write mode only

functional program

`split_whitespace()` `"Hello World _ _ _"` → `["Hello", "World", "I"]`

Extra info

Reading inputs: `let it = String::new || io::stdin().read_line(&mut it).expect("failed to read")`

Parse input: `io::stdin().trim().parse().expect("")`

collecting args: `let a: Vec<String> = args().collect()`

headers:

- use `std::fs::File`
- use `std::io::Write` `Read`
- use `std::env::args` (collect Arg info)

- Use `std::io::stdin` or

tips

takes 2 param return `i32` No semicol returns
`fn my_func(a: i32, b: i32) → i32 { a + b }`

data vector push // break : task

```
6  Run | Debug
7  fn main() {
8      let a: Vec<String> = args().collect();
9      if a.len() != 2 {
10         println!("Usage: {} <filename>", &a[0]);
11         return;
12     }
13     let mut file_name: File = File::open(path: &a[1]).expect(msg: "Unable to open file");
14     let mut buffer: String = String::new();
15     file_name.read_to_string(buf: &mut buffer).expect(msg: "Unable to read file");
16
17     let tasks: Vec<&str> = vec!["coding", "design", "planning", "testing"];
18     let breaks: Vec<&str> = vec!["snack", "break", "nap"];
19     let mut task_data: Vec<f32> = Vec::new();
20     let mut break_data: Vec<f32> = Vec::new();
21
22     for line: &str in buffer.lines() {
23         let parts: Vec<&str> = line.split_whitespace().collect();
24         if parts.len() != 2 {
25             continue;
26         }
27         let task_name: &str = parts[0];
28         let hours: f32 = match parts[1].parse() {
29             Ok(h: f32) => h,
30             Err(_) => continue,
31         };
32
33         if tasks.contains(&task_name) {
34             task_data.push(hours);
35         } else if breaks.contains(&task_name) {
36             break_data.push(hours);
37         }
38     };
39     let task_sum: f32 = task_data.iter().sum();
40     let break_sum: f32 = break_data.iter().sum();
41     if task_sum > break_sum {
42         println!("Very productive \n");
43     } else {
44         println!("Try to focus more")
45     }
46 }
47
48 fn main
```

create vectors that contain &str
f32

ignore parts : try parsing the numb.

push data into the newly created
vectors

ON/off in Rust

```
use std::io::Read;
use std::fs::File;
use std::env::args;

Run | Debug
fn main() {
    let a: Vec<String> = args().collect();
    if a.len() < 2 {
        println!("Usage: {} <filename>", &a[0]);
        return;
    }
    let mut filename: File = File::open(path: &a[1]).expect(msg: "Unable to open file");
    let mut buffer: String = String::new();
    filename.read_to_string(buf: &mut buffer).expect(msg: "Unable to read file");

    let mut counter: i32 = 2;
    let mut data: Vec<i32> = Vec::new();

    for line: &str in buffer.lines() { // + This should be buffer.split('\n') or similar
        let line: &str = line.trim();

        if line.is_empty() {
            continue;
        }
        // Check for On/Off commands first
        if line == "On" {
            counter += 1;
            continue;
        }
        if line == "Off" {
            counter -= 1;
            continue;
        }
        // If not a command, try to parse as number
        match line.parse::<i32>() {
            Ok(num: i32) => {
                if counter == 2 {
                    data.push(num);
                }
            },
            Err(_) => continue,
        }
    }

    let num_sum: i32 = data.iter().sum();
    println!("Sum: {}", num_sum);
}
fn main
```