ECE 421
Assignment 1

Arun Woosaree
XXXXXXX

January 27, 2020

1

Functional programming is a declarative paradigm. Computation is treated as the evaluation of mathematical functions. Expressions and declarations are used as opposed to statements. Given a set of inputs, the output will always be the same. Data is immutable, and as a result, functions have no 'side-effects'.

2

The function is summing the numbers from 0 to 100. Right at the beginning, we see that the program produces a side-effect, which is using IO to output some data. This contradicts the idea of a 'purely' functional language. However, if a program never has side effects, it would be pretty boring. Also, we see that the program is written in an imparative style, even though Haskell is generally considered a purely functional language where code is usually written in a declarative style. That is, it looks like a C programmer wrote some code here. The looping part of the program is recursive, and the value i is updated using a monad. The program still is functional, in the sense that the function does the same thing every time, and the dirty work of updating variables is achieved with the monad. Also, it does not mutate anything outside of the function.

3

Immutability is nice, because it makes things predictable. For example, in concurrent applications, shared data is a lot more predictable, since nothing can mutate the data. Immutability also helps with producing more readable code, because values don't change. Also, when you call a function, you know that it has not mutated anything. This is what makes things more predictable.

4

a)

```
int x = y + z;
```

b)

```
let x = y + z
```

c)

All languages must allow for mutability at some level, otherwise they would never produce useful output. When code is compiled, it generates mutable machine code since that's what computers (currently) understand.

5

a)

- sqrtx takes an input, and returns the squared value of the input
- imparativefun returns the sum of squares of its input, which is a list of numbers
- functionalfun does the same thing, but does so in a declarative manner instead of an imparative way

b)

- there seems to be no difference in reliability between the two implementations
- there may be a slight difference in efficiency between the two implementations, but it is probably negligible
- I would argue that the functionalfun is more maintainable as it's easier to understand what the function is doing. Even though the functions do the same thing, functionalfun has less "clutter" around it, which is what makes it easier to understand.
- there will be no difference in portability between the two implementations.

c)

```
let fourthpower x = x |> sqrt |> sqrt
```

6

- changing the file system is a side effect, so it is not a pure function
- inserting a record cannot be a pure function, because it results in the database mutating
- making an http call cannot be a pure function because you cannot guarantee the same thing will be returned on every function call. The status message can change, the server could go down, etc.
- printing to the screen requires a side-effect, namely I/O, so it is not a pure function
- querying the DOM can be a pure function, provided the DOM is an input parameter to the function, and the output of the function depends only on this input, and if the function does not mutate anything
- the system state will be different (for example, the time will be different) on every call to the function, so it will return something different each time, so it cannot be a pure function
- pure functions return the same output every time for a given set of inputs. Any randomness violates this, so Math.random() is not a pure function

7

```
fn functionalfun (x : isize) -> isize {
    (1..=x).map(|i| i*i + 2).sum()
}
```

8

```
fn volume_of_sphere(radius: f64) -> f64{
    4_f64/3_f64 * std::f64::consts::PI * radius * radius * radius
}
```

9

The Scheme function returns a sum of the number of leaf nodes in a nested list that are not #f. That is, no matter how nested the elements in the lists are, it returns a count of all the elements in a list, and its nested lists not including #f.

10

a)

The GHC compiler produces no error when presented this code. This is because Haskell supports things like partial function applications and incomplete pattern matching. The program will compile with no errors by default, but the flag `-fwarn-incomplete-patterns` can be used to show a warning at compile time.

b)

The Rust compiler will scream at you for this. It will catch that the pattern Blue defined by Colour is not covered in the pattern matching statement and tell you to fix it. Till then, it will refuse to compile the program.