ECE 522 Assignment 1

Zhaoyi Wang 1689747

Question 1

I think of functional programming like writing code as a formula, or a series of nested function calls. It is different from imperative programming where the code is run line by line from top to bottom. Functional programming is like putting values into a formula, where the only obvious things are the inputs and outputs. In addition, each function is independent and they do not change the values of external variables, or rather, it does not need to take into account thread deadlocks.

Question 2

The purpose of this code is to calculate the sum of 1 to 100. First it defines a *totalRef*, then it goes through a loop to calculate the sum from 1 to 100, and finally it passes this result to *total* and prints it out. The Haskell language treats each operation as a function, they are independent of external code, and no changes are made to external variables during execution.

Question 3

First, immutable variables prevent variables from being accidentally modified, reducing unnecessary debug time due to accidental modifications. In addition, immutable variables make the code logic clearer. For example, if a variable is modified multiple times, it would take a lot of time for the reader to look up how the variable was modified in order to understand the code logic.

Question 4

For Question a):

```c
#include <stdio.h>
int main()
{
    int x;
    int y = 1, z = 2;

    int R1, R2, R3;

    R1 = y;
    R2 = z;
    R3 = R1 + R2;

    x = R3;
    printf("x is %d",x);
}
```

For Question b):

```
main = do
    let sum x y = x + y
    return sum
```

For Question c):

Regardless of what language this code is translated into, its purpose does not change, it is to assign values and sum. It is possible that different languages write the code differently to describe this function.

Question 5

For Question a):

For the first single line, it aims to find the square of x.

For imperative programming, first, a variable is created to hold the final result. Then, a loop was created to square each number in the loop and add it to the previous result (to find the sum). The final result is returned as total.

For functional programming, it uses mapping, which maps all values to the function and then calculates the sum.

For Question b):

Reliability
Functional programming is more reliable than imperative programming. This is because functional programming does not involve changing external variables, which greatly reduces errors and troubleshooting time due to frequent variable changes.

Efficiency
The code style of functional programming is more concise and easier to understand.

Maintainability
From the above two features we can conclude that maintainability is also better with functional programming.

Portability
Functional programming is more portable because it is more independent and more focused on the problem itself.

For Question c):

```
let imperativefun list =
    let mutable total = 0
    for i in list do
        let x = sqrtx i
        let y = sqrtx x
        total <- total + y
    total
```

Question 6

- All of them are not

Question 7

```rust
use std::io;

fn main() {
    let user_input = handle_input();
    println!("{}", sum(user_input))
}

fn square_add(x: i32) -> i32 {
    let result = x * x + 2;
    result
}

fn sum(num: i32) -> i32 {
    let mut i = 1;
    let mut sum_part = 0;
    let mut sum_all = 0;
    while i < (num + 1) {
        sum_part = square_add(i);
        sum_all = sum_all + sum_part;
        i = i + 1;
    }
    sum_all
}

fn handle_input() -> i32 {
    println!("Input a number");
    let mut number = String::new();
    io::stdin().read_line(&mut number).expect("Can't recognize");
    let num: i32 = match number.trim().parse() {
        Ok(num) => num,
        Err(_) => panic!(),
    };
    num
}
```

Question 8

```rust
fn main() {
    println!("{}",calculate_sphere(3.0));
}

fn calculate_sphere(radius: f64) -> f64{
    let pi = 3.14;
    let result = (4.0/3.0)*pi*radius*radius*radius;
    result
}
```

Question 9

First, we determine that whether the *x* is a list. If it is empty, the function returns a zero. If it is not a list, we have two choices. If the *car* is *#f*, we recursive on the *cdr* and return that. Or, we will count the *car* as 1 and add that to the result of the recursive on the *cdr*. Finally, we do a recursive on both *car* and *cdr*, and the add them together.

Question 10

For Question a):

The program does not print out any results and it shows a error `main.hs:(3,5)-(5,24): Non-exhaustive patterns in case`.

For Question b):

It shows an error: `non-exhaustive patterns: &Blue not covered`