Question 1

In computer science, functional programming is a declarative programming paradigm in which programs are constructed by applying and creating functions. Purely functional programming treats all functions as deterministic mathematical functions. When a function is called with some input, it will always return the same result and is not affected by any mutable state or side effects (memory or I/O). Usually in functional programming, programs are written in a declarative and composable style.

Question 2

The given Haskell program sums up the numbers from 0 to 100. This is not a pure function since the beginning of the program produces a side-affect by using I/O. Although Haskell is a pure functional programming language, the program seems to be written in an imperative style instead of a declarative style.

Question 3

Immutable object are useful because they are inherently thread-safe and good for sharing information between threads. Programs that use immutable objects are easier to understand since you don't need to think about how objects may evolve over time.

Question 4

a. `int x = y + z;`

b. `let x = y + z`

c. All language allow mutability to some degree – even functional programming languages such as Haskell.

Question 5

a.
- `imperativefun` sums the square of each value in the list and returns the sum
- `functionalfun` also sums the square of each value in the list and returns the sum but is written in a declarative style.

b. **Reliability:** there is no difference in reliability between the two
Efficiency: the efficiency between functional and imperative programs are the same.
Maintainability: `functionalfun` seems shorter and easier to understand which can help it be more maintainable.
Portability: There is no difference in portability.

c. `let fourthpower x = sqrtx x * sqrtx x;`

Question 6

- Changing the file system is not a pure function. It has a side effect.
- Inserting a record into a database is not a pure function.
- Making an http call is not a pure function. A given input may not return the same output.
- Mutations is not a pure function
- Printing to the screen / logging is not a pure function. It uses I/O, a side effect.
- Obtaining user input is not a pure function. It also uses I/O, a side affect.
- Querying the DOM could be a pure function if the output only depends on the DOM and does not mutate anything
- Accessing system state is not a pure function. The system state can change and return something different for a given input.
- Math.random() is not a pure function. It returns something random every time, so the output is different for a given input.

Question 7

```
fn another_function(x: i32) -> i32 {
  let mut sum = 0;
  for i in 1..x+1 {
      sum = sum + (i*i + 2)
  }
  return sum;
}
```

Question 8

```
fn volumeofsphere(r: f32) -> f32 {
  (4.0 / 3.0) * (3.14) * r * r * r
}
```

Question 9

The Scheme function returns a count of all the elements in a list and its nested lists not including #f.

Question 10

a. There doesn't seem to be any compiler errors when we run the code. Partial application in Haskell involves passing less than the full number of arguments to a function that takes multiple arguments. This does not throw an error.

b. Running the given Rust code causes the compiler to throw an error. The errors says `error[E0004]: non-exhaustive patterns: &Blue not covered`. It is unhappy that the pattern `Blue` defined by `Color` is not covered in the pattern matching statement.