2nd Problem Set Justin Cesarini

Task 1

In order to run a program, the Runtime Stack must be called. The run time stack stores and handles nonstatic variables. The Runtime Stack also stores functions in a sequential way starting with the main. The Runtime Stack is often compared to a stack of plates. Each plate has its own food and it cannot go from one plate to another. It is placed in order from top to bottom.

Task 2

Allocation of memory is when memory is collected by the operating system. Deallocation of memory is when the operating system frees up memory in order to allow a program to run faster or perhaps create room for more memory to be stored

Garbage collection is when a programming language spots memory that is currently not being used and therefore gets rid of it. A perfect example is a variable that is initialized but never used. A language like Lisp would see that unused variable and clear it from memory in order to allow the program to run faster and free up memory.

Task 3

1. Rust is better than Haskell because Rust gives more control of the allocation of memory in a persons program.
2. They view the compiler as a key element in testing the correctness of our program and both embrace useful syntactic features like sum types, typeclasses, polymorphism, and type inference.

3. Basic primitives in Rust include Various sizes of integers, signed and unsigned (i32, u8, etc.)

   Floating point types f32 and f64.

   Booleans (bool)

   Characters (char). Note these can represent unicode scalar values (i.e. beyond ASCII)

4. While variables are statically typed, it is typically unnecessary to state the type of the variable. This is because Rust has type inference, like Haskell.

5.
```
addExpression :: Int -> Int -> Int
addExpression x y = x + y

addWithStatements ::Int -> Int -> IO Int
addWithStatements x y = do
  putStrLn "Adding: "
  print x
  print y
  return $ x + y
```

Rust has both these concepts. But it's a little more common to mix in statements with your expressions in Rust. Statements do not return values. They end in semicolons. Assigning variables with let and printing are expressions.

6. Like Haskell, Rust has simple compound types like tuples and arrays (vs. lists for Haskell).

7. Statements do not return values. They end in semicolons. Assigning variables with let and printing are expressions. Expressions return values. Function calls are expressions. Block statements enclosed in braces are expressions.

8. Another concept relating to collections is the idea of a slice. This allows us to look at a contiguous portion of an array. Slices use the & operator though.

9. Arrays and tuples composed of primitive types are themselves primitive! This makes sense, because they have a fixed size.

10. Another concept relating to collections is the idea of a slice. This allows us to look at a contiguous portion of an array. Slices use the & operator though.

Task 4

1. When we declare a variable within a block, we cannot access it after the block ends. (In a language like Python, this is actually not the case)

2.Another important thing to understand about primitive types is that we can copy them. Since they have a fixed size, and live on the stack, copying should be inexpensive

3. We've dealt with strings a little by using string literals. But string literals don't give us a complete string type. They have a fixed size. So even if we declare them as mutable, we can't do certain operations like append another string. This would change how much memory they use

4.What's cool is that once our string does go out of scope, Rust handles cleaning up the heap memory for it.

5. In Rust, here's what would happen with the above code. Using `let s2 = s1` will do a shallow copy. So `s2` will point to the same heap memory. But at the same time, it will **invalidate** the `s1` variable. Thus when we try to push values to `s1`, we'll be using an invalid reference. This causes the compiler error.

6. In general, passing variables to a function gives up ownership.

7. Like in C++, we can pass a variable by **reference**.

8. You can only have a single mutable reference to a variable at a time. Otherwise your code won't compile. This helps prevent a large category of bugs.

9. Once the `x` value gets assigned its value, we can't assign another! We can change this behavior though by specifying the `mut` (mutable) keyword.

10. If you want a *mutable* reference, you can do this as well. The original variable must be mutable, and then you specify `mut` in the type signature.

Task 5

1. Haskell has one primary way to declare a new data type: the `data` keyword.

2. The name struct is a throwback to C and C++. But to start out we can actually think of it as a distinguished product type in Haskell. That is, a type with one constructor and many named fields.

3. Rust also has the notion of a "tuple struct". These are like structs except they do not name their fields.

4. The last main way we can create a data type is with an "enum".

5. Rust also has the idea of a "unit struct". This is a type that has no data attached to it.

6. In Haskell, we typically use this term to refer to a type that has many constructors with no arguments. But in Rust, an enum is the general term for a type with many constructors, no matter how much data each has.

7. Pattern matching isn't quite as easy as in Haskell. We don't make multiple function definitions with different patterns. Instead, Rust uses the match operator to allow us to sort through these. Each match must be exhaustive, though you can use _ as a wildcard, as in Haskell.

8. But unlike Haskell, Rust allows us to attach implementations to structs and enums. These definitions can contain instance methods and other functions. They act like class definitions from C++ or Python.

9. As in Haskell, we can also use generic parameters for our types.

10. For the final topic of this article, we'll discuss traits. These are like typeclasses in Haskell, or interfaces in other languages. They allow us to define a set of functions.

Task 6

Paragraph 1: Rust does not contain garbage collection. Rust actually has its own way of organizing and dealing with memory issues or problems. It was meant to be in between languages that have garbage collection and ones that do not.

Paragraph 2: To avoid dangerous, security-relevant errors involving references, Rust enforces a programming discipline involving ownership, borrowing, and lifetimes. Also, most type-safe languages use garbage collection to prevent the possibility of using a pointer after its memory has been freed. Rust prevents this without garbage collection by enforcing a strict ownership-based programming

Paragraph 3: Since the memory guarantees of Rust can cause it to be conservative and restrictive, Rust provides escape hatches that permit developers to deactivate some, but not all, of the borrow checker and other Rust safety checks. Unsafe functions and methods are not safe in all cases or for all possible inputs.