CS4402 Discussion assignment 7

Question 1: Discuss the concept of lazy evaluation and why this is a powerful advantage of functional programs.

You are expected to make a minimum of 3 responses to your fellow student’s posts.

Question 2: Describe and Discuss the concepts of Higher order functions and currying in a functional programming language such as Haskell (or Standard ML) and describe why these concepts are important.

You are expected to make a minimum of 3 responses to your fellow student’s posts.

Write a brief discussion about the concept of lazy evaluation and the reason for this being the powerful feature in functional programming. Use Haskell as example

Write a brief discussion about the concept of higher order functions and currying in functional programming like Haskell and describe their importance.

And provide a code example

In a nutshell, lazy evaluation means that expressions are not evaluated until their results are actually needed. This contrasts with eager evaluation, where expressions are evaluated immediately.

Below are the advantages of lazy evaluation(*Lazy Evaluation - HaskellWiki*, n.d.)

**Efficiency**: Lazy evaluation can save resources. In Haskell, you can create infinite lists, and because the values are only computed when needed, you can work with these lists without worrying about running out of memory.

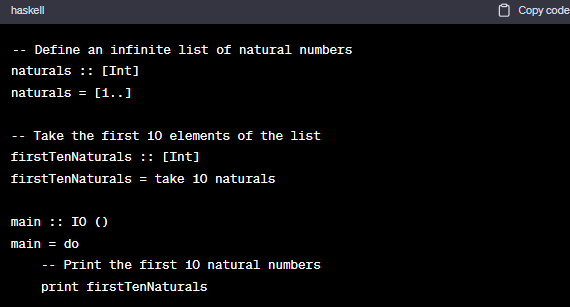
**Modularity**: Lazy evaluation enhances modularity. Functions can be designed to produce infinite data structures, and other functions can consume only as much as they need. This separation of concerns can lead to more modular and reusable code.

**Expressiveness**: It enables a more expressive style of programming. You can define complex operations in a more declarative way without worrying about the order of evaluation. This can make your code easier to read and reason about.

**Infinite Data Structures**: As mentioned, Haskell allows you to create infinite data structures. For instance, you can define an infinite list of Fibonacci numbers without worrying about the list's size, which is not possible in languages with eager evaluation.

**Non-Strict Semantics**: Haskell's non-strict semantics, which includes lazy evaluation, encourage more abstract and concise programming. You can work with data at a higher level of abstraction and delay making decisions about what to compute until necessary.

However, it's important to note that lazy evaluation can introduce challenges, like space leaks if not used carefully.

**Code** example as below 

We define naturals, which is an infinite list starting from 1 ([1, 2, 3, ...]). However, Haskell doesn't compute this entire list immediately; it generates elements as they are needed.

We then use the take function to extract the first 10 elements from the infinite list, creating firstTenNaturals. At this point, only the first 10 natural numbers are computed.

In the main function, we print the firstTenNaturals list. The system will not throw any error for the infinite list.

**Higher-Order Functions**: In functional programming, functions are first-class citizens. This means you can treat functions just like any other data type, passing them as arguments to other functions or returning them as results. Functions that either take one or more functions as arguments or return functions as results are called higher-order functions(*Chapters - Learn You a Haskell for Great Good!*, n.d.).

The importance of higher-order functions (*Chapters - Learn You a Haskell for Great Good!*, n.d.):

**Abstraction**: Higher-order functions allow for a high level of abstraction. They enable you to express general concepts that can be applied to a wide range of specific situations. This promotes code reusability and conciseness.

**Modularity**: They promote modularity by breaking down complex operations into smaller, composable functions. This makes your code easier to read, test, and maintain.

**Functional Composition**: Higher-order functions facilitate functional composition. You can chain functions together to create more complex behaviors, which is a fundamental concept in functional programming.

**Currying**: Currying is a technique in functional programming where a function that takes multiple arguments is transformed into a series of functions, each taking a single argument. This allows for partial application, where you can call a function with fewer arguments than it expects, creating a new function that awaits the remaining arguments.

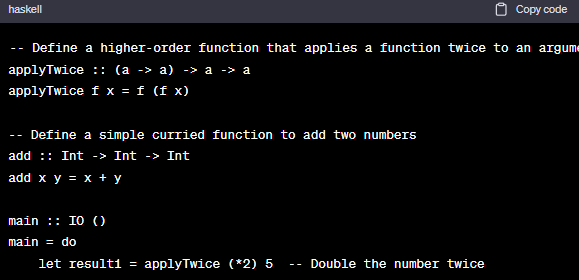
The importance of currying(*Chapters - Learn You a Haskell for Great Good!*, n.d.):

**Partial Application**: Currying enables partial function application. You can create specialized versions of a function by fixing some of its arguments, making your code more flexible and expressive.

**Currying and Higher-Order Functions**: Curried functions naturally work well with higher-order functions. They make it easier to pass and return functions, as you can partially apply them as needed.

Here's an example in Haskell to illustrate both higher-order functions and currying:

haskellCopy code example



**Explanation of code**

In this code, applyTwice is a higher-order function that takes a function f and applies it twice to an argument x. add is a curried function that takes two arguments.

We demonstrate partial application by partially applying the add function to create a specialized function addFive, which adds 5 to any number. This showcases the power of currying and how it works seamlessly with higher-order function

**Reference**

*Chapters - Learn You a Haskell for Great Good!* (n.d.). Retrieved October 23, 2023, from http://learnyouahaskell.com/chapters

*Lazy evaluation - HaskellWiki*. (n.d.). Retrieved October 23, 2023, from https://wiki.haskell.org/Lazy\_evaluation