Lecture Summary: Lists and Recursion

Generated by LectureMate

February 22, 2025

1 Introduction

The lecture began with an introduction to the peer support service called Impulse, run by undergraduate students. This service offers assistance with information and extracurricular skills, operating three days a week in Appleton Tower. Students are encouraged to participate in skill sessions, including topics like Git and LaTeX, which are essential for their university careers.

2 Overview of the Lecture

The main focus of the lecture was on lists and recursion in programming, particularly in Haskell. The instructor discussed the differences between list comprehensions and recursive programming, emphasizing that recursion is a powerful tool for manipulating lists.

2.1 Lists in Haskell

- Lists are fundamental data structures in Haskell.
- The operation `cons` (written as `(:)`) adds an element to the front of a list.
- The operation `append` (written as `++`) combines two lists.

2.2 Type Safety

Haskell enforces strict type checking, which can lead to type errors if operations are misapplied. The instructor highlighted the importance of understanding types to avoid such errors.

3 Recursion

Recursion is defined as a method where a function calls itself to solve smaller instances of the same problem. The instructor explained that recursion can express any computation that can be performed by a Turing machine, as stated in the Church-Turing thesis.

3.1 Defining Lists Recursively

A list can be defined recursively as follows:

- The empty list is defined as `[]`.
- A non-empty list can be defined as `(x : xs)`, where `x` is the head and `xs` is the tail.

3.2 Pattern Matching

Pattern matching is a technique used in Haskell to deconstruct lists. The instructor provided examples of how to match against lists to extract elements.

4 Recursive Functions

The lecture included examples of recursive functions, such as calculating the square of each element in a list. The recursive definition was compared to a list comprehension version.

4.1 Example: Squaring Elements

The function to square elements in a list can be defined recursively as:

$$\text{squares\_rec } [] = []\text{squares\_rec } (x : xs) = (x \times x) : \text{squares\_rec } xs$$

4.2 Example: Summing Elements

The sum of a list can also be defined recursively:

$$\text{sum } [] = 0\text{sum } (x : xs) = x + \text{sum } xs$$

5 Conclusion

The lecture concluded with a discussion on the importance of recursion in functional programming. The instructor encouraged students to practice writing recursive functions and to utilize tools like QuickCheck for testing their implementations.