Lecture Summary: Algebraic Data Types in Haskell

Generated by LectureMate

February 22, 2025

1 Introduction

The lecture began with a light-hearted introduction involving music and a mention of the textbook *The Land of Lisp*. The instructor highlighted the unique syntax of Lisp, characterized by its extensive use of parentheses, and transitioned into the main topic of the day: Algebraic Data Types (ADTs) in Haskell.

2 Algebraic Data Types

Algebraic Data Types are a fundamental concept in functional programming, particularly in Haskell. The term "algebraic" may sound intimidating, but it refers to a way of constructing new types from existing ones. The instructor emphasized that understanding types is crucial in functional programming.

2.1 Basic Types

The lecture reviewed basic built-in types in Haskell, such as:

- Integers
- Booleans
- Characters
- Tuples
- Lists

These types can be combined to create more complex types, such as lists of integers or tuples of booleans.

2.2 Defining New Types

The instructor demonstrated how to define new types using the `data` keyword. The first example was the Boolean type:

```
data Bool = False | True
```

This definition introduces a new type `Bool` with two constructors: `False` and `True`. The use of capital letters for constructors is a convention in Haskell.

2.3 Pattern Matching

Pattern matching allows for concise function definitions based on the structure of data types. The instructor provided examples of functions that operate on the `Bool` type:

```
not :: Bool -> Bool
not False = True
not True = False
```

3 Examples of Algebraic Data Types

The lecture continued with several examples of ADTs.

3.1 Seasons

A new type `Season` was defined to represent the four seasons:

```
data Season = Winter | Spring | Summer | Fall
```

Functions were defined to determine the next season and whether a season is warm.

3.2 Shapes

The instructor introduced a more complex type, `Shape`, which can represent circles and rectangles:

```
data Shape = Circle Float | Rect Float Float
```

This definition allows for functions to compute the area of a shape:

```
area :: Shape -> Float
area (Circle r) = pi * r * r
area (Rect w h) = w * h
```

3.3 Lists

The lecture also covered how to define lists using ADTs. A parameterized type for lists was introduced:

```
data List a = Nil | Cons a (List a)
```

This recursive definition allows for the construction of lists of any type.

3.4 Natural Numbers

Finally, the instructor discussed defining natural numbers using ADTs:

```
data Nat = Zero | Succ Nat
```

Functions for addition and multiplication were defined recursively, illustrating how ADTs can represent mathematical concepts.

4 Conclusion

The lecture concluded with a reminder of the importance of understanding ADTs in functional programming. The instructor encouraged students to explore the examples provided and to practice defining their own types and functions.