Lecture Summary: Lists and Comprehensions in Haskell

Generated by LectureMate

February 22, 2025

1 Introduction

In this lecture, we explored the concept of lists and list comprehensions in Haskell, emphasizing their importance as a fundamental data structure in functional programming. We compared lists to arrays in conventional languages like Java and C++, highlighting their role in collecting simpler data items.

2 Lists

2.1 Definition and Creation

A list is a compound data structure used to collect items of the same type. For example, a list of integers can be defined as follows:

$$\text{nums} = [1, 2, 3]$$

The type signature for this list is `[Int]`, indicating it is a list of integers. The order of elements in a list is significant, distinguishing it from sets in mathematics.

2.2 Types of Lists

- **List of Integers:** Example: `nums = [1, 2, 3]`
- **List of Characters:** In Haskell, characters are defined using single quotes. For example, `['I', 'n', 'f', 'A']` is a list of characters, which can also be represented as a string: `"InfA"`.
- **List of Lists:** Lists can contain other lists. For example, `[[1], [2, 2, 4], [], [3, 5]]` is a list of lists of integers.
- **List of Functions:** Functions can also be stored in lists, e.g., `[invert, flipV]` where both functions have the type `Picture -> Picture`.

2.3 Type Errors

An example of a type error occurs when trying to create a list with mixed types:

```
[1, "string", [1, 2]]
```

This results in a type error since all elements must be of the same type.

2.4 Special Notations

Haskell provides special notations for generating lists:

- `[1..10]` generates a list of integers from 1 to 10.
- `['a'..'z']` generates a list of lowercase letters.
- For even numbers, `[0, 2..100]` generates all even numbers from 0 to 100.

3 List Operations

3.1 Constructing Lists

The `cons` operator (`::`) is used to add an element to the front of a list:

```
1 :: [2, 3] = [1, 2, 3]
```

3.2 Decomposing Lists

The `head` and `tail` functions are used to access the first element and the rest of the list, respectively:

- `head [1, 2, 3] = 1`
- `tail [1, 2, 3] = [2, 3]`

3.3 Pattern Matching

Pattern matching is a powerful feature in Haskell. For example, the definition of `head` can be expressed as:

```
head (x : xs) = x
```

4 List Comprehensions

List comprehensions provide a concise way to create lists based on existing lists. The syntax is similar to set comprehensions in mathematics.

4.1 Basic Examples

- To create a list of squares:

  `[x * x | x <- [1, 2, 3]]` results in $[1, 4, 9]$

- To convert a string to lowercase:

  ```
  [toLower c | c <- "Hello World"]
  ```

4.2 Using Pairs and Tuples

List comprehensions can also generate pairs:

`[(x, even x) | x <- [1, 2, 3]]` results in $[(1, False), (2, True), (3, False)]$

4.3 Conditional Expressions

List comprehensions can include conditional expressions:

`[if even x then x else x + 1 | x <- [4, 5, 6]]` results in $[4, 6, 6]$

4.4 Guards

Guards can be used to filter results:

`[x | x <- [1..10], x  mod  2 == 0]` results in $[2, 4, 6, 8, 10]$

5 Built-in Functions

Haskell provides built-in functions like `sum` and `product` to operate on lists:

- `sum [1, 2, 3]` results in 6.
- `product [1, 2, 3]` results in 6.

6 Conclusion

In this lecture, we covered the fundamentals of lists and list comprehensions in Haskell, including their creation, manipulation, and practical applications. Understanding these concepts is crucial for effective functional programming.