Lists, Tuples, Pattern Matching, and Parameterized Types

Week 2, Lecture 2 of HMC CS 131 focuses on lists, tuples, pattern matching, and parameterized types in Haskell. There are practical and pedagogical reasons to study lists.

Why Lists?

1. Practical reason: Collecting data together in a single data structure is convenient. Lists in Haskell are similar to arrays in Java and C++, linked lists in Java and C++, and lists in Python. For example:

```
first_val  = 5
second_val = 3
third_val  = 19

vals = [5, 3, 19]
```

2. Pedagogical reason: Lists are a recursive data structure, which is a key concept in functional languages like Haskell. A list is either empty or an element followed by another list. Functional languages make heavy use of recursion, as shown in this example of a factorial function:

```
fact 0 = 1
fact n = n * fact (n-1)
```

Later in this lecture, we'll explore the powerful combination of functional languages, recursive data structures, and pattern matching. Lists are the simplest useful data structure to emphasize this point.

Lists

In Haskell, lists are a fundamental data structure. They are homogenous, meaning all elements of the list must be of the same type. Lists can be manipulated using various functions, including `head`, `tail`, `length`, and `reverse`.

Tuples

Tuples are another data structure in Haskell. Unlike lists, tuples can contain elements of different types. Tuples have a fixed size, and their length cannot be changed after creation.

Pattern Matching

Pattern matching is a powerful feature in Haskell that allows you to destructure data and match specific patterns. Pattern matching can be used with lists, tuples, and other data structures, making it an essential tool for writing concise and expressive code.

Parameterized Types

Parameterized types, also known as polymorphic types, allow you to write more general and reusable code by abstracting over the specific types used in functions or data structures. For example, a list can be parameterized over its element type, allowing you to create lists of integers, lists of characters, or lists of any other type.

This lecture covers essential concepts in Haskell, including lists, tuples, pattern matching, and parameterized types. These concepts will be vital for understanding functional programming and writing efficient, elegant code in Haskell.

Lists in Haskell: Recap and New Concepts

Haskell Syntax for Lists

Lists in Haskell use square brackets with contents separated by commas. They can contain various types of elements, including numbers, booleans, and even other lists. However, Haskell lists are homogeneous, meaning all elements within a list must be of the same type.

```
[4, 9, -13, 5, 1]
[False, True, False, False]
[3.1, 3.14, 3.141, 3.1419, 3.14159, 3.141592]
[]  -- an empty list
[[1, 2, 3], [4, 5, 6, 7], [8, 9]]  -- a list of lists
```

Building Lists in Haskell

The cons operator `(:)` is used to construct lists. For example, `3 : [7, 15, 4]` creates the same list as `[3, 7, 15, 4]`. The cons operator associates to the right, meaning the rightmost colons are given higher precedence. Lists can be recursively defined in Haskell:

```
3 : 7 : [15, 4]
3 : 7 : 15 : 4 : []
```

A list in Haskell is either an empty list `[]` or a value `x` 'consed' onto a list `xs`: `x : xs`.

Lists and Types

When working with the cons operator `(:)`, it's important to consider the types of its arguments. For example, if `x : xs`, the left side (x) has a certain type, and the right side (xs) has a list of that type:

```
Integer : [Integer]
Animal : [Animal]
Char : [Char]
Bool : [Bool]
```

This holds true for more complex types as well:

```
[String] : [[String]]
(Integer, Shape) : [(Integer, Shape)]
```

As you work with Haskell lists, understanding types and how they interact with list operations will help you create and manipulate lists effectively.

Building Lists in Haskell

Cons Operator

The cons operator `(:)` is used to construct lists. For example, `3 : [7, 15, 4]` is the same list as `[3, 7, 15, 4]`. The cons operator has a left side and a right side:

- The left side has type `a`
- The right side has type `[a]`

Both sides must match. In this instance, `a` is a type variable and can stand for any type. Parameterized types will be discussed later.

Concatenation Operator

The concatenation operator `(++)` returns the result of concatenating two lists. For example, `[1, 3] ++ [7, 15, 4]` results in `[1, 3, 7, 15, 4]`. The concatenation operator has a left side and a right side:

- The left side has type `[a]`
- The right side has type `[a]`

Both sides must match. Again, `a` is a type variable and can stand for any type.

Identity Element

An interesting observation: `[]` is the identity element for `++` on Haskell lists, similar to adding 0 to a number:

```
some_list ++ [] == some_list
[] ++ some_list == some_list
```

Practice: Haskell List Basics

Check your understanding of Haskell list basics. What do each of the following expressions evaluate to? If they produce an error, can you explain why? (You can try them out in GHCi if you want to!)

```
[1,2,3] : 5
["hi"] : [["my"], ["name", "is"]]
9 ++ [12, 15, 18, 21, 24]
(4 : []) ++ (3 : 2 : 1 : [])
[] : []
[True] ++ []
'H' : "ello," ++ "friend!"
```

Apologies for that. Let's continue with the "Exercise: emphasizeList".

Exercise: emphasizeList

Emphasize every element in a String list.

```haskell
emphasizeList :: [String] -> [String]
emphasizeList list =
  if null list
  then list
  else (++ "!") (head list) : emphasizeList (tail list)
```

Example:

```haskell
emphasizeList ["hi", "bye", "ahoy"] == ["hi!", "bye!", "ahoy!"]
```

Using map

We can shorten the `doubler` function using `map`:

```haskell
map (* 2) [19, 7, 21]
```

Exercise: strlens

Write a function `strlens` to compute the length of every string in a list.

```haskell
strlens :: [String] -> [Int]
strlens list = map length list
```

Example:

```haskell
strlens ["Haskell", "Python", "Java"] == [7, 6, 4]
```

Useful List Function: filter

Filter elements in a list that satisfy a test (a.k.a. a predicate):

```haskell
filter even [8, -13, 25, 16, 0, 2]
filter (<3) [8, -13, 25, 16, 0, 2]
filter ((<3) . length) ["oh", "hi", "there!"]
```

Useful List Functions: minimum, maximum, sum, product

These all do what you probably think they do for lists. To find out for sure, try them out! At this point, when performing operations on lists, you should think to yourself, "is this something common enough to already have been written and included in Haskell?" Look it up! Unless we specifically tell you not to use some function, feel free to look up and utilize existing Haskell list operations!

Pattern Matching for Lists

Pattern matching provides a more elegant and concise way to work with lists in Haskell.

Warmup Here's how you would write your own `length` and `filter` functions using pattern matching:

```haskell
myLength :: [a] -> Int
myLength [] = 0
myLength (_:xs) = 1 + myLength xs

myFilter :: (a -> Bool) -> [a] -> [a]
myFilter _ [] = []
myFilter test (x:xs)
  | test x    = x : myFilter test xs
  | otherwise = myFilter test xs
```

Pattern matching makes the code more readable by removing the need for explicit `if` statements and `tail` function calls. Instead, we deconstruct the list directly in the function definition.

Additional Exercises Using pattern matching, write your own `map` function:

```haskell
myMap :: (a -> b) -> [a] -> [b]
myMap _ [] = []
myMap f (x:xs) = f x : myMap f xs
```

Using pattern matching, write your own `take` function:

```haskell
myTake :: Int -> [a] -> [a]
myTake _ [] = []
myTake n (x:xs)
  | n <= 0    = []
  | otherwise = x : myTake (n-1) xs
```

Using pattern matching, write your own `drop` function:

```haskell
myDrop :: Int -> [a] -> [a]
myDrop _ [] = []
myDrop n (_:xs)
  | n <= 0    = xs
  | otherwise = myDrop (n-1) xs
```

These examples demonstrate how pattern matching can simplify your code and make it easier to understand. In general, when working with lists in Haskell, consider using pattern matching to deconstruct the list and process its elements.

Tuples and Pattern Matching

Tuples are used to combine different types of data into one structure. Unlike lists, tuples can have a fixed number of elements with different types.

Pattern matching can also be used with tuples. Here are some examples:

```haskell
prof1 = ("Lucas Bang", 131, False)
(name, course, acted) = prof1

isRightTriangle (a, b, c) = a * a + b * b == c * c

getCourse (_, course, _) = course
getCourse prof2 -- 131
```

Parameterized Types

Parameterized types, also known as polymorphic types, allow a single type to be parameterized by other types. This means that the same type can be used with different underlying types.

Here are the types for some common list functions:

```haskell
(:) :: a -> [a] -> [a]

(++) :: [a] -> [a] -> [a]

map :: (a -> b) -> [a] -> [b]

filter :: (a -> Bool) -> [a] -> [a]
```

These functions use type variables `a` and `b` to represent any type. This makes them more flexible and reusable.

Now, let's see the types of some other common functions:

- `head`: This function returns the first element of a list.

  ```haskell
  head :: [a] -> a
  ```

- `tail`: This function returns the list without its first element.

  ```haskell
  tail :: [a] -> [a]
  ```

- `take`: This function takes an integer `n` and a list, and returns the first `n` elements of the list.

  ```haskell
  take :: Int -> [a] -> [a]
  ```

- `drop`: This function takes an integer `n` and a list, and returns the list without its first `n` elements.

  ```haskell
  drop :: Int -> [a] -> [a]
  ```