Lec 8: List (2)

CS220: Programming Principles

Sang Kil Cha

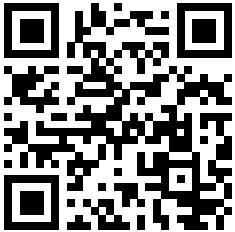
Recap

- List is a chain of cons cells.
- List is a recursive data structure.
- We use a type constructor to define a generic list type.

Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don't accept late responses.



Built-in List Type

No Need to Define Our Own List

- F# has a built-in list type: `'a list`.
- The cons operator is `::`.
- The empty list is `[]`.
- The car and cdr operators are `List.head` and `List.tail`.

No Need to Define Our Own List

- F# has a built-in list type: `'a list`.
- The cons operator is `::`.
- The empty list is `[]`.
- The car and cdr operators are `List.head` and `List.tail`.
- There are many built-in functions for lists in the `List` module, e.g.,
  - `List.length`
  - `List.append`
  - etc.

Length of a List

```
let rec length lst =
  match lst with
  | [] -> 0
  | _ :: tl -> 1 + length tl
```

Length of a List

```
let rec length lst =
  match lst with
  | [] -> 0
  | _ :: tl -> 1 + length tl
```

Can you make it tail-recursive?

Tail-recursive `length`

```
let length lst =
  let rec loop cnt = function
    | [] -> cnt
    | _ :: tl -> loop (cnt + 1) tl
  loop 0 lst
```

In-Class Activity #08

Preparation

We are going to use the same git repository as before. Just in case you don't have it, clone the repository using the following command.

1. Clone the repository to your machine.
   - `git clone https://github.com/KAIST-CS220/CS220-Main.git`
2. Move in to the directory `CS220-Main/Activities`
   - `cd CS220-Main`
   - `cd Activities`

Problem 1: Checking List of List

Given a list of lists, write a function `countEmptyList` that returns the number of empty lists in the given list.

Problem 2: Equivalence of Lists

Write a function `equal` that takes in two lists as input and returns a boolean indicating whether the two lists have the same sequence of elements or not.

Structural Equality vs. Physical Equality

F# uses ***structural equality*** by default. This means that two types are equal if they have the same structure. For example, `let a = [1; 2; 3]` and `let b = [1; 2; 3]` are structurally equal.

Structural Equality vs. Physical Equality

F# uses ***structural equality*** by default. This means that two types are equal if they have the same structure. For example, `let a = [1; 2; 3]` and `let b = [1; 2; 3]` are structurally equal.

On the other hand, physical equality means that two types are equal if they are the same object in the memory. For example, `let a = [1; 2; 3]` and `let b = [1; 2; 3]` are not physically equal.

Physical Equality in F#?

`LanguagePrimitives.PhysicalEquality` is a function that checks physical equality of two values.

```
let (==) = LanguagePrimitives.PhysicalEquality
```

Physical Equality $\rightarrow$ Structural Equality

In functional world:

- Physical equality implies structural equality.
- Structural equality does not imply physical equality.

This means, one can efficiently check equality of two values by checking physical equality first. This allows us to write an extremely efficient data structure, e.g., hash consing.

Recursion is the Key to Handle Lists

When writing a function that handles a list, you should think recursively.

Pattern Matching vs. Built-in Functions

Which one is better?

1. Using car/cdr (`List.head` and `List.tail`).
2. Using pattern matching.

Other Type Constructors

Another Type Constructor: Option

Option type is a built-in union type that represents either a valid value or an invalid (or missing) value.

Something or nothing.

```
type IntOrNothing =
  | Int of int
  | NoInt

type StringOrNothing =
  | Str of string
  | NoStr
```

Option Type Constructor

Something or nothing (generic type).

```
/// This is a built-int type: no need to define this.
type Option<'T> = // We often write this as 'a option
  | Some of 'T
  | None

let validInt = Some 42
let invalidInt = None
```

Option Type Example

Suppose you have a database of movies. You want to develop an API (`findMovie`) that takes in a title as input and returns information about a movie that matches the given title. What would be the signature of the function?

```
val findMovie:  string -> Movie option
```

`'a option` vs. `Option<'a>`

For historical reasons, we prefer `'a option` than `Option<'a>`, and prefer `'a list` than `list<'a>`. However, other than these two cases, we prefer to put type parameters after type constructors.

List of List & Option of Option

```
[ [1]; [2; 3]; [4; 5] ] // A list of integer lists
Some (Some 42) // An option of an integer option
```

Result Type Constructor

```
type Result<'T, 'TError> =
  | Ok of 'T
  | Error of 'TError
```

Result Type vs. Option Type

- Option type is useful when you want to represent a missing value.
- Result type is useful when you want to represent an error.

Usage of Result Type

```
type Request = { Name: string; Email: string }

let validateName req =
  match req.Name with
  | "" -> Error "Name is empty"
  | "bananas" -> Error "Bananas is not a name."
  | _ -> Ok req

let validateEmail req =
  match req.Email with
  | "" -> Error "Email is empty"
  | s when s.EndsWith("bananas.com") -> Error "bananas.com is not allowed."
  | _ -> Ok req

let validateRequest reqResult =
  reqResult |> Result.bind validateName |> Result.bind validateEmail
```

Question?

Further Reading

- Do more exercises here: `http://www.fssnip.net/an/title/NinetyNine-F-Problems-Problems-1-10-Lists`.
- Read: `https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/results`.