Lec 6: List

CS220: Programming Principles

Sang Kil Cha
List
List

List is a finite sequence of values.

Extremely useful data type!
### Example Lists.

```plaintext
[] // Empty list  
[1; 2; 3] // A list of three integers.  
["a"; "b"; "c"; "d"] // A list of four strings.  
[1; 2; "abc"] // ?
```
Cons

1. **Cons** constructs a value, which is often referred to as a **cons cell**, holding a pair of values. We say, “cons x onto y” when we construct a new pair where y is followed by x.

2. The first element of a cons cell is **car**.

3. The second element of a cons cell is **cdr**\(^1\).

\(^1\)Why car and cdr? See: https://en.wikipedia.org/wiki/CAR_and_CDR
List = a Sequence of Cons Cells

- There is a value that represents an empty list (often called as nil).
- A singleton list is a cons cell of a value and an empty list.
- A list with two elements can be constructed by consing two cons cells.
- ...

![Diagram of list structure](image-url)
List Consing Operator (::)

```latex
let (::) a b = // Prepend a to the list b.

1 :: [2] // Returns [1; 2]
2 :: [4; 6] // Returns [2; 4; 6]
"abc" :: [] // Returns ["abc"]

1 :: 2 :: 3 :: 4 :: [] // [1; 2; 3; 4]
```

Note: [1; 2; 3; 4] is just syntactic sugar for 1 :: 2 :: 3 :: 4 ::[].
List car and cdr.

```ocaml
let car lst = 
  match lst with 
  | hd :: _ -> hd 
  | ... // ?

let cdr lst = 
  match lst with 
  | _ :: tl -> tl 
  | ... // invalidArg ?
```
List Range Expressions

A convenient way to construct lists.

Example list range expressions.

- \([ 1 .. 5 ] // \text{Returns } [1; 2; 3; 4; 5]\)
- \([ -1 .. 1 ] // \text{Returns } [-1; 0; 1]\)
- \([ 1 .. 2 .. 5 ] // \text{Returns } [1; 3; 5]\)
- \([ 1.0 .. 3.2 ] // \text{Returns } [1.0; 2.0; 3.0]\)
- \([ 1 .. -2 .. 5 ] // ?\)
Appending an Element to a List

Can we “append” an element to a list? What does it mean to append an element to a list?

Values are immutable in F#!
Write Your Own List Type
Let’s Define a Cons Cell Type for Integers

IntList can only be either a nil or a cons of two elements.

How would you combine two seemingly different types?
Let’s Define a Cons Cell Type for Integers (cont’d)

Integer list type.

```plaintext
type IntList =
  | Nil
  | Cons of int * IntList
```

Can you now construct a list \([1; 2; 3]\) with the newly defined type?
Write basic operators.

```ocaml
let empty =
  // ?

let cons elt lst =
  // ?

let car lst =
  // ?

let cdr lst =
  // ?
```
Write infix operator.

Using `cons` for constructing large lists is inconvenient.

Infix operator for consing:

- `let (::) elt lst = ... // Error!`
- `let (+++) elt lst = Cons (elt, lst) // First trial.`
- `let (^+^) elt lst = Cons (elt, lst) // Second trial.`

Read:

Difference between `IntList` vs. ’a list?

There is a space in the ’a list type! 😊 Ours can only take integers, whereas ’a list can take any types.

Can we make our list implementation generic?
Generic Types

A type constructor is a function that takes in a type and returns a type. For example, `list` is a type constructor that takes in `a` as a parameter. For the `int` type, it returns the `int list` type. We sometimes call a type constructor as generic.

```typescript
/// One way to define a generic type.
type 'a MyList =
| Nil
| Cons of 'a * 'a MyList

/// Another way to define a generic type.
type MyList<T> =
| Nil
| Cons of T * MyList<T>
```
Exercise: isEmpty

Write a function `isEmpty` that checks if a given list (`MyList<T>`) is empty or not.
In-Class Activity #06: List Length Function

Write a function `myfunc` that takes in a list (`MyList<’T>`) and returns the length of the list.

1. It is obvious that the function will return 0 for an empty list.
2. For a given cons cell, we can recursively compute the length of its `cdr`, and adds one to the result (as `car` of the list would only contain a single element anyways).
Exercise: Membership Test Function

Write a function `isMember` that takes in an element (of type 'T) and a list (`MyList<T>`) and returns a boolean indicating whether the element is a member of the given list or not.

1. It is obvious that the function will return false for an empty list.
2. For a given cons cell, we can recursively compare the equality between the given value and the `car` of the list, and then recurse into the `cdr` of the list.
Exercise: List Append

The infix operator (@) joins two lists. For example, [1; 2] @ [3; 4] returns [1; 2; 3; 4]. Write your own append operator over MyList<T>!

1. append [1; 2] [3; 4]
Exercise: List Append

The infix operator (@) joins two lists. For example, [1; 2] @ [3; 4] returns [1; 2; 3; 4]. Write your own append operator over MyList<'T>!

1. append [1; 2] [3; 4]
2. 1 :: (append [2] [3; 4])
Exercise: List Append

The infix operator (@) joins two lists. For example, [1; 2] @ [3; 4] returns [1; 2; 3; 4]. Write your own append operator over MyList<T>!

1. append [1; 2] [3; 4]
2. 1 :: (append [2] [3; 4])
3. 1 :: 2 :: (append [] [3; 4])
Exercise: List Append

The infix operator (@) joins two lists. For example, [1; 2] @ [3; 4] returns [1; 2; 3; 4]. Write your own append operator over MyList<'T>!

1. append [1; 2] [3; 4]
2. 1 :: (append [2] [3; 4])
3. 1 :: 2 :: (append [] [3; 4])
4. 1 :: 2 :: [3; 4]
Exercise: List Append

The infix operator (@) joins two lists. For example, [1; 2] @ [3; 4] returns [1; 2; 3; 4]. Write your own append operator over MyList<'T>!

1. append [1; 2] [3; 4]
2. 1 :: (append [2] [3; 4])
3. 1 :: 2 :: (append [] [3; 4])
4. 1 :: 2 :: [3; 4]
5. 1 :: [2; 3; 4]
Exercise: List Append

The infix operator (@) joins two lists. For example, [1; 2] @ [3; 4] returns [1; 2; 3; 4]. Write your own append operator over MyList<'T>!

1. append [1; 2] [3; 4]
2. 1 :: (append [2] [3; 4])
3. 1 :: 2 :: (append [] [3; 4])
4. 1 :: 2 :: [3; 4]
5. 1 :: [2; 3; 4]
6. [1; 2; 3; 4]
Make it Tail-Recursive!

Both examples were not tail-recursive. Why?

Can you make them tail-recursive?
Exercise: List Reverse

Write a function `rev` that takes in a list (`MyList<'T>`) and returns a reversed list.
In-Class Activity #07

Write a function `equal` that takes in two lists of the `MyList<'T>` type and returns a boolean indicating whether the lists have the same sequence of elements or not.
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.

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

type StringOrNothing =
  | Str of string
  | NoStr
```
Option Type Constructor

Something or nothing (generic type).

```typescript
/// 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?

```ocaml
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

\[
[ [1]; [2; 3]; [4; 5] ] \quad // \quad \text{A list of integer lists}
\]

Some (Some 42) \quad // \quad \text{An option of an integer option}
Conclusion
1. We can create our own list datatype with data abstraction techniques we learned so far.

2. List type is commonly used, and F# has a built-in support for lists.

3. List can be generic.

4. Type constructors is a function that takes in a type and returns a type. It is used to construct generic types such as List and Option.
Further Reading

See more examples from
Question?