This lecture covers lists, which are used extensively in functional programming. It also covers recursion and pattern matching in more depth.
Structured types: lists

- Lists are structured types for ordered collections of values.
- `int list` is the type of lists containing integers
  - alternatively written as: `list<int>`
- More generally:
  - `T list` is the type of lists containing values of type `T`, where `T` can be any F# type.
- List values are constructed using `[]` and `::`
- `[]` is an empty list (of any type).
- `x :: xs` is a `T list` if `x` has type `T` and `xs` is a `T list`.
  - This constructs a list with `x` in front of the elements in `xs`.
  - `x` is called the head and `xs` the tail.
Functional nature of lists

- Note that `::` does not modify the tail list.
  - Instead it always creates a new list.
  - In FP, generally data structures are not modified, instead new ones are created.

- Generally lists are implemented as linked lists.
  - I.e., objects containing a head and a tail reference/pointer.
  - Many lists may share the same tail:
    ```
    let xs = 1 :: 2 :: 3 :: 4 :: []
    let ys = 0 :: xs
    let zs = 9 :: xs
    ```
  - Usually you wouldn’t do this in C/Java because the first list might be modified later, affecting the other lists.
  - This is quite a major difference, and can be more efficient because memory is saved due to sharing.
Creating and using lists

- Lists can be deconstructed by pattern matching.
  - Every list is constructed using either [] or ::
  - So, often we match these two cases – empty and non-empty lists.
  - And, often we use recursion to process the tail of the list.

```ocaml
let rec SumList = function
  | []        -> 0
  | x::xs     -> x + SumList xs
```

- There’s a handy abbreviation for creating a list by giving the elements:

  ```ocaml```
  - let as = [1; 2; 3; 4]       // Abbreviation
  - let as = 1 :: 2 :: 3 :: 4 :: [] // Without the abbreviation

- Lists can also be constructed using ranges.

  ```ocaml```
  - > [1 .. 10];;
    val it : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
  - > [1 .. 2 .. 10];;
    val it : int list = [1; 3; 5; 7; 9]
  - > ['a' .. 'e'];;
    val it : char list = ['a'; 'b'; 'c'; 'd'; 'e']
List comprehensions

- List comprehensions are an elegant way of defining lists using other lists.
- `for` is used to for generators, which enumerate the elements of another list.
- `if` allows filtering based on a Boolean.
- `yield` adds an element to the list being created.

```plaintext
let factors n = [ for x in 1 .. n do
                  if n % x = 0 then yield x
                 ]
factors 40;;
val it : int list = [1; 2; 4; 5; 8; 10; 20; 40]
```

- Multiple generators can be used, with an effect similar to a nested loop.

```plaintext
let commonFactors m n = [ for x in factors m do
                          for y in factors n do
                          if x=y then yield x
                        ]
commonFactors 40 110;;
val it : int list = [1; 2; 5; 10]
```
List comprehensions (cont.)

- can be used with `for ... in` as an abbreviation of `do yield`

```ml
let squares n = [ for x in 1 .. n -> (x, x*x) ]
squares 4;;
val it : int list = [ (1,1); (2,4); (3,9); (4,16); (5,25) ]
```

- `for` allows patterns:

```ml
let squareAdd n = [ for (x,sq) in squares 4 -> x + sq ]
squareAdd 8;;
val it : int list = [ 2; 6; 12; 20; 30 ]
```

- `yield!` puts a whole list of values into the output list.

```ml
[ for a in 1 .. 5 do
  match a with
  | 3 -> yield! ["a"; "b"]
  | _ -> yield a.ToString()  // "_" matches anything
];;
val it : string list = ["1"; "2"; "a"; "b"; "4"; "5"]
```