Recursion

- Using pure functions we don't have access to variables. So how do we loop? Recursion!

- In F# we can define recursive functions using let rec ... =

- Recusive functions have two cases:
  - The recursive case, which "loops" further by calling itself
  - The base case which terminates the recursion, and starts unwinding

- But isn't recursion terribly inefficient? Yes and no! Regular recursion is quite expensive in terms of memory, Stack and function calls.

  But! We can often rearrange recursion into a form when tail call optimization can be performed → reursion becomes equivalent to imperative loops.

(How can we recurse in pure λ-calculus?
The Y-combinator!)

```
let rec fact n =
    if n = 0 then
      1
    else
      n * fact (n-1)
```

With TCO:

```
let fact n =
  let f acc n =
    if n = 0 then
      1
    else
      f (n * acc) (n-1)
  f 0 n
```

How does TCO work? (using mutation)

```
f acc n =
top:
  if n = 1
    acc
  else
    acc ← n * acc
    n ← n-1
    goto top
```

⟹

```
acc ← 1
while n > 0:
    acc ← n * acc
    n ← n-1
```

Recursive data structures

- Data types can also be recursive:

```
type List<'a> =
    | Cons of 'a * List<'a>
    | Nil
```

This is the canonical representation of a linked list using product and sum types.

Note how elegant it is! No pointers or references to the next element. Just clean recursion!

"To loop is human, to recurse is divine"

- We can construct a list like this:

```
let l = Cons (1, Cons (2, Cons (3, Nil)))
```

and we destruct it using recursion and pattern matching

```
let rec sum x = function
    | Cons (n, l) -> sum (x+n) l
    | Nil -> x
```

Mutual recursion

- In F# we must always define things before we use them. But what if we can't?

```
let f x = if x>0 then g(x-1)else x
let g x = if x>0 then f(x-1)else x
```

⇒
```
let rec f x = ...
    and g x = ...
```

- For types:

```
type Folder = {
    path : string
    files : File list
}
and File = {name: string; folder : Folder}
```

- Sometimes when we have many mutually recursive cases it's easier to define a recursive module:

```
module rec Stuff = ...
```

This makes everything in the module mutually recursive. Use with <u>care!!</u>

Collection types

List: a linked list which can be appended to with (::) (cons)

Array: A fixed size array. Much more performant than lists, and much less flexible api:s

Map: A immutable dictionary (key, value)

Set: A collection of unique elements stored in a binary tree. Fast lookups.

Seq: seq<'T> is an alias for IEnumerable<'T>
All of above are instances of seq!

- All collection types are well documented, also in IntelliSense / FSAC

- Notable functions:

  takes from/to
  ↙

  Seq.empty
  Seq.singleton
  Seq.ofList
  Seq.ofArray

Looping part 2

- The most common cause for looping is traversing and transforming data structures (collections)
- For this purpose we have 3 work horses for collection types: map, filter, fold

- map f [...] → [f x_1; f x_2; f x_3 ...]
  map transforms, while retaining shape

- filter p [ ] → [ ]
  (p: true?/false?)
  filter removes elements which do not satisfy the predicate p

- fold (fun acc x → ...) m [...]
  (acc: accumulator; m: initial value and type; x: next element)
  fold is very general and can map, filter reduce, expand, contract data structures!
- The "normal" fold is a left fold.
- There is also a right fold:

  foldBack (fun x acc → ...) [ ] m

- Right folds are primitive recursive and can encode any recursion!

Mutable state (be ware!)

- In F# mutable variables are declared and modified explicitly:

```
let mutable V = 0
V <- 1
```

- Always think twice before using mutables! We do need them, sometimes for performance, but most often we do not.

- F# also have reference cells:

```
let r = ref 42
r := 42 + 1
printfn "%d" !r
```

- Reference cells are defined like this

```
type Ref<'a> = {mutable v : 'a}
let ref x = {v = x}
```

- What is the difference:

```
let a = ref 5
let b = a
b := 10 // modifies both
```

```
let mutable a = 5
let mutable b = a
b <- 10 // only b
```

Reference cells are actual values!