Programovací jazyky F# a OCaml

Chapter 4.

Generic and recursive types

Generic types

Generic types

» Types that can carry values of any type

Note: Built-in tuple is "generic" automatically

```
let tup1 = ("hello", 1234)
let tup2 = (12.34, false)
```

» Declaring our own generic types

```
type MyOption<'a> =
  | MySome of 'a
  | MyNone
```

Generic type parameter(s)

Actual type argument will be used here

OCaml-compatible syntax

```
type 'a MyOption =
  | MySome of 'a
  | MyNone
```

Using generic types

» Type inference infers the type automatically

```
> let optNum = MySome(5);;
val optNum : MyOption<int> = MySome 5
```

Inferred type argument

```
> let optStr = MySome("abc");;
val optStr : MyOption<string> = MySome "abc"

> let opt = MyNone;;
val opt : MyOption<'a>
```

Tricky thing: generic value – we don't know the actual type argument yet

```
> optStr = opt;;
val it : bool = false
```

We can use it with any other option

Writing generic functions

» Just like ordinary functions – “it just works”!

```
> let getValue opt =
    match opt with
    | MySome(v) -> v
    | _ -> failwith "error!";;
val getValue : MyOption<'a> -> 'a

> getValue (MySome "hey");;
val it : string = "hey"
```

Inferred as generic argument

Doesn’t matter what type the value has

The type of “getValue” is generic function

» Automatic generalization

An important aspect of F# type inference

Finds the most general type of a function

Recursive data types

Type declarations

» **Question**

Can we create a data type that can represent datasets of arbitrary size, unknown at compile time (using what we've seen so far)?

If yes, how can we do that?

Recursive type declarations

» Using type recursively in its declaration

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

let list = Cons(0, Cons(1, Cons(2, ...)))
```

Recursive usage

This looks like a problem!

» We also need to terminate the recursion

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

let list = Cons(0, Cons(1, Cons(2, Nil)))
```

Represents an empty list

F# list type

» Same as our previous list – two constructors:

- `[]` – creates an empty list
- `::` – creates list from element and a list
- operator `@` – concatenates two lists (inefficient)

```
let data = (1::(2::(3::(4::(5::[])))))
let data = 1::2::3::4::5::[]
let data = [1; 2; 3; 4; 5]
```

Right-associative

Simplified syntax

» Processing lists using pattern-matching

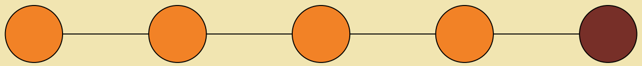
```
let firstElement =
  match data with
  | [] -> -1
  | x::_ -> x
```

Complete pattern match – covers both cases

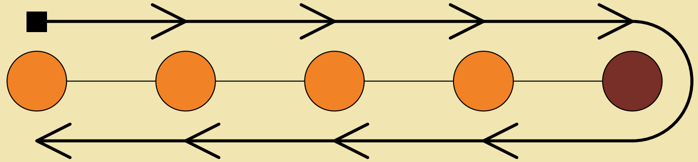
Structural recursion

» Processing data structures recursively

List defines the structure of the data:



We can follow this structure when processing:



Processing will always terminate (if data is finite)!

» We can express many standard operations

Filtering, projection, aggregation (aka folding)

Structurally recursive functions

» Processing functions have similar structure

```
let rec removeOdds list =
  match list with
  | [] -> []
  | x::xs ->
     let rest = removeOdds xs
     if x%2=0 then rest else x::rest
```

Return [] for empty list

Recursively process the rest

Join current element with the processed

```
let rec sumList list =
  match list with
  | [] -> 0
  | x::xs -> x + (sumList xs)
```

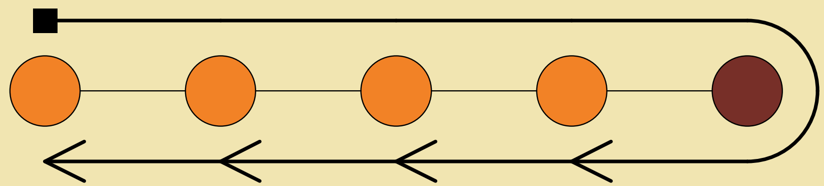
Return 0 for empty list

Join current element with the sum

Recursively sum the rest

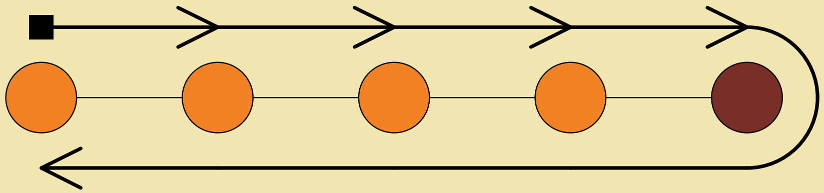
Processing lists

- Sum, filtering – calculate on the way back



Value is returned as the result from the function

- Other operations calculate on the way forward



Pass the value as argument to the recursive call

Structurally recursive functions

» Calculating on the way forward – reverse

```
let rec reverse' res list =
  match list with
  | [] -> []
  | x::xs -> reverse' (x::res) list

let reverse list = reverse' [] list
```

Return [] for empty list

Recursive call

Perform calculation before recursion

» Technique called *accumulator argument*

We accumulate the result of the function

Important concept that allows *tail-recursion*

Homework #1

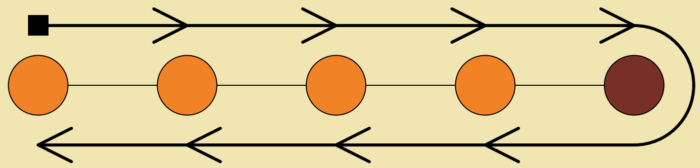
» *Write a function that counts the number of elements in the list that are larger than or equal to the average (using integer division for simplicity).*

```
foo [1; 2; 3; 4] = 3 // average 2
foo [1; 2; 3; 6] = 2 // average 3
foo [4; 4; 4; 4] = 4 // average 4
```

Using just a single traversal of the list structure!

You can define a utility function `foo'` *if you need to…*

Hint:



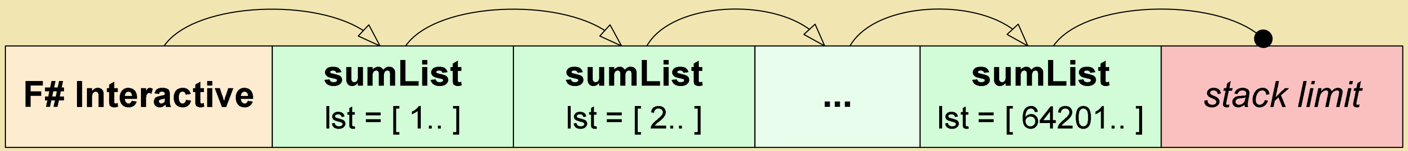
Tail-recursion

Sum list – non-tail-recursive

```
let test2 = List.init 100000 (fun _ -> rnd.Next(- 50, 51);;
let rec sumList list =
    match list with
    | [] -> 0
    | x::xs -> x + sumList xs
```

Performs addition after recursion

» Every recursive call adds a single stack frame



We'll can get StackOverflowException

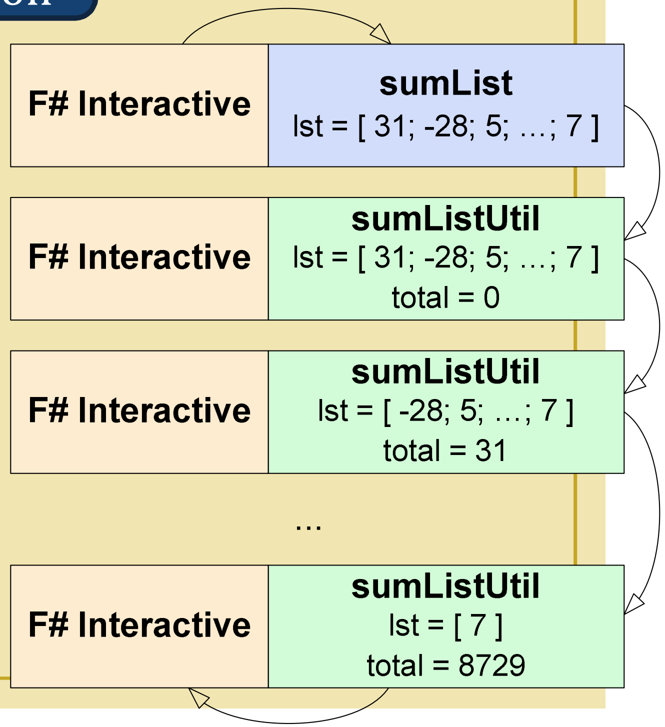
How to rewrite the function using tail-recursion?

Sum list – tail-recursive

```
let rec sumList' total list =
       match list with
        | [] -> total
        | x::xs ->
          let ntotal = x + total
          sumList' ntotal xs
let sumList list = sumList' 0 list
```

Recursive call is the last thing!

Performs addition **before** recursion



» Process while going forward
  - We can drop the current stack frame when performing a tail-recursive call

Homework #2

- *Write a tail-recursive function that takes a list and "removes" all odd numbers from the list.*

  (e.g. removeOdds [1; 2; 3; 5; 4] = [2; 4])

- ***Hints:***

  1. *Tail-recursive functions do all processing when traversing the list forward.*

  2. *You'll need to do this during two traversals of some list (both of them use accumulator and are tail-recursive)*