Programovací jazyky F# a OCaml

Chapter 5.

Hiding recursion using function-as-values

Hiding the recursive part

- Writing recursive functions explicitly

```
let rec sum list =
  match list with | [] -> 0 | x::xs -> x + (sum xs)
let rec mul list =
  match list with | [] -> 1 | x::xs -> x * (mul xs)
```

  How to avoid repeating the same pattern?

- Parameterized and higher-order functions

  Initial value: 1 for mul and 0 for sum

  Aggregation function: * for mul and + for sum

List processing with HOFs

» Generalized function for aggregation

```
> let rec aggregate f initial list =
    match list with
    | [] -> initial
    | x::xs -> f (aggregate f initial xs) x;;
val aggregate : ('a -> 'b -> 'a) -> 'b -> 'a list -> 'b
```

"some" state

» Automatic generalization

Infers the most general type of the function!

```
> aggregate (+) 0 [1 .. 10];;
val it : int = 55
> aggregate (fun st el -> el::st) [] [1 .. 10];;
val it : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

Previously unexpected use!

Processing options using HOFs

Reading two integers

» Read two integers and add them

Fail (return None) when the input is invalid

```
let readInput() =
  let (succ, num) = Int32.TryParse(Console.ReadLine())
  if succ then Some(num) else None

let readAndAdd() =
  match readInput() with
  | None    -> None
  | Some(n) ->
     match (readInput()) with
     | None    -> None
     | Some(m) -> Some(n + m)
```

First input is wrong

Read second value

Second input is wrong

Finally!

Simplifying using HOFs

map – apply calculation to a value (if there is any) and wrap the value in option with same structure

```
let readAndAdd() =
  match readInput() with
  | None    -> None
  | Some(n) -> readInput() |> Option.map ((+) n)
```

bind – apply calculation to a value (if there is any), the calculation can fail (returns another option)

```
let readAndAdd() =
  readInput() |> Option.bind (fun n ->
    readInput() |> Option.map ((+) n))
```

DEMO

Working with options in F#

F# library functions

- Working with options in F#
  - **map** – Calculates a new value if there is a value
  - **bind** – Calculates a new option if there is a value
  - **exists** – True if a value exists & matches predicate
  - **fold** – Aggregates "all" values into a single value

List processing using HOFs

F# library functions

» Processing lists in F#

- **map** – Generates a new value for each element
- **filter** – Creates list filtered using predicate
- **fold** – Aggregate all elements into "some state"
- **foldBack** – same as fold, but from the end
- **collect** (aka **bind**) – generate list of data for every element and merge all created lists
- **rev** – reverse the list
- **Seq.unfold** – builds a sequence (convertible to list)

DEMO

Working with lists in F#

Pipelining and composition

» Pipelining (|> operator)

```
[1 .. 10] |> List.filter (fun n -> n%2=0)
          |> List.map (fun n -> n*n)

// Implementation is very simple!
let (|>) v f = f v
```

Pass the value eon the left side to the function on the right

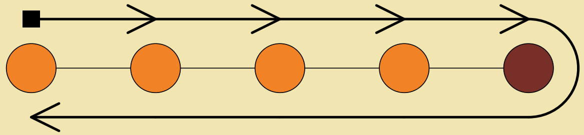
» Composition (>> operator)

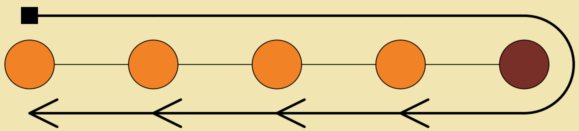
```
[(1, "one"); (2, "two"); (3, "three")]
   |> List.map (snd >> String.length)

// Implementation is very simple!
let (>>) f g x = g (f x)
```

Creates a function that performs *f*, then *g*.

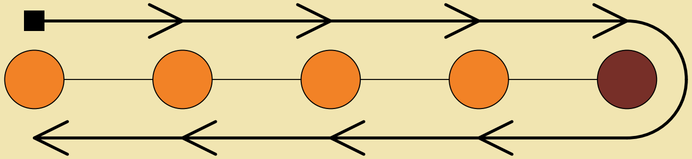
Homework #1

- ***fold*** *processes data on the way to the front,* ***foldBack*** *on the way back (to the beginning).*





- *Write a more general function that allows us to do both things at once. Use it to implement:*
  - ***fold*** *&* ***foldBack***
  - ***traverse*** *homework (from Ch. 4, slide 14)*



Abstract data types

∈

Algebraic definitions

- Defines a set and an operation on the set
- **Monoid**: is a set **M**, operation •, element *e*:

  Operation: $\bullet : \boldsymbol{M} \times \boldsymbol{M} \rightarrow \boldsymbol{M}$

  Associativity: $(a \bullet b) \bullet c = a \bullet (b \bullet c)$

  Identity: $e \bullet a = a \bullet e = a$

- **Natural numbers**: set **N**, 0 ∈ **N**, operation S

  Successor: $S : \boldsymbol{N} \rightarrow \boldsymbol{N}$

  Plus a lot of axioms defines natural numbers

Abstract data type

- Describe type using operations we can use
- For example, a list is **L<'a>** and operations:

  Operation: *map : ('a → 'b) → L<'a> → L<'b>*

  Operation: *fold : ('a → 'b → ‘a) → 'a → L<‘b> → ‘a*

  There are also some axioms…

  map g (map f l) == map (f >> g) l

  *fold g v (map f l) == fold (**fun** s x -> g s (f x)) v l*

- Abstract description of types (as in algebra ☺)

Back to monoids…

- What is a monoid in programming language?

  Monoid is **M** and e ∈ **M** operation *f* : **M -> M -> M**

Associativity:	$f\ (f\ a\ b)\ c = f\ a\ (f\ b\ c)$
Identity:	$f\ e\ a = f\ a\ e = e$

- Which F# data types are monoids?

Numeric :	Int `(+, 0)`, Int `(*, 1)`	
Strings:	`(+, "")`	+ is concatenation
Lists:	`(@, [])`	@ is concatenation

Representing other data structures