List Functions, and Higher-Order Functions

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
http://www.idt.mdh.se/˜blr/

List Functions

We have seen some list functions already

There are some important ones left

We'll define `List.append` (@) and `List.zip` here

List.append

(We've seen it in use before)

```
let rec (@) xs ys =
  match xs with
  | []    -> ys
  | x::xs ->  x :: (xs @ ys)
```

Thus,

```
[1;2] @ [3;4;5] =  1 :: 2 :: [] @ 3 :: 4 :: 5 :: []
                => 1 :: (2 :: [] @ 3 :: 4 :: 5 :: [])
                => 1 :: 2 :: ([] @ 3 :: 4 :: 5 :: [])
                => 1 :: 2 :: 3 :: 4 :: 5 :: []
                =  [1;2;3;4;5]
```

Note that `List.append` takes time proportional to length of first argument

List.zip

`List.zip` takes two lists and returns a list of pairs of their respective elements (like closing a zipper):

```
zip : 'a list -> 'b list -> ('a * 'b) list
let rec zip l1 l2 =
  match (l1,l2) with
  | (x::xs,y::ys) -> (x,y) :: zip xs ys
  | ([],[])       -> []
  | _             -> failwith "Lists have different length"
```

Thus,

```
List.zip [1;2;3] ["allan";"tar";"kakan"] =>
         [(1,"allan");(2,"tar");(3,"kakan")]
```

So we can for instance use `List.zip` to put a number on each element in a list

`List.zip` requires the argument lists to be of equal length

Exercise: define a version that accepts lists of different length! Let the resulting list be as long as the shortest argument list

Higher-Order Functions

F# is a *higher order* language

This means that functions are data just as data of any other “ordinary” type

They can be stored in data structures, passed as arguments, and returned as function values

Functions as arguments provides a way to *parameterize* function definitions, where common computational structure is “factored out”

Functions that take functions as arguments are called *higher-order functions*

Common computational patterns can be captured as higher order functions

We’ll show some important examples here

Three Common Higher-Order Functions over Lists

- **map**: apply a function to all elements in a list
- **filter**: remove all elements not satisyfing a given condition
- **fold** (several versions): combine all elements using a function with two arguments (like binary operators)

These functions capture common computation patterns

They allow these patterns to be reused

All functional languages have them

Can also be defined for other data types, like arrays and trees

A First Example: List.map

A common pattern is to apply a function to each element in a list

An example: a function that adds one to each element in a list of integers:

```
inclist : int list -> int list
let rec inclist l =
  match l with
  | []    -> []
  | x::xs -> x + 1 :: inclist xs

inclist [2;1;3;4] => [3;2;4;5]
```

Computation pattern captured by a higher-order function `List.map`:

```
let rec map f l =
  match l with
  | []    -> []
  | x::xs -> f x :: map f xs
```

`List.map` applies an *arbitrary* function `f` to the elements in a list

```
List.map : ('a -> 'b) -> 'a list -> 'b list
```

Note that the type of `List.map` is polymorphic. This is common for higher-order functions

We can now define `inclist` through `List.map` instead:

```
let inclist l = let inc n = n + 1 in map inc l
```


A Second Example: filter

`List.filter` removes all elements from a list that do not satisfy a given predicate:

```
filter : ('a -> bool) -> 'a list -> 'a list
let rec filter p l =
  match l with
  | []    -> []
  | x::xs -> if p x then x :: filter p xs
                    else filter p xs
```

For instance: if `even` returns `true` for exactly the even numbers, then

```
filter even [0;1;2;3;4;5] => [0;2;4]
```


Some Syntax: Guarded Patterns

Here is another way to define `List.filter` in F#:

```
let rec filter p l =
  match l with
  | []             -> []
  | x::xs when p x -> x :: filter p xs
  | x::xs          -> filter p xs
```

This definition uses a *guard*: a condition that "filters out" a certain case

The keyword "`when`" specifies a guard

`pattern when guard -> expr` will return `expr` when the pattern is matched and the guard becomes true

(Guards are just syntactic sugar)

Folds

Rather than applying a function to each single member of a list, we might want to apply a function with two arguments successively to all elements

An instance of this is summing all numbers in a numeric list, recall `List.sum`:

```
let rec sum l =
  match l with
  | []    -> 0
  | x::xs -> x + sum xs
```

Applies $+$ successively to all elements, "collecting" them into their sum

Now consider *multiplying* the numbers in a list:

```
let rec product l =
  match l with
  | []    -> 1
  | x::xs -> x * product xs
```

Or, ANDing together a list of booleans:

```
let rec all l =
  match l with
  | []    -> true
  | x::xs -> x && all xs
```

There's something in common here!

All these functions are instances of `List.foldBack`:

```
foldBack : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
let rec foldBack f l init =
  match l with
  | []    -> init
  | x::xs -> f x (foldBack f xs init)
```

We can now define:

```
let sum xs = List.foldBack (+) xs 0
let product xs = List.foldBack (*) xs 1
let all xs = List.foldBack (&&) xs true
let some xs = List.foldBack (||) xs false
```

Can you think of any other functions that can be defined with `List.foldBack`? (an example on next slide)

Remember `words2string`?

```
let rec words2string ws =
  match ws with
  | []        -> ""
  | w :: rest -> w + " " + words2string rest

words2string : string list -> string
```

How define it with `List.foldBack`?

(Solution next slide)

Define

```
let conc_words w1 w2 = w1 + " " + w2
conc_words : string -> string -> string
```

`conc_words` can now be used as a “binary operator” on strings

Now, we can define

```
let words2string ws =
  let conc_words w1 w2 = w1 + " " + w2
  in List.foldBack conc_words ws ""
```

(With nameless functions we could have avoided the explicit declaration of `conc_words`. More about this later)

The `List` module actually contains *two* folds:

`List.foldBack`

`List.fold`, defined as:

```
fold : ('b -> 'a -> 'b) -> 'b -> 'a list -> 'b
let rec fold f init l =
  match l with
  | []    -> init
  | x::xs -> fold f (f init x) xs
```

`List.foldBack` = “fold from the right”

`List.fold` = “fold from the left”

Note the accumulating argument for `List.fold`, where the “sum” is collected

Why two Folds?

Why two folds? Sometimes, one can be more efficient than the other

Also, they have slightly different types, there are cases where one will work but not the other

However, under some conditions they will compute the same answer (more on this later)

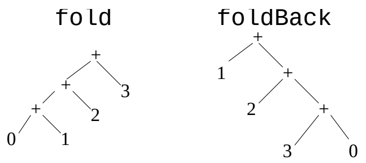
How do the Folds Work?

Let's compare the evaluation of `List.foldBack (+) [1;2;3] 0` and `List.fold (+) 0 [1;2;3]`:

```
List.foldBack (+) [1;2;3] 0 => 1 + List.foldBack (+) [2;3] 0
                    => 1 + (2 + List.foldBack (+) [3] 0)
                    => 1 + (2 + (3 + List.foldBack (+) [] 0))
                    => 1 + (2 + (3 + 0))
                    => 6

List.fold (+) 0 [1;2;3] => List.fold (+) (0 + 1) [2;3]
                    => List.fold (+) ((0 + 1) + 2) [3]
                    => List.fold (+) (((0 + 1) + 2) + 3) []
                    => (((0+ 1) + 2) + 3)
                    => 6
```


Note how `List.fold` and `List.foldBack` build the expression tree in different ways:



Since + is associative these give the same result. The same holds for `*`, &&, `||`.

If the operator is not associative, then `List.fold` and `List.foldBack` can yield different results

Efficiency of List.fold vs. List.foldBack

For operators on atomic types, such as + (`int`, `float`, etc.), and && (`bool`), `List.fold` is more efficient than `List.foldBack`

Reason: since F# is call-by-value, the accumulating argument of `List.fold` will be evaluated for each new call

Therefore, the expression tree never grows higher than one level

Less stack memory is needed to hold the expression tree

Also, `List.fold` is *tail recursive*: a good compiler can compile tail recursive functions into loops

Thus, `sum`, `product`, etc. are better defined as:

```
let sum xs = List.fold (+) 0 xs
let product xs = List.fold (*) 1 xs
let all xs = List.fold (&&) true xs
let some xs = List.fold (||) false xs
```