Higher order functions

Predefined Higher Order Functions

- Three important predefined higher-order functions:
  - `List.map`
  - `List.fold`
  - `List.foldBack`
- List.fold and List.foldBack are similar – not identical
- Defined for List module
- `https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-listmodule.html`

The map Function

Example

```
> List.map (fun i -> -i) [1;2;3];;
val it : int list = [-1; -2; -3]

> List.map (fun x -> x+1) [1;2;3];;
val it : int list = [2; 3; 4]

> List.map (fun (a,b) -> a+b) [(1,2); (3,4)];;
val it : int list = [3; 7]

> List.map (fun i -> i*2) [1..5];;
  val it : int list = [2; 4; 6; 8; 10]
```

- Apply *unary* function to every element of a List, and collect a list of results

Exercise

What are the results?

```
List.map (fun x -> x*x) [1;2;3];;

List.map (fun x -> if x<0 then -x else x) [-1;0;1];;

List.map (fun x -> ("A", x)) [1;2;3];;

let f a L = List.map (fun x -> (a, x)) L;;
f "A" [1;2;3];;
```

The foldBack Function

- Very similar to our `foldback`
- Used to combine all the elements of a list
- Given binary function `f`, a starting value `c`, and a list `x` = $x_1 \ldots x_n$:
- $f(x_1, f(x_2, \ldots f(x_{n-1}, f(x_n, c)) \ldots))$

Example

```
> List.foldBack (fun a b -> a+b) [1;2;3] 0;;
val it : int = 6  // calculates 1+(2+(3+0))
```

foldBack Examples

Example

```
> List.foldBack (fun x c -> x-c) [1;2;3;4] 0;;
val it : int = -2 // (1-(2-(3-(4-0))))
> List.foldBack (fun x c -> if x<c then x else c)
    [4;2;8;5] 9999;;
val it : int = 2  // min function
> List.foldBack (fun x c -> x::c) [1;2;3] [10];;
int list = [1; 2; 3; 10] // 1::(2::(3::[10]))
```

- $f(x_1, f(x_2, \ldots f(x_{n-1}, f(x_n, c))\ldots))$

The fold Function

Example

```
> List.fold (fun c x -> x+c) 0 [1;2;3];;
val it : int = 6

> List.fold (fun c x -> x*c) 1 [1;2;3;4];;
val it : int = 24
```

- Used to combine all the elements of a list
- Same results as `List.foldBack` in certain cases
- Note base case *c* and List elements *x* are switched

fold Functionality

Example

```
> List.fold (fun c x -> x+c) 0 [1;2;3];;
val it : int = 6 // evaluates as ((0+1)+2)+3 = 6
```

- Takes a function `f`, a starting value `c`, list `x` = $(x_1, \ldots, x_n)$ and computes:
- $f(f(\ldots f(f(c, x_1), x_2) \ldots, x_{n-1}), x_n)$
- Remember, `List.foldBack` calculated $1 + (2 + (3 + 0)) = 6$

fold Functions

- `fold` starts at the left, `foldBack` starts at the right
- Difference does not matter when the function is associative and commutative, like + and *
- For other operations, such as – or /, order matters

Example

```
> List.fold (fun x c -> x-c) 0 [1;2;3];;
val it : int = -6  // ((0-1)-2)-3
> List.foldBack (fun x c -> x-c) [1;2;3] 0;;
val it : int = 2 // 1-(2-(3-0))
```

fold Comparisons

Example

```
> let cons a L = a::L;;
> let L = [];;

> List.foldBack (fun x c -> cons x c) [1;2;3] L;;
val it : int list = [1; 2; 3]
// cons (1, cons(2, cons(3, [])))

> List.fold (fun c x -> cons x c) L [1;2;3];;
val it : int list = [3; 2; 1]
// cons (3, cons(2, cons(1, [])))
```

Exercises

- fold: $f(f(\ldots f(f(c,x_1),x_2)\ldots,x_{n-1}),x_n)$
- foldBack: $f(x_1,f(x_2,\ldots f(x_{n-1},f(x_n,c))\ldots))$

Evaluate

```
List.foldBack (fun x c -> x/c) [128;16;4] 1;;

List.fold (fun c x -> x/c) 1 [128;16;4];;

List.fold (fun c x -> x::c) [] [1;2;3;4];;

List.fold
  (fun c x -> List.map (fun a -> a*a) (x::c))
  [] [1;2;3;4];;

let f L = List.fold (fun c x -> x::c) [] L;;
f [(1, "A"); (2, "B"); (3, "C"); (4, "D")];;
```

Exercises

Evaluate

```
let reverse (s:string) =
      new string(Array.rev (s.ToCharArray()));;
let toUpper (s: string) = s.ToUpper();;
let appendBar (s: string) = s + "bar";;

appendBar (toUpper (reverse "Foo"));; //OOFbar
reverse (toUpper (appendBar "Foo"));; //RABOOF

List.fold (fun c xf -> (xf c)) "foo"
  [reverse; toUpper; appendBar];;

List.foldBack (fun xf c -> (xf c))
  [reverse; toUpper; appendBar] "foo";;
```

Tail Recursion and Continuations

Tail recursion

A function call is tail recursive if there is nothing to do after the function returns except return its value.

Example (Not tail recursive)

```
> let rec factorial n =
    match n with
      | 1 -> 1
      | n -> factorial (n-1) * n;;
```

Example (Tail recursive)

```
> let rec tailFact a n =
    match n with
      | 1 -> a
      | n -> tailFact (a*n) (n-1);;
```

Tail recursion

- Non-tail recursion accumulates returned results after recursion.
- Tail recursion accumulates results before recursion.

Equivalent factorial

Example

```
> let rec tailFact a n =
    match n with
      | 1 -> a
      | n -> tailFact (a*n) (n-1);;
> let factorial n = tailFact 1 n;;

> let factorial (n:int):int =
    let rec tailFact a n =
      match n with
      | 1 -> a
      | n -> tailFact (a*n) (n-1)
      in
      tailFact 1 n;;
```

Exercises

Convert to tail-recursive

```
let rec sum L =
  match L with
    | [] -> 0
    | h::t -> h+(sum t);;
```

Convert to tail-recursive

```
let rec rdc L =
  match L with
    | h::[] -> []
    | h::t -> h::rdc t;;
```

Exercise Solution

Example

```
let rec sumTail a L =
  match L with
    | [] -> a
    | h::t -> sumTail (a+h) t;;
```

Example

```
let rec rdcTail a L =
  match L with
    | h::[] -> a
    | h::t -> rdcTail (h::a) t;;
```

Do `rdc` and `rdcTail` work the same?

Tail recursion problem

Example

```
let rec id L =
  match L with
    | [] -> []
    | h::t -> h::id t;;
let rec idTail a L =
  match L with
    | [] -> a
    | h::t -> idTail (h::a) t;;

id [1;2;3;4];; // 1::2::3::4::[]
idTail [] [1;2;3;4];; // 4::3::2::1::[]
```

- Tail recursive but :: is not associative – results are in reverse order.

Using Iteration

FACT(*int n*)

```
1: if n=1 then
2:     fact ← 1
3: else
4:     fact ← FACT(n-1) * n
```

FACTORIAL_ITERATIVE(*int n*)

```
1: int factorial ← 1
2: while n ≠ 1 do
3:     factorial ← factorial * n
4:     n ← n – 1
```

- change *if* to *while*
- change while test to the **not** of *if* test (De Morgan's Laws)
- change recursive call to assignment statement that modifies value of variable controlling *while* loop