Lists and higher-order functions

Dr. Giuseppe Maggiore

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Lecture topics

- Recursive definition of lists as union types
- Recursive traversal of lists with pattern matching
- Generalization of lists to multiple data types
- Generalization of list traversals

```
type IntList =
  | Empty
  | Cons of int * IntList
```

```
let l = Cons(0, Cons(1, Cons(2, Empty)))
```

```
let rec listBetween l u =
  if l > u then Empty
  else
    Cons(l, listBetween (l+1) u)
```

```
let (++) h t = Cons(h, t)
```

```
let l' = 0 ++ (1 ++ (2 ++ Empty))
```

```
let rec listBetween' l u =
  if l > u then Empty
  else
    l ++ listBetween' (l+1) u
```

```
let rec length l =
  match l with
  | Empty -> 0
  | Cons(h,t) -> 1 + length t
```

```
let rec sum l =
  match l with
  | Empty -> 0
  | Cons(h,t) -> h + sum t
```

```
let rec product l =
  match l with
  | Empty -> 1
  | Cons(h,t) -> h * sum t
```

```
let rec add k l =
  match l with
  | Empty -> Empty
  | Cons(h,t) -> (h+k) ++ add k t
```

```
let rec mul k l =
  match l with
  | Empty -> Empty
  | Cons(h,t) -> (h*k) ++ add k t
```

```
let rec negate l =
  match l with
  | Empty -> Empty
  | Cons(h,t) -> (-h) ++ negate t
```

```
let rec removeEven l =
  match l with
  | Empty -> Empty
  | Cons(h,t) when h % 2 = 0 -> removeEven t
  | Cons(h,t) -> h ++ removeEven t
```

```
let rec removeOdd l =
  match l with
  | Empty -> Empty
  | Cons(h,t) when h % 2 = 1 -> removeOdd t
  | Cons(h,t) -> h ++ removeOdd t
```

```
let rec minElem l =
  match l with
  | Empty -> failwith "Empty list has no minimum
      element"
  | Cons(h, Empty) -> h
  | Cons(h,t) -> min h (minElem t)
```

Introduction

Generic lists

- We have only seen lists of int's
- How about lists of strings, float's, tuples?
- How about lists of lists of tuples of lists?
- How about ...

```
type List<'T> =
  | Empty
  | Cons of 'T * List<'T>
```

```
let (++) h t = Cons(h, t)
```

```
let rec length l =
  match l with
  | Empty -> 0
  | Cons(h,t) -> 1 + length t
```

```
let rec sum l =
  match l with
  | Empty -> 0
  | Cons(h,t) -> h + sum t
```

Introduction

Generic list traversals

- There is a lot of repetition in the code above
- We wish to reduce this repetition
- We need to factor out the common parts, leaving the changing bits parameterized
- The parameterization is done with "functions as parameters" (called *higher order functions*)

```
let rec map f l =
  match l with
  | Empty -> Empty
  | Cons(h,t) -> (f h) ++ map f t
```

```
let add k l = map (fun x -> x + k) l
```

```
let mul k l = map (fun x -> x * k) l
```

```
let negate l = map (fun x -> -x) l
```

```
let rec filter p l =
  match l with
  | Empty -> Empty
  | Cons(h,t) when p h -> h ++ filter p t
  | Cons(h,t) -> filter p t
```

```
let removeEven l = filter (fun x -> x % 2 = 0) l
```

```
let removeOdd l = filter (fun x -> x % 2 = 1) l
```

```
let rec fold z f l =
  match l with
  | Empty -> z
  | Cons(h,t) -> f h (fold z f t)
```

```
let sum l = fold 0 (fun x y -> x + y) l
```

```
let product l = fold 1 (fun x y -> x * y) l
```

```
let rec reduce f l =
  match l with
  | Empty -> failwith "Cannot reduce empty list"
  | Cons(h, Empty) -> h
  | Cons(h,t) -> fold h f t
```

```
let minElem l = reduce (fun x y -> min x y) l
```

```
let maxElem l = reduce (fun x y -> max x y) l
```

Introduction

Generic generic list traversals

- `fold` is the most powerful of the list traversals we have seen so far
- It is so powerful that `map` and `filter` can be expressed as `folds`
- (`reduce` is already expressed as a `fold`)

```
let map f l = fold Empty (fun h t -> Cons(f h,t)) l
```

```
let filter p l = fold Empty (fun h t -> if p h then
    Cons(h,t) else t) l
```

Introduction

Built-in lists and sequences

- F# has built-in lists and lazy lists
- They have type `List<'a>` and `Seq<'a>`
- Plus a huge library of combinator functions
- Plus shortcut syntax for some combinators

```
let l = [1..100]
```

```
let l = List.map (fun i -> i * i) [1..100]
```

```
let l =
  [
    for i = 1 to 100 do
    yield i * i
  ]
```

```
let l = List.filter (fun i -> i % 2 = 0) (List.map (
    fun i -> i * i) [1..100])
```

```
let l = [1..100] |> List.map (fun i -> i) |> List.
    filter (fun i -> i % 2 = 0)
```

```
let l =
  [
    for i = 1 to 100 do
      if i % 2 = 0 then
        yield i * i
  ]
```

```
let l =
  [
    for i = 1 to 100 do
    yield i * i
  ]
```

```
let l =
  [
    for i = 1 to 100 do
    for j = 1 to 100 do
    yield i,j
  ]
```

```
let l =
  [
    for i = 1 to 100 do
      if i % 2 = 0 then
        for j = 1 to 100 do
          if j > i then
            yield i,j
  ]
```

```
let l =
  seq{
    for i = 1 to 100 do
      if i % 2 = 0 then
        for j = 1 to 100 do
          if j > i then
            yield i,j
  }
```

Introduction

Conclusions and assignment

- The assignments are on Natschool
- **Restore** the games to a working state
- Hand-in a **printed** report that only contains your **sources** and the associated **documentation**

Introduction

Conclusions and assignment

- Any book on the topic will do
- I did write my own (Friendly F#) that I will be loosely following for the course, **but it is absolutely not mandatory or necessary to pass the course**

Dit is het

The best of luck, and thanks for the attention!