CSCI 2041

Sebastiaan Joosten

September 20th: Folds

University of Minnesota
Driven to Discover®

Overview

- Summing a list
- Multiplying a list
- Reversing a list (inefficiently)
- A generalization (fold_right)
- Reversing more efficiently
- Summing and multiplying
- Another generalization (fold_left)
- Some fold crazyness

Summing a list

You may read this line as:

let rec sum lst = match lst with

```
let rec sum = function
  | [] -> 0
  | h :: t -> h + sum t

val sum : int list -> int = <fun>
```

- sum [1;2]
  = sum [1;2]
  = (function [] -> 0 | h :: t -> h + sum t) [1;2]
  = 1 + sum [2]
  = 1 + (function [] -> 0 | h :: t -> h + sum t) [2]
  = 1 + (2 + sum [])
  = 1 + (2 + 0) = 1 + 2 = 3

Multiplying a list

What do we change?

```
let rec sum = function
  | [] -> 0
  | h :: t -> h + sum t

val sum : int list -> int = <fun>
```

Multiplying a list

What do we change?

```
let rec product = function
  | [] -> 1
  | h :: t -> h * product t

val sum : int list -> int = <fun>
```

Reversing a list

```
let rec reverse = function
```

Reversing a list

```
let rec reverse = function
  | [] ->
  | (h::tl) ->
```

Reversing a list

```
let rec reverse = function
  | [] -> []
  | (h::tl) ->
```

Reversing a list

```
let rec reverse = function
  | [] -> []
  | (h::tl) -> (reverse tl) @ [h]
```

A generalization…

```
let rec sum = function
  | [] -> 0
  | h :: t -> h + sum t
let rec product = function
  | [] -> 1
  | h :: t -> h * product t
let rec reverse = function
  | [] -> []
  | (h::tl) -> (reverse tl) @ [h]
```

A generalization…

```
let rec sum = function
  | [] -> 0
  | h :: t -> h + sum t
let rec product = function
  | [] -> 1
  | h :: t -> h * product t
let rec reverse = function
  | [] -> []
  | h :: tl -> (reverse tl) @ [h]

let rec fold_right = function
  | [] ->
  | h :: tl ->
```

A generalization…

```
let rec sum = function
  | [] -> 0
  | h :: t -> h + sum t
let rec product = function
  | [] -> 1
  | h :: t -> h * product t
let rec reverse = function
  | [] -> []
  | h :: tl -> (reverse tl) @ [h]

let rec fold_right = function
  | [] -> a
  | h :: tl -> f
```

A generalization…

```
let rec sum = function
  | [] -> 0
  | h :: t -> h + sum t
let rec product = function
  | [] -> 1
  | h :: t -> h * product t
let rec reverse = function
  | [] -> []
  | h :: tl -> (reverse tl) @ [h]

let rec fold_right f lst a = match lst with
  | [] -> a
  | h :: tl -> f h (fold_right f tl a)
```

Using fold_right

```
let sum lst = List.fold_right (+) lst 0

let product lst = List.fold_right (*) lst 1

let reverse lst
  = List.fold_right (fun rtl h -> rtl @ [h]) lst []


let rec fold_right f lst a = match lst with
  | [] -> a
  | h :: tl -> f h (fold_right f tl a)
```

A visual way to understand fold_right

- fold_right (») (a::(b::(c::(d::[])))) z

  = a»(b»(c»(d»z)))

Quiz time…

https://tinyurl.com/caml07

```
fold_right (») (a::(b::(c::(d::[])))) z
=                    a»(b»(c»(d»z)))
```

Reversing a list.. why is it inefficient?

```
let rec reverse = function
  | [] -> []
  | (h::tl) -> (reverse tl) @ [h]

reverse [1;2;3]
= reverse [1;2;3]
= reverse [2;3] @ [1]
= (reverse [3] @ [2]) @ [1]
= ((reverse [] @ [3]) @ [2]) @ [1]
= (([] @ [3]) @ [2]) @ [1]
```

Reversing a list: repeated appending

```
let rec (@) lst1 lst2 = match lst1 with
  | [] -> lst2
  | (h :: tl) -> h :: (lst1 @ lst2)

reverse [1;2;3]
= reverse [1;2;3]
= reverse [2;3] @ [1]
= (reverse [3] @ [2]) @ [1]
= ((reverse [] @ [3]) @ [2]) @ [1]
= (([] @ [3]) @ [2]) @ [1]
```

Reversing a list

```
let rec (@) lst1 lst2 = match lst1 with
  | [] -> lst2
  | (h :: tl) -> h :: (lst1 @ lst2)

reverse [1;2;3]
= reverse [1;2;3]
= reverse [2;3] @ [1]
= (reverse [3] @ [2]) @ [1]
= ((reverse [] @ [3]) @ [2]) @ [1]
= (([] @ [3]) @ [2]) @ [1]
= ([3] @ [2]) @ [1]
= (3 :: ([] @ [2])) @ [1]
= [3,2] @ [1] = 3::([2] @ [1]) = 3::(2::([] @ [1]))
= [3,2,1]
```

A more efficient reverse

- The ‘rev_append’ function reverses its first argument, then append its second.
- If we can implement it efficiently, we can define:
  let reverse lst = rev_append lst []

A more efficient reverse

- Quick implementation:
  rev_append lst1 lst2 = (reverse lst1) @ lst2
- Towards a recursive implementation:
  - rev_append [] lst2 = (reverse []) @ lst2
    = [] @ lst2 = lst2
  - rev_append (h::tl) lst2 = (reverse (h::tl)) @ lst2
    = (reverse tl @ [h]) @ lst2
    = reverse tl @ ([h] @ lst2)
    = reverse tl @ (h::lst2) = rev_append tl (h::lst2)

More efficient reverse

- let rec rev_append lst1 lst2 = match lst1 with
  | [] -> lst2
  | (h::tl) -> rev_append tl (h::lst2)
- let reverse lst = rev_append lst []

- (these are List.rev_append and List.reverse)

Summing and multiplying

```
let rec sum_plus lst a = match lst with
  | [] -> a
  | h :: t -> sum_plus t (a + h)

let sum lst = sum_plus lst 0

let rec product_times lst a = match lst with
  | [] -> a
  | h :: t -> product_times t (a * h)

let product lst = product_times lst 1
```

Another generalization

```
let rec sum_plus lst a = match lst with
  | [] -> a
  | h :: t -> sum_plus t (a + h)

let rec product_times lst a = match lst with
  | [] -> a
  | h :: t -> product_times t (a * h)

let rec rev_append lst1 lst2 = match lst1 with
  | [] -> lst2
  | (h::tl) -> rev_append tl (h::lst2)

let rec fold_left f acc lst =
```

Another generalization

```
let rec sum_plus lst a = match lst with
  | [] -> a
  | h :: t -> sum_plus t (a + h)

let rec product_times lst a = match lst with
  | [] -> a
  | h :: t -> product_times t (a * h)

let rec rev_append lst1 lst2 = match lst1 with
  | [] -> lst2
  | (h::tl) -> rev_append tl (h::lst2)

let rec fold_left f acc lst = match lst with
  | [] -> …
  | h::tl ->
```

Another generalization

```
let rec sum_plus lst a = match lst with
  | [] -> a
  | h :: t -> sum_plus t (a + h)

let rec product_times lst a = match lst with
  | [] -> a
  | h :: t -> product_times t (a * h)

let rec rev_append lst1 lst2 = match lst1 with
  | [] -> lst2
  | (h::tl) -> rev_append tl (h::lst2)

let rec fold_left f acc lst = match lst with
  | [] -> acc
  | h::tl -> …
```

Another generalization

```
let rec sum_plus lst a = match lst with
  | [] -> a
  | h :: t -> sum_plus t (a + h)

let rec product_times lst a = match lst with
  | [] -> a
  | h :: t -> product_times t (a * h)

let rec rev_append lst1 lst2 = match lst1 with
  | [] -> lst2
  | (h::tl) -> rev_append tl (h::lst2)

let rec fold_left f acc lst = match lst with
  | [] -> acc
  | h::tl -> fold_left f (…) tl
```

Another generalization

```
let rec sum_plus lst a = match lst with
  | [] -> a
  | h :: t -> sum_plus t (a + h)

let rec product_times lst a = match lst with
  | [] -> a
  | h :: t -> product_times t (a * h)

let rec rev_append lst1 lst2 = match lst1 with
  | [] -> lst2
  | (h::tl) -> rev_append tl (h::lst2)

let rec fold_left f acc lst = match lst with
  | [] -> acc
  | h::tl -> fold_left f (f acc h) tl
```

Using fold_left

```
let sum lst = List.fold_left (+) 0 lst

let product lst = List.fold_left ( * ) 1 lst

let reverse lst
  = List.fold_left (fun x y -> y::x) [] lst

let rec fold_left f acc lst = match lst with
  | [] -> acc
  | h::tl -> fold_left f (f acc h) tl
```

What is fold_left?

- fold_left (») z (a::(b::(c::(d::[])))) = (((z»a)»b)»c)»d
- Note that the parentheses are the other way around…
- This corresponds to traversing in opposite order

- fold_left is very powerful, and can be very efficient, but often slightly harder to work with than fold_right

Fold-crazyness

- ocaml has a function fold_left2, which traverses over two lists, it is defined like this:

```
let rec fold_left2 f accu l1 l2 =
    match (l1, l2) with
      ([], []) -> accu
    | (a1::l1, a2::l2) -> fold_left2 f (f accu a1 a2) l1 l2
    | (_, _) -> invalid_arg "List.fold_left2"
```

- This could have been defined non-recursively as:

```
let fold_left2 f accu lst1 lst2
   = let (acc,[]) = List.fold_left
                (fun (acc, y::tl) x -> (f acc x y, tl))
                (accu, lst2) lst1 in acc
```

- Why wasn't it defined non-recursively like this?

Outlook

- going beyond lists