CMSC 330: Organization of Programming Languages

Maps and Folds
Anonymous Functions
Project 3 and Project 4

• You may use the **List** module
  – There is a link on the “Resources” page
  – For exams, however, you should be able to implement any of the functions in List
    • With the exception of the sorting functions

• You may also use the **Char, String, Printf** and **Scanf** modules
  – Make sure your output matches the project spec!

• You may NOT use the **Array** or **Hashtbl** modules
  – Or any other modules not mentioned here
Review

• sum l (* sum of elts in l *)
  let rec sum l = match l with
    [] -> 0
    | (x::xs) -> x + (sum xs)

• negate l (* negate elements in list *)
  let rec negate l = match l with
    [] -> []
    | (x::xs) -> (-x) :: (negate xs)

• last l (* last element of l *)
  let rec last l = match l with
    [x] -> x
    | (x::xs) -> last xs
Recursion Exercises (cont'd)

(* return a list containing all the elements in the
list l followed by all the elements in list m *)

• append (l, m)

• rev l  (* reverse list; hint: use append *)

• rev takes $O(n^2)$ time. Can you do better?
Recursion Exercises (cont'd)

(* return a list containing all the elements in the list l followed by all the elements in list m *)

• append (l, m)
  let rec append (l, m) = match l with
      [] -> m
    | (x::xs) -> x::(append (xs, m))

• rev l  (* reverse list; hint: use append *)
  let rec rev l = match l with
      [] -> []
    | (x::xs) -> append ((rev xs), [x])

• rev takes $O(n^2)$ time. Can you do better?
A Clever Version of Reverse

```ocaml
let rec rev_helper (l, a) = match l with
  | [] -> a
  | (x::xs) -> rev_helper (xs, (x::a))
let rev l = rev_helper (l, [])
```

- Let’s give it a try
  ```
  rev [1; 2; 3] →
  rev_helper ([1;2;3], []) →
  rev_helper ([2;3], [1]) →
  rev_helper ([3], [2;1]) →
  rev_helper ([], [3;2;1]) →
  [3;2;1]
  ```
More Examples

- `flattenPairs l (* ('a * 'a) list -> 'a list *)`

- `take (n, l) (* return first n elts of l *)`
More Examples

• flattenPairs l (* ('a * 'a) list -> 'a list *)
  let rec flattenPairs l = match l with
  [] -> []
  | ((a, b)::t) -> a :: b :: (flattenPairs t)

• take (n, l) (* return first n elts of l *)
  let rec take (n, l) =
  if n = 0 then []
  else match l with
  [] -> []
  | (x::xs) -> x :: (take (n-1, xs))
Working with Lists

• Several of these examples have the same flavor
  – Walk through the list and do something to every element
  – Walk through the list and keep track of something

• Wouldn’t it be nice to have a generic way of doing this?
Higher-Order Functions

- In OCaml you can pass functions as arguments, and return functions as results

```ocaml
let plus_three x = x + 3
let twice (f, z) = f (f z)
twice (plus_three, 5)       (* returns 11 *)
twice : ('a->'a) * 'a  ->  'a
```
The map Function

• Let’s write the map function
  – Takes a function and a list, applies the function to each element of the list, and returns a list of the results

```ocaml
let rec map (f, l) = match l with
  [] -> []
| (h::t) -> (f h)::(map (f, t))
```

```ocaml
let add_one x = x + 1
let negate x = -x
map (add_one, [1; 2; 3])
map (negate, [9; -5; 0])
```

• Type of map?  
  map : ('a -> 'b) * 'a list -> 'b list
Anonymous Functions

• Passing functions around is very common
  – Often we don’t want to bother to give them names

• Use fun to make a function with no name

\[
\text{fun } x \rightarrow x + 3
\]

\[
\text{map } ((\text{fun } x \rightarrow x + 13), [1; 2; 3])
\]
\[
\text{twice } ((\text{fun } x \rightarrow x + 2), 4)
\]
Pattern Matching with fun

• **match** can be used within **fun**

  ```
  map ((fun l -> match l with (h::_) -> h),
       [[1; 2; 3]; [4; 5; 6; 7]; [8; 9]])
  (* [1; 4; 8] *)
  ```

  – For complicated matches, though, use named functions

• Standard pattern matching abbreviation can be used

  ```
  map ((fun (x, y) -> x + y), [(1, 2); (3, 4)])
  (* [3; 7] *)
  ```
Function Binding

• Functions are first-class, so you can bind them to other names as you like
  – `let f x = x + 3`
  – `let g = f`
  – `g 5 (* returns 8 *)`

• `let` for functions is just a shorthand
  – `let f x = x + 3` stands for
  – `let f = fun x -> x + 3`

– Punchline: all functions are anonymous!
Examples

• let next \( x = x + 1 \)
  – Short for let next = fun x -> x + 1

• let plus (x, y) = x + y
  – Short for let plus = fun (x, y) -> x + y
  – Which is short for
    • let plus = fun z ->
      (match z with (x, y) -> x + y)

• let rec fact n =
  if n = 0 then 1 else n * fact (n-1)
  – Short for let rec fact = fun n ->
    (if n = 0 then 1 else n * fact (n-1))
The fold Function

• Common pattern: iterate through a list and apply a function to each element, keeping track of the partial results computed so far

```ocaml
let rec fold (f, a, l) = match l with
  [] -> a
| (h::t) -> fold (f, f (a, h), t)
```

- a = “accumulator”
  - this is usually called “fold left” to remind us that f takes the accumulator as its first argument

• What's the type of fold?

```
fold : ('a * 'b -> 'a) * 'a * 'b list -> 'a
```
The fold Function

- What if \( f \) took the accumulator as the second argument instead of the first?

```ocaml
let rec fold_right (f, a, l) = match l with
  [] -> a
| (h::t) -> f (h, fold_right (f, a, t))
```

- This is called “fold right”
- It is not tail-recursive, so it is slower than “fold left”
**Example**

```plaintext
let rec fold (f, a, l) = match l with
  [] -> a
| (h::t) -> fold (f, f (a, h), t)
```

```plaintext
let add (a, x) = a + x
fold (add, 0, [1; 2; 3; 4]) →
fold (add, 1, [2; 3; 4]) →
fold (add, 3, [3; 4]) →
fold (add, 6, [4]) →
fold (add, 10, []) →
10
```

We just built the `sum` function!
Another Example

```
let rec fold (f, a, l) = match l with
  [] -> a
  | (h::t) -> fold (f, f (a, h), t)
```

```
let next (a, _) = a + 1
fold (next, 0, [2; 3; 4; 5]) →
fold (next, 1, [3; 4; 5]) →
fold (next, 2, [4; 5]) →
fold (next, 3, [5]) →
fold (next, 4, []) →
4
```

We just built the `length` function!
Using fold to Build rev

\[
\text{let rec fold } (f, a, l) = \begin{cases} 
[] \rightarrow a \\
(h::t) \rightarrow \text{fold } (f, f (a, h), t) 
\end{cases}
\]

- Can you build the reverse function with fold?

\[
\begin{align*}
\text{let prepend } (a, x) & = x::a \\
\text{fold } (\text{prepend}, [], [1; 2; 3; 4]) & \rightarrow \\
\text{fold } (\text{prepend}, [1], [2; 3; 4]) & \rightarrow \\
\text{fold } (\text{prepend}, [2; 1], [3; 4]) & \rightarrow \\
\text{fold } (\text{prepend}, [3; 2; 1], [4]) & \rightarrow \\
\text{fold } (\text{prepend}, [4; 3; 2; 1], []) & \rightarrow \\
[4; 3; 2; 1]
\end{align*}
\]
Basic Partial Evaluation

- Partial evaluation is the process of consuming multiple inputs by creating temporary anonymous functions.

```ml
let add a b = a + b
```

```
add 3 4
= (add 3) 4
= (fun x -> 3 + x) 4
= 3 + 4
= 7
```
Basic Partial Evaluation

• Each input causes a partial evaluation
• This is also called “currying”
• Actually, it creates a slightly more complex structure than just an anonymous function
• These structures are called “closures”

• We’ll talk more about this next week