Recursive Functions

Recursive Functions

```ocaml
# let rec factorial n =  
  if n = 0 then 1 else n * factorial (n - 1);; 
val factorial : int -> int = <fun> 
# factorial 5;; 
- : int = 120 
# (* rec is needed for recursive function declarations *)
```

Recursion Example

```ocaml
Compute \( n^2 \) recursively using:
\[
  n^2 = (2 \times n - 1) + (n - 1)^2
\]

```ocaml
# let rec nthsq n =  
  match n with 0 -> 0 | n -> (2 * n -1) + nthsq (n -1) ;;
val nthsq : int -> int = <fun> 
# nthsq 3;; 
- : int = 9
```

Structure of recursion similar to inductive proof

List

```
List can take one of two forms:
- Empty list, written []
- Non-empty list, written x :: xs
  - x is head element, xs is tail list, :: called “cons”
- Syntactic sugar: [x] == x :: []
- [x1; x2; ...; xn] == x1 :: x2 :: ... :: xn :: []
```

```ocaml
# let fib5 = [8;5;3;2;1;1];; 
val fib5 : int list = [8; 5; 3; 2; 1; 1] 
# let fib6 = 13 :: fib5;; 
val fib6 : int list = [13; 8; 5; 3; 2; 1; 1] 
# (8::5::3::2::1::1::[]) = fib5;; 
- : bool = true 
# fib5 @ fib6;; 
- : int list = [8; 5; 3; 2; 1; 1; 13; 8; 5; 3; 2; 1; 1]
```
Lists are Homogeneous

# let bad_list = [1; 3.2; 7];;
Characters 19-22:
let bad_list = [1; 3.2; 7];;

This expression has type float but is here used with type int

Question

Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [[“hi”; “there”]; [“wahcha”]; [ ]; [“doin”]]

Answer

Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [[“hi”; “there”]; [“wahcha”]; [ ]; [“doin”]]

3 is invalid because of last pair

Functions Over Lists

# let rec double_up list =
match list
with [] -> []  (* pattern before ->, expression after *)
| (x :: xs) -> (x :: x :: double_up xs);;
val double_up : 'a list -> 'a list = <fun>

# let fib5_2 = double_up fib5;;
val fib5_2 : int list = [8; 8; 5; 5; 3; 3; 2; 2; 1; 1; 1; 1]

Functions Over Lists

# let silly = double_up ["hi"; "there"];;
val silly : string list = ["hi"; "hi"; "there"; "there"]

# let rec poor_rev list =
match list
with [] -> []
| (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>

# poor_rev silly;;
val poor_rev_silly : string list = ["there"; "there"; "hi"; "hi"]

Structural Recursion

Functions on recursive datatypes (eg lists) tend to be recursive

Recursion over recursive datatypes generally by structural recursion

- Recursive calls made to components of structure of the same recursive type
- Base cases of recursive types stop the recursion of the function
Question: Length of list

Problem: write code for the length of the list

- How to start?

```ml
let rec length list =
```

- What patterns should we match against?

```ml
let rec length list =
  match list with
    | [] -> 0
    | (a :: bs) ->
```

- What result do we give when list is empty?
Question: Length of list
- Problem: write code for the length of the list
- What result do we give when list is not empty?
  ```ml
  let rec length list =
      match list with [] -> 0
             | (a :: bs) -> 1 + length bs
  ```

Structural Recursion: List Example
```ml
# let rec length list = match list
  with [] -> 0 (* Nil case *)
     | a :: bs -> 1 + length bs;; (* Cons case *)
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```
- Nil case [] is base case
- Cons case recurses on component list bs

Same Length
- How can we efficiently answer if two lists have the same length?
  ```ml
  let rec same_length list1 list2 =
      match list1 with [] -> true
             | (y::ys) -> false
             | (x::xs) -> (match list2 with [] -> false
                             | (y::ys) -> same_length xs ys)
  ```

Your turn: doubleList : int list -> int list
- Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2
  ```ml
  let rec doubleList list =
  ```
Your turn: `doubleList : int list -> int list`

Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2

```ocaml
let rec doubleList list =
  match list with 
  | [] -> []
  | x :: xs -> (2 * x) :: doubleList xs
```

Higher-Order Functions Over Lists

```ocaml
# let rec map f list =
  match list with [] -> []
  | (h::t) -> (f h) :: (map f t);
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x - 1) fib6;;
- : int list = [12; 7; 4; 2; 1; 0; 0]
```

Mapping Recursion

Can use the higher-order recursive map function instead of direct recursion

```ocaml
# let doubleList list =
  List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

Same function, but no explicit recursion

```ocaml
# let doubleList list =
  List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```
Folding Recursion

- Another common form “folds” an operation over the elements of the structure

```ocaml
# let rec multList list = match list
with [ ] -> 1
| x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
```

Computes \((2 \times (4 \times (6 \times 1)))\)

---

Folding Recursion: Length Example

- Let rec length list = match list
  with [ ] -> 0 (* Nil case *)
  | a :: bs -> 1 + length bs;; (* Cons case *)
val length : 'a list -> int = <fun>

```ocaml
# length [5; 4; 3; 2];;
- : int = 4
```

Nil case \([\ ]\) is base case, 0 is the base value
Cons case recurses on component list \(bs\)
What do \texttt{multList} and \texttt{length} have in common?

---

Forward Recursion

- In Structural Recursion, split input into components and (eventually) recurse
- Forward Recursion form of Structural Recursion
- In forward recursion, first call the function recursively on all recursive components, and then build final result from partial results
- Wait until whole structure has been traversed to start building answer

```ocaml
# let rec double_up list =
match list
with [ ] -> [ ]
| (x :: xs) -> (x :: x :: double_up xs);;
val double_up : 'a list -> 'a list = <fun>
```

Base Case Operator Recursive Call

```ocaml
# let rec poor_rev list =
match list
with [ ] -> [ ]
| (x::xs) -> let r = poor_rev xs in r @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```

Base Case Operator Recursive Call

---

Forward Recursion: Examples

```ocaml
# let rec double_up list =
match list
with [ ] -> [ ]
| (x :: xs) -> (x :: x :: double_up xs);;
val double_up : 'a list -> 'a list = <fun>
```

Base Case Operator Recursive Call

```ocaml
# let rec poor_rev list =
match list
with [ ] -> [ ]
| (x::xs) -> let r = poor_rev xs in r @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```

Base Case Operator Recursive Call

---

Recursing over lists

```ocaml
# let rec fold_right f list b =
match list
with [ ] -> b
| (x :: xs) -> f x (fold_right f xs);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

```ocaml
# fold_right
(fun s -> fun () -> print_string s)
["hi"; "there"]
();;
therehi- : unit = ()
```

---

The Primitive Recursion Fairy
Folding Recursion: Length Example

```ocaml
# let rec length list = match list
  with [ ] -> 0 (* Nil case *)
  | a :: bs -> 1 + length bs;; (* Cons case *)
val length : 'a list -> int = <fun>
# let length list =
  fold_right (fun a -> fun r -> 1 + r) list 0;;
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

Folding Recursion

multList folds to the right
Same as:

```ocaml
# let multList list =
  List.fold_right (fun x -> fun p -> x * p)
  list 1;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

Terminology

Available: A function call that can be executed by the current expression
The fastest way to be unavailable is to be guarded by an abstraction (anonymous function, lambda lifted).

- if (h x) then f x else (x + g x)
- if (h x) then (fun x -> f x) else (g (x + x))

Not available

Tail Recursion

A recursive program is tail recursive if all recursive calls are tail calls
Tail recursive programs may be optimized to be implemented as loops, thus removing the function call overhead for the recursive calls
Tail recursion generally requires extra “accumulator” arguments to pass partial results
May require an auxiliary function

```
let length list =
  let rec length_aux list acc_length =
  match list
  with [ ] -> acc_length
  | (x::xs) ->
    length_aux xs (1 + acc_length)
  in length_aux list 0
```

Can replace recursion by fold_right in any forward primitive recursive definition
- Primitive recursive means it only recurses on immediate subcomponents of recursive data structure
- Can replace recursion by fold_left in any tail primitive recursive definition