***Benjamin Fenster***
CS-103 (Skalka)
5 October 2003

# Higher Order Functions

I. List
   a. We want to double all numbers in a list
      i. `let rec double_all l = match l with [] -> [] | (x::xs) -> (2*x)::(double_all(xs))`
      ii. ` : int list -> int list`
   b. Convert all numbers in list to floats
      i. `let rec double_all l = match l with [] -> [] | (x::xs) -> (float x)::float_all(xs)`
      ii. `:int list -> float list`
   c. Structure of these two is nearly identical
   d. Can be seen as an abstract pattern of control. We could define a function that captures that pattern.
   e. `let rec map f l = match l with [] -> [] | (x::xs) -> (f x)::(map f xs)`
      i. NB: This is implemented as List.map in the OCaml library
      ii. `map : (‘a -> ‘b) -> ‘a list -> ‘b list`
      iii. `let double_all = map (fun x -> 2 * x)`
      iv. `let float_all = map (fun x -> float x)`
   f. Example
      i. `let graph = [(1.0, 3.5); (2.2, 4.6); (4.8, 9.2)]`
      ii. `let xcoords = map (fun (x, y) -> x)`
      iii. `xcoords graph ⇓ [1.0; 2.2; 4.8]`
      iv. `graph : ‘a * ‘b list -> ‘a list`

II. Mathematical Induction
   a. `let rec fact n = match n with 0 -> 1 | n -> n * fact(n – 1)`
   b. `let rec expt n = match n with 0 -> 1 | n -> 2 * expt(n – 1)`
   c. Again, very similar structure. Can capture this mathematical induction definition in a higher-order function.
   d. `let rec math_ind basis step n = match n with 0 -> basis | n -> step(n, math_ind basis step (n – 1)`
      i. `let fact = math_ind 1 (fun (n, n’) -> n * n’)`
      ii. `let expt = math_ind 1 (fun (n, n’) -> 2 * n’)`
      iii. `step` defines how we combine the nth element with the result of the recursive call
      iv. `math_ind : ‘a -> ((int * ‘a) -> ‘a) -> int -> ‘a`

III. List Induction
   a. Mathematical induction (i.e. induction on natural numbers) is just a special case of induction.
   b. Now we’ll consider list induction.
   c. Prove that a property p holds for an arbitrary list
      i. Basis: Prove that p([]) holds
      ii. IH: Assume that P(l) holds for some list l
      iii. Induction step: Prove that p(x::l) holds.
   d. Example
      i. let rec length l = match l with [] -> 0 | (x::xs) -> 1 + length(xs)
         1. Basis: Immediate since n = 0 and length [] ⇓ 0 by definition.
         2. IH: length [$v_2$; …; $v_n$] ⇓ n –1
         3. Step: length [$v_1$; …; $v_n$] = 1 + length[$v_2$; …; $v_n$]. By the IH, length [$v_2$; …; $v_n$] ⇓ n – 1 … et cetera
      ii. let rec list_sum l = match l with [] -> 0 | (x::xs)->x + list_sum(xs)
      iii. Again, we’ll generalize it.

1. `let rec list_ind basis step l = match l with [] -> basis | (x::xs) -> step(x, list_ind basis step xs)`
2. `let length = list_ind 0 (fun (x, x’) -> 1 + x’)`
3. `let list_sum = list_ind 0 (fun (x, x’) -> x + x’)`
4. `list_ind : ‘a -> ((‘b*‘a) -> ‘a) -> ‘b list -> ‘a`
5. `length : ‘a list -> int`
6. `list_sum : int list -> int`
7. NB: list_ind is usually called foldr in the community

iv. `let forall p l = foldr true (fun (x, x’) -> p(x) && x’)`

v. `let exists pl = (* left as exercise *)`

e. NB: `->` operator is *right* associative

ERROR: undefinedfilename
OFFENDING COMMAND: </FONT>

STACK: