Lecture 3: Mutual Recursion & Tail Recursion

CS 6371: Advanced Programming Languages

January 21, 2020

`#let rec length =`
`  function [] -> 0 | _::t -> (length t)+1;;`
`length : 'a list -> int = <fun>`
	"function … -> …" is an abbreviation for "fun x -> (match x with … -> …)"

`#type staff = Programmer | Manager of dept`
` and dept = Outsourced | Staffed of staff;;`
`Type staff defined.`
`Type dept defined.`
`#Manager (Staffed Programmer);;`
`- : staff = Manager (Staffed Programmer)`
	Mutually recursive types are separated by the word "and". Notice that there is no ";;" before the "and" and there is no second "type" keyword. You can string as many mutually recursive types together as you wish with "and".

`#let rec staff2str s =`
`  (match s with`
`  Programmer -> "Peon"`
`  | Manager d ->`
`  "Dictator["^(dept2str d)^"]")`
`and dept2str d =`
`  (match d with Outsourced -> "Exiled"`
`  | Staffed s -> staff2str s);;`
`staff2str : staff -> string = <fun>`
`dept2str : dept -> string = <fun>`
	Mutually recursive functions are also defined with "and". The first function in the group begins with "let rec". Each subsequent function begins with "and" (and no "let rec"). The only ";;" appears at the end of the whole group.

`#type 'a btree = BNull`
`  | BNode of ('a * 'a btree * 'a btree);;`
`Type btree defined.`
`#BNode (3,BNull,BNull);;`
`-:int btree = BNode (3, BNull, BNull)`
`#BNode ("foo",BNull,BNull);;`
`-:string btree = BNode ("foo",BNull,BNull)`
`#BNode("foo",BNode(3,BNull,BNull),BNull);;`
`Toplevel input:`
`>BNode("foo",BNode(3,BNull,BNull),BNull);;`
`>  ^^^^^^^^^^^^^^^^^^^`
`This expression has type int btree,`
`but is used with type string btree.`
	Polymorphic variants define a type constructor that is parameterized by a type variable.

`#let rec tree2list t =`
` (match t with`
`  BNull -> []`
`  | BNode (x,t1,t2) ->`
`  (tree2list t1) @ (x::(tree2list t2)));;`
`tree2list : 'a btree -> 'a list = <fun>`
`#tree2list (BNode(1,BNode(2,BNull,BNull),`
`  BNode(3,BNull,BNull)));;`
`- : int list = [2; 1; 3]`
	Here's an example of a function that converts a polymorphic binary tree to a polymorphic list (with list elements given in prefix order). The "@" operator concatenates two lists. This differs from the "::" operator, which inserts an *element* onto the head of a list.

`#let rec fold_left f b l =`
`  (match l with`
`  [] -> b`
`  | h::t -> fold_left f (f b h) t);;`
`fold_left : ('a -> 'b -> 'a) -> 'a -> 'b`
`list -> 'a = <fun>`
`#fold_left (fun x y -> x+y) 0 [1;2;3];;`
`- : int = 6`
	"Fold" is an extremely important list operation in functional programming. (fold_left f b [w;x;y;z]) computes the formula f(f(f(f(b,w),x),y),z). Parameter 'b' is called the "base case".

`#fold_left (fun b x -> b || (x>2))`
`  false [1;2;3];;`
`- : bool = true`
`#let exists f l =`
	From "fold" one can derive many useful list functions, such as existence and forall functions that check if a given condition holds for any or all

`fold_left (fun b x -> b || (f x))`
`false l;;`
`exists:('a->bool)->'a list->bool = <fun>`
`#let for_all f l =`
`fold_left (fun b x -> b && (f x))`
`true l;;`
`for_all:('a->bool)->'a list->bool = <fun>`
`#for_all (fun x -> x>2) [1;2;3];;`
`- : bool = false`
	of the elements of a list.

`#let rec fold_right f l b =`
`(match l with`
`[] -> b`
`| h::t -> f h (fold_right f t b));;`
`fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>`
`#fold_right (fun x y -> x-y) [1;2;3] 0;;`
`- : int = 2`
`#fold_left (fun x y -> x-y) 0 [1;2;3];;`
`- : int = -6`
	There is another operation called "fold_right" that applies function f starting with the rightmost element. That is, (fold_right f [w;x;y;z] b) computes f(w,f(x,f(y,f(z,b)))).

`#let rec fold_left f b l =`
`(match l with`
`[] -> b`
`| h::t -> fold_left f (f b h) t);;`
`#let rec fold_right f l b =`
`(match l with`
`[] -> b`
`| h::t -> f h (fold_right f t b));;`
	A function is "tail recursive" if the value that it returns is the value returned by a direct recursive call to itself. Note that fold_left is tail-recursive but fold_right is not. Try to write tail-recursive functions whenever possible, since these can be optimized much better by functional compilers.

`List.length, List.map, List.fold_left, List.fold_right, List.exists, List.for_all`
 	
`#fst ("foo",3);;`
`- : string = "foo"`
`#snd ("foo",3);;`
`- : int = 3`
	Many of the functions we've defined for lists are defined for you in standard libraries, including the ones listed to the left. The "fst" and "snd" functions are also useful for manipulating pairs.

`#exception ImplErr of string;;`
`Exception ImplErr defined.`
`#raise (ImplErr "Help!");;`
`Uncaught exception: ImplErr "Help!"`
	Exceptions are defined like types, except that you use the keyword "exception" in place of "type". Use the "raise" command to throw an exception.

`#let head l =`
`(match l with x::_ -> x | [] -> raise`
`(ImplErr "head of empty list"));;`
`head : 'a list -> 'a = <fun>`
`#head [1;2;3];;`
`- : int = 1`
`#head [];;`
`Uncaught exception: ImplErr "head of empty list"`
	An expression's type declares its return type IF the function or expression returns normally. When you raise an exception, you don't need to satisfy the return type of the enclosing expression because the expression is not returning normally. **Warning: If you program using exceptions, you lose many of the benefits of functional programming! I recommend avoiding them.**

`#let foo l =`
`(try (head l) with ImplErr _ -> 0);;`
`foo : int list -> int = <fun>`
	Catch exceptions with "try … with …". The "with" part is a pattern-match on the exception type. Each value returned by the right side of an arrow must be of the same type that would be returned if no exception was thrown.