References

```
let imperative_fact n =
  let result = ref 1 in
  for i = 1 to n do
    result := !result * i
  done;
  !result
```

- More complicated than the purely functional version; harder to reasont about
- Considered bad style in a functional language

```
let fact n =
let rec f n result =
  if n = 0 then acc
  else f (n-1) (n*acc)
in
  f n 1
```

```
let rec fact n =
  if n = 0 then 1
  else  n * f (n-1)
```

```
type counter_object =
  {tick : unit -> int ;
   reset: unit -> unit}

let newCounter () =
  let counter = ref 0 in
    {tick = (fun () -> counter := !counter + 1; !counter)
     reset = fun () -> counter := 0}
```

Imperative Append on Linked Lists

```
let rec rapp (r1, r2) = match !r1 with
  | Empty -> r1 := !r2
  | RCons (x,xs) -> rapp (xs, r2)
```

In contrast to our former declarative list append:

```
let rec app l1 l2 = match l1 with
  | [] -> l2
  | x::xs -> x::(app xs l2)
```

Continuations

```
let rec app_tr l1 l2 k =
  match l1 with
  | []    -> k l2
  | h::t -> app_tr t l2 (fun r -> k (h::r))
```

Append_tr uses the heap

```
let rec findAll' p t sc = match t with
  | Empty -> sc []
  | Node(l,d,r) ->
     (if (p d) then
       findAll' p l
         (fun el -> findAll' p r
                      (fun er ->  sc (el@(d::er))))
     else
       findAll' p l
         (fun el -> findAll' p r
                      (fun er ->  sc (el@er)))

     )
```

```
(* VERSION 2: Exceptions *)
let rec find_exc p t = match t with
  | Empty -> raise Fail
  | Node (l,d,r) ->
    if (p d) then Some d
    else (try find_exc p l with Fail -> find_exc p r)

let find_ex p t = (try find_exc p t with Fail -> None)
```

```
(* VERSION 3: Success & failure continuation *)
let rec find_tr p t fail succeed = match t with
  | Empty -> fail ()
  | Node(_, d, _) when p d -> succeed d
  | Node(l, _, r) ->
     find_tr p l
        (fun () -> find_tr p r fail succeed)
        succeed

let find' p t =
  find_tr p t (fun () -> None) (fun x -> Some x)
```

Exceptions

Reference calls to support mutable storage

Continuation is a functional accumulator

```
let rec change coins amt =
  if amt = 0 then  []
  else
    begin match coins with
      | [] -> Raise Change
      | coin::cs -> if coin > amt then
                     Change cs amt
                   else
                     try  Coin :: (change coins  (amt-coin))
                     with Change ->  Change cs amt
    end
```

```
let find_path (g: 'a graph) (a: 'a) (b: 'a) : ('a list * weight) =
  (* Helper function to find the neighbors of a node *)
  let neighbours g vertex =
    List.fold_left (fun acc (v1, v2, w) ->
        if v1 = vertex then (v2, w) :: acc else acc
      ) [] g
  in

  let rec aux_node (node: 'a * int) (visited : 'a list) : ('a list * int) =
    let (current, cost) = node in
    if current = b then ([b], cost)  (* If we've reached the goal, return the path ar
    else
      (* Try to find a path from the list of neighbors *)
      let path, total_cost = aux_list (neighbours g.edges current) (current :: visite
      (current :: path, cost + total_cost)

  and aux_list (nodes: ('a * int) list) (visited: 'a list) : ('a list * int) =
    match nodes with
    | [] -> raise Fail
    | (next, w) :: rest ->
        if List.mem next visited then
          aux_list rest visited  (* Skip if the node is already visited *)
        else
          try aux_node (next, w) visited
          with Fail -> aux_list rest visited  (* Backtrack if path not found *)

  in

  aux_node (a, 0) []  (* Start from node 'a' with an initial cost of 0 *)
```

```
(* Backtracking function to find a path to the goal *)
let rec find_path maze start goal visited =
  if start = goal then [goal]  (* Base case: if start is the goal, return the path *)
  else
    let (row, col) = start in
    let moves = [(row + 1, col); (row - 1, col); (row, col + 1); (row, col - 1)] in
    let rec try_moves = function
      | [] -> raise Not_found  (* No moves left to try, backtrack *)
      | next_pos :: rest ->
          if is_valid_move maze next_pos && not (List.mem next_pos visited) then
            try
              start :: find_path maze next_pos goal (next_pos :: visited)  (* Recursi
            with Not_found -> try_moves rest  (* Backtrack and try the next move *)
          else
            try_moves rest
    in
    try_moves moves
```