Exceptions:

```
let rec find_path d t = match t with
    | Empty -> raise NotFound
    | Node (x, l, r) -> begin
        if x = d then
            H
        else
            try L (find_path d l)
            with NotFound -> R (find_path d r)
    end
let rec change coins amt = if amt = 0 then [] else begin
    match coins with
    | [] -> raise Change
    | coins::cs ->
        if coin > amt then change cs amt
        else try coin :: change coins (amt-coin)
        with Change -> change cs amt end
```

Continuations:

```
let rec appendC l k =
    let rec appendC' l k c = match l with
        | [] -> c k
        | h::t -> appendC' t k (fun r -> c (h::r))
    in
        appendC' l k (fun r -> r)

let rec findPathC d t cont sc = match t with
    | Empty -> cont ()
    | Node (x, l, r) -> begin
        if x = d then sc H
        else
            findPathC d l
            (fun () -> findPathC d r cont
(fun p -> sc (R p)))
            (fun p -> sc (L p))
    end
let findPath d t = findPathC d t (fun () -> raise NotFound) (fun p -> p)

let map f l c m = match l with
  | [] -> c []
  | h::t -> map f t (fun x -> sc ((f h)::x))

let rec cchange coins amt sc fc =
  if amt = 0 then sc []
  else
    match coins with
    | [] -> fc ()
    | coin :: cs -> begin
        if coin > amt then
          cchange cs amt sc fc
        else
            cchange coins (amt - coin)
            (fun cl -> sc (coin :: cl))
            (fun () -> cchange cs amt (fun cl -> sc cl) fc)
      end
```

HOF:

```
let rec eprod a b = List.map (fun be -> List.map (fun ae -> ae * be) a) b
let sum f n = fold_left (fun a b -> a + b) 0 (tabulate n f)
```

```
let nth_catalan n = match n with
    | 0
    | 1 -> 1
    | _ -> sum (fun i -> (nth_catalan i) * (nth_catalan (n - 1 - i))) (n-1)

let rec subtree t interval = begin
    tree_fold
        (fun ((k, v), l, r) -> match compare_interval k interval with
            | Within -> Node ((k, v), l, r)
            | Less -> r
            | Greater -> l)
        t
        Empty
end
```

Lazy programming:

```
type 'a susp = Susp of (unit -> 'a)
type 'a str = { hd: 'a; tl: ('a str) susp }
let force (Susp f) = f ()
let rec mult s1 s2 =
{ hd = s1.hd *. S2.hd; tl = Susp (fun () -> mult (force s1.tl) (force s2.tl)) }
let rec seq i =
{ hd = float (2 * (2 * i + 1)) /. float (i + 2); tl = Susp (fun () -> seq (i + 1)) }
let s = seq 0
let rec get i s = match i with
    | 0 -> s.hd
    | _ -> get (i - 1) (force s.tl)
let rec catalan = { hd = 1.; tl = Susp (fun () -> mult catalan (seq 0)) }
let rec psums s = { hd = s.hd; tl = Susp (fun () -> add (psums s) (force s.tl)) }
let rec fib = { hd = 0; tl = Susp (fun () -> fib') }
and fib' = { hd = 1; tl = Susp (fun () -> add fib fib') }
```

Objects and references:

```
let make_lock () = begin
    let lock = ref Close in
    fun action -> match !lock, action with
        | Open, Close -> lock := Close
        | Close, Open -> lock := Open
        | Open, Open
        | Close, Close -> raise (Error "Bapbapaba")
end
```

Free variables and substitution:

let x = 5 in let y = x + 3 in y + y ≡ let x = 5 in let x = x + 3 in x + x
==> 8 + 8 = 16

Subtyping:

C-produces: S1 <= T1 and S2 <= T2 ==> S1 -> S2 <= T1 -> T2
Records: ^same but from 1 to n
Fun: S1 >= T1 and S2 <= T2 ==> S1 -> S2 <= T1 -> T2
(T1 -> T2 being replaced)
Ref: S = T ==> S <= T

Trees:

```
let rec size t = match t with
    | Empty -> 0
    | Node (a, l, r) -> size l + size r + 1
let rec insert ((x,d) as e) t = match t with
    | Empty -> Node(e, Empty, Empty)
    | Node ((y,d'), l, r) ->
        if x = y then Node(e, l, r)
        else (if x < y then Node((y,d'), insert e l, r)
        else Node((y,d'), l, insert e r))
let rec lookup x t = match t with
    | Empty -> None
    | Node ((y,d), l, r) ->
        if x = y then Some(d)
        else (if x < y then lookup x l
        else lookup x r)
```

Currying:

```
(* curry : (( 'a * 'b) -> 'c) -> 'a -> 'b -> 'c *)
let curry f = (fun x y -> f (x , y ) )
(* uncurry: ('a -> 'b -> 'c) -> (('a * 'b) -> 'c) *)
let uncurry f = (fun (y,x) -> f y x)
```

Prefix compression:

```
let chars_of_string s = begin
    let len = String.length s in
    let rec aux acc = function
        | -1 -> acc
        | n -> aux (s.[n] :: acc) (n - 1)
    in aux [] (len - 1)
end
let string_of_chars cl =
List.fold_left (fun c1 c2 -> c1 ^ (Char.escaped c2)) "" cl

let prefixes l = List.fold_right (fun el ll -> [] :: List.map (fun l -> el :: l) ll) l [[]]
```

Regex matcher

```
type regexp =
  Char of char | Times of regexp * regexp | One | Zero |
  Plus of regexp * regexp | Star of regexp
let rec acc r clist k = match r , clist with
    | Char c        , []    -> false
    | Char c        , c1::s -> (c = c1) && (k s)
    | Times(r1, r2) , s     -> acc r1 s (fun s' -> acc r2 s' k)
    | One           , s     -> k s
    | Plus(r1, r2)  , s     -> acc r1 s k || acc r2 s k
    | Zero          , s     -> false
    | Star r        , s     ->
         (k s) || acc r s (fun s' -> not(s = s') && acc (Star r) s' k)
```