LAP 2024-11

April 23, 2024

1 Linguagens e Ambientes de Programação

1.1 (Aula Teórica 11)

1.1.1 Agenda

- Merge sort
- Arvores n-árias

[ ]:
```
let rec split_n n l =
  if n = 0 then ([], l)
  else match l with
  | [] -> ([], l)
  | x::xs -> let (l1,l2) = split_n (n-1) xs in (x::l1,l2)

let split l =
  let n = List.length l in
  split_n (n/2) l
```

[ ]:
```
val split_n : int -> 'a list -> 'a list * 'a list = <fun>
```

[ ]:
```
val split : 'a list -> 'a list * 'a list = <fun>
```

[ ]:
```
let _ = assert (split [1;2;3;4;5;6] = ([1;2;3], [4;5;6]))
let _ = assert (split_n 0 [1;2;3;4;5;6] = ([], [1;2;3;4;5;6]))
let _ = assert (split_n 1 [1;2;3;4;5;6] = ([1], [2;3;4;5;6]))
let _ = assert (split_n 10 [1;2;3;4;5;6] = ([1;2;3;4;5;6],[]))
```

[ ]:
```
- : unit = ()
```

[ ]:
```
- : unit = ()
```

[ ]:
```
- : unit = ()
```

[ ]:
```
- : unit = ()
```

[ ]:
```
let rec merge l1 l2 =
  match l1, l2 with
  | [], l -> l
  | l, [] -> l
  | x::xs, y::ys -> if x < y then x::(merge xs l2) else y::(merge l1 ys)
```

[ ]:
```
val merge : 'a list -> 'a list -> 'a list = <fun>
```

[ ]:
```
let _ = assert (merge [1;3;5] [2;4;6] = [1;2;3;4;5;6])
let _ = assert (merge [1;3;5;7] [2;6] = [1;2;3;5;6;7])
let _ = assert (merge [] [1;3] = [1;3])
let _ = assert (merge [1;3] [] = [1;3])
```

[ ]:
```
- : unit = ()
```

[ ]:
```
- : unit = ()
```

[ ]:
```
- : unit = ()
```

[ ]:
```
- : unit = ()
```

[ ]:
```
let rec mergesort l =
  match l with
  | [] | [_] -> l
  | _ -> let (l1,l2) = split l in merge (mergesort l1) (mergesort l2)
```

[ ]:
```
val mergesort : 'a list -> 'a list = <fun>
```

[ ]:
```
let _ = assert (mergesort [3; 2; 1; 4; 5; 6; 7; 8; 9; 10] = [1; 2; 3; 4; 5; 6;␣
 ↪7; 8; 9; 10])
```

[ ]:
```
- : unit = ()
```

[ ]:
```
let rec mergesort l =
  let rec split l = match l with
    | [] -> ([], [])
    | [x] -> ([x], [])
    | x::y::tl -> let (l1, l2) = split tl in (x::l1, y::l2)
```

```
  in let rec merge l1 l2 = match (l1, l2) with
    | ([], l) -> l
    | (l, []) -> l
    | (x::tl1, y::tl2) -> if x < y then x::(merge tl1 l2) else y::(merge l1 tl2)
  in match l with
  | [] | [_] -> l
  | _ -> let (l1,l2) = split l in merge (mergesort l1) (mergesort l2)
```

[ ]:
```
val mergesort : 'a list -> 'a list = <fun>
```

[ ]:
```
let _ = assert (mergesort [3; 2; 1; 4; 5; 6; 7; 8; 9; 10] = [1; 2; 3; 4; 5; 6;␣
 ↪7; 8; 9; 10])
```

[ ]:
```
- : unit = ()
```

[ ]:
```
let split_n n l =  (* tail recursive *)
  let rec split_n_rec n l acc =
    if n = 0 then (List.rev acc, l)
    else match l with
    | [] -> (List.rev acc, l)
    | x::xs -> split_n_rec (n-1) xs (x::acc)
  in split_n_rec n l []
```

[ ]:
```
val split_n : int -> 'a list -> 'a list * 'a list = <fun>
```

[ ]:
```
let _ = assert (split_n 0 [1;2;3;4;5;6] = ([], [1;2;3;4;5;6]))
let _ = assert (split_n 1 [1;2;3;4;5;6] = ([1], [2;3;4;5;6]))
let _ = assert (split_n 10 [1;2;3;4;5;6] = ([1;2;3;4;5;6],[]))
```

[ ]:
```
- : unit = ()
```

[ ]:
```
- : unit = ()
```

[ ]:
```
- : unit = ()
```

[ ]:
```
let merge l1 l2 = (* tail recursive *)
  let rec merge_rec l1 l2 acc =
    match l1,l2 with
    | [], l -> (List.rev acc)@l
    | l, [] -> (List.rev acc)@l
    | x::xs, y::ys -> if x < y then merge_rec xs l2 (x::acc) else merge_rec l1␣
 ↪ys (y::acc)
```

```
  in merge_rec l1 l2 []
```

[ ]:
```
val merge : 'a list -> 'a list -> 'a list = <fun>
```

[ ]:
```
let _ = assert (merge [1;3;5] [2;4;6] = [1;2;3;4;5;6])
let _ = assert (merge [1;3;5;7] [2;6] = [1;2;3;5;6;7])
let _ = assert (merge [] [1;3] = [1;3])
let _ = assert (merge [1;3] [] = [1;3])
```

[ ]:
```
- : unit = ()
```

[ ]:
```
- : unit = ()
```

[ ]:
```
- : unit = ()
```

[ ]:
```
- : unit = ()
```

1.2 BST

[ ]:
```
type 'a bst = Leaf | Node of 'a * 'a bst * 'a bst
```

[ ]:
```
type 'a bst = Leaf | Node of 'a * 'a bst * 'a bst
```

[ ]:
```
let rec insert x t =
  match t with
  | Leaf -> Node(x, Leaf, Leaf)
  | Node(y, l, r) -> if x < y then Node(y, insert x l, r) else Node(y, l,␣
 ↪insert x r)
```

[ ]:
```
val insert : 'a -> 'a bst -> 'a bst = <fun>
```

[ ]:
```
let rec remove_min t =
  match t with
  | Leaf -> failwith "Tree is empty"
  | Node(y, Leaf, r) -> (y, r)
  | Node(y, l, r) -> let (m, l') = remove_min l in (m, Node(y, l', r))
```

[ ]:
```
val remove_min : 'a bst -> 'a * 'a bst = <fun>
```

```
let _ = assert (remove_min (Node(5, Node(3, Leaf, Leaf), Node(7, Leaf, Leaf)))
  ↪= (3, Node(5, Leaf, Node(7, Leaf, Leaf))))
```

```
- : unit = ()
```

```
let rec remove x t =
  match t with
  | Leaf -> Leaf
  | Node(y,Leaf,r) when x = y -> r (* maybe superfluous *)
  | Node(y,l,Leaf) when x = y -> l
  | Node(y,l,r) when x = y -> let (m, r') = remove_min r in Node(m, l, r')
  | Node(y,l,r) -> if x < y then Node(y, remove x l, r) else Node(y, l, remove
  ↪x r)
```

```
val remove : 'a -> 'a bst -> 'a bst = <fun>
```

```
let _ = assert (remove 3 (Node(5, Node(3, Leaf, Leaf), Node(7, Leaf, Leaf))) =
  ↪Node(5, Leaf, Node(7, Leaf, Leaf)))
let _ = assert (remove 7 (Node(5, Node(3, Leaf, Leaf), Node(7, Leaf,
  ↪Node(9,Leaf,Leaf)))) = Node(5, Node(3, Leaf, Leaf), Node(9, Leaf, Leaf)))
```

```
- : unit = ()
```

```
- : unit = ()
```

1.3 Árvores n-árias

```
type 'a ntree = Leaf | Node of 'a * 'a ntree list

let nt = Node (1, [Node (2, [Node (3, [])]); Node (4, [])])
```

```
type 'a ntree = Leaf | Node of 'a * 'a ntree list
```

```
val nt : int ntree = Node (1, [Node (2, [Node (3, [])]); Node (4, [])])
```

```
let rec sum_tree t =
  match t with
  | Leaf -> 0
  | Node (v, l) -> v + List.fold_left (fun acc t -> acc + sum_tree t) 0 l
```

```
val sum_tree : int ntree -> int = <fun>
```

[ ]:
```
let _ = sum_tree nt
```

[ ]:
```
- : int = 10
```

[ ]:
```
let rec map_ntree f nt =
  match nt with
  | Leaf -> Leaf
  | Node (v, l) -> Node (f v, List.map (map_ntree f) l)
```

[ ]:
```
val map_ntree : ('a -> 'b) -> 'a ntree -> 'b ntree = <fun>
```

[ ]:
```
let _ = map_ntree ((+) 1) nt
```

[ ]:
```
- : int ntree = Node (2, [Node (3, [Node (4, [])]); Node (5, [])])
```

[ ]:
```
let rec prefix_fold_ntree f acc nt =
  match nt with
  | Leaf -> acc
  | Node (v, l) -> let acc_n = f acc v in List.fold_left (fun acc t ->␣
 ↪prefix_fold_ntree f acc t) acc_n l
```

[ ]:
```
val prefix_fold_ntree : ('a -> 'b -> 'a) -> 'a -> 'b ntree -> 'a = <fun>
```

[ ]:
```
let _ = prefix_fold_ntree (+) 0 nt
```

[ ]:
```
- : int = 10
```