Notes Tps Pf ;

```ocaml
let rec padovan n =
match n with
| 0 -> 0
| 1 -> 0
| 2 -> 1
| _ -> padovan (n-2) + padovan (n-3)

let estPremier n =
if n <= 1 then false
else
let rec aux d =
(d*d) <= n && ((n mod d = 0) || (aux (d+1)))
in not (aux 2)

let rec insert ordre elt l =
match l with
| [] -> [elt]
| t::q -> if (ordre elt t) then elt::l
else t::insert ordre elt q

let rec tri_insertion ordre l =
match l with
| [] -> []
| t::q -> insert ordre t (tri_insertion ordre q)

let rec scinde l =
match l with
| [] -> ([], [])
| [t] -> ([t], [])
| t::e::q -> let (l1, l2) = (scinde q) in ((t::l1), (e::l2))

let rec fusionne ordre l1 l2 =
match l1 with
| [] -> l2
| t::q -> fusionne ordre q (insert ordre t l2)
let rec tri_fusion ordre l =
match l with
| [] -> []
| [_] -> l
| _ -> let (l1, l2) = scinde l in fusionne ordre (tri_fusion ordre l1) (tri_fusion ordre l2)

let partition n =
let rec aux p t =
if p = t then [[t]]
else if p < t then []
else
(List.map (fun s -> t::s) (aux (p-t) t))@(aux p (t+1))
(* in if n = 1 then [[1]] else aux n 1 *)
in aux n 1
```

```
let rec permutations l =
match l with
| [] -> [[]]
| t::q -> List.flatten(List.map (fun s -> insertion t s)(permutations q))

let rec insertion e l =
match l with
| [] -> [[e]]
| t::q -> [e::l]@(List.map (fun s -> t::s) (insertion e q))

let rec combinaison k l =
match (k, l) with
| (0, _) -> [[]]
| (_, []) -> []
| (k, t::q) -> (List.map (fun s -> t::s) (combinaison (k-1) q))@(combinaison k q)

let rec gray_code n =
match n with
| 0 -> [[]]
| _ -> let q = gray_code (n-1) in (List.map (fun l -> (0::l)) q)@(List.map (fun l -> (1::l)) (List.rev q))

let rec retrait_arbre lc (Noeud (b, lb)) =
match lc with
| [] -> Noeud (false, lb)
| tc::qc ->
let newArbre =
let l = recherche tc lb in
match l with
| None -> Noeud (false, [])
| Some a -> a
in Noeud (b, maj tc (retrait_arbre qc newArbre) lb)

let rec parcours_arbre (Noeud (b, lb)) =
if b then
([]::List.flatten(List.map (fun x -> let (c, arbre) = x in List.map (fun x -> c::x ) (parcours_arbre arbre)) lb))
else (List.flatten(List.map (fun x -> let (c, arbre) = x in List.map ( fun x -> c::x) (parcours_arbre arbre))
lb) )

let rec normaliser (Noeud (b, lb)) =
if lb = [] then Noeud(b, lb)
else
let lbNorm =
List.fold_right (fun (c, sArb) qlb -> let normArb = normaliser sArb in if normArb = Noeud(false,[]) then qlb
else (c, normArb)::qlb) lb [] in
if (not b) && (lbNorm = []) then
Noeud(false,[])
else
Noeud(b, lbNorm)
type 'a hlist =
| Nil : nil hlist
| Cons : 'p * 'a hlist -> ('p * 'a) hlist
```

```
let rec tail : type a p. (a * 'p) hlist -> 'p hlist = function
| Cons (_, q) -> q

let%test _ = tail Cons (1, Cons (true, Nil)) = Cons (true, Nil)

let add : type a. (int * (int * a)) hlist -> (int * a) hlist = function
| Cons (t1, Cons (t2, q)) -> Cons (t1+t2, q)

type 'a t = 'a flux = Tick of ('a * 'a t) option Lazy.t;;

let vide = Tick (lazy None);;

let cons t q = Tick (lazy (Some (t, q)));;

let uncons (Tick flux) = Lazy.force flux;;
let rec apply f x =
Tick (lazy (
match uncons f, uncons x with
| None , _ -> None
| _ , None -> None
| Some (tf, qf), Some (tx, qx) -> Some (tf tx, apply qf qx)));;

let rec unfold f e =
Tick (lazy (
match f e with
| None -> None
| Some (t, e') -> Some (t, unfold f e')));;

let rec filter p flux =
Tick (lazy (
match uncons flux with
| None -> None
| Some (t, q) -> if p t then Some (t, filter p q)
else uncons (filter p q)));;
let rec append flux1 flux2 =
Tick (lazy (
match uncons flux1 with
| None -> uncons flux2
| Some (t1, q1) -> Some (t1, append q1 flux2)));;

type 'a t = unit -> 'a option

(* à compléter *)
let ( ++ ) iter1 iter2 =
fun () -> if Random.bool ()
then
match iter1 () with
| None -> iter2 ()
| r -> r
else
```

```
match iter2 () with
| None -> iter1 ()
| r -> r
let zero = fun () -> None

let ( >>= ) iter f =
fun () ->
match iter () with
| None -> None
| Some a -> f a ()

let return iter = fun () -> Some iter

let map f iter =
fun () ->
match iter () with
| None -> None
| Some a -> Some (f a)

type res =
| Fork of (unit -> unit) * (unit -> res)
| Yield of (unit -> res)
| Done;;

(* Fonction de continuation : k : unit->res pour le syntaxe de shift*)
let prompt0 = new_prompt ();;
(* handle : interpreter *)
let scheduler proc_init =
(* des queue de (unit -> res ) Queue.t*)
let queue = Queue.create () in
let rec handle result =
match result with
| Done -> if Queue.is_empty queue then () else
let k = Queue.pop queue in handle (k ())
(* [p] -> p [] -> [p]*)
| Yield k -> Queue.push k queue; let k' = Queue.pop queue in handle (k' ())
| Fork (p, k) -> Queue.push k queue; run p

and run prog =
handle (Delimcc.push_prompt prompt0 (fun () -> prog (); Done))
in run prog_init
(* j'autorise le scheduler de prendre la main (j'arrete un petit
moment et je revient aprés)*)
let yield () = Delimcc.shift prompt0 (fun k -> Yield k);;
let fork proc = Delimcc.shift prompt0 (fun k -> Fork (proc, k));;
let exit () = Delimcc.shift prompt0 (fun _ -> Done);;

(* Grammaire LL1 des programmes LOGO:
P -> begin I end
I -> /\
I -> C ; I
```

```
C -> repeat entier P
C -> move entier
C -> turn entier
C -> on
C -> off
*)
(* les parsers mutuellement récursifs pour la grammaire ci-dessus: à compléter *)
let rec parse_P : (char, prog) parser = fun flux ->
(
p_begin >>= (fun _ -> parse_I >>= (fun i -> p_end >>= (fun _ -> return i)))
) flux
and parse_I : (char, inst) parser = fun flux ->
(
((return []) ++ (parse_C >>= (fun cmd -> p_ptvirg >>= (fun _ -> parse_I >>= (fun i -> return
(cmd::i))))))
) flux
and parse_C : (char, cmd) parser = fun flux ->
(
(
(p_off >>= (fun _ -> return Off)) ++
(p_on >>= (fun _ -> return On)) ++
(p_turn >>= (fun _ -> p_entier >>= (fun cmd -> return (Turn cmd)))) ++
(p_move >>= (fun _ -> p_entier >>= (fun cmd -> return (Move cmd)))) ++
(p_repeat >>= (fun _ -> p_entier >>= (fun n -> parse_P >>= (fun p -> return (Repeat (n, p)))))))
) flux
```

```
let rec fold_right f l e =
match l with
| Nil –> e
| Cons (t, q) –> f t (fold_right f q e)
| Append (l, r) –> fold_right f l (fold_right f r e)
```

```
type res =
| Done
| Send of int * (unit –> res)
| Receive of ( int –> res);;
let p = Delimcc.new_prompt ();;
let send v = Delimcc.shift p (fun k –> Send (v, k));;
let receive () = Delimcc.shift p (fun k –> Receive k);;
let scheduler procs =
let rec loop result others rcvs medium =
match result with
| Done –> comm others rcvs medium
| Send (v, k) –> comm (k::others) rcvs (medium@[v])
| Receive k –> comm others (rcvs@[k]) medium
and comm others rcvs medium =
match rcvs, medium, others with
| kr :: qr, vm::qm, _ –> loop (kr vm) others qr qm
| _ , _ , o :: qo –> loop (o ()) qo rcvs medium
| _ , _ , [] –> Done
in ignore (Delimcc.push_prompt p (fun () –> loop Done procs [] []));;
```

```
let map : ('a -> 'b) -> 'a maplist -> 'b maplist =
  fun f l ->
    let rec aux acc l =
      match l with
      | Nil -> acc
      | Cons (x, rest) -> aux (Cons (f x, acc)) rest
```

```
      | ConsF (g, x, rest) -> aux (ConsF (fun a -> f (g a), x, acc)) rest
    in
    aux Nil l
```

```
type ('a, 'b) maplist = Cons of 'a * ('a, 'b) maplist | ConsF of ('a -> 'b) * 'a * ('a, 'b) maplist | Nil

let uncons : ('a, 'b) maplist -> ('a * ('a, 'b) maplist) option = function
  | Cons (x, xs) -> Some (x, xs)
  | ConsF (f, x, xs) -> Some (f x, xs)
  | Nil -> None

let to_list : 'a maplist -> 'a list =
  fun l ->
    let rec aux acc l =
      match l with
      | Nil -> List.rev acc
      | Cons (x, rest) -> aux (x :: acc) rest
      | ConsF (g, x, rest) -> aux (g x :: List.map g (to_list rest) @ acc) Nil
    in
    aux [] l

let fold_right : ('a -> 'b -> 'b) -> 'a maplist -> 'b -> 'b =
  fun f l acc ->
    match l with
    | Nil -> acc
    | Cons (x, rest) -> f x (fold_right f rest acc)
    | ConsF (g, x, rest) -> f (g x) (List.fold_right g (to_list rest) acc)
```

```
let rec combinaisons k l =
match l, k with
| [], _ -> ND.zero
| _, 0 -> ND.return []
| t::q, _ -> ND.(combinaisons k q ++ (combinaisons (k-1) q >>= fun combinaison -> return
(t::combinaison)))

let rec insertion e l =
match l with
| [] -> ND.return [e]
| t::q -> ND.(return (e::l) ++ (insertion e q >>= fun inser -> return (t::inser)))

let rec permutations l =
match l with
| [] -> ND.return []
| t::q -> ND.(permutations q >>= (fun perm -> insertion t perm))


let partitions n =
let rec aux_partitions p t =
if p = t then ND.return [t]
else if p < t then ND.zero
else ND.((aux_partitions (p-1) t >>= fun partition -> return (t::partition)) ++ aux_partitions p (t+1))
in aux_partitions n 1
```