```
(* TP Syntaxe et Sémantique *)

(* Exemple : expressions logiques *)

type prop =  Vrai | Faux | A | B
           | Not of prop
           | Ou of prop * prop
           | Et of prop * prop
           | Imp of prop * prop
           | Eq of prop * prop ;;


let exple = Eq ( Imp  ( Not(A),
                        Not(B)
                      ),
                 Imp ( B,
                       A
                      )
               ) ;;

(* L'exemple représente le raisonnement par contraposée. *)


(* Partie I : Calcul formel *)

type expr =  Var of char
           | Const of float
           | Exp of expr
           | Somme of expr * expr
           | Produit of expr * expr ;;

(* 1 *)

let rec calcule e = match e with
  |Var _ -> failwith "variable"
  |Const a -> a
  |Exp e1 -> exp (calcule e1)
  |Somme (e1,e2) -> (calcule e1) +. (calcule e2)
  |Produit (e1,e2) -> (calcule e1) *. (calcule e2);;

(* 2 *)
let rec evalue e phi = match e with
  |Var x -> phi x
  |Const a -> a
  |Exp e1 -> exp (evalue e1 phi)
  |Somme (e1,e2) -> (evalue e1 phi) +. (evalue e2 phi)
  |Produit (e1,e2) -> (evalue e1 phi) *. (evalue e2 phi);;
```

```
(* 3 *)
let rec derive x e = match e with
  |Var c when c=x -> Const 1.
  |Var c  -> Const 0.
  |Const _ -> Const 0.
  |Exp a -> Produit (derive x a,Exp a)
  |Somme(a,b) -> Somme(derive x a,derive x b)
  |Produit(a,b) -> Somme(Produit(derive x a,b),Produit(a,derive x b));;

(* 4 *)
let exple4 = (Exp (Somme (Var 'x',Const 1.)));;
let res_exple4 = derive 'x' exple4 ;;

(* Caml renvoie :
 - : expr = Produit (Somme (Const 1.0, Const 0.0), Exp (Somme (Var 'x', Const 1.0)))
 Cette forme est bien compliquee, on pourrait faire des simplifications ! Par
 exemple, Produit ((Const 1.),a) pourrait se simplifier en a...
*)

let rec simplifie e = match e with
  |Var _ -> e
  |Const _ -> e
  |Exp a -> begin match (simplifie a) with
      |Const x -> Const (exp x)
      |b -> Exp b
      end
  |Somme (a,b) -> begin match (simplifie a, simplifie b) with
      |Const x, Const y -> Const (x +. y)
      |Const 0., bb -> bb
      |aa, Const 0. -> aa
      |aa,bb -> Somme (aa,bb)
      end
  |Produit (a,b) -> begin match (simplifie a, simplifie b) with
      |Const x, Const y -> Const (x *. y)
      |Const 0., _ -> Const 0.
      |_,Const 0. -> Const 0.
      |Const 1.,bb -> bb
      |aa, Const 1. -> aa
      |aa,bb -> Produit (aa,bb)
      end;;



simplifie (derive 'x' exple4);;
```

```
(* Partie II. Syntaxe concrète *)

type expr =  C of string
           | U of string * expr
           | B of string * expr * expr ;;

(* 5 *)

(* On trouve : "x 1 + exp" *)

(* 6 *)
let rec ecritpost e = match e with
  |C s -> print_string s
  |U (s,e1) -> ecritpost e1; print_string s
  |B (s,e1,e2) -> ecritpost e1; ecritpost e2; print_string s;;


let exple5 = U ("exp", B("+", C "x", C "1" ));;

ecritpost exple5;;

(* 7 *)
(* On peut imaginger deux solutions : utiliser une reference a la pile p qui se modifie a
 mesure qu'on lit la liste (avec une fonction recursive), ou utiliser une
 fonction recursive qui prend en argument la pile et la liste.*)

(* Premiere solution, plus longue mais assez naturelle : *)

let arbre_de_liste l=
  let p=ref [] in
  let rec traite l = match l with
    |[] -> ()
    |(0,s)::ll -> p:= (C s)::!p ; traite ll
    |(1,s)::ll -> let t = List.hd !p
                  and pp = List.tl !p
                  in p:=U(s,t)::pp ; traite ll
    |(2,s)::ll -> let  t2 = List.hd !p
                   and t1 = List.hd(List.tl !p)
                   and pp = List.tl(List.tl !p) in
                  p:=B(s,t1,t2)::pp ; traite ll
    |_ -> failwith "forme postfixe incorrecte"
  in
  traite l;
  match !p with
    | [x] -> x
    | _ -> failwith "forme incorrecte"
  ;;
```

```
let exple7 = [(0,"x") ; (0,"1") ; (2,"+"); (1,"exp") ];;

arbre_de_liste exple7 ;;

(* Deuxieme solution, plus concise mais il faut y penser : *)

let arbre_de_liste2 l =
  let rec traite l p = match (l,p) with
    |[], t::[] -> t (* La liste est vide, donc la procedure est finie *)
    |(0,s)::ll, _ -> traite ll ((C s)::p)
    |(1,s)::ll, t::pp -> traite ll ((U (s,t))::pp)
    |(2,s)::ll, t2::t1::pp -> traite ll ((B (s,t1,t2))::pp)
    |_ -> failwith "forme postfixe incorrecte"
  in
  traite l [];;


arbre_de_liste2 exple7;;

(* 8 *)
(* On a besoin d'une sous-fonction "litmot" qui prend en argument une chaine de
 caracteres et un indice, et renvoie le mot commencant a cet indice et
 s'arretant au premier 'C', 'U' ou 'B' trouve (cette derniere lettre n'apparait
 pas dans le mot renvoye). Cette sous-fonction renverra aussi la taille du mot
 qu'on renvoie.
 On ecrit ensuite une fonction recursive "aux" qui prend en argument un indice
 et renvoie la liste obtenue pour le sous-mot commencant a cet indice.
*)

let parse s =
  let n = String.length s in
  let litmot i =
    let t = ref 0 in
    while i+ !t<n && s.[i+ !t]!='U' && s.[i+ !t]!='B' && s.[i+ !t]!='C' do
      incr t
    done;
    String.sub s i !t, !t
  in
  let rec aux i = match i with
   | _ when i=n -> []
   | _     -> begin
    let (mot,t) = litmot (i+1) in match s.[i] with
|'C'-> (0,mot) :: (aux (i+t+1))
|'U'-> (1,mot) :: (aux (i+t+1))
|'B'-> (2,mot) :: (aux (i+t+1))
|_ -> failwith "forme incorrecte"
     end
  in
  aux 0;;
```

```
let arbre_de_chaine s = arbre_de_liste (parse s);;


arbre_de_chaine "CxC1B+Uexp";;

(* 9 *)
(* Non, par exemple, l'expression infixe "1 - 2" s'ecrit en postfixe "1 2 -" et
 en prefixe "- 1 2". *)

(* 10 *)
(* On ecrit une sous-fonction recursive qui fait ce qui est dit dans l'indication. *)

let arbre_de_liste3 l =
  let rec debut l = match l with
    |(0,s)::q -> C s, q
    |(1,s)::q -> let (a,q1)=debut q in U(s,a),q1
    |(2,s)::q -> let (a1,q1)=debut q in let (a2,q2)=debut q1 in B(s,a1,a2), q2
    |_ -> failwith "forme prefixe incorrecte"
  in
  match (debut l) with
    |a,[] -> a
    |_ -> failwith "forme prefixe incorrecte";;


arbre_de_liste3 [(1, "exp") ; (2,"+") ; (0, "x") ; (0 , "1") ];;


(* Partie III. Expressions rationnelles *)

(* 11 *)

type regexp = Vide | Epsilon
            | Lettre of char
            | Etoile of regexp
            | Union of regexp * regexp
            | Concat of regexp * regexp ;;


(* 12 *)
let rec avide e = match e with
  |Vide -> false
  |Epsilon -> true
  |Lettre _ -> false
  |Etoile _ -> true
  |Union (a1,a2) -> (avide a1) || (avide a2)
  |Concat (a1,a2) -> (avide a1) && (avide a2);;
```

```
(* 13 *)
let arden a b = match avide a with
  |true -> failwith "probleme trop difficile"
  |false -> Concat (Etoile a, b);;

(* 14 *)
(* On va resoudre le systeme par substitution. Grace au lemme d'Arden applique a
 la premiere equation, on exprime X1 en fonction des autres langages :
 X1 = A11*(A12 X2 + ... + A1n Xn + B1).
 On remplace X1 par cette ecriture dans toutes les autres lignes, et on applique
 les regles habituelles de langages, pour obtenir un systeme sur X2, ..., Xn.
 On verifie facilement que ce nouveau systeme verifie toujours la propriete "le
 mot vide n'appartient pas aux A'ij". On peut donc recommencer pour X2,
 substituer dans le systeme, etc.

 On obtient la procedure suivante, ou les indices commencent a 1 comme dans
 l'enonce :
  Pour k de 1 a n-1 (on va substituer Xk dans la suite)
   Pour i de k+1 a n (on substitue dans la ligne i)
    Pour j de k+1 a n
     Aij <- Aik.Akk*.Akj + Aij
    Fin
    Bi <- Aik.Akk*.Bk + Bi
   Fin
  Fin

 A la fin de cet algorithme, on peut appliquer encore le lemme d'Arden pour
 trouver Xn = Ann*Bn, puis, sur la n-1eme ligne, on trouve
 X(n-1) = A(n-1)(n-1)* ( A(n-1)n.Xn + B(n-1) ), etc. de maniere generale, on a
 Xk=Akk*. (Ak(k+1).X(k+1) + ... + Akn.Xn + Bk) ce qui permet de trouver les Xk
 par ordre decroissant via la procedure :
  On initialise le tableau Xk a Bk, puis :
  Pour k de n a 1 en ordre decroissant (c'est important)
   Pour i de k+1 a n
    Xk <- Xk + Aki.Xi
   Fin
   Xk <- Akk*.Xk
  Fin

 En fait au lieu de creer un tableau des Xk pour faire ca, on peut le faire
 directement sur le tableau des Bk et renvoyer le resultat !
 Pour coder tout ca, je vais diminuer de 1 les indices des procedures donnees en
 pseudo-code pour me retrouver entre 0 et n-1. Comme on veut modifier des
 tableaux, j'ecris une fonction "copie_tab" et une fonction "copie_mat" qui
 renvoient un copie de, respectivement, un tableau et une matrice de langages.
 Ca evite de modifier les tableaux qu'on me donne en argument.
*)
```

```
let copie_tab v =
  let n = Array.length v in
  let v2 = Array.make n Vide in
  for i=0 to n-1 do
    v2.(i) <- v.(i)
  done;
  v2;;

let copie_mat m =
  let n = Array.length m in
  let m2 = Array.make n [||] in
  for i=0 to n-1 do
    m2.(i) <- copie_tab m.(i)
  done;
  m2;;

let syst a1 b1 =
  let a=copie_mat a1 and b=copie_tab b1 in
  let n= Array.length a in
  for k=0 to n-2 do
    for i=k+1 to n-1 do
      for j=k+1 to n-1 do
        a.(i).(j) <- Union (a.(i).(j), Concat(a.(i).(k),Concat(Etoile a.(k).(k), a.(k).(j)))
      done;
      b.(i) <- Union (b.(i), Concat(a.(i).(k),Concat(Etoile a.(k).(k), b.(k))))
    done
  done;
  for k=n-1 downto 0 do
    for i=k+1 to n-1 do
      b.(k) <- Union(b.(k), Concat(a.(k).(i),b.(i)))
    done;
    b.(k) <- Concat(Etoile a.(k).(k), b.(k))
  done;
  b;;

(* 15 *)
(* Pour tous les etats i de l'automate, on appelle Xi le langage reconnu par
 l'automate "en partant de i". Alors on peut etablir un systeme sur les Xi
 semblable au systeme precedent : pour toute fleche etiquetee par 'a' entre
 l'etat i et un etat j, on a Xi = aXj + (le reste). On ecrit comme ca une somme.
 Si i est un etat terminal, il faut aussi ajouter "epsilon" a cette somme. C'est
 bien un systeme verifiant les hypotheses du 4 : les Aij sont vides ou reduits a
 des lettres, et les Bi sont vides ou reduits a "epsilon". On resout ce systeme,
 et le langage reconnu par l'automate est alors l'union des Xi pour les etats
 initiaux i.
```

```
 Exercice : ecrire l'algorithme qui trouve le langage reconnu par un automate !
 Vous pouvez prendre la representation d'un automate que vous preferez
 (attention, il n'est pas suppose deterministe ou complet).

 Bien sur, le resultat risque d'etre assez sale et il faut encore ecrire une
 fonction de simplification. On peut aussi effectuer les simplifications a la
 volee, au moment de resoudre le systeme dans la fonction syst : par exemple, au
 lieu de faire betement le constructeur Union, on peut appeler une fonction
 union qui fait une union intelligente : ne rien faire si on fait l'union avec
 le vide, ne rien faire non plus si on ajoute epsilon a un langage qui le
 contient,...
*)
```