Correction du devoir du Vendredi 13 Juin

Exercice 1 (Du cours)

1. (a) Voici les primitives d'accès demandées :

```
let creer_pile () = {contenu = []} ;;

let etre_pile_vide p = match p.contenu with
    | [] -> true
    | _ -> false
;;

let empiler e p = p.contenu <- e::p.contenu ;;

let depiler p = match p.contenu with
    | [] -> failwith "pile vide"
    | t::q -> p.contenu <- q; t
;;
```

(b) Voici la fonction `insere_pile` :

```
let rec insere_pile x p =
  if etre_pile_vide p then
    {contenu = [x]}
  else
    begin
      let t = depiler p in
        if x <= t then
          empiler x (empiler t p)
        else
          empiler t (insere_pile x p)
    end
;;
```

(c) Voici la fonction `tri_insertion_pile p` :

```
let rec tri_insertion_pile p =
  if etre_pile_vide p then
    p
  else
    insere_pile depiler p (tri_insertion_pile p)
;;
```

2. (a) Voici les instructions pour définir l'arbre de l'énoncé :

```
let a = N(2, N(1, Vide, Vide), N(4, N(7, Vide, Vide), Vide)) ;;
```

(b) Voici la fonction `noeuds` :

```
let rec noeuds a = match a with
  | Vide -> 0
  | N(_, Vide, Vide) -> 0
  | N(_, g, d) -> 1 + noeuds g + noeuds d
;;
```

(c) Voici la fonction `hauteur` :

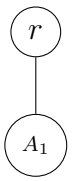
```
let rec hauteur a = match a with
  | Vide -> -1
  | N(_, g, d) -> 1 + max (hauteur g) (hauteur d)
;;
```

(d) Par induction structurelle à partir de la définition récursive d'arbre.

Cas de base : Soit A un arbre binaire à 0 noeud. A possède est de hauteur 0 et on a bien la relation demandé.

Etape d'induction : On considère un arbre binaire A à n noeuds. Notons, pour tout arbre B, $N_n(B)$ le nombre de noeuds de B et $h(B)$ sa hauteur. On a deux cas :

— Soit A est de la forme



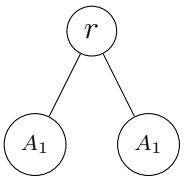
où r est la racine d'arité 1 et A_1 est un arbre binaire non vide vérifiant la propriété. Alors, en utilisant l'hypothèse d'induction structurelle :

$$h(A) = 1 + h(A_1) \leqslant 1 + N_n(A_1) = N_n(A)$$

et

$$\begin{aligned} N_n(A) &= N_n(A_1) + 1 \leqslant 2^{h(A_1)} - 1 + 1 = 2^{h(A_1)} \\ &\leqslant 2^{h(A_1)} + \underbrace{2^{h(A_1)} - 1}_{0\leqslant} = 2^{h(A_1)+1} - 1 = 2^{h(A)} - 1. \end{aligned}$$

— Soit A est de la forme



où r est la racine d'arité 2 et A_1, A_2 sont des arbres binaires non vides vérifiant la propriété. Supposons par exemple $h(A_1) \geqslant h(A_2)$. Alors, en utilisant l'hypothèse d'induction structurelle :

$$h(A) = 1 + h(A_1) \leqslant 1 + N_n(A_1) \leqslant 1 + N_n(A_1) + N_n(A_2) = N_n(A)$$

et

$$\begin{aligned} N_n(A) &= 1 + N_n(A_1) + N_n(A_2) \leqslant 1 + 2^{h(A_1)} - 1 + 2^{h(A_2)} - 1 \\ &\leqslant 2^{h(A_1)} + 2^{h(A_1)} - 1 = 2^{h(A)} - 1. \end{aligned}$$

Par induction structurelle, le résultat est vrai.

(e) Pour parcourir un arbre en profondeur en mode préfixe, on procède récursivement en visitant entièrement les sous-arbres de gauche à droite et on traite les sommets visités à la première visite.

(f) Voici la fonction `prefixe` demandée :

```
let rec prefixe a = match a with
  | N(r, Vide, Vide) -> [r]
  | N(r, g, d) -> (r :: prefixe g) @ (prefixe d)
;;
```

Exercice 2 (Logique et calcul des propositions)

1. L'ensemble des mintermes sur (x_1, x_2) est :

$$x_1 \wedge x_2, \quad x_1 \wedge \neg x_2, \quad \neg x_1 \wedge x_2, \quad \neg x_1 \wedge \neg x_2.$$

L'ensemble des maxtermes sur (x_1, x_2) est :

$$x_1 \vee x_2, \quad x_1 \vee \neg x_2, \quad \neg x_1 \vee x_2, \quad \neg x_1 \vee \neg x_2.$$

2. Voici la table de vérité du connecteur de Sheffer :

x_1	x_2	$\neg x_1$	$\neg x_2$	$x_1 \diamond x_2$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

3. (a) Avec les lois de Morgan, $x_1 \diamond x_2 \equiv \neg x_1 \vee \neg x_2 \equiv \neg(x_1 \wedge x_2)$.

(b) $x_1 \diamond x_1 \equiv \neg x_1 \vee \neg x_1 \equiv \neg x_1$.

(c) Pour $\wedge$:

$$x_1 \wedge x_2 \equiv \neg(\neg x_1 \vee \neg x_2) \equiv \neg(x_1 \diamond x_2) \equiv (x_1 \diamond x_2) \diamond (x_1 \diamond x_2).$$

Pour $\vee$:

$$x_1 \vee x_2 \equiv \neg\neg x_1 \vee \neg\neg x_2 = \neg(x_1 \diamond x_1) \vee \neg(x_2 \diamond x_2) \equiv (x_1 \diamond x_1) \diamond (x_2 \diamond x_2).$$

Pour $\Rightarrow$:

$$x_1 \Rightarrow x_2 \equiv \neg x_1 \vee x_2 \equiv \neg x_1 \vee \neg\neg x_2 \equiv \neg x_1 \vee \neg(x_2 \diamond x_2) \equiv x_1 \diamond (x_2 \diamond x_2).$$

4. (a) $x_1 \diamond x_2 \equiv \neg x_1 \vee \neg x_2$ est une forme normale conjonctive constituée d'un seul maxterme $\neg x_1 \vee \neg x_2$.

(b) On a :

$$\begin{aligned} x_1 \diamond x_2 &\equiv \neg x_1 \vee \neg x_2 \\ &\equiv (\neg x_1 \wedge (x_2 \vee \neg x_2)) \vee (\neg x_2 \wedge (x_1 \vee \neg x_1)) \\ &\equiv (\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2) \vee (\neg x_2 \wedge x_1) \vee (\neg x_2 \wedge \neg x_1) \\ &\equiv (\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2) \vee (x_1 \wedge \neg x_2) \end{aligned}$$

C'est une forme normale disjonctive constituée de trois mintermes $\neg x_1 \wedge x_2$, $\neg x_1 \wedge \neg x_2$ et $x_1 \wedge \neg x_2$.

5. (a) Par induction structurelle (en utilisant les résultats de la question 3) :

Cas de base.

Soit P appartenant à $(x_1, \ldots, x_n)$, alors P est sémantiquement équivalente à P, formule propositionnelle ne faisant intervenir que le connecteur $\diamond$.

Étape d'induction.

— Considérons une formule propositionnelle de la forme $P = \neg P_1$. Par hypothèse d'induction, P_1 est sémantiquement équivalente à une formule Q_1 ne faisant apparaître que le connecteur $\diamond$. Donc

$$P \equiv \neg Q_1 \equiv Q_1 \diamond Q_1,$$

et cette dernière expression ne fait apparaître que le connecteur $\diamond$.

— Considérons une formule propositionnelle de la forme $P = P_1 \vee P_2$. Par hypothèse d'induction, P_1 et P_2 sont sémantiquement équivalentes à des formules Q_1 et Q_2 ne faisant apparaître que le connecteur $\diamond$. Donc

$$P \equiv Q_1 \vee Q_2 \equiv (Q_1 \diamond Q_1) \diamond (Q_2 \diamond Q_2),$$

et cette dernière expression ne fait apparaître que le connecteur $\diamond$.

— Considérons une formule propositionnelle de la forme $P = P_1 \wedge P_2$. Par hypothèse d'induction, P_1 et P_2 sont sémantiquement équivalentes à des formules Q_1 et Q_2 ne faisant apparaître que le connecteur $\diamond$. Donc

$$P \equiv Q_1 \wedge Q_2 \equiv (Q_1 \diamond Q_2) \diamond (Q_1 \diamond Q_2),$$

et cette dernière expression ne fait apparaître que le connecteur $\diamond$.

— Considérons une formule propositionnelle de la forme $P = P_1 \Rightarrow P_2$. Par hypothèse d'induction, P_1 et P_2 sont sémantiquement équivalentes à des formules Q_1 et Q_2 ne faisant apparaître que le connecteur $\diamond$. Donc

$$P \equiv Q_1 \Rightarrow Q_2 \equiv Q_1 \diamond (Q_2 \diamond Q_2),$$

et cette dernière expression ne fait apparaître que le connecteur $\diamond$.

— Considérons une formule propositionnelle de la forme $P = P_1 \Leftrightarrow P_2$. Par hypothèse d'induction, P_1 et P_2 sont sémantiquement équivalentes à des formules Q_1 et Q_2 ne faisant apparaître que le connecteur $\diamond$. Donc

$$\begin{aligned} P &\equiv Q_1 \Leftrightarrow Q_2 \\ &\equiv (Q_1 \Rightarrow Q_2) \wedge (Q_2 \Rightarrow Q_1) \\ &\equiv (Q_1 \diamond (Q_2 \diamond Q_2)) \wedge (Q_2 \diamond (Q_1 \diamond Q_1)) \\ &\equiv ((Q_1 \diamond (Q_2 \diamond Q_2)) \diamond (Q_2 \diamond (Q_1 \diamond Q_1))) \\ &\quad \diamond ((Q_1 \diamond (Q_2 \diamond Q_2)) \diamond (Q_2 \diamond (Q_1 \diamond Q_1))) \end{aligned}$$

et cette dernière expression ne fait apparaître que le connecteur $\diamond$.

(b) On a avec la question 3 :

$$\begin{aligned} & x_1 \vee (\neg x_2 \wedge x_3) \\ \equiv\ & x_1 \vee ((x_2 \diamond x_2) \wedge x_3) \\ \equiv\ & x_1 \vee (((x_2 \diamond x_2) \diamond x_3) \diamond ((x_2 \diamond x_2) \diamond x_3)) \\ \equiv\ & (x_1 \diamond x_1) \diamond ((((x_2 \diamond x_2) \diamond x_3) \diamond ((x_2 \diamond x_2) \diamond x_3)) \diamond (((x_2 \diamond x_2) \diamond x_3) \diamond ((x_2 \diamond x_2) \diamond x_3))). \end{aligned}$$

Exercice 3 (Sur les permutations)

1. Voici la fonction `test_permutation_1` :

```
let test_permutation_1 p =
  let n = Array.length p and b = ref true in
    for i = 1 to n-1 do
      let occ := ref 0 in
        for j = 1 to n-1 do
          if p.(j) = i then
            occ := !occ + 1
        b := !b && (!occ = 1);
        done;
    done;
    b
;;
```

2. Voici la fonction `test_permutation_2` :

```
let test_permutation_2 p =
  let n = Array.length p and i = ref 1 in
    let v = Array.make n false in
      while (!i < n) && (not v.(p.(!i))) do
        v.(p.(!i)) <- true ;
        i := !i+1;
      done;
  !i = n
;;
```

3. Voici la fonction `ident` :

```
let ident n =
  let p = Array.make (n+1) 0 in
    for k=1 to n do
      p.(k) <- k
    done;
    p
;;
```

4. Voici la fonction `composition` :

```
let composition p q =
  let n = Array.length p in
    let r = Array.make n 0 in
      for k = 1 to n-1 do
        r.(k) <- p.(q.(k))
      done;
      for k = 1 to n-1 do
        p.(k) <- r.(k)
      done;
      p
;;
```

On choisit ici de stocker la composée $p \circ q$ dans le tableau p pour la prochaine question.

5. Voici la version itérative de `power` :

```
let power p m =
  let n = Array.length p - 1 in
    let r = ident n in
      for k=1 to m do
        composition r p
      done;
      r
;;
```

Voici la version récursive de `power` :

```
let rec power p m =
  let n = Array.length p - 1 in
    math m with
      | 0 -> ident n
      | _ -> composition p (power p (m-1))
;;
```

6. Voici la fonction `signature` :

```
let signature p =
  let n = Array.length p - 1 and s = ref 1 in
    for i = 1 to n do
      for j = i+1 to n do
        if p.(i)>p.(j) then s := - !s
      done;
    done;
    !s
;;
```

7. (a) Le code de Lehmer associé à la permutation

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 6 & 1 & 5 & 2 & 3 \end{pmatrix}$$

est $c = (3, 4, 0, 2, 0, 0)$.

(b) Voici la fonction `code` :

```
let code p =
  let n = Array.length p - 1 in
    let c = Array.make (n+1) 0 in
      for i = 1 to n do
        let compt = ref 0 in
          for j = i+1 to n do
            if (p.(j) < p.(i)) then compt := !compt + 1
          done;
          c.(i) <- !compt
      done;
      c
;;
```

8. (a) La permutation dont le code de Lehmer est $(2, 0, 3, 1, 0, 0)$ est :

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 1 & 6 & 4 & 2 & 5 \end{pmatrix}.$$

(b) Voici la fonction `decode` :

```
let decode c =
  let n = Array.length c - 1 in
    let q = ident n and p = Array.make (n+1) 0 in
      for i = 1 to n do
        let fin = c.(i)+1 in
          let j = ref 0 and k = ref 0 in
            while (!j < fin) do
              k := !k+1;
              if q.(!k) <> 0 then j := !j +1;
            done;
            p.(i) <- !k;
            q.(!k) <- 0
      done;
      p
;;
```

9. Pour $n = 3$, on a :

$$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix} \to \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} \to \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix} \to$$
$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \to \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} \to \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

On a bien toutes les permutations de E_3 car on sait que $card(E_3) = 3! = 6$ et on remarque qu'elles sont bien dans l'ordre lexicographique.

10. Comme $\sigma \neq \sigma_{\max}$, l'ensemble fini $\{j \in [\![1, n-1]\!] \mid \sigma(j) < \sigma(j+1)\}$ est non vide. Il admet donc un plus grand élément i, ce qui prouve son existence.
Par définition de l'entier i, $\sigma(i) < \sigma(i+1)$ donc l'ensemble fini $\{\sigma(j)/j \in [\![i+1, n]\!], \sigma(i) < \sigma(j)\}$ est non vide. Il admet donc un plus petit élément k, ce qui prouve son existence.

11. On aura besoin de la fonction `echange` suivante :

```
let echange p i j =
  let aux = p.(i) in
    p.(i) <- p.(j);
    p.(j) <- aux
;;
```

Voici la fonction `next` :

```
let next p =
  let n = Array.length p - 1 in
    let i = ref (n-1) in
      while (!i >= 0) && (p.(!i) > p.(!i+1)) do
        i := !i - 1;
      done;
      if (!i > 0) then
        begin
          let m = ref (!i+1) in
            for k = (!i+2) to n do
              if (p.(k) < p.(!i)) && (p.(k) <= !m) then m := k
            done;
            echange p !i !m;
            for u = !i+1 to n do
              m := u;
              for v = u+1 to n do
                if p.(v) < p.(!m) then m :=v
              done;
              echange p u !m;
            done;
        end;
;;
```

12. On remarque que $\sigma(1) = \sigma'(1), \sigma(2) = \sigma'(2), \ldots, \sigma(i-1) = \sigma'(i-1)$. Par définition, $\sigma(i) < \sigma(k)$ donc $\sigma'(i) = \sigma(k) > \sigma(i)$. Donc $\sigma' >_{\text{lex}} \sigma$.

13. L'application répétée de l'algorithme donne une suite strictement croissante de permutations appartenant à E_n. Comme E_n est finie, cette suite de permutations est finie et on génère ainsi en un temps fini des permutations distinctes.

    On commence par appliquer notre algorithme à la permutation identité qui est la plus petite permutation pour l'ordre lexicographique. On admet de plus que, à chaque application de notre algorithme, la permutation obtenue est le successeur immédiat de la précédente par l'ordre lexicographique. Comme E_n est totalement ordonné par l'ordre lexicographique, on obtient donc toutes les permutations de E_n.

14. On aura besoin de la fonction `factorielle` suivant :

```
let rec factorielle n = match n with
  | 0 -> 1
  | _ -> n * (factorielle (n-1))
;;
```

On peut alors définir la fonction `list` (en sachant que $card(E_n) = n!$) :

```
let list n =
  let p = ident n and N = factorielle(n) in
    for k = 1 to N do
      afficher p;
      next p;
    done
;;
```