Devoir d'Informatique n°1

10 mai 2025

*
* *

Durée de l'épreuve :
3 heures (non tiers-temps)
4 heures (tiers-temps)

L'usage de tout dispositif électronique est interdit.

Consignes

Pour répondre à une question, il vous est permis de réutiliser le résultat d'une question antérieure même si vous n'avez pas réussi à établir ce résultat.

Quand l'énoncé demande de coder une fonction, sauf demande explicite, il n'est pas nécessaire de justifier la correction ou la terminaison de cette fonction, ou de la commenter.

Vous attacherez la plus grande importance à la clarté, à la précision et à la concision de la rédaction.

Si vous repérez ce qu'il vous semble être une erreur d'énoncé, signalez-le sur votre copie et poursuivez la composition en expliquant les éventuelles initiatives que vous aurez pris.

Vous devrez traiter les questions de programmation dans le langage OCaml.

Exercices

Q1. Donner une expression OCaml pour chacun des types suivants.

a) `('a * 'b) -> ('b * 'a)`
b) `'a -> int`
c) `'a -> ('a -> 'b) -> 'b`
d) `'a -> ('b -> 'a)`

Q2. Donner le type de chacune des fonctions suivantes.

```
1  let q2a f g h =
2    f 0 + g 0 + h
3
4  let q2b x y z t w =
5    w ((x y) (z t))
6
7  let rec q2c m p =
8    let a, b, c = m 0 in
9    if p a = p b then c
10   else q2c m (fun x -> p x - 1)
```

Q3. Déterminer la valeur de chacune des expressions suivantes, en expliquant votre raisonnement et/ou vos calculs.

```
1  let q3a =
2    let rec f la lb =
3      match la with
4      | [] -> lb
5      | x :: q -> f q (x :: lb)
6    in f [2; 3] [4; 5]
7
8  let q3b =
9    let rec f la = match la with
10     | [] -> failwith "Erreur"
11     | [x] -> x
12     | x::y::q -> f ((if x < y then x else y) :: q)
13   in f [8;4;1;3;7]
14
15 let q3c =
16   let p a b x = x a b in
17   let g = p 5 3 in
18   let e c x = c (fun u v -> x 2 (u-v)) in
19   let s c = c (fun u v -> u) + c (fun u v -> v) in
20   s g * s (e g)
```

Écriture binaire (adapté du début de Centrale 2019)

Pour $x \in \mathbb{N}$, une **représentation binaire** de x est une suite finie $a_0, a_1, \ldots, a_{p-1}$ vérifiant :

$$x = \sum_{k=0}^{p-1} a_k 2^k$$

Par exemple, 6 admet comme représentation binaire $(0, 1, 1)$ car $6 = 0+2+4$. Notons qu'une telle représentation n'est pas unique. Par exemple, $(0, 1, 1, 0, 0, 0)$ est aussi une représentation binaire valide de 6. Pour $x \in \mathbb{N}$ et $(a_0, \ldots, a_{p-1})$ une de ses représentations binaires, les a_i sont appelés les **bits** de x. a_0 est appelé son **bit de poids faible**.

Q1. Donner les entiers représentés par les suites de bits suivantes :

a) $(1, 0, 1, 1)$
b) $(0, 0, 0, 0, 1)$
c) $(1, 1, 1, 1)$

Q2. Donner une représentation binaire pour chacun des entiers suivants :

a) 13
b) 128
c) 31
d) 2^k pour $k \in \mathbb{N}$
e) $2^k - 1$ pour $k \in \mathbb{N}$

Q3. Soit $x \in \mathbb{N}$ et $(a_0, \ldots, a_{p-1}), (b_0, \ldots, b_{p-1})$ deux représentations binaires de x de mêmes longueurs. Montrer que les deux représentations sont identiques. On pourra commencer par justifier que $a_0 = b_0$ en exprimant chacun comme le reste d'une division euclidienne.

En OCaml, on représente les suites de bits par des listes de booléens. Afin de simplifier le code, on stocke le bit de poids faible en début de liste.

Q4. Écrire une fonction `lire: bool list -> int` qui étant donné une suite de bits, renvoie l'entier qu'elle représente. Par exemple :

```
1 assert (lire_entier [0; 1; 1] = 6);;
2 assert (lire_entier [1; 0; 1; 1] = 13);;
3 assert (lire_entier [] = 0);;
4 assert (lire_entier [0; 1; 1; 0; 0] = 6);;
```

Cette fonction devra être de complexité $\mathcal{O}(p)$ avec p la taille de la liste en entrée, ce que l'on justifiera brièvement.

Q5. Écrire une fonction `ecrire: int -> bool list` effectuant l'opération inverse. On rappelle l'existence des opérateurs infixes `mod` et `/` servant à effectuer respectivement le reste et le quotient de la division euclidienne.

Q6. Écrire une fonction `add_one: bool list -> bool list * bool` ajoutant 1 à son entrée. Cette fonction prendra en entrée une liste L formant une représentation binaire d'un entier n, et renverra un couple (L', c) avec L' une représentation binaire de $n+$, et c un booléen indiquant si la liste a changé de taille. Par exemple :

```
assert (add_one [0; 1; 1] = ([1; 1; 1], false));;
assert (add_one [1; 1; 1] = ([0; 0; 0; 1], true));;
```

Q7. Écrire une fonction `enum: int -> unit` qui prend en entrée $p \in \mathbb{N}$ et affiche tous les entiers de 0 à $2^p - 1$ en binaire, avec le bit de poids faible à **droite**. Par exemple, `enum 3` affichera :

```
000, 001, 010, 011, 100, 101, 110, 111
```

Il est conseillé de s'appuyer sur des fonctions auxiliaires, que vous décrirez précisément.

Q8. Déterminer la complexité de la fonction `enum` ?

Pour des raisons techniques, il peut être avantageux d'énumérer les entiers dans un ordre où deux termes successifs n'ont qu'un bit de différence. Cela permet d'éviter des erreurs dues à des états transitoires intermédiaires dans des systèmes électroniques par exemple. Les codes de Gray forment un ordre d'énumération ayant cette propriété.

Par exemple, l'énumération classique des entiers sur 2 bits, qui ne possède pas la propriété voulue, est $00, 01, 10, 11$. Le passage de 01 à 10 cause **deux** changements de bits. Le code de Gray d'ordre 2 est $00, 01, 11, 10$: chaque passage d'un terme au suivant ne nécessite qu'un seul changement de bit.

Pour produire la liste des termes du code de Gray d'ordre $p+1$, on commence par produire la liste L_p des termes du code de Gray d'ordre p. On construit alors la liste L_{p+1} en ajoutant un 0 en tête de chaque terme de L_p, puis en aoutant un 1 en tête de chaque terme de L_p parcourue à l'envers.
Le code de Gray d'ordre 1 est $0, 1$. D'après le procédé décrit ci-dessus, le code de Gray d'ordre 2 est donc $00, 01, 11, 10$, et celui d'ordre 3 est $000, 001, 011, 010, 110, 111, 101, 100$.

Q9. Écrire une fonction `prefixer: bool list list -> bool -> bool list list` prenant en entrée L une liste de booléens, b un booléen, et renvoyant la liste L' obtenue en ajoutant b en tête de chaque élément de L.

Q10. On rappelle l'existence de la fonction `List.map: ('a -> 'b) -> 'a list -> 'b list` telle que pour f une fonction et $L = [x_1; x_2; \ldots; x_n]$ une liste, `List.map f L` renvoie la liste obtenue en appliquant f à chaque élément de L, i.e. $[f(x_1); f(x_2); \ldots; f(x_n)]$. En utilisant la fonction `List.map`, proposer une deuxième version de la fonction `ajout`.

Q11. Écrire une fonction OCaml `gray: int -> bool list list` prenant en entrée $p \in \mathbb{N}$ et renvoyant la liste des termes du code de Gray d'ordre p.

Flots d'arbres (adapté du début de X 2001)

Soit E un ensemble. On définit les arbres binaires étiquetés par E de la façon suivante :

— Pour $e \in E$, la feuille $F(e)$ est un arbre ;

— Pour $e \in E$ et g, d deux arbres étiquetés par E, le noeud interne $N(e, g, d)$ est un arbre.

On s'intéresse à deux types d'arbres dans ce problème :

— Les arbres étiquetés par $\mathbb{N}$;

— Les arbres étiquetés par le singleton $\{()\}$, aussi appelés les **squelettes** d'arbres.

Dans un squelette d'arbre, seule la **structure** de l'arborescence importe. En OCaml, on utilisera les types suivants :

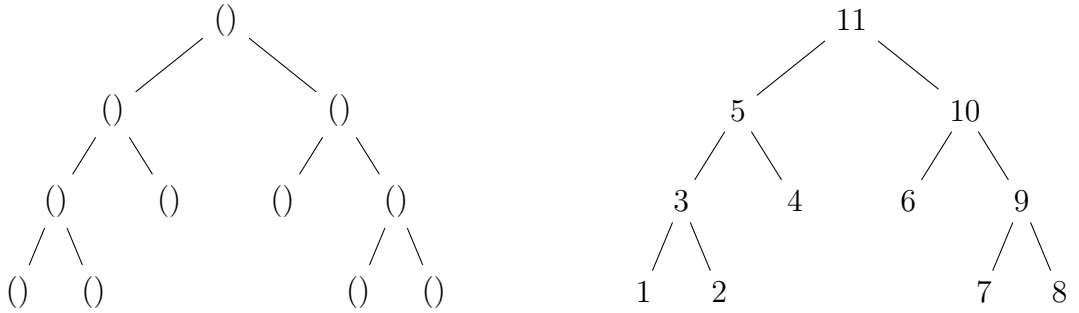
```
(* arbres binaires *)
type 'a ab =
  | F of 'a
  | N of 'a * 'a ab * 'a ab

type arbre = int ab
type squelette = unit ab
```

Q1. Écrire deux fonctions `compte_N: 'a ab -> int` et `compte_F: 'a ab -> int` comptant respectivement le nombre de noeuds internes et de feuilles apparaissant dans un arbre binaire.

Q2. Montrer par induction structurelle sur les arbres que pour tout arbre binaire a, on a $\mathbf{compte\_F}(a) = \mathbf{compte\_N}(a) + 1$.

En guise d'objectif préliminaire servant à se familiariser avec ces structures, on se propose d'écrire une fonction permettant de **numéroter** un squelette, en inscrivant sur chaque feuille ou noeud interne son numéro dans l'ordre postfixe.

Par exemple, voici un squelette a_0, et sa version numérotée :



Q3. Écrire une fonction `post_print: arbre -> unit` qui prend en entrée un arbre étiqueté par des entiers, et qui affiche chacun de ses éléments, dans l'ordre postfixe. Par exemple, sur l'arbre ci-dessus, cette fonction afficherait $1, 2, \ldots, 10, 11$ puisque les noeuds sont numérotés précisément dans l'ordre postfixe.

Q4. Écrire une fonction `numeroter: squelette -> arbre` numérotant un squelette. On pourra passer par la fonction auxiliaire suivante :

```
(* numeroter indice s p numérote s dans l'ordre postfixe à partir de l'indice p.
   Renvoie un couple (a, p') avec
   - a l'arbre obtenu en numérotant s à partir de p dans l'ordre postfixe
   - p' le premier numéro non-utilisé dans la numérotation de s
   Par exemple, si l'appel à numeroter_indice s 10 renvoie un arbre a
   utilisant les étiquettes 10, 11, 12, 13, 14, alors p' vaudra 15. *)
let rec numeroter_indice (s: squelette) (p: int) : arbre * int = ...
```

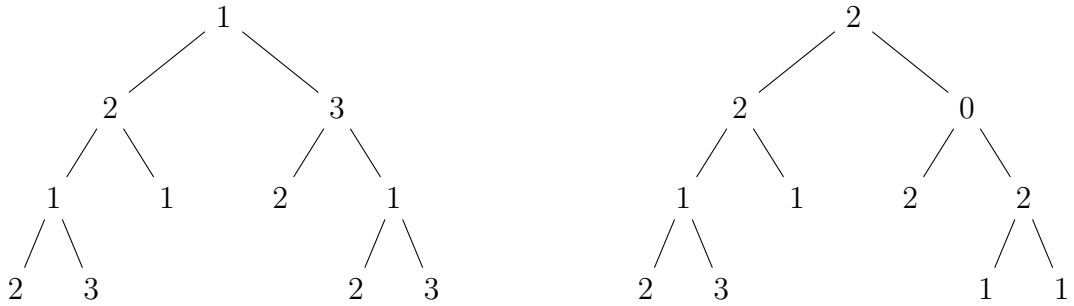
Dans la suite, on note A_n l'ensemble des squelettes possédant n feuilles ($n \geq 1$).

Un **flot** f de taille n est une suite de n entiers égaux à 1, 2 ou 3. Pour un squelette $a \in A_n$, un flot f induit une fonction des noeuds de a vers les entiers, notée f également, et définie comme suit :

— La i-ème feuille (de gauche à droite) est associée au i-ème entier du flot f

— $f(N((), g, d) = f(g) + f(d) \bmod 4$ pour un noeud interne.

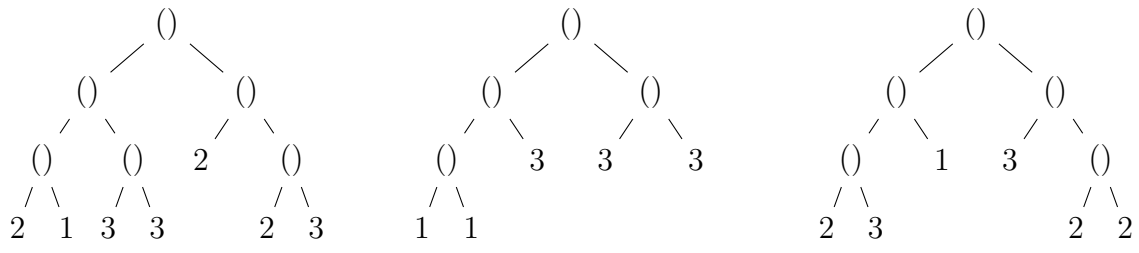
Ce procédé transforme le squelette a en un arbre étiqueté par des entiers. Par exemple, voici le squelette de l'arbre a_0 précédent, après application du flot $f_1 = [2; 3; 1; 2; 2; 3]$ et après application du flot $f_2 = [2; 3; 1; 2; 1; 1]$



Un flot est dit **compatible** avec un squelette $a \in A_n$ si pour tout nœud x de a, $f(x) \neq 0$. Dans l'exemple ci-dessus, le flot f_1 est compatible avec a, mais le flot f_2 ne l'est pas. On note F_n l'ensemble des flots de taille n. En OCaml, on représentera les flots par des listes d'entiers :

```
1 type flot = int list
```

Q5. Dans chacun des exemples suivants, étendre le flot placé sur les feuilles aux autres noeuds et déterminer quels flots sont compatibles :



a_1 a_2 a_3

Q6. En s'inspirant de la **Q4**, écrire une fonction `place_flot: squelette -> int list -> arbre` qui étant donné un squelette s et un flot f, renvoie un arbre étiqueté par N, où les noeuds internes sont tous étiquetés par 0, et où les feuilles ont été correctement étiquetés par f.

Q7. Écrire une fonction `propage_flot: arbre -> arbre * bool`. Cette fonction prend en entrée un arbre a où les feuilles ont été étiquetées par un flot, et renvoie un couple (a', b) avec b un booléen indiquant si le flot inscrit sur les feuilles est compatible avec a, et a' définit comme suit.

— Si $b =$ **true**, a' est l'arbre obtenu après avoir propagé totalement le flot des feuilles ;

— sinon, a' est un arbre arbitraire.

De plus, la fonction `propage_flot` calculera les nœuds dans l'ordre postfixe, et devra s'arrêter d'effectuer des additions modulo 4 dès qu'un nœud interne x vérifiant $f(x) = 0$ est découvert.

Q8. Soit $a \in A_n$ un squelette et $f \in F_n$ un flot compatible avec a. On suppose que a n'est pas réduit à une feuille, et on écrit $a = N((), g, d)$. Donner les valeurs possibles du couple $(f(g), f(d))$ selon les valeurs possibles de $f(a)$.

Q9. Soit $a \in A_n$. Soit $v \in \{1, 2, 3\}$. Calculer $F(a, v)$, nombre de flots f compatibles avec a et tels que $f(a) = v$. En déduire le nombre de flots compatibles avec a.

On fixe a un squelette, et on s'intéresse au coût de la fonction `propage_flot` sur a. Pour $f \in F_n$, on note $N_+(a, f)$ le nombre d'additions modulo 4 effectuées par l'appel de la fonction `propage_flot` sur a et f. Les calculs étant effectués dans l'ordre postfixe, sur l'exemple plus haut avec l'arbre a_0 et les flots f_1, f_2, on a $N_+(a_0, f_1) = 5$, car tous les nœuds internes sont calculés, mais $N_+(a_0, f_2) = 4$, car le calcul de compatibilité s'arrête lorsque le 0 est calculé, juste avant le calcul de la racine.

On admet que N_+ est une bonne mesure de la complexité de la fonction `propage_flot`.

Q10. Quelle est la valeur exacte de $N_+(a, f)$ dans le pire cas ?

Néanmoins, on s'intéresse au calcul de la complexité **moyenne** de la fonction `propage_flot`, c'est à dire à la valeur moyenne de $N_+(a, f)$ en considérant une répartition équiprobable des 3^n flots de F_n.

Pour $k \in [\![1, n-1]\!]$, on définit $\nu_k(a)$ le nombre de flots f incompatibles avec a et tels que $N_+(a, f) = k$.

Q11. Calculer $\nu_1(a)$. Sa valeur dépend-elle de la forme de a ?

Q12. Montrer qu'il existe un squelette a' possédant $n-1$ feuilles et tel que pour $k \in [\![2, n-1]\!]$, $\nu_k(a) = 2\nu_k(a')$. En déduire la valeur de $\nu_k(a)$ dans le cas général.

Q13. On définit la complexité moyenne de l'appel de `propage_flot` sur a comme

$$\sum_{f \in F_n} \frac{1}{3^n} N_+(a, f)$$

Calculer cette complexité moyenne, et montrer que sa limite lorsque $n \to +\infty$ vaut 3. On pourra utiliser la formule suivante sans la démontrer :

$$\sum_{k=1}^{n} k\alpha^{k-1} = \frac{1 - \alpha^n(n + 1 - n\alpha)}{(1-\alpha)^2}$$

Ainsi, malgré le fait que le coût pire cas est linéaire, le temps moyen d'exécution de la fonction `propage_flot` est **constant** lorsque n grandit.