Université Paris Cité — Programmation Fonctionnelle
L3 Informatique — Année 2022-2023

Examen du jeudi 5 janvier 2023 – Durée 2 heures

Cet énoncé a 3 pages. Tout document papier est autorisé. Les ordinateurs et les téléphones portables doivent être éteints et rangés, ainsi que tout autre moyen de communication.

Les fonctions demandées doivent être rédigées en fonctionnel pur : ni références, ni tableaux, ni boucles `for` ou `while`, ni champs mutables. Chaque question ci-dessous peut utiliser les fonctions prédéfinies et/ou les fonctions des questions précédentes. À titre indicatif, toutes les fonctions demandées peuvent s'écrire en moins de dix lignes.

Exercice 1 (**Expressions, types, valeurs).** Pour chacune des expressions suivantes, indiquez si OCaml l'accepte, et donnez dans ce cas le type et la valeur calculés par OCaml. Si une valeur fonctionnelle est présente dans le résultat, la noter `<fun>` sans détailler plus. Si OCaml signale une erreur, la décrire succinctement. On ne demande pas alors le message d'erreur exact, mais l'idée essentielle, par exemple "ceci est de type `string`, mais `int` était attendu ici".

1.1 `[1; 2; 3.14]`

1.2 `if false then if true then false else true`

1.3 `(fun x y -> (x,y)) 1 2`

1.4 `(fun x y -> (x,y)) 1`

1.5 `(fun x y -> (x,y)) 1 2 3`

1.6 `let x = 3 in let f x = x*x in f x + x`

1.7 `List.map (fun x -> Some x) [2; 3; 4]`

1.8 `List.map (fun x -> Some x) [None; Some 1]`

1.9 `(fun f -> f true) (fun b -> not b)`

1.10 `try List.assoc 2 [(1,"ok");(2,"yes")] with Not_found -> "no"`

Exercice 2. La *compression gauche* d'une liste d'entier est la transformation suivante :

`[`x_1`; `x_2`; `x_3`; ...; `x_n`]` donne `[`x_1 `+ `x_2`; `x_3`; ...; `x_n`]`

Autrement dit, cette transformation fusionne les deux premiers éléments d'une liste d'entiers en les remplaçant par leur somme. La compression gauche d'une liste de moins de deux éléments gardera cette liste intacte.

2.1 Écrire une fonction `comp_gauche : int list -> int list` retournant la compression gauche de la liste reçue en argument. Par exemple `comp_gauche [1;2;3] = [3;3]`.

Les *compressions gauches successives* d'une liste `xs` sont chacune des listes obtenues en appliquant zéro, une ou plusieurs compressions gauches à `xs`.

2.2 Écrire une fonction `comps_gauches : int list -> (int list) list` telle que pour toute liste d'entiers `xs`, `comps_gauches xs` renvoie toutes les compressions gauches successives de `xs`. Par exemple `comps_gauches [1;2;0;4] = [[1;2;0;4]; [3;0;4]; [3;4]; [7]]`

On généralise maintenant ce qui précède en appelant *compression* chacune des transformations suivantes :

`[`x_1`; ...; `x_i`; `x_{i+1}`; ...; `x_n`]` donne `[`x_1`; ...; `x_i `+ `x_{i+1}`; ...; `x_n`]`

Autrement dit, ces transformations fusionnent deux éléments consécutifs d'une liste d'entiers en les remplaçant par leur somme. Là encore, la compression d'une liste de moins de deux éléments garde cette liste intacte. Enfin, on appelle *compressions successives* d'une liste `xs` toutes les listes obtenues en appliquant zéro, une ou plusieurs compressions à `xs`. Voici par exemple toutes les compressions successives de `[1; 1; 1; 1]` :

```
[1; 1; 1; 1]                    (* zero compression *)
[2; 1; 1]  [1; 2; 1]  [1; 1; 2]  (* une compression *)
[3; 1]  [2; 2]  [1; 3]          (* deux compressions *)
[4]                             (* trois compressions *)
```

2.3 Écrire une fonction `comps : int list -> (int list) list` telle que pour toute liste d'entiers `xs`, `comps xs` renvoie une liste de toutes les compressions successives de `xs`, dans l'ordre de votre choix. La liste de listes générée pourra contenir des répétitions.

Exercice 3 (**Mots bien parenthésés).** On considère ici les mots formés sur l'alphabet des parenthèses, avec donc deux lettres possibles seulement : parenthèse ouvrante et parenthèse fermante. Voici la représentation OCaml que nous utiliserons :

```
type parenthese = PO | PF (* Parenthèse Ouvrante, Parenthèse Fermante *)
type mot = parenthese list
let exemple1 = [PO;PO;PF;PF;PO;PF]  (* mot (())() *)
let exemple2 = [PO;PF;PF;PO;PF;PO]  (* mot ())()) *)
```

Parmi les deux exemples précédents, le mot `exemple1` représentant `(())()` est ce qu'on appelle un *mot bien parenthésé* (m.b.p. en abrégé) tandis que `exemple2` représentant `())()(` est un mot qui n'est pas bien parenthésé. Plus généralement, les m.b.p. sont exactement les mots que l'on obtient via les règles suivantes :

— Le mot vide est un m.b.p ;

— Si un mot `m` est un m.b.p, alors le mot `(m)` obtenu en ajoutant une parenthèse ouvrante et une parenthèse fermante autour de `m` est aussi un m.b.p ;

— Si les mots $\mathtt{m}_1$ et $\mathtt{m}_2$ sont deux m.b.p, alors leur concaténation $\mathtt{m}_1\mathtt{m}_2$ est aussi un m.b.p.

3.1 Dans le code suivant, la fonction `test_mbp` détermine si un mot `m` est un m.b.p. en énumérant tous les m.b.p. de la longueur de `m`. Donner les types OCaml des fonctions `entoure`, `sigma`, `apps` et `mbps` ci-dessous.

```
let entoure m = [PO] @ m @ [PF]
let rec sigma n f = if n <= 1 then [] else sigma (n-1) f @ f (n-1)
let rec apps f l1 l2 = match l1 with
  | [] -> []
  | x1::r1 -> (List.map (f x1) l2) @ (apps f r1 l2)
let rec mbps = function
  | 0 -> [[]]
  | 1 -> []
  | n -> List.map entoure (mbps (n-2))
         @ sigma n (fun k -> apps (@) (mbps k) (mbps (n-k)))
let test_mbp m = List.mem m (mbps (List.length m))
```

Cette première implémentation de `test_mbp` est proche de la définition des m.b.p. mais a une très mauvaise complexité. Voici une caractérisation plus commode des m.b.p. (que l'on admettra) :

— L'*ouverture* d'un mot est son nombre de parenthèses ouvrantes moins son nombre de parenthèses fermantes.

— Un mot `m` est un m.b.p ssi son ouverture vaut 0 et si tout préfixe de `m` a une ouverture positive ou nulle.

Par exemple `exemple2` admet `[PO;PF;PF]` comme préfixe ayant une ouverture de -1, donc `exemple2` n'est donc pas un m.b.p. On rappelle qu'un préfixe de `m` est un mot constitué des premières lettres de `m`, et de longueur inférieure ou égale à celle de `m`.

3.2 Implémenter cette caractérisation en une fonction `test_mbp_opt : mot -> bool` déterminant si un mot est un m.b.p. ou non. Un bonus sera attribué si le mot est parcouru une unique fois par cette fonction (et ses éventuelles sous-fonctions).

Voici maintenant un type OCaml encodant spécifiquement les m.b.p. :

```
type mbp =
 | Vide
 | Paren of mbp
 | Concat of mbp * mbp
let exemple1_mbp = Concat (Paren (Paren Vide), Paren Vide)
```

La construction `Paren m` représente le m.b.p. `m` entouré d'une parenthèse ouvrante et fermante. Dans cette représentation, l'exemple `(())()` est encodé par `exemple1_mbp` ci-dessus et l'exemple `())()(` n'est pas représentable.

3.3 Implémenter `ecrire_mbp : mbp -> mot` convertissant un `mbp` en la représentation `mot` précédente. Par exemple `ecrire_mbp exemple1_mbp = exemple1`.

3.4 Réciproquement, implémenter `lire_mbp : mot -> mbp` qui convertit un `mot` en un `mbp` correspondant, ou bien lance une exception de votre choix si ce mot n'est pas bien parenthésé. On notera que tout préfixe d'un m.b.p. peut se décomposer en un ou plusieurs m.b.p. séparés par des parenthèses ouvrantes. Il pourra donc être intéressant de manipuler également une liste de `mbp` correspondant à la décomposition du préfixe en cours de lecture, et de rallonger (resp. raccourcir) cette liste à chaque parenthèse ouvrante (resp. fermante).

Le type OCaml `mbp` a un inconvénient : il n'est pas *canonique*, c'est-à-dire que deux valeurs `m` et `p` de type `mbp` peuvent redonner le même mot par `ecrire_mbp` sans être égales au départ. On notera `m` $\approx$ `p` pour abréger le fait que `ecrire_mbp m = ecrire_mbp p`. Exemples de telles situations :

— `Concat(Vide,m)` $\approx$ `m` $\approx$ `Concat(m,Vide)`

— `Concat(m,Concat(p,q))` $\approx$ `Concat(Concat(m,p),q)`

On cherche donc maintenant à *normaliser* les valeurs de type `mbp` en enlevant les `Vide` superflus, et en choisissant de mettre les `Concat` le plus à droite possible.

3.5 Écrire une fonction `concats : mbp list -> mbp` concaténant une liste de `mbp` de la manière suivante :

— `concats [] = Vide`

— `concats [m] = m`

— `concats [`m_1`;`m_2`...`m_{n-1}`;`m_n`] = Concat (`m_1`,Concat(`m_2`,...Concat(`m_{n-1}`,`m_n`)...)`

3.6 Dans un `mbp m`, un noeud `Concat` est dit *sommital* s'il n'est englobé par aucun noeud `Paren`. Implémenter une fonction `aplatir : mbp -> mbp list` qui déconstruit tous les `Concat` sommitaux d'un `mbp` et regroupe les morceaux obtenus, de gauche à droite. On enlèvera également `Vide` de la liste résultat, ce qui fait que tous les `mbp` présents dans cette liste résultat devront commencer par `Paren` (sans chercher ici à modifier l'intérieur de ces `Paren`). Par exemple :

```
aplatir (Concat (exemple1_mbp, Concat (Vide, Vide)))
 = [Paren (Paren Vide); Paren Vide]
```

3.7 Implémenter une fonction `normalise : mbp -> mbp` qui procède via `aplatir` puis normalise récursivement l'intérieur des fragments `Paren` obtenus, puis les regroupe via `concats`. De la sorte, si `m' = normalise m` alors `m'` $\approx$ `m` et de plus `m'` est l'unique mot issu de `normalise` vérifiant cette propriété.

3.8 Proposer un type OCaml `mbpcan` permettant de coder exactement les m.b.p. tout en étant canonique dès le départ cette fois-ci.