

Examen – Session 1

lundi 2 janvier 2017

Tout document papier est autorisé. Les ordinateurs, les téléphones portables, comme tout autre moyen de communication vers l'extérieur, doivent être éteints. Le temps à disposition est de **3 heures**.

Les exercices doivent être rédigés **en fonctionnel pur** : ni références, ni tableaux, ni boucles `for` ou `while`, pas d'enregistrements à champs mutables. Une fonction écrite en style impératif ne rapportera aucun point.

Les questions sont indépendantes mais il est parfois nécessaire (ou recommandé) d'utiliser les fonctions définies précédemment (même si elles sont non traitées) ou prédéfinies dans la bibliothèque standard (notamment dans le module sur les listes).

Certaines questions, plus difficiles, sont marquées d'une étoile : ★.

On recommande de *bien lire* l'énoncé d'un exercice avant de commencer à le résoudre. Cet énoncé a 4 pages plus **appendice**.

Exercice 1 (Expressions). Qu'affiche le toplevel Caml quand on entre l'une après l'autre les expressions suivantes ? Donner pour chaque expression ou définition : son type, sa valeur, et, le cas échéant, son effet de bord, ou l'exception levée. Si l'expression est mal typée ou pas correcte, donner le message d'erreur. Justifier si nécessaire.

1. `let x = "bonne" and y = "chance" in x^y;;`
2. `x^y;;`
3. `let y = let x = 3 in (let x = "thread is String.length x" in x);;`
Handwritten: $E = 3$, $E + 3 = 8$
4. `let rec fact x = if x < 0 then print_string "On se trompe?"
 else if x = 0 then 1
 else x * (fact (x-1));;`
Handwritten: (int, int) → int
5. `let t = [(1,2);(3,4,5)];;`
Handwritten: (int, int) list
6. `let t = [(1,2);(3,4,5)];;`
Handwritten: int list, vect = (1, 2, ..., 5)
7. `let u = t;;`
Handwritten: u is t
8. `u.(0) <- t.(1); u.(1) <- t.(0);;`
Handwritten: 2; 1
9. `u;;`
10. `t;;`
11. `let comp f g = function x -> f(g x) ;;`
Handwritten: comp ((a → b) → (c → a) → c → b) = (f ∘ g)
12. `let map_comp f l =
 let rec aux g l = match l with
 | [] -> []
 | h::tl -> (g h) :: (aux (comp f g) tl)
 in aux f l;;`

13. `map_comp (fun x -> 2*x) [1;2;3];;`

★ 14. `let rec boom () = boom ()
and gnam x = ();;`

★ 15. `gnam boom;;`

★ 16. `gnam (boom ());;`

Exercice 2. Écrire une version du programme suivant en style fonctionnel pur, c.à-d. qui n'utilise ni boucles, ni références, ni tableaux, ni autres données mutables. Attention ! la fonction doit avoir exactement le même type et le même comportement.

Indication : Vous pouvez déclarer des fonctions auxiliaires si vous le souhaitez.

```
let palindrome n =  
  let lignes = Array.make n "" in  
  print_endline ("Entrez "^(string_of_int n)^" lignes :");  
  for i = 0 to n-1 do  
    lignes.(i) <- read_line ()  
  done ;  
  print_endline "Voici vos lignes :";  
  for i = n-1 downto 0 do  
    print_endline (lignes.(i))  
  done ;
```

Exercice 3 (Les nombres parfaits). Cet exercice a pour but de programmer un test pour les nombres parfaits et deux extensions : les nombres semi-parfaits et super-parfaits.

Indication : L'utilisation de la récurrence terminale dans cet exercice sera appréciée.

Un nombre *parfait* est un entier naturel n tel que $\sigma(n) = 2 \times n$, où $\sigma(n)$ est la somme des diviseurs positifs de n . Par exemple, 6 est parfait car $\sigma(6) = 1 + 2 + 3 + 6 = 12 = 2 \times 6$.

1. Définir tout d'abord une fonction `add_divisor : int -> int -> int list -> int list` telle que l'expression `add_divisor n x l` renvoie la liste `x::l` si x est un diviseur de n , sinon elle renvoie `l`.

Indication : rappelez vous que $n \bmod x$ calcule le reste de la division entière de n par x

2. Définir ensuite une fonction auxiliaire `divisors : int -> int list` qui calcule la liste des diviseurs positifs d'un entier donné. Par exemple, `divisors 6` s'évaluera à `[1; 2; 3; 6]` (ou à n'importe quelle autre liste contenant les mêmes éléments).
3. En utilisant `divisors`, définir une fonction `perfect : int -> bool` qui détermine si un entier donné est parfait ou non. Par exemple, `perfect 6` s'évaluera à `true`, tandis que `perfect 5` s'évaluera à `false`.
4. Un nombre *m-super-parfait* (où m est un entier naturel strictement plus grand que 0) est un entier naturel n tel que :

$$\underbrace{\sigma(\dots \sigma(n) \dots)}_{m \text{ fois}} = 2 \times n.$$

Les nombres parfaits sont donc les nombres 1-super-parfaits. Un exemple de nombre 2-super-parfait est l'entier 16, car $\sigma(16) = 1 + 2 + 4 + 8 + 16 = 31$ et $\sigma(31) = 1 + 31 = 32$, donc $\sigma(\sigma(16)) = 32 = 2 \times 16$.

Définir une fonction `super_perfect: int -> int -> bool` tel que `super_perfect m n` renvoie `true` si n est un nombre m -super-parfait, et `false` sinon. Dans le cas où $m < 1$, la fonction devra déclencher une exception.

- ★ 5. Une définition équivalente de nombre parfait est la suivante : n est un nombre parfait si et seulement s'il est égal à la somme de ses diviseurs *stricts*, c'est-à-dire la somme de tous les diviseurs de n distincts de n . Par exemple, les diviseurs stricts de 6 sont 1, 2 et 3, et l'on a bien $6 = 1 + 2 + 3$.

Un nombre *semi-parfait* est un entier naturel n égal à la somme d'une partie de ses diviseurs stricts. Bien sur, tout nombre parfait est semi-parfait. Le nombre 12 est semi-parfait : ses diviseurs stricts sont 1, 2, 3, 4, 6, et l'on a bien $12 = 2 + 4 + 6$.

Définir une fonction `semi_perfect: int -> bool` déterminant si un nombre donné est semi-parfait ou non.

Exercice 4. Nous allons dans cet exercice modéliser des expressions arithmétiques, et implémenter quelques fonctions permettant de les manipuler. Les expressions considérées seront soit des constantes entières, soit la somme de deux expressions, soit le produit de deux expressions. Elle seront représentées à l'aide du type suivant :

```
type expr =
| Int of int
| Add of expr * expr
| Mul of expr * expr
```

- Donner la déclaration d'une variable `e1` de type `expr`, dont la valeur représente l'expression $((2 + 1) \times (3 \times 4))$.
- Définir une fonction `eval : expr -> int` calculant la valeur d'une expression en résolvant toutes ses opérations internes. Par exemple, pour la variable `e1` déclarée ci-dessus, `eval e1` s'évaluera en 36.
- On souhaite étendre le type `expr` afin de représenter des expressions pouvant contenir tous les éléments précédentes, ainsi que :
 - des inconnues, identifiées par des chaînes de caractères :

```
type unknown = string
```

— la division entière.

Étendre le type `expr` pour permettre la représentation de ces expressions étendues. Donner la déclaration d'une variable `e2` de type `expr`, représentant l'expression $\frac{2 \times e1}{x} + 1$, où `e1` est toujours la variable ci-dessus et `"x"` est une chaîne de caractères, identifiant d'une inconnue de type `expr`.

- Nous allons à présent écrire une nouvelle version de la fonction `eval` adaptée à ces expressions étendues. Ceci implique de prendre en compte deux nouveaux facteurs :
 - pour évaluer une expression contenant des inconnues, il est nécessaire de disposer d'une *affectation*, c'est-à-dire d'une liste d'association spécifiant la valeur d'une inconnue à partir de son identifiant. On ajoute donc à `eval` un nouveau paramètre de type `(unknown * int) list` représentant une telle affectation ;
 - l'évaluation d'une expression peut échouer, soit parce que le dénominateur d'une division vaut 0, soit parce que l'affectation fournie ne spécifie pas toutes valeurs des inconnues de l'expression. Pour traiter ces cases, on utilisera le type `'a option` comme type de retour de `eval` : si l'évaluation d'une expression échoue `eval` retournera `None`, sinon elle retournera `Some n`, où n est la valeur calculée pour l'expression.

Donner la nouvelle implémentation de cette fonction, de type

`eval : (unknown * int) list -> expr -> int option`

Par exemple, `eval [("x", 2)] e2` s'évaluera à `Some 37`, tandis que `eval [("x", 0)] e2` s'évaluera à `None`.

Indication : lorsqu'on applique une opération arithmétique (par exemple, le +), à deux sous-expressions, la valeur de l'expression résultante n'est bien définie que si celles des sous-expressions sont toutes les deux bien définies. Par exemple, `Add (e1, e2)` s'évalue en `Some (n + m)` si `e1` et `e2` s'évaluent respectivement en `Some m` et `Some n`. Si l'une au moins des deux expressions `e1`, `e2` s'évalue en `None`, alors `Add(e1, e2)` s'évalue aussi en `None`. Il peut être donc utile de définir une fonction auxiliaire

`lift : ('a -> 'b -> 'c) -> 'a option -> 'b option -> 'c option`

permettant d'étendre une fonction à deux arguments (comme (+) dans l'exemple ci-dessus) en une fonction opérant sur des types options. Rappelez-vous également qu'une division entière par 0 soulève l'exception `Division_by_zero`

- ★ 5. Nous souhaitons à présent étendre le type `expr` pour permettre la représentation de sommes *n*-aires, comme par exemple $\sum_{i=3}^5 3 \times i$, correspondant à $(3 \times 3) + (3 \times 4) + (3 \times 5)$. Les bornes inférieure et supérieure d'une somme *n*-aire pourront être des expressions quelconques, éventuellement avec des inconnues, comme par exemple :

$$\sum_{i=x+2}^y 3 \times i + x \quad (1)$$

La valeur de la somme ci-dessus dépend bien sûr de l'affectation choisie pour les inconnues *x* et *y* (le paramètre *i* n'est utilisé que par l'opérateur de somme *n*-aire).

Par exemple, si l'affectation choisie est `[(x, 2); (y, 6)]`, la somme (1) correspondra à $(3 \times 4 + 2) + (3 \times 5 + 2) + (3 \times 6 + 2)$ qui vaut 51 (plus exactement, `Some 51`), alors qu'avec l'affectation `[(x, 2); (y, 3)]`, la même expression correspond à une somme vide, de valeur 0 (`Some 0`). Quel devrait être la valeur de cette expression avec l'affectation vide ?

Étendre le type `expr` afin de permettre la représentation de sommes *n*-aires de la forme $\sum_{i=e_1}^{e_2} e_3$, où *i* est une inconnue, et *e*₁, *e*₂ et *e*₃ sont trois expressions quelconques. Donner la valeur de type `expr` correspondant à la somme (1).

- ★ 6. Enfin, donner une nouvelle version de la fonction `eval` pour qu'elle prenne en compte ce nouveau constructeur.

*Indication : vous pourriez avoir besoin de définir une fonction auxiliaire en recursion mutuelle avec `eval`, calculant la liste des valeurs de tous les termes d'une somme *n*-aire...*

~~A. Quelques fonctions utiles de la librairie d'OCaml~~

- `Array.make : int -> 'a -> 'a array`
`Array.make n x` returns a fresh array of length `n`, initialized with `x`. All the elements of this new array are initially physically equal to `x`.
- `List.assoc : 'a -> ('a * 'b)list -> 'b`
`List.assoc a l` returns the value associated with key `a` in the list of pairs `l`. That is, `List.assoc a [ ...; (a,b); ...] = b` if `(a,b)` is the leftmost binding of `a` in list `l`. Raise `Not_found` if there is no value associated with `a` in the list `l`.
- `List.exists : ('a -> bool)-> 'a list -> bool`
`List.exists p [a1; ...; an]` checks if at least one element of the list satisfies the predicate `p`. That is, it returns `(p a1) || (p a2) || ... || (p an)`.
- `List.fold_left : ('a -> 'b -> 'a)-> 'a -> 'b list -> 'a`
`List.fold_left f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`.
- `List.iter : ('a -> unit)-> 'a list -> unit`
`List.iter f [a1; ...; an]` applies function `f` in turn to `a1; ...; an`. It is equivalent to `begin f a1; f a2; ...; f an; ()end`.
- `List.map : ('a -> 'b)-> 'a list -> 'b list`
`List.map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`. Not tail-recursive.
- `List.mem : 'a -> 'a list -> bool`
`List.mem a l` is true if and only if `a` is equal to an element of `l`.