Types, listes et premiers

OPTION INFORMATIQUE - Devoir n° 1 - Olivier Reynet

Toutes les réponses à cet examen sont à fournir en langage OCaml.

A Types

A1. Indiquez, en justifiant votre réponse, le type des expressions `e` suivantes :

(a) `let e = 42`

Solution : `val e : int = 42`

(b) `let e = 42.`

Solution : `val e : float = 42.`

(c) `let e = true`

Solution : `val e : bool = true`

(d) `let e = [3 ; 5]`

Solution : `val e : int list = [3 ; 5]`

(e) `let e = let a = 3 in let b = 4 in a + b`

Solution : `val e : int = 7`

(f) `let e r = let pi = 3.14 in pi *. r *. r`

Solution : `val e : float -> float = <fun>`

(g) `let e = let pi2 = 3.14 in let pi5 = 3.14159 in pi5 - pi2`

Solution : Il faut écrire `-.` car : :
`Error : This expression has type float but an expression was expected of type int`

(h) `let e a b = let s = ref 0 in for i = 1 to b do s := !s + a done; !s`

Solution : `val e : int -> int -> int = <fun>`

(i) `let e a b = if a < b then a else b`

Solution : Cette fonction es polymorphe, elle s'applique à tout ce qui est comparable.

```
val e : 'a -> 'a -> 'a = <fun>
```

(j) `let k g b = if b = 0 then g else k (g*g)(b-1)`

Solution : Celle-ci est discutable... La bonne réponse est certainement que `e` ne possède aucun type... Pour les autres, il manque le mot clef `rec` car la fonction ainsi définie est récursive.

```
Error : Unbound value k. Hint: If this is a recursive definition, you should add the 'rec ' keyword on line 1
```

(k) `let e = let n = ref 3 in n := !n + 1`

Solution : `val e : unit = ()`

(l) `let e = let n = ref 0 in while !n < 10 do n := !n +1 done`

Solution : `val e : unit = ()`

B Listes et filtrage de motif

L'usage des fonctions du module List d'OCaml n'est pas autorisé.

B2. Écrire une fonction **récursive** de signature `length : 'a list -> int` qui renvoie la longueur d'une liste.

Solution :

```
let rec length l =
    match l with
    | [] -> 0
    | _ :: t -> 1 + length t;;
```

B3. Quelle est la complexité temporelle de la fonction `length` ?

Solution : La récurrence associée à la complexité de la fonction est $T(n) = 1 + T(n-1)$ si n est la longueur de la liste. Il s'agit d'une suite arithmérique de raison 1 et on a : $T(n) = n+1$. La complexité est linéaire en $O(n)$.

B4. Écrire une fonction de signature `second : 'a list -> 'a` qui renvoie le deuxième élément d'une liste s'il existe et lève une exception de message `"Action impossible"` dans le cas contraire.

Solution :

```
let second l =
    match l with
    | [] -> failwith "Impossible de renvoyer un deuxième élément"
    | [_] -> failwith "Impossible de renvoyer un deuxième élément"
    | _ :: b :: _ -> b;;
```

On suppose maintenant qu'on manipule une liste d'entiers.

B5. Écrire une fonction de signature `sum : int list -> int` qui renvoie la somme des entiers d'une liste d'entiers.

Solution :

```
let rec sum l =
    match l with
    | [] -> 0
    | e :: t -> e + sum t;;
```

B6. Écrire une fonction de signature `parite : int list -> bool list` qui renvoie la liste des parités de chaque élément d'une liste d'entiers. Par exemple, `parite [2 ;3 ;5 ;42]` renvoie `[true; false; false; true]`.

Solution :

```
let rec parite l =
    match l with
    | [] -> []
    | e :: t -> (e mod 2 = 0) :: parite t;;
(* let parite l= List.map (fun e -> e mod 2 = 0) l *)
```

B7. Écrire une fonction de signature `minimum : int list -> int` qui renvoie le minimum d'une liste d'entiers.

Solution :

```
let rec minimum l =
    match l with
        | [] -> failwith "Impossible, car liste vide !"
        | [e] -> e
        | e :: t -> let m =  minimum t  in if e < m then e else m;;
(* let minimum l = List.fold_left (fun acc e -> min acc e) (List.hd l) l *)
```

C Premiers

C8. Écrire une fonction de signature `nbdivinf : int -> int -> int` qui renvoie le nombre de diviseurs d'un nombre entier naturel inférieurs ou égaux à un autre. La fonction s'utilise ainsi : `nbdivinf n d`

où n et d sont des entiers naturels. Par exemple, `nbdivinf 40 9` renvoie 5 car 40 est divisible par les nombres 1, 2, 4, 5 et 8 qui sont inférieurs à 9. On fera attention de bien gérer le cas où le diviseur est zéro en levant une exception. On pourra garantir par une assertion que le diviseur est inférieur ou égal au nombre.

Solution :

```
let rec nbdivinf n d =
   assert (d <= n) ;
   match d with
      | 0 -> failwith "Division par zéro"
      | 1 -> 1
      | d when n mod d = 0 -> 1 + nbdivinf n (d-1)
      | d -> nbdivinf n (d-1) ;;
```

C9. Démontrer la correction de la fonction `nbdivinf n d`.

Solution : Tout d'abord on peut affirmer que la fonction termine. En effet, d est un entier naturel qui est décrémenté de 1 à chaque appel récursif. Donc la suite des paramètres des appels récursifs est entière, positive et strictement décroissante : elle atteint nécessairement 1 qui est la condition d'arrêt et l'algorithme se termine.

Pour la correction partielle, on procède par récurrence sur d.

Démonstration. Soit la propriété $\mathcal{P}(d)$: La fonction `nbdivinf n d` est correcte. Soit n un entier quelconque non nul.

(Initialisation) Pour $d = 1$, la fonction renvoie 1 qui est le seul diviseur de 1.

(Hérédité) Supposons que la fonction soit correcte pour un certain $d < n$. Soit m le nombre de diviseurs renvoyé par `nbdivinf n d`. Pour $d+1$, la fonction `nbdivinf n (d+1)` renvoie soit :

- $1+m$ si $(d+1)$ est facteur de n, ce qui est correct : tous les diviseurs inférieurs à d plus $d+1$.
- m sinon, c'est-à-dire tous les diviseurs de n inférieurs à d.

Le nombre de diviseur de n inférieur ou égal à $d+1$ et donc correctement calculé.

(Conclusion) La propriété est vérifié pour $d = 1$ et l'hérédité est vérifiée. Donc, la fonction se terminant, elle est correcte.

■

C10. En utilisant la fonction précédente, écrire une fonction de signature `nbdiviseurs : int -> int` qui renvoie le nombre de diviseurs d'un nombre.

Solution :

```
let nbdiviseurs x = nbdivinf x x
```

C11. En déduire une fonction de signature `estpremier : int -> bool` qui statue sur le fait qu'un nombre est premier. On rappelle que 1 n'est pas un nombre premier.

Solution :

```
let estpremier x =
  assert (x > 0);
  nbdiviseurs x = 2;;
```

C12. Quelle est la complextié temporelle de `estpremier` dans le pire des cas ?

Solution : $O(n)$ car la complexité de `nbdiviseurs` est linéaire par rapport au diviseur (récurrence reconnue $T(d) = 1 + T(d-1)$).

C13. Écrire une fonction de signature `listediviseurs : int -> int list` qui renvoie la liste des diviseurs d'un nombre. Par exemple, `listediviseurs 40` renvoie `[40 ; 20 ; 10 ; 8 ; 5 ; 4 ; 2 ; 1]`.

Solution :

```
let listediviseurs n =
  assert (n > 0);
  let rec aux d =
    match d with
      | 1 -> [1]
      | d when n mod d = 0 -> d :: aux (d-1)
      | d -> aux (d-1)
  in aux n;;
```