Paradigmes de programmation

-

Licence 3e année

-

Année 2011-2012

Responsable : René Thoraval

Inférence de type

Toutes les questions sont indépendantes et il faut considérer que pour chacune d'elles les évaluations se font dans une session OCaml dédiée.

1. Toutes les expressions OCaml suivantes sont syntaxiquement correctes. Pour chacune, dire si elle a une valeur. Dans ce cas : laquelle et de quel type ? Dans le cas contraire : pourquoi ?

```
5 * 4 * 3 * 2 * 1 ;;
true ;;
if 2 > 0 then "true"
         else "false" ;;
if 2 > 0 then true
         else false ;;
if 2 > 0 then "oui"
         else "non" ;;
if 2 > 0 then oui
         else non ;;
if 2 > 0 then true
         else "false" ;;
3.14 +. 2. ;;
3.14 +. 2 ;;
3.14 + 2.0 ;;
3.14 > 2. ;;
3.14 > 2 ;;
int_of_float (3.14) > 2 ;;
3.14 > float_of_int (2) ;;
3 > 2 ;;
"trois" > "deux" ;;
"deux" > "un" ;;
5 / 2 ;;
5. /. 2. ;;
sqrt (4.) ;;
sqrt (4) ;;
sqrt ;;
exp (1.) ;;
log (exp (1.)) ;;
5 / 0 ;;
if true then 5 / 0
        else 3 ;;
if true then 3
        else 5 / 0 ;;
if false then 5 / 0
         else 3 ;;
if false then 3
         else 5 / 0 ;;
(2 / 0 = 5) && true ;;
true && (2 / 0 = 5) ;;
(2 / 0 = 5) && false ;;
false && (2 / 0 = 5) ;;
(2 / 0 = 5) || true ;;
true || (2 / 0 = 5) ;;
(2 / 0 = 5) || false ;;
false || (2 / 0 = 5) ;;
"s" ^  "i" ;;
‘s‘ ^  "i" ;;
```

```
‘s‘ ^  ‘i‘ ;;
"s" ^ i ;;
si ;;
"if" ;;
function x -> x > 0 ;;
(function x -> x > 0) (12) ;;
function x -> 2 * x ;;
(function x -> 2 * x) (7) ;;
function (x, y) -> (x + y, x - y) ;;
function x -> x / 0 ;;
(function x -> x / 0) (12) ;;
(function x -> x) (5) ;;
(function x -> x) ("oui") ;;
function x -> x ;;
(function x -> 1) (5) ;;
(function x -> 1) (5.) ;;
function x -> 1 ;;
(function (x, y) -> (y, x)) (1, 2) ;;
(function (x, y) -> (y, x)) (1, "oui") ;;
(function (x, y) -> (y, x)) ("oui", "oui") ;;
function (x, y) -> (y, x) ;;
function x -> (x, x) ;;
(function x -> (x, x)) (1) ;;
let aaa = 2 and bbb = 3 * aaa in aaa + bbb ;;
let aaa = 2 in let bbb = 3 * aaa in aaa + bbb ;;
let aaa = 2 in let bbb = 3 * aAa in aaa + bbb ;;
```

2. Toutes les phrases OCaml suivantes sont syntaxiquement correctes. On suppose qu'elles sont interprétées l'une après l'autre. Pour chacune, dire si c'est une définition globale `let` $\ldots = e$ ou une expression e. Dire si e a une valeur. Dans ce cas : laquelle et de quel type ? Dans le cas contraire : pourquoi ?

```
let absolu = function x -> if x >= 0 then x else - x ;;
absolu (absolu (-12)) ;;
let comp = function (x, y, z) -> (x <= y, y <= z) ;;
(comp (3, 2, 4), comp (2, 3, 4)) ;;
comp ("jaune", "rouge", "vert") ;;
let approx = function (a, b) -> let (b1, b2) = comp (a - 1, b, a + 1) in b1 && b2 ;;
approx (12, 13) ;;
approx ("mauve", "violet") ;;
let ff = function x -> x + 1 in ff ;;
ff (5) ;;
let ff = function x -> x + 1 ;;
ff (5) ;;
let ff = function x -> x + 1. ;;
ff (5) ;;
let ff = function x -> x +. 1. ;;
ff (5) ;;
(let ff = function x -> x + 1 in ff) (5) ;;
ff (5) ;;
ff ;;
let fff = function b -> b ;;
fff (3) ;;
fff ("oui") ;;
let ffff c = c ;;
ffff (3) ;;
ffff ("oui") ;;
```

3. Donner une valeur de chacun des types suivants (il est inutile de la nommer), l'évaluer, puis l'appliquer :

```
int -> int
int * int -> bool
string -> bool
float * float -> string
bool * bool -> bool
int -> int * bool
```

4. Définir `f` et `g` (autrement dit, lier chacun de ces noms à une valeur) pour que l'expression suivante ait une valeur :

```
f (f (g (2, true), 4))
```

5. Toutes les phrases OCaml suivantes sont syntaxiquement correctes. On suppose qu'elles sont interprétées l'une après l'autre. Pour chacune, dire si c'est une définition globale `let` ... $= e$ ou une expression e. Dire si e a une valeur. Dans ce cas : laquelle et de quel type ? Dans le cas contraire : pourquoi ?

```
let a = 5
in 2 * a ;;
a ;;
let a = 7
in (a, a) ;;
let a = 1 and b = 3 and c = 4
in a + b + c ;;
let (a, b, c) = (1, 3, 4)
in a + b + c ;;
let a = 5 ;;
a ;;
let a = 3
in 2 * a ;;
a ;;
let a = "oui" ;;
a ;;
let x = 5 ;;
let x = 3
in 2 * x ;;
let y = x + 1 ;;
let x = 4 and y = 2
in x * y ;;
y ;;
let y = let x = 7 and y = 9
        in y - x ;;
let y = y = 9 ;;
failwith ;;
```

6. Même question.

```
let f = function
  x -> if x <> 0 then x / abs (x)
                 else failwith ("f : s'applique a une valeur non nulle") ;;
(f (-7), f (5)) ;;
f (0) ;;
let rec fois = function (a, b) -> if b = 0 then 0
                                           else a + fois (a, b - 1) ;;
```

```
fois (5, 7) ;;
let rec modulo8 = function n -> if n < 0 then modulo8 (n + 8)
                                         else if n >= 8 then modulo8 (n - 8)
                                                        else n ;;
modulo8 (-13) ;;
let rec repete = function (mot, n) -> if n = 0 then "!"
                                               else mot ^ repete (mot, n - 1) ;;
repete ("Ah ", 5) ;;
let rec f = function x -> f (x) ;;
f (3) ;;
let rec puissance = function (x, n) ->
  if n = 0 then 1.0
           else if n < 0 then 1.0 /. puissance (x, - n)
                         else x *. puissance (x, n - 1) ;;
puissance (1.2, 4) ;;
puissance (5.0, -3) ;;
let rec est_pair = function n -> if n = 0 then true
                                          else est_pair (n - 2) ;;
est_pair (12) ;;
est_pair (11) ;;
```

7. Toutes les phrases OCaml suivantes sont syntaxiquement correctes. On suppose qu'elles sont interprétées l'une après l'autre. Pour chacune, dire si c'est une définition globale `let` ... $= e$ ou une expression e. Dire si e a une valeur. Dans ce cas : laquelle et de quel type ? Dans le cas contraire : pourquoi ?

```
open List ;;
let li_1 = 7 :: (3 :: (11 :: [])) ;;
2 :: 3 ;;
[] :: 7 ;;
let li_2 = 9 :: li_1 in li_2 ;;
li_1 ;;
li_2 ;;
[] ;;
let li_2 = 6 :: 5 :: 9 :: [] ;;
li_2 :: (li_1 :: []) ;;
6 :: (li_1 :: []) ;;
let li_3 = 6 :: (5.3 :: (9 :: [])) ;;
let li_4 = "ours" :: "vache" :: "renard" :: [] ;;
let li_5 = "miel" :: "herbe" :: "poule" :: [] ;;
li_4 :: li_5 ;;
tl (tl ("oui" :: [])) ;;
(2, 3.5) :: (0, 4.) :: [] ;;
hd (tl ((2 = 0) :: (if 4 > 2 then 3 < 5 else 3 > 5) :: [])) ;;
(if 4 > 2 then 3 < 5 else 3 > 5) :: [] ;;
li_1 :: [] :: (4 :: li_2) :: [] ;;
tl (5 :: []) ;;
let rec f1 = function li -> if tl (li) = [] then hd (li) else f1 (tl (li)) ;;
let rec f2 = function (li, n) -> if n = 1 then hd (li) else f2 (tl (li), n - 1) ;;
[] :: [] ;;
([] :: []) :: [] ;;
sqrt :: log :: exp :: [] ;;
() ;;
(function () -> []) :: [] ;;
[] :: (function () -> []) ;;
[] :: ((function () -> []) ()) ;;
```

```
(function () -> []) :: (function () -> []) :: [] ;;
let rec f3 = function (li, li1) ->
  if li = []
    then li1
    else if li1 = []
           then li
           else if hd (li) <= hd (li1)
                  then hd (li) :: f3 (tl (li), li1)
                  else hd (li1) :: f3 (li, tl (li1)) ;;
let rec f4 = function li ->
  if li = []
    then (li, [])
    else if tl (li) = []
           then (li, [])
           else let
                  (h1, h2) = f4 (tl (tl (li)))
                in
                  ((hd (li) :: h1), (hd (tl (li)) :: h2)) ;;
let rec f5 = function li ->
  if li = [] then li
             else if tl (li) = []
                    then li
                    else let
                           (h1, h2) = f4 (li)
                         in
                           f3 (f5 (h1), f5 (h2)) ;;
let rec f6 (li, li1 ) =
  if li = [] && li1 = []
    then []
    else ((hd (li), hd (li1)) :: f6 (tl (li), tl (li1))) ;;
let rec f7 = function li -> if li = [] then f7 (0 :: li) else f7 (li) ;;
f7 (2 :: []) ;;
let rec f8 (li) = if li = [] then tl (li) else f8 (tl (li)) ;;
f8 (2 :: []) ;;
let rec f9 = function
  li -> if li = []
          then []
          else if tl (li) = [] then hd (li) else tl (li) ;;
f9 (2 :: []) ;;
let rec f10 li = if li = [] then li else hd (li) :: f10 (tl (li)) ;;
f10 (3 :: 2 :: []) ;;
hd (tl (tl (li_1))) ;;
3.5 :: [] = [] ;;
tl (tl (tl (li_4))) = [] ;;
(f1 (li_1), f1 (li_4)) ;;
let
  li = f5 (f3 (li_1, li_2))
in
  (f2 (li, 1), f2 (li, 2), f2 (li, 3), f2 (li, 4), f2 (li, 5), f2 (li, 6)) ;;
li ;;
let
  li = f5 (f3 (li_4, li_5))
in
  (f2 (li, 1), f2 (li, 2), f2 (li, 3), f2 (li, 4), f2 (li, 5), f2 (li, 6)) ;;
let
  li = f6 (li_4, li_5)
```

```
in
  (f2 (li, 1), f2 (li, 2), f2 (li, 3)) ;;
[] :: [[]] ;;
[] :: [] ;;
[] :: [[[]]] ;;
[[]] :: [] :: [[[]]] ;;
[] :: [[[]]] ;;
[[]] :: [] :: [[[]]] :: [] ;;
```

8. Toutes les phrases OCaml suivantes sont syntaxiquement correctes. On suppose qu'elles sont interprétées l'une après l'autre. Pour chacune, dire si c'est une définition globale `let` ... $= e$ ou une expression e. Dire si e a une valeur. Dans ce cas : laquelle et de quel type ? Dans le cas contraire : pourquoi ?

```
open List ;;
(if 1 > 0 then
            if 2 > 3 then 1
                     else 0
          else 5)
+
(if 1 > 2 then 4
          else
            if 4 > 3 then 7
                     else 8) ;;
let
  li = 4 :: 3 :: []
in
  (if hd (li) > hd (tl (li)) then 2 else 3)
  +
  (if hd (li) > 1 then 2 else 3) ;;
(function (x, y) -> x :: (x :: y)) (2, []) ;;
let rec f = function (x, n) -> if n = 0 then []
                                       else x :: f (x, n - 1) ;;
f (3, 2) ;;
f (f (3, 2), 1) ;;
f (f (f (3, 2), 1), 0) ;;
if (function x -> x + 1) (2) > 2 then (function x -> 5 * x)
                                 else (function x -> 3 * x + 4) ;;
(if 3 > 2 then (function x -> 3 * x)
          else (function x -> 2 * x)) (5) ;;
let rec
  g = function n -> if n = 0 then 1 else n * g (n - 1)
in
  g ;;
g ;;
(let rec
   g n = if n = 0 then 1 else n * g (n - 1)
 in
   g)
(5) ;;
g ;;
```

9. Toutes les phrases OCaml suivantes sont syntaxiquement correctes. On suppose qu'elles sont interprétées l'une après l'autre. Pour chacune, dire si c'est une définition globale `let` ... $= e$ ou une expression e. Dire si e a une valeur. Dans ce cas : laquelle et de quel type ? Dans le cas contraire : pourquoi ?

```
function x -> (function y -> x + y) ;;
(function x -> (function y -> x + y)) (3) ;;
((function x -> (function y -> x + y)) (3)) (4) ;;
(function x -> "function y -> x + y") (3) ;;
function x -> "function y -> x + y" ;;
(exp (1.), log (1.), log (exp (1.))) ;;
let mul = function (f, x, y) -> exp (f (log (x), log (y))) ;;
let entier = function (f, x) -> int_of_float (f (float_of_int (x))) ;;
let reel = function (f, x) -> float_of_int (f (int_of_float (x))) ;;
let divisible_par = function n -> (function k -> k mod n = 0) ;;
let commente = function f -> (function x -> "f(|x|)=" ^ string_of_int (f (abs (x)))) ;;
let test = function c -> (function x -> x = c) ;;
mul ((function (a, b) -> a +. b), 4., 5.) ;;
mul ((function (a, b) -> a *. b), 2., 5.) ;;
(entier (sqrt, 25), reel ((function x -> x * x), 5.)) ;;
let par6 = divisible_par (6) and par4 = divisible_par (4) ;;
(function n -> par6 (n) && par4 (n)) (48) ;;
(commente (function x -> entier (sqrt, x))) (- 25) ;;
let non = test ("negation") ;;
non ("vrai") || non ("faux") ;;
```

10. Donner au moins une valeur de chacun des types suivants, dont l'évaluation n'engendre pas l'avertissement `<< Attention: ce filtrage n'est pas exhaustif. >>` :

`int -> int -> int`	c'est-à-dire : `int -> (int -> int)`
`(int -> int) -> int`	
`int -> int -> bool`	c'est-à-dire : `int -> (int -> bool)`
`(int -> int) -> bool`	
`int * (int -> int) -> bool -> int`	c'est-à-dire : `(int * (int -> int)) -> (bool -> int)`

11. Toutes les phrases OCaml suivantes sont syntaxiquement correctes. On suppose qu'elles sont interprétées l'une après l'autre. Pour chacune, dire si c'est une définition globale `let` ... $= e$ ou une expression e. Dire si e a une valeur. Dans ce cas : laquelle et de quel type ? Dans le cas contraire : pourquoi ?

```
open List ;;
let lx = [1 ; 2 ; 3] ;;
map (function x -> x + 7) ;;
(map (function x -> x + 7)) (lx) ;;
let f1 = map (function x -> (function y -> x + y)) ;;
f1 (lx) ;;
let applique_les_fonctions = function (lf, x) -> (map (function f -> f (x))) (lf) ;;
applique_les_fonctions (f1 (lx), 5) ;;
let rec accumule (op, neutre) =
  (let rec
     f lx = if lx = []
              then neutre
              else op (hd (lx), f (tl (lx)))
   in
     f) ;;
let ajout (x, lx) = x :: lx ;;
let f2 = function x, y -> (accumule (ajout, y)) (x) ;;
f2 (lx, lx) ;;
let f3 = function x -> (accumule (f2, [])) (x) ;;
let lxx = ajout (lx, ajout (lx, ajout (lx, ajout (lx, [])))) ;;
f3 (lxx) ;;
```

```
let f4 = function
  ly -> (accumule ((function x, y -> x + y), 0)) (map (function x -> 1) (ly)) ;;
f4 (lx) ;;
f4 (f3 (lxx)) ;;
let filtre = function
  pred -> let
            f = function x, y -> if pred (x) then x :: y else y
          in
            accumule (f, []) ;;
(filtre (function x -> x = 1)) (f3 (lxx)) ;;
let filtre_bis = function
  pred -> let
            f = function
              x -> if pred (x) then x :: [] else []
          in
            function lx -> f3 ((map (f)) (lx)) ;;
(filtre_bis (function x -> x = 1)) (f3 (lxx)) ;;
```

12. Donner au moins une valeur de chacun des types suivants, dont l'évaluation n'engendre pas l'avertissement `<< Attention: ce filtrage n'est pas exhaustif. >>` :

`(int -> int) -> int -> int`	c'est-à-dire : `(int -> int) -> (int -> int)`
`(int -> bool) -> int -> bool`	c'est-à-dire : `(int -> bool) -> (int -> bool)`

13. Toutes les phrases OCaml suivantes sont syntaxiquement correctes. On suppose qu'elles sont interprétées l'une après l'autre. Pour chacune, dire si c'est une définition globale `let` ... $= e$ ou une expression e. Dire si e a une valeur. Dans ce cas : laquelle et de quel type ? Dans le cas contraire : pourquoi ?

```
open List ;;
let mul = function f -> (function (x, y) -> exp (f (log (x), log (y)))) ;;
let entier = function f -> (function x -> int_of_float (f (float_of_int (x)))) ;;
let rec pf = function f -> (function x -> if x = f (x) then x else (pf (f)) (f (x))) ;;
let comp = function
  (f1, f2, f3) -> (function x -> let (a, b) = f1 (x) in f3 (f2 (a), f2 (b))) ;;
let fois = mul (function (a, b) -> a +. b) ;;
fois (3., 5.) ;;
let racine = entier (sqrt) ;;
racine (81) ;;
(pf (function x -> (x +. 2.) /. (x +. 1.))) (0.) ;;
let c = comp ((function x -> x / 4, x mod 4),
              (function x -> x * 3),
              (function q, r -> 4 * q + r)) ;;
c (9) ;;
let rec accumule = function
  (op, neutre) -> (let rec
                     f = function
                       lx ->  if lx = []
                                then neutre
                                else op (hd (lx), f (tl (lx)))
                   in
                     f) ;;
let decurryfie = function f -> (function (x, y) -> (f (x)) (y)) ;;
let echange = function f -> (function x -> (function y -> (f (y)) (x))) ;;
let accumule_bis = function
  f -> echange (function neutre -> accumule (decurryfie (f), neutre)) ;;
```

```
accumule_bis (function x -> (function y -> x + y)) ;;
(accumule_bis (function x -> (function y -> x + y))) ([1 ; 2 ; 3]) ;;
((accumule_bis (function x -> (function y -> x + y))) ([1 ; 2 ; 3])) (7) ;;
((accumule_bis (function x -> (function y -> x - y))) ([1 ; 2 ; 3])) (7) ;;
fold_left ;;
fold_right ;;
((fold_right (function x -> (function y -> x + y))) ([1 ; 2 ; 3])) (7) ;;
((fold_right (function x -> (function y -> x - y))) ([1 ; 2 ; 3])) (7) ;;
```