Exercice 1 : Expressions, expressions, expressions

Pour chacune des expressions suivantes, donner son type, sa valeur et les étapes de calcul dans le modèle équationnel (il est possible que certaines expressions non correctes se soient glissées dans la liste, repérez les) :

(a)
```
1
```

(b)
```
1.5 *. 3. +. 2.
```

(c)
```
(2, 4)
```

(d)
```
(2, if true then true else false)
```

(e)
```
(3 * 5) + 2
```

(f)
```
((fun x -> 2 * x) 7)
```

(g)
```
((fun x -> 2 * x) (3 * 5))
```

(h)
```
(((fun x -> (fun y -> x * y)) 3) 7)
```

(i)
```
((fun x -> fun y -> x * y) 3 7)
```

(j)
```
(fun f -> fun x -> (f x) + x) (fun x -> x - 1) 3
```

(k)
```
let (x, y) = (1, 2) in
x + y
```

(l)
```
let a = (1, 2) in
let (x, y) = a in
x + y
```

(m)
```
let f x = x + 1 in
(f, 3)
```

(n)
```
let b = true in
if b then (fun x -> x)
else 0
```

(o)
```
let b = true in
((if b then (fun x -> x + 1)
  else (fun x -> x - 1)) 5)
```

(p)
```
((fun b ->
    ((fun x -> fun y -> 2 * x + 3 * y) (if b then 1 else 2) 5)
  ) true)
```

(q)
```
let f = fun x -> not x in
let g = fun f -> fun x -> fun y -> (f (not x)) && (f y) in
(g f true false)
```

Exercice 2 : 'a, 'b ?

Q. 1 Quel est le type de `fun x -> x`, le tester dans votre machine. Même question avec `fun f -> (fun x -> f x)`.

Q. 2 Définir la fonction identité `id` prenant en argument une valeur `x` et calculant cette même valeur.

Q. 3 Pouvez-vous appliquer cette fonction à des éléments de types `int` et à des éléments de types `float` ? Même question avec des éléments de types `float * int` et `int -> int`.

Exercice 3 : Ordre supérieur.

Q. 1 On souhaite définir une fonction `application` prenant en arguments une fonction `f` et un élément `x` et calculant `f(x)`. Donner le type le plus général possible pour la fonction `application`. Implanter la fonction `application`.

Q. 2 On souhaite définir une fonction `compose` prenant en arguments une fonction `f` et une fonction `g` et calculant la fonction $f \circ g$. Donner le type le plus général possible pour la fonction `compose`. Implanter la fonction `compose`.

Q. 3 On souhaite définir une fonction `f_ou_ident` prenant en arguments une fonction `f` et un booléen `x` et calculant : la fonction `f` si le booléen est vrai et la fonction identité sinon. Donner le type le plus général possible pour la fonction `f_ou_ident`. Implanter la fonction `f_ou_ident`.

Exercice 4 : N-uplets

Q. 1 Définir les fonctions `fst: ('a * 'b) -> 'a` (resp. `snd: ('a * 'b) -> 'b`) projetant une paire sur sa première (resp. seconde) composante.

Q. 2 Définir une fonction `paire: 'a -> 'b -> ('a * 'b)` qui, à partir de deux éléments a et b construit la paire (a, b).

Q. 3 Déduire de la question précédente une fonction `paire_true: 'a -> (bool * 'a)` qui à un élément a associe la paire (`true`, a).

Q. 4 Définir une fonction `curry: (('a * 'b) -> 'c) -> ('a -> 'b -> 'c)` prenant en argument une fonction `f` attendant une paire comme argument, et calculant une fonction de deux arguments `g` telle que pour tout `x`, `y`, on ait `(g x y) = (f (x, y))`.

Q. 5 Définir une fonction `uncurry: ('a -> 'b -> 'c) -> (('a * 'b) -> 'c)` prenant en argument une fonction `g` à deux arguments, et calculant une fonction `f` attendant une paire telle que pour tout `x`, `y`, on ait `(f (x, y)) = (g x y)`.

Exercice 5 : Puissances

Le langage OCaml ne fournit pas de fonction prédéfinie permettant le calcul de x^n.

Q. 1 Donner une mise en équation récursive de x^n où x et n sont des entiers.

Q. 2 Implanter une fonction `pow : int -> int -> int` telle que `(pow x n)` $= x^n$.

L'expression OCaml `failwith "message d'erreur"`, lors de son évaluation, arrête l'évaluation courante et retourne à l'utilisateur le message d'erreur `"message d'erreur"`.

Q. 3 Au moyen de la fonction `failwith` redéfinissez votre fonction `pow : int -> int -> int` pour la protéger contre un utilisateur qui lui passerait en argument un exposant négatif. Votre fonction devra faire comprendre à l'utilisateur qu'il a fait une mauvaise utilisation de votre fonction `pow`.

Exercice 6 : Récurrences mutuelles

On considère les deux suites définies par

$$\begin{array}{ll} u_0 = 0 & u_{n+1} = v_n \\ v_0 = 1 & v_{n+1} = v_n + u_n \end{array}$$

Q. 1 Donner les 3 premiers termes des suites (u_n) et (v_n).

Q. 2 Donner les définitions (mutuellement récursives) des fonctions `uu : int -> int` et `vv : int -> int` calculant respectivement le n-ième terme de la suite (u_n) (resp. de la suite (v_n)) où n est l'argument de ces deux fonctions.

Q. 3 Donner la suite d'équations représentant le calcul lors de l'appel `(uu 3)`.

Exercice 7 : Sommes

Q. 1 Définir la fonction de signature `sum (n:int) : int` qui calcule la somme des n premiers nombres entiers strictement positifs. Par exemple : `(sum_n 5)` $= 5+4+3+2+1$ et `(sum_n 0)` $= 0$.

Q. 2 En utilisant une fonction locale, redéfinir la fonction `sum_n` de manière à ce qu'elle lève une exception si `n` est négatif.

Q. 3 Définir la fonction de signature `sum_p (n:int) : int` qui calcule la somme des n premiers nombres pairs positifs.

Q. 4 Définir la fonction de signature `sum_f (f:int -> int) (n:int) : int` qui calcule la somme `(f n) + (f (n-1)) + ... + f(0)`.

Q. 5 Utiliser cette fonction pour redéfinir `sum_p`. Vérifiez que c'est bien la *même fonction* que vous avez définie.

Exercice 8 : Ordre supérieur et polynômes

Dans cet exercice on souhaite utiliser le pouvoir d'expression de l'ordre supérieur afin de représenter une collection de valeurs. Nous définissons un polynôme comme une paire `int * (int -> int)` ayant le sens suivant : la paire `(d, f)` représente le polynôme :

$$\sum_{i=0}^{d-1} \texttt{(f i)} X^i$$

Q. 1 Définir le polynôme nul `nul`.

Q. 2 Définir une fonction prenant en argument un entier n et retournant le polynôme $\sum_{k=0}^{n} k! X^k$.

Q. 3 Définir la fonction `eval : (int * (int -> int)) -> int -> int` prenant en arguments un polynôme P et un entier n et calculant $P(n)$.

Q. 4 Définir une fonction `add : (int * (int -> int)) -> (int * (int -> int)) -> (int * (int -> int))` calculant la somme de deux polynômes.

Q. 5 Définir une fonction `mul : (int * (int -> int)) -> (int * (int -> int)) -> (int * (int -> int))` calculant le produit de deux polynômes.