Expressions et Fonctions

Exercice 1

Vérifiez si les expressions suivantes sont correctes et si c'est le cas donnez leur type et leur valeur :

```
(3 * (7 - 5))                 7.1 +. 2.2
7.2 / 2                       7. /. 2.
7 / 2                         3 * 4
3. *. 4.                      3 * 4.
7. mod 2.                     7 mod 2
int_of_float(10.0 /. 2.5)     int_of_float(10.0) / int_of_float(2.5)
(true || false)               true && false
not (true && false)           not (true) && not(false)
(1>2) || (1<2)                (12.0 +. 30. = 42.) && (3 + 4 = 4 + 4)
true = false                  "bon" ^ "jour"
not("true")                   true = "true"
if 1<2 then 3+1 else 5/2      if 1 < 3/2 then if 2 < 3/2 then 1 else 2 else 3
if 3/2 < 2 then 4.1           if 3/2<2 then 4<5 else 4+5
```

Exercice 2

Soient f et g deux fonctions de signature f : int $\longrightarrow$ bool et g : int$\times$bool $\longrightarrow$ int.
Indiquez si les expressions suivantes sont bien écrites et bien typées et si c'est le cas calculez leur type :

```
 2+f(3)          f(3)||f(4)   g(2,f(2))
 f(g(2,f(2)))    f(2,g(3))    g(2,f(2))+f(2)
if f(2) then g(1,true) else 3/4
if f(2) then f(2) else g(2,f(2))
if g(1,true)/2 > 5 then f(2) else 5/3<2
```

Exercice 3

Calculez les types et valeurs des expressions :

Expression	Type	Valeur
`let x=5 in x*x`		
`let a=2 and b=true in b && (a>0)`		
`let a=2 and b=5 in a+2*b`		
`let a=5 and b=a+1 in a+2*b`		
`let a=3 and b=5 in (a<b) && ((a>b)/2)`		
`let a=1 and b=1 and c=2 in a=b=c`		

Exercice 4

Écrivez l'expression $\frac{\sqrt{2,5}}{\sqrt{2,5}+1}$ en OCaml en n'utilisant qu'une fois la fonction `sqrt` (racine carrée).
Écrivez en OCaml l'expression $x^2 + x^4 + x^5$ en n'utilisant que 3 multiplications. On suppose x de type entier.

Exercice 5

— De quel type doivent être x et y pour que chacune des expressions ci–dessous soit correctement typée ?

```
x+1              x||y                  (x+1)||y                  x=y
(x=y)||(x=y+1)   if x then y else 3    not(x)||((x mod 2)=0)
```

— Quelle doit être la signature des fonctions f et g pour que chacune des expressions ci–dessous soit correctement typée ?

```
f(1)+1   g(1,true)&&f(1)   f(f(1))   f(true)+.g(2)   g(1+f(3))   f(x) || (f(x)=0)
```

Définition de fonction

Exercice 6
Écrivez les définitions OCaml des fonctions `f, g, cube` définies par :

$$f : \begin{array}{ccc} \mathbb{Z} & \longrightarrow & \mathbb{Z} \\ x & \longmapsto & 2x+1 \end{array} \qquad g : \begin{array}{ccc} \mathbb{R}\times\mathbb{R} & \longrightarrow & bool \\ (x,y) & \longmapsto & x < \frac{y}{2} \end{array} \qquad cube : \begin{array}{ccc} \mathbb{R} & \longrightarrow & \mathbb{R} \\ x & \longmapsto & x^3 \end{array}$$

Exercice 7
Quelle est la signature des fonctions :

```
let f1 = function (x,y) →
  if y then x+1 else 2*x
```

```
let f2 = function (x,y) →
  if x > y then x > y else y > 2*x
```

Exercice 8
Soit la fonction calculant le périmètre d'un cercle de rayon r. Écrivez sous forme mathématique cette fonction, puis sa définition OCaml. Vous prendrez 3.14 comme valeur pour π.
Faites de même pour la fonction vérifiant si deux entiers sont de même signe.

Exercice 9
Définissez en OCaml les fonctions qui calculent :
- la valeur absolue d'un entier [1]
- le maximum de deux entiers [2]
- le maximum de trois entiers
- la valeur médiane de trois entiers

Exercice 10
Définissez en OCaml les fonctions qui testent si trois réels sont les mesures :
- d'un triangle équilatéral
- d'un triangle

Exercice 11
Définissez la fonction correspondant à l'opérateur booléen "implication" dont la sémantique est :

a	b	implication(a,b)
true	true	true
true	false	false
false	false	true
false	true	true

Vous donnerez une définition utilisant l'expression conditionnelle `if..then..else..` et une définition n'utilisant que les opérateurs booléens `not || et &&`.

Récursivité

Exercice 12
Soit la fonction $f : \begin{array}{ccc} \mathbb{N} & \longrightarrow & \mathbb{N} \\ n & \longmapsto & \text{5 si n=0} \\ & & \text{2n + f(n-1) sinon} \end{array}$
- calculez `f(1), f(2), f(3)`
- de manière générale combien vaut `f(n)` ?
- écrivez cette fonction en OCaml

1. La fonction valeur absolue est déjà définie en OCaml (`abs`).
2. Cette fonction est également prédfinie en OCaml (`max`) mais nous le verrons sa signature est différente.

Exercice 13

Définissez par récurrence la fonction $som : \begin{array}{ccc} \mathbb{N} & \longrightarrow & \mathbb{N} \\ n & \longmapsto & 0+1+2\cdots+n \end{array}$:

- **Cas de base** quand n=0 `som(n)` =
- **Récurrence** quand n>0 `som(n)` =

Traduisez en OCaml.

Exercice 14

Écrivez en OCaml la fonction $somInt : \begin{array}{ccc} \mathbb{N}\times\mathbb{N} & \longrightarrow & \mathbb{N} \\ (a,b) & \longmapsto & \text{somme des entiers de l'intervalle [a ;b]} \end{array}$

Exercice 15

Écrivez la fonction `mul` qui calcule le produit de 2 entiers naturels, sans utiliser l'opérateur $\star$, en n'utilisant qu'additions et soustractions. Commencez par définir `mul(a,b)` par récurrence sur `a`, puis traduisez en OCaml.

- **Cas de base** quand `a=0`, `mul(a,b)` =
- **Équation de récurrence :** quand `a>0` `mul(a,b)` =

Exercice 16

On cherche un algorithme calculant le carré d'un entier naturel, sans utiliser de multiplication (en n'utilisant qu'additions et soustractions). Appelons `carre` cette fonction.

Pour calculer `carre(n)` on utilise un schéma récursif :

- **Cas de base :** quand `n=0` que vaut `carre(n)` ?
- **Équation de récurrence :** quand `n`$\neq$`0` essayez de définir `carre(n)` en fonction de `carre(n-1)` (pour cela développez l'expression $(n-1)^2$).

Écrivez la fonction `carre`.

Exercice 17

Pour tester si un entier n est pair il suffit de vérifier si `n mod 2 = 0`. Écrivez la fonction `estPair` qui teste si un entier naturel est pair, sans utiliser de division, que ce soit `mod` ou `/`.

Exercice 18

Un entier $n \in \mathbb{N}$ est une puissance de 10, s'il s'écrit $n = 10^k$ ($k \in \mathbb{N}$). Vous devez écrire la définition d'une fonction qui teste si un entier est une puissance de 10, sans utiliser l'opération `log`. Pour cela posez-vous les questions suivantes :

- quelle est la plus petite puissance de 10 ?
- est-ce que les autres puissances de 10 sont des multiples de 10 ?
- est-ce qu'un multiple de 10 est une puissance de 10 ? Quelle condition faut-il ajouter pour obtenir une condition nécessaire et suffisante au fait d'être une puissancede 10 ?

Exercice 19

On cherche à calculer le nombre de façons de répartir n objets dans b boîtes. Les boites sont **différenciées** et peuvent être **vides**. Par exemple les répartitions de 3 objets dans 2 boîtes sont | 0 | 3 | | 1 | 2 | | 2 | 1 | | 3 | 0 | et donc le nombre de répartitions est 4.

On appelle `nbrep(n,b)` ce nombre. Pour trouver l'équation de récurrence définissant `nbrep(n,b)` on distingue :

- les répartitions où la première boite est vide. Dans ce cas il faut répartir l'ensemble des objets dans les autres boites
- les répartitions où le première boite contient au moins 1 objet. Dans ce cas il faut répartir les autres objets dans l'ensemble des boites

En déduire l'équation de récurrence puis les cas de base (les nombres de boites et d'objets sont des entiers naturels). Définissez la fonction OCaml calculant ce nombre de répartitions.

Comment modifier la définition pour ne compter que les répartitions dans des boîtes différenciées **non vides** ? Il n'y a que 2 façons de répartir 3 objets dans 2 boîtes non vides.

Exercice 20

1. L'expression ci–dessous définit la fonction à 2 paramètres $g : \begin{array}{ccc} \mathbb{R}\times\mathbb{R} & \longrightarrow & bool \\ (x,y) & \longmapsto & x < \frac{y}{2} \end{array}$

```
let g: float*float -> bool = function
  (x,y) -> x < (y /. 2.)
```

Nous pouvons la définir comme une fonction a un seul paramètre, de signature $gg : \mathbb{R} \to \mathbb{R} \to bool$ comme suit :

```
let gg: float → float → bool = function
    x → function
        y → x < (y /. 2.)
```

Quel est le type de l'expression gg 3.2 ? Quelle expression avec `gg` permet d'obtenir la valeur de `g(3.2,4.4)` ? On peut simplifier l'expression OCaml définissant `gg` comme suit :

```
let gg: float → float → bool = fun
    x y → x < (y /. 2.)
```

2. (Re)définissez la fonction `min` de signature $int \to int \to int$ calculant le minimum de 2 entiers.
3. Définissez la fonction `somInt` de l'exercice 14 avec la signature $int \to int \to int$.

Exercice 21

On cherche à définir une fonction vérifiant si un entier est un nombre premier (nombre ayant exactement 2 diviseurs). Pour cela commencez par définir la fonction récursive `nbDiv` de signature $int \to int \to int$ telle que, `n` et `i` étant 2 entiers $1 \leq i \leq n$, l'évaluation de `nbdiv n i` donne le nombre de diviseurs de n inférieurs ou égaux à i.
Écrivez alors la fonction `estPremier` qui teste si un nombre est premier.

Les Listes

Exercice 22

Évaluez les expressions ci–dessous :

```
let li=[2;3;2;4]
List.tl(List.tl(li))
List.hd(List.tl(List.tl(li)))
List.hd(List.tl(List.tl(List.tl(List.tl(li)))))
7 :: li
7 :: List.tl(li)
List.hd(li) :: List.tl(li)
List.hd(List.tl(li)) :: List.tl(li)
List.hd ["liste";"entiers"] :: ["chaines"]
```

Exercice 23

Écrivez une fonction qui inverse l'ordre des deux premiers éléments d'une liste d'entiers de longueur au moins deux.

```
let inverserElem: int list → ....
```

Modifiez la définition pour que la fonction s'applique à tout type de liste (ayant au moins deux éléments).

Exercice 24

Soit la fonction `listeEntiers`, qui étant donné un entier naturel `n`, calcule la liste `[n;n-1;...;1;0]`. Complétez la récurrence :

— **Cas de base :** quand `n=0` que vaut `listeEntiers(0)` ?
— **Équation de récurrence :** quand `n>0`, définissez `listeEntiers(n)` en fonction de `listeEntiers(n-1)`.

Écrivez la définition OCaml de la fonction récursive `listeEntiers`.

Exercice 25

Définissez la fonction `derListe` qui calcule la valeur du dernier élément d'une liste d'entiers non vide.
Commencez par écrire la récurrence :

— **Cas de base :** quand `li` ne contient qu'un élément que vaut `derListe(li)` ?
— **Équation de récurrence :** quand `li` contient plus d'un élément, définissez `derListe(li)` en fonction de `derListe(List.tl(li))`.

Écrivez la définition OCaml de `derListe`.

```
let rec derListe: int list → int =
```

Modifiez cette définition pour qu'elle s'applique à tout type de liste non vide.

Exercice 26

On cherche à écrire un algorithme `maxListe` qui étant donné `li`, une liste non vide d'entiers, calcule le plus grand de ses éléments. Pour cela complétez la récurrence :

- **Cas de base :** quand `li` ne contient qu'un élément que vaut `maxListe(li)` ?
- **Équation de récurrence :** quand `li` contient plus d'un élément, définissez `maxListe(li)` en fonction de `maxListe(List.tl(li))` en utilisant la fonction `max` calculant le maximum de 2 nombres.

Complétez alors la définition :

let rec *maxListe* : *int list* →

Modifiez la définition pour que `maxListe` s'applique à n'importe qu'elle s'applique aux listes de réels, chaînes de caractères, booléens. Rappelez-vous de la signature de la fonction prédéfinie max : $'a \rightarrow 'a \rightarrow 'a$

Exercice 27

Définissez les fonctions

- `tousPairs` qui étant donnée une liste d'entiers vaut `true` si tous ses éléments sont pairs, `false` sinon.
  Par exemple, `tousPairs([2;8;2])` et `tousPairs[])` valent `true`, tandis que `tousPairs([2;5;2])` vaut `false`.
- `exiPairs` qui teste si une liste d'entiers contient au moins un élément pair.
  Par exemple, `exiPair([2])` et `exiPair[3;2;5])` valent `true`. `exiPair([7;5])` et `exiPair([])` valent `false`.
- `liPairs` qui extrait d'une liste d'entiers la liste des nombres pairs.
  Par exemple `liPairs([5;8;1;2])=[8;2]`.
- `oterPairs` qui supprime les nombres pairs d'une liste d'entiers.
  Par exemple `oterPairs([5;8;1;2])=[5;1]`.

Exercice 28

On souhaite définir la fonction `liDiv` qui calcule la liste des diviseurs d'un entier naturel non nul.

- Commencez par définir la fonction liDivInf : int ⟶ int⟶ int list, tel que
  `(liDivInf n i)` donne comme résultat la liste des diviseurs de `n` inférieurs ou égaux à `i`.
  Par exemple `(liDivInf 20 4)=[4;2;1] (liDivInf 7 5)=[1] (liDivInf 4 4)=[4;2;1]`.
- Définissez `liDiv` à partir de `liDivInf`.

Exercice 29

Définissez la fonction `ajouFin` qui ajoute un élément à la fin d'une liste. Le type de l'élément doit correspondre au type des éléments de la liste, mais peut être quelconque.
Par exemple `(ajouFin 4 [5;4;2])=[5;4;2;4]` et `(ajouFin "deux" ["un"])=["un";"deux"]`

Exercice 30

Définissez la fonction `inverseLi` qui inverse l'ordre des éléments d'une liste.
Par exemple `(inverseLi [1;4;2]) = [2;4;1]`.
Vous pouvez utiliser la fonction `ajouFin` de la question précédente, ...ou pas.

Exercice 31

Définitions : soient `l1` et `l2` deux listes :

- `l1` est **préfixe** de `l2` si `l2` est composée des éléments de `l1` dans le même ordre, puis d'éléments en nombre et valeur quelconques.
- `l1` est **suffixe** de `l2` si `l2` est composée d'éléments quelconques, suivis des éléments de `l1` dans le même ordre.
- `l1` est **facteur** de `l2` si `l2` est composée d'éléments quelconques, suivis des éléments de `l1` dans le même ordre, suivis d'éléments quelconques.

Exemples :

- `[3;2;3]` est préfixe et facteur de la liste `[3;2;3;4;2]`.
- `[3;2;3]` est préfixe, suffixe et facteur de la liste `[3;2;3]`.
- `[]` est préfixe, suffixe et facteur de toutes les listes.

— `[3;2;3]` est suffixe et facteur de la liste `[4;3;2;3]`
— `[3;2;3]` est facteur de la liste `[4;2;3;2;3;8]`.
Définissez les fonctions `prefixe`, `suffixe` et `facteur`.

Le filtrage

Exercice 32
Quelles sont les valeurs des expressions :

```
let n=0;;
match n with
| 1 → 1
| i when i<0 → 0
| i when i mod 2=0 → n/2
| i when i mod 2=1 → i+1
| 3 → 0
| _ → 2;;
```

```
let n=6;;
match n with
| 1 → 1
| i when i<0 → 0
| i when i mod 2=0 → n/2
| i when i mod 2=1 → i+1
| 3 → 0
| _ → 2;;
```

```
let n=3;;
match n with
| 1 → 1
| i when i<0 → 0
| i when i mod 2=0 → n/2
| i when i mod 2=1 → i+1
| 3 → 0
| _ → 2;;
```

Exercice 33
Comparez les fonctions `z1, z2, z3.`

```
let z1= function
  x →
    if x = 0 then true
    else false
```

```
let z2= function
  x →
    match x with
    |0 → true
    |_ → false
```

```
let z3= function
    |0 → true
    |_ → false
```

Exercice 34
En utilisant le filtrage, redéfinissez les fonctions `implication` et `f` vues aux exercices 11, et 12, ainsi que les fonctions `exiPair` et `liPairs` de l'exrecice 27.

Exercice 35
Définissez la fonction `nbOcc` qui compte le nombre d'occurrence d'une valeur dans une liste.
Par exemple `(nbOcc 3 [3;8;9;3;0])=2 (nbOcc 1 [])=0 (nbOcc 1.5 [3.2;1.5;1.5;2.;1.5])=3.`
Quelle est la signature de `nbOcc` ?

Exercice 36
Tous les éléments d'une liste doivent être de même type. Ce type est quelconque, par exemple une listes d'entiers. Ainsi [[1;2]; [3;1;0]; []; [8]] est une liste de listes d'entiers. On manipule ces listes comme les autres listes avec les fonctions `List.hd, Liste.tl, ::.`

— Quels sont les résultats des évaluations des expressions suivantes ?

```
let li = [ [1;2]; [3;1;0]; []; [8] ];;
List.hd li;;
List.tl li;;
3:: li;;
[3]::List.tl li;;
```

— Écrivez la fonction `nbOccLiLi` qui étant donnés un entier `n` et une liste de listes d'entiers `li` calcule le nombre d'occurrences de `n` dans les listes de `li`. Vous utiliserez la fonction `nbOcc` de l'exercice 35.
Par exemple `(nbOccLiLi 6 [[6;2;6]; [3;6;0]; []; [6]]) = 4`
— Écrivez la fonction `liSuff` qui calcule la liste des suffixes d'une liste.
Par exemple `(liSuff [1;2;3;4]) = [[1; 2; 3; 4]; [2; 3; 4]; [3; 4]; [4]; []]`

Polymorphisme

Exercice 37
Inférer le type des fonctions suivantes :

```
let id x = x

let f e l = e :: l

let g c x l =
  if (fst c) then (snd c)::l
  else [x]

let h c x y = function
  | [] → []
  | e::tl as l →
     if c then (x, y)::l
     else e::(x, y)::tl

let rec j x = j x

let rec k a = function
  | [] → a
  | e::tl →
     if (e mod 2 = 0) then k a tl
     else failwith "stop"
```

Exercice 38
Écrire la fonction `make_couple` qui, étant donnés deux paramètres `x` et `y`, rend le couple `(x,y)`.
Écrire la fonction `proj1` qui, étant donné un couple c passé en paramètre, rend la première composante de c.
Écrire la fonction `proj2` qui, étant donné un couple c passé en paramètre, rend la deuxième composante de c.
Quels sont les types des fonctions `make_couple`, `proj1` et `proj2` ?

Exercice 39
Écrire la fonction `mem_assoc` (sans utiliser celle prédéfinie de la bibliothèque `List`), qui étant données une clé `a` et une liste de couples `l`, rend `true` s'il existe un couple `(a,b)`, où `b` est une valeur quelconque, dans la liste `l` et `false` sinon.

Exemples d'exécution :
- `(mem_assoc 2 [(1,'a');(2,'b');(3,'c')])` s'évalue en `true`.
- `(mem_assoc 4 [(1,'a');(2,'b');(3,'c')])` s'évalue en `false`.
- `(mem_assoc 2 [])` s'évalue en `false`.

Quel est le type de cette fonction ?

Exercice 40
Écrire une fonction `dup_list` qui, étant donnés un entier `n` et une liste `l`, rend une nouvelle liste où chaque élément est dupliqué `n` fois. Si `n` est nul ou si la liste `l` est vide, alors la fonction rend une liste vide.

Exemples d'exécution :
- `(dup_list 2 [1;2;3])` s'évalue en `[1;1;2;2;3;3]`.
- `(dup_list 3 [1;2;3])` s'évalue en `[1;1;1;2;2;2;3;3;3]`.
- `(dup_list 2 [])` s'évalue en `[]`.
- `(dup_list 0 [1;2;3])` s'évalue en `[]`.

Quel est le type de cette fonction ?
Quel est le résultat de l'évaluation de `(dup_list 2 [1;2;3;4;5])` ?
Quel est le résultat de l'évaluation de `(dup_list 3 ['a';'b';'c'])` ?

Types sommes

Exercice 41

On considère le type somme `number`, vu en cours et défini de la façon suivante :

```
type number =
| Int of int
| Float of float
```

Écrire la fonction `split_number_list` qui, étant donnée une liste d'éléments de type `number`, rend la liste des éléments entiers (de la forme `Int ...`) et la liste des éléments flottants (de la forme `Float ...`). La fonction rend donc un couple de résultats.
Écrire la fonction `sum_split_number_list` qui, étant donnée une liste d'éléments de type `number`, rend la somme des éléments entiers et la somme des éléments flottants (la fonction rend à nouveau un couple de résultats).

Exercice 42

On considère le type somme `carte`, vu en cours et défini de la façon suivante :

```
type enseigne = Trefle | Carreau | Coeur | Pique
type valeur = As | Deux | Trois | Quatre | Cinq | Six | Sept
            | Huit | Neuf | Dix | Valet | Dame | Roi
type carte = Carte of enseigne * valeur
```

Dans les questions qui suivent, on se place dans le cadre d'un jeu de poker. En particulier, une main est une liste de 5 cartes. Dans la suite, pour simplifier, on ne testera pas cependant la cardinalité des listes de cartes (on supposera qu'elles contiennent bien 5 cartes).

Écrire une fonction qui teste si une main contient au moins un as.
Écrire une fonction qui teste si une main contient au moins une carte d'une valeur donnée.
Écrire une fonction qui teste si une main contient au moins deux as.
Écrire une fonction qui teste si une main contient au moins une paire.
Écrire une fonction qui teste si une main est une couleur (5 cartes de la même enseigne).

Exercice 43

On considère le type somme `nat` des entiers de Peano, vu en cours et défini de la façon suivante :

```
type nat = O | S of nat
```

Écrire la fonction `to_dec` qui, étant donné un entier de Peano, rend un entier décimal (de type `int`).
Écrire la fonction `from_dec` qui, étant donné un entier décimal (de type `int`) rend l'entier de Peano correspondant.

Écrire la fonction `mul_nat`, qui réalise la multiplication de deux entiers de Peano.

Pour ce faire, on utilisera les deux équations suivantes provenant de l'axiomatisation de Peano :
- `(mul_nat x O) = O`.
- `(mul_nat x (S y)) = (add_nat (mul_nat x y) x)`.

où `add_nat` est la fonction d'addition sur les entiers de Peano (vue en cours), et où `x` et `y` sont deux entiers de Peano.

Exercice 44

On considère le type somme `tree` des arbres binaires, vu en cours et défini de la façon suivante :

```
type 'a tree =
| Empty
| Node of 'a * 'a tree * 'a tree
```

Écrire la fonction `height`, qui étant donné un arbre, rend sa hauteur. On considérera que la hauteur d'un arbre binaire est le plus grand nombre de nœuds des chemins de la racine vers les feuilles. La hauteur de l'arbre vide est 0.
Écrire la fonction `is_balanced`, qui étant donné un arbre, teste si l'arbre est équilibré ou non. Un arbre est équilibré si pour tout nœud, le fils gauche et le fils droit ont la même hauteur. L'arbre vide est équilibré.

Exercice 45

On considère un petit langage des expressions arithmétiques et booléennes avec variables. Ce langage est défini par la grammaire suivante (en utilisant une présentation à la BNF) :

k	$::= \quad n$	
	$\mid \quad \mathsf{true} \mid \mathsf{false}$	*Constantes*
uop	$::= \quad -$	
	$\mid \quad \mathsf{not}$	*Opérateurs unaires*
bop	$::= \quad + \mid - \mid \times \mid /$	
	$\mid \quad \mathsf{and} \mid \mathsf{or}$	*Opérateurs binaires*
	$\mid \quad < \mid \leq \mid = \mid \neq \mid \geq \mid >$	
e	$::= \quad k \mid x$	
	$\mid \quad \mathsf{uop}\ e \mid e\ \mathsf{bop}\ e$	*Expressions*

Écrire la syntaxe abstraite correspondant à cette grammaire en utilisant uniquement les types sommes et les types primitifs (entiers, chaînes de caractères, produit cartésien, etc.). On utilisera un type somme par entrée de la grammaire.

On souhaite maintenant évaluer les expressions exprimées dans cette syntaxe abstraite. Étant donné que le langage permet l'utilisation de variables, on considérera des environnements d'évaluation, qui seront des listes de couples (nom de variable, valeur). Par ailleurs, l'évaluation d'une expression pourra échouer pour un certain nombre de raisons (application d'opérateurs arithmétiques à des booléens et vice-versa, utilisation d'une variable non définie dans l'environnement d'évaluation, etc.).

Écrire le type (somme) des valeurs des expressions.
Écrire la fonction `eval_expr` qui, étant donnés un environnement d'évaluation et une expression, rend le résultat de l'évaluation de cette expression dans l'environnement d'évaluation.
Donner quelques exemples d'utilisation de la fonction `eval_expr`.

Types enregistrements

Exercice 46

On considère le type enregistrement `personne`, vu en cours et défini de la façon suivante :

```
type personne = {nom : string; prenom : string; age : int}
```

Écrire la fonction `afficher_personne`, qui étant donnée une personne, affiche ses informations selon la forme : « <Prénom> <Nom> a <Age> ans ».

Exemples d'exécution :

- `(afficher_personne {nom = "Church"; prenom = "Alonzo"; age = 92})` affichera : Alonzo Church a 92 ans

Écrire la fonction `changer_nom`, qui étant donnés un nom et une personne, crée une nouvelle personne avec ce nouveau nom (le reste des champs est inchangé).
Écrire la fonction `plus_age`, qui étant données deux personnes `p1` et `p2`, teste si `p1` est plus âgée que `p2` ou non. Cette fonction rendra `false` si `p1` et `p2` ont le même âge.

Définir le type `personne2`, qui ajoute une information concernant le sexe de la personne (masculin ou féminin). Le type de cette information sera modélisé par un type somme (à définir au préalable).
Écrire les fonctions de conversion `p_vers_p2` et `p2_vers_p`, qui permettent respectivement de traduire une valeur de type `personne` en `personne2` et inversement. Donner des exemples d'évaluation de ces fonctions.

Exceptions

Exercice 47
Soient les fonctions `f` et `g` suivantes :

```
let g n =
  if n = -1 then failwith "-1"
  else if n > -10 then n
  else raise (Invalid_argument (string_of_int n));;

let f n =
  match n with
  | i when i >= 0 → 1 / i
  | i when i >= -10 && i <= -1 →
     (try g n with
     | Failure _ → 0)
```

Que rend l'évaluation de `f` appliquée à : 2, 0, -1, -5, -10 et -11 ?

Exercice 48
Écrire la fonction `som` qui, étant donné un entier `n`, calcule la somme des `n` premiers entiers naturels (nous avons déjà écrit cette fonction dans l'exercice 13). Cette fonction n'est pas définie pour les entiers négatifs et on veillera à lever l'exception `Failure` avec un message d'erreur approprié lorsque l'entier donné en argument est négatif.

Écrire la fonction `maxListe` qui, étant donnée une liste non vide, calcule le maximum (le plus grand de ses éléments) de cette liste (nous avons déjà écrit cette fonction dans la partie dédiée aux listes). Comme cette fonction n'est pas définie pour la liste vide, on veillera à lever l'exception `Invalid_argument` avec un message d'erreur approprié lorsque la liste donnée en argument est vide.
Quel est le type de cette fonction ?

Exercice 49
Écrire la fonction `my_tl` qui, étant donnée une liste `l`, rend la queue de la liste. Si la liste `l` est vide, la fonction rendra une liste vide (contrairement à la fonction prédéfinie `List.tl`). Pour écrire cette fonction, on utilisera la fonction `List.tl` et on rattrapera l'exception `Failure` quand la liste donnée en argument est vide.

Écrire la fonction `my_nth` qui, étant donnés une liste `l` et un entier `n`, rend l'élément d'indice `n` dans la liste (le premier élément de la liste est à l'indice 0). Si `n` correspond à une position plus grande que la position du dernier élément de la liste, on rendra le dernier élément de la liste. Si `n` est négatif, on rendra le premier élément de la liste. Comme précédemment, pour écrire cette fonction, on utilisera la fonction prédéfinie `List.nth`, qui lève l'exception `Failure` si la position indiquée est trop grande et l'exception `Invalid_argument` si la position indiquée est négative.

Exercice 50
Écrire la fonction `somme_sous_liste` qui, étant donnés un entier `s` et une liste d'entiers `l`, rend une sous-liste de `l` dont la somme des éléments est égale à `s`.

Pour ce faire, on définira, au préalable, une exception nommée `Impossible`. En effet, pour une liste et un entier donnés, le problème n'admet pas forcément de solution et il faudra lever l'exception `Impossible` dans ce cas.

Pour vous aider, on vous donne l'algorithme pour résoudre le problème. Cet algorithme est basé sur la notion d'essais successifs ou de *backtrack* (en anglais). Le principe est le suivant :

- Si la liste est vide, alors si la somme est nulle la solution est la liste vide, sinon il n'y a pas de solution ;
- Si la liste n'est pas vide, donc de la forme `e::tl`, deux choix :
  - Essayer une solution composée de `e` et d'une sous-liste de la liste `tl` dont la somme des éléments fasse `s-e` ;
  - Ou répondre au problème initial en ne choisissant les éléments de la solution que parmi les éléments de `tl`.

Exercice 51

Écrire la fonction `plus_petit_que` qui, étant donnés un entier `n` et une liste d'entiers `l`, rend le premier entier de la liste `l` plus petit (strictement) que `n`, ou lève l'exception `Not_found` (prédéfinie) s'il n'y a aucun élément plus petit que `n` dans la liste `l`.

Pour ce faire, on définira, au préalable, l'exception `Found` qui prend un entier en argument. Puis, on écrira la fonction `plus_petit_que_exc` qui, étant donnés un entier `n` et une liste d'entiers `l`, lève l'exception `Found` avec le premier entier de la liste `l` plus petit que `n`, ou lève l'exception `Not_found` (prédéfinie) s'il n'y a aucun élément plus petit que `n` dans la liste `l`. La fonction `plus_petit_que` devra utiliser la fonction `plus_petit_que_exc`.

Modifier le code afin que la fonction `plus_petit_que` rende non seulement le premier entier de la liste `l` plus petit que `n`, mais aussi la position de cet élément dans la liste (les positions commencent à 0).

Fonctions d'ordre supérieur

Exercice 52

Inférer le type des fonctions suivantes :

```
let f g x = g x
let g f x y = f (x,y)
let h f (x,y) = f x y
```

Que font les fonctions `f`, `g` et `h` ?
Donner des exemples d'application de ces fonctions.

Exercice 53

Écrire la fonction `my_map` qui, étant données une fonction `f` et une liste d'éléments `l`, rend la liste des applications de `f` aux éléments de `l` sans utiliser la fonction prédéfinie `List.map`. Plus précisément, `my_map f [a1; ...; an]` rendra la liste `[f a1; ...; f an]`.

Écrire la fonction `my_filter` qui, étant donnés un prédicat `f` et une liste d'éléments `l`, rend la liste des éléments de `l` vérifiant le prédicat `f` dans le même ordre sans utiliser la fonction prédéfinie `List.filter`.

Écrire la fonction `my_fold_right` qui, étant donnés une fonction `f`, une liste d'éléments `[a1; ...; an]` et un élément `init`, rend la suite d'applications `f a1 (f a2 (... (f an init) ...))` sans utiliser la fonction prédéfinie `List.fold_right`.

Exercice 54

En OCaml, une façon d'implémenter les fonctions partielles est d'utiliser le type `option`, défini de la manière suivante :

```
type 'a option =
| None
| Some of 'a
```

Ainsi, partout où elle est définie, la fonction rend `(Some v)`, où `v` est la valeur de la fonction, et lorsqu'elle n'est pas définie, la fonction rend `None` (au lieu de lever une exception).

Soit la fonction factorielle définie de la façon suivante :

```
let rec fact n =
  if n < 0 then failwith "Pas d'entiers négatifs !"
  else if n = 0 then 1
  else n * (fact (n - 1));;
```

Écrire la fonction `fact_securisee` qui utilise la fonction `fact`, mais qui rend une valeur de type `option`. En particulier, lorsque la fonction n'est pas définie, elle rendra la valeur `None`.

Écrire la fonction `securiser_fonction` qui transforme une fonction d'arité 1 en une fonction qui rend un type `option` selon la même idée que précédemment pour la fonction `fact` (on ne se posera pas trop de questions et on rattrapera toutes les exceptions potentiellement levées par la fonction).
Quel est le type de la fonction `securiser_fonction` ?
Réécrire la fonction `fact_securisee` en utilisant la fonction `securiser_fonction`.

Peut-on généraliser cette approche à une fonction d'arité quelconque ? Si oui, donner le code correspondant, sinon expliquer pourquoi et ce qui nous manquerait.

Exercice 55

Dans le λ-calcul pur (uniquement des fonctions, pas de structures de données), Alonzo Church a eu l'idée d'encoder les structures de données à l'aide de fonctions. En particulier, pour les entiers, l'idée est d'utiliser la notion d'itération (ou d'application) de fonction. Ainsi, l'entier n est une fonction qui itère (applique) n fois une certaine fonction. Ces entiers, encodés de cette manière, sont ainsi appelés itérateurs de Church. Plus précisément, on a la correspondance suivante :

— $0 \equiv \textbf{fun}\ f\ x \to x$
— $1 \equiv \textbf{fun}\ f\ x \to f\ x$
— $2 \equiv \textbf{fun}\ f\ x \to f\ (f\ x)$
— ...
— $n \equiv \textbf{fun}\ f\ x \to f\ (f\ (\ldots (f\ x))) = \textbf{fun}\ f\ x \to f^n\ x$, avec f itérée n fois

Écrire la fonction `make_church` qui, étant donné un entier `n`, rend l'itérateur (une fonction) correspondant à `n` selon la correspondance indiquée ci-dessus.
Écrire la fonction `succ` qui, étant donné un itérateur de Church `n`, rend l'itérateur de Church correspondant à `n+1`. Pour ce faire, il y a deux solutions différentes : ajouter une itération de la fonction soit en tête, soit en queue (vous donnerez les deux solutions).

Exercice 56

Les types sommes en OCaml sont des outils très puissants. Leur aspect récursif permet en particulier d'itérer et d'écrire des fonctions récursives sans passer par le `let rec` habituel d'OCaml en encodant ce que l'on appelle un combinateur de point fixe. Voici le code qui permet de réaliser cela :

```
type 'a mu = Roll of ('a mu -> 'a)
let unroll (Roll x) = x
let fix f = (fun x a -> f (unroll x x) a) (Roll (fun x a -> f (unroll x x) a))
```

La fonction `fix` est le combinateur qui va nous permettre d'itérer. Pour l'appliquer, il suffit d'écrire la fonctionnelle correspondant à notre fonction récursive, c'est-à-dire une fonction paramétrée par une fonction représentant les appels récursifs. Par exemple, pour la fonction factorielle, on écrira :

```
let fact f n =
  if n = 0 then 1
  else n * f (n - 1);;
```

Quels sont les types des fonctions `unroll`, `fix` et `fact` ?
Quel est le résultat de l'évaluation de `fix fact 3` ?

Itérateurs sur les listes

Exercice 57
Que font les fonctions suivantes (vous donnerez leurs types, ainsi que des exemples d'exécution) ?

```
let f l = List.fold_left (fun a e → a ^ (Char.escaped e)) "" l

let g l = List.fold_right (fun e a → e::a) l []

let h l = List.for_all (fun e → e) (List.map (fun e → e >= 0) l)

let i e l = List.length (List.filter (fun x → x = e) l)

let j l =
  let l1,l2 = List.partition (fun e → e >= 0) l in
  List.length l1 >= List.length l2
```

Exercice 58
Réécrire toutes les fonctions `tousPairs`, `exiPairs`, `liPairs` et `oterPairs` de l'exercice 27 en utilisant uniquement des itérateurs sur les listes.

Exercice 59
Écrire la fonction `somme_pairs` qui, étant donnée une liste d'entiers `l`, réalise la somme des entiers pairs de la liste `l` en utilisant uniquement des itérateurs sur les listes.

Écrire la fonction `union` qui, étant donnée deux listes d'éléments sans doublons `l1` et `l2`, concatène les deux listes en une liste sans doublons également en utilisant uniquement des itérateurs sur les listes. L'idée ici est d'utiliser l'itérateur `List.fold_left` (ou `List.fold_right`) sur `l1` pour supprimer tous les éléments de `l1` dans `l2` en filtrant avec `List.filter`. On obtient ainsi une liste `l2` filtrée qui peut être concaténée directement à `l1`.

Écrire la fonction `est_triee` qui, étant donnée une liste d'éléments `l`, teste si la liste `l` est triée ou non en utilisant uniquement des itérateurs sur les listes. L'idée ici est d'utiliser l'itérateur `List.fold_left` en mémorisant à la fois la réponse et le dernier élément traité (la base est donc un couple). Attention à bien traiter le cas de la liste vide à part.