

Université de Montréal

IFT 2035 - Exercices

Vincent Archambault-Bouffard

Version du 8 juin 2019

Ce document est dédié au domaine public via CC0 1.0.

Pour obtenir le code source de ce document :

- `https://github.com/archambaultv/IFT2035-UdeM`
- `vincent.archambault-bouffard@umontreal.ca`

Remerciement

Je tiens à remercier les personnes suivantes (par ordre alphabétique) :

Frédéric Hamel Pour avoir fourni plusieurs solutions.

Stefan Monnier Pour avoir fourni plusieurs exercices.

1 Introduction

Petit recueil d'exercices pour IFT 2035. Vous trouverez les solutions à la fin, lorsqu'elles sont disponibles.

2 Grammaire

Exercice 2.1

Pour chaque expression infixe ci-dessous, réécrire l'expression en notation préfixe et postfixe. Dessiner également l'arbre de syntaxe abstraite (ASA).

1. $a + b + c$
2. $a + (b + c)$
3. $a \cdot b + c \cdot d$
4. $a + b < a \cdot (c + d)$
5. $\sqrt{b \cdot b - 4 \cdot a \cdot c}$

Exercice 2.2

Soit les expressions suivantes en notation postfixe :

1. `a b + c - d *`
2. `a b && c d == ||`
3. `a b + c d * == not`

1. Écrire ces expressions en format infixe, préfixe et en arbre de syntaxe abstraite. `==` est l'opérateur de comparaison, `&&` est l'opérateur de conjonction et `||` est l'opérateur de disjonction.
2. Il est possible d'évaluer les expressions postfixes facilement à l'aide d'une pile. Montrer comment évaluer l'expression numéro 3 de la page précédente avec une pile. Utiliser les valeurs numériques suivantes : $a = 1$, $b = 2$, $c = 3$, $d = 4$.

Exercice 2.3

Nous allons voir dans cette section plusieurs grammaires BNF possibles pour l'addition et la multiplication. Vous verrez qu'il est facile de générer une grammaire BNF ambiguë ou ne respectant pas la priorité et l'associativité des opérations.

Partie I — Grammaire 1

Soit la grammaire ci-dessous :

$$\begin{aligned}\langle \text{expr} \rangle &::= \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \langle \text{chiffre} \rangle \\ \langle \text{chiffre} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\end{aligned}$$

1. Montrer que dans cette grammaire, pour une même expression, il est possible de choisir une dérivation où l'opérateur est associatif à gauche et une dérivation où l'opérateur est associatif à droite. Cela démontre que la grammaire est ambiguë.

Partie II — Grammaire 2

Voici comment on pourrait transformer la grammaire 1 pour lever l'ambiguïté.

$$\begin{aligned}\langle \text{expr} \rangle &::= \langle \text{expr} \rangle + \langle \text{terme} \rangle \mid \langle \text{chiffre} \rangle \\ \langle \text{terme} \rangle &::= '(' \langle \text{expr} \rangle ')' \mid \langle \text{chiffre} \rangle \\ \langle \text{chiffre} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\end{aligned}$$

1. Montrer que cette grammaire est associative à gauche.
2. Donner l'arbre de dérivation de l'expression $1 + (1 + 1) + 1$.

Noter que cette grammaire est récursive à gauche et associative à gauche.

Partie III — Grammaire 3

Donner une grammaire BNF similaire à la grammaire 1 mais cette fois-ci associative à droite.

Partie IV — Grammaire 4

La grammaire 2 est associative à gauche et non ambiguë. Voici une première tentative d'ajouter l'opérateur de multiplication.

$$\begin{aligned}\langle \text{expr} \rangle &::= \langle \text{expr} \rangle + \langle \text{terme} \rangle \mid \langle \text{expr} \rangle * \langle \text{terme} \rangle \mid \langle \text{chiffre} \rangle \\ \langle \text{terme} \rangle &::= '(' \langle \text{expr} \rangle ')' \mid \langle \text{chiffre} \rangle \\ \langle \text{chiffre} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\end{aligned}$$

1. Montrer que cette grammaire est non ambiguë.
2. Cette grammaire ne respecte pas la priorité des opérations. Donner l'arbre de dérivation des expressions $1 + 2 * 3$ et $1 * 2 + 3$.

Partie V — Grammaire 5

Soit la grammaire ci-dessous qui corrige le défaut de la grammaire 4.

$$\begin{aligned}\langle \text{expr} \rangle &::= \langle \text{expr} \rangle + \langle \text{terme} \rangle \mid \langle \text{terme} \rangle \\ \langle \text{terme} \rangle &::= \langle \text{terme} \rangle * \langle \text{facteur} \rangle \mid \langle \text{facteur} \rangle \\ \langle \text{facteur} \rangle &::= '(' \langle \text{expr} \rangle ')' \mid \langle \text{chiffre} \rangle \\ \langle \text{chiffre} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\end{aligned}$$

1. Cette grammaire respecte la priorité des opérations. Donner l'arbre de dérivation des expressions $1 + 2 * 3$ et $1 * 2 + 3$.

Partie VI — Grammaire 6

Voici comment ajouter la soustraction et la division.

$$\begin{aligned} \langle \text{expr} \rangle &::= \langle \text{expr} \rangle + \langle \text{terme} \rangle \mid \langle \text{expr} \rangle - \langle \text{terme} \rangle \mid \langle \text{terme} \rangle \\ \langle \text{terme} \rangle &::= \langle \text{terme} \rangle * \langle \text{facteur} \rangle \mid \langle \text{terme} \rangle / \langle \text{facteur} \rangle \mid \langle \text{facteur} \rangle \\ \langle \text{facteur} \rangle &::= '(' \langle \text{expr} \rangle ')' \mid \langle \text{chiffre} \rangle \\ \langle \text{chiffre} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

1. Pourquoi peut-on ajouter la soustraction (division) dans la même catégorie que l'addition (multiplication) plutôt que de créer une nouvelle catégorie ?

Partie VII — Grammaire 7 - Pour les curieux

La grammaire 5 est non ambiguë et respecte la priorité et l'associativité des opérateurs. Toutefois, elle est récursive à gauche pour la catégorie $\langle \text{expr} \rangle$ et $\langle \text{terme} \rangle$. Une grammaire récursive à gauche peut engendrer des boucles infinies.

En effet, dans un parseur avec analyse descendante (top down), la portion du programme devant lire la catégorie $\langle \text{expr} \rangle$ doit d'abord faire appel à la portion du programme qui doit lire la catégorie $\langle \text{expr} \rangle$ qui doit d'abord faire appel à la portion du programme qui doit lire la catégorie $\langle \text{expr} \rangle$ qui doit d'abord faire appel ...

Il faut donc transformer cette grammaire en une grammaire équivalente mais non récursive à gauche.

$$\begin{aligned} \langle \text{expr} \rangle &::= \langle \text{terme} \rangle \langle \text{expr}' \rangle \\ \langle \text{expr}' \rangle &::= + \langle \text{terme} \rangle \langle \text{expr}' \rangle \mid \epsilon \\ \langle \text{terme} \rangle &::= \langle \text{facteur} \rangle \langle \text{terme}' \rangle \\ \langle \text{terme}' \rangle &::= * \langle \text{facteur} \rangle \langle \text{terme}' \rangle \mid \epsilon \\ \langle \text{facteur} \rangle &::= '(' \langle \text{expr} \rangle ')' \mid \langle \text{chiffre} \rangle \\ \langle \text{chiffre} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

Il est possible de montrer que cette grammaire est équivalente à la grammaire du numéro 5.

Exercice 2.4

Voici une grammaire pour if then else. Les règles $\langle E \rangle$ et $\langle X \rangle$ représente des expressions et autres règles de la grammaire dont il n'est pas important de spécifier.

$$\begin{aligned} \langle S \rangle &::= \langle X \rangle \\ &\mid 'if' \langle E \rangle 'then' \langle S \rangle \\ &\mid 'if' \langle E \rangle 'then' \langle S \rangle 'else' \langle S \rangle \end{aligned}$$

Cette grammaire est ambiguë.

1. Donner un exemple d'ambiguïté.
2. Donner la grammaire non ambiguë qui associe les else avec le if le plus proche.

3 Variables

Exercice 3.1

1. Donner une définition de la portée statique (lexicale) et de la portée dynamique.
2. Donner un exemple de code en utilisant la syntaxe Haskell qui sera évalué différemment si la portée est lexicale ou dynamique. Expliquez clairement pourquoi votre code se comporte différemment d'une portée à l'autre.

Exercice 3.2

Soit le code suivant dans un langage hypothétique dont la syntaxe est la même que celle de Haskell :

```
let z = 1
    x = 2
    f1 y = z + x + y
    f2 x = f1 (x + 1)
    f3 z = f3 (z + 2)
in
    f3 5
```

Montrer les étapes de l'évaluation dans chacun des deux cas : le cas où le langage utilise la portée dynamique et le cas où il utilise la portée statique.

Exercice 3.3

Vous devez implanter deux évaluateurs pour un langage minimaliste. L'un aura la portée lexicale et l'autre la portée dynamique. Les expressions du langage peuvent être décrites avec le datatype `Exp`.

```
data Exp = EInt Int
        | Var String
        | Lambda String Exp
        | App Exp Exp
        deriving (Show)
```

Dans les deux cas, l'évaluation d'une expression `App e1 e2` doit réduire `e1` à une valeur de type fonction sinon l'évaluation ne peut plus se poursuivre.

Pour vous aider, l'évaluateur avec portée dynamique a un environnement, des valeurs et une signature définie comme suit :

```
type Env a = [(String, a)]
```

```
data ValueD = VVar String  
            | VInt Int  
            | VDLambda String Exp  
            deriving (Show)
```

L'évaluateur avec portée lexicale possède le même environnement mais a des valeurs et une signature définie comme suit :

```
data Value = VVar String  
           | VInt Int  
           | VLambda String Exp (Env Value)  
           deriving (Show)
```

4 Fonctions

Exercice 4.1

Écrire un programme qui donne un résultat différent pour chacune des combinaisons possibles entre les 4 méthodes de passage de paramètres et les 2 types de portées (8 combinaisons en tout). Pour rappel, les 4 types de passage de paramètres sont :

- passage de paramètres par valeur
- passage de paramètres par référence
- passage de paramètres par valeur-résultat
- passage de paramètres par nom

Exercice 4.2

Soit le morceau de code suivant dans un langage hypothétique qui utilise une syntaxe de style C, et où `f` est une fonction quelconque *que l'on ne connaît pas* :

```
{  
  int table[2] = {0, 1};  
  int size = 2;  
  int tmp = 0;  
  
  f (table, size);  
  ...  
}
```

On aimerait savoir si certaines conditions sont nécessairement toujours vraies après l'appel à `f`. On s'intéresse plus particulièrement aux conditions suivantes :

- `table[0] == 0`
- `size == 2`
- `tmp == 0`

Indiquer lesquelles de ces trois conditions sont nécessairement vraies dans chacun des cas suivants :

1. Le langage est exactement comme C : portée statique, passage d'arguments par valeur, affectation autorisée.

2. Le langage est comme C sauf que l'affectation (autre que l'initialization) est interdite.
3. Le langage est comme C sauf que les arguments sont passés par référence.
4. Le langage est comme C mais avec portée dynamique.

5 Récursion

Exercice 5.1

Écrire en Haskell les fonctions suivantes manipulant des listes de nombres :

length Retourne la longueur d'une liste. Exemples :

- `length [] == 0`
- `length [1] == 1`
- `length [1, 2] == 2`

concat Retourne la concaténation de deux listes. Exemples

- `concat [] [] == []`
- `concat [1] [2] == [1, 2]`
- `concat [] [2] == [2]`
- `concat [1, 2] [] == [1, 2]`

member Trouve un nombre dans la liste. Exemples

- `member 1 [] == False`
- `member 1 [2, 3, 1] == True`

reverse Inverse la liste reçue en argument

- `reverse [5, 4, 3, 2, 1] == [1, 2, 3, 4, 5]`
- `reverse [] == []`

subList Indique si la première liste est incluse dans la seconde. Les éléments de la première liste doivent apparaître dans l'ordre dans la seconde. Exemples :

- `subList [] [5, 4, 3, 2, 1] == True`
- `subList [2, 1, 3] [5, 4, 3, 2, 1] == False`
- `subList [2, 1, 3] [5, 2, 1, 3, 4] == True`
- `subList [2] [] == False`

6 Fermeture

Exercice 6.1

Cet exercice a pour objectif de vous faire pratiquer les fermetures et de vous convaincre qu'elles peuvent servir de structure de données. Nous allons utiliser les encodages de Church pour écrire les structures de données usuelles sous forme de fonctions. Chaque fonction peut être écrite sur une ligne, mais ce sont des lignes qui ne sont pas toujours facile à trouver.

Détail technique

Nous allons redéfinir plusieurs identificateurs de la librairie standard. Je vous conseille de mettre la ligne suivant au début de votre fichier Haskell pour éviter que le compilateur ou ghci vous indique que vous faites ombrage aux définitions standards.

```
import Prelude hiding (Bool, succ, fst, snd, head, tail)
```

Partie I — Booléen

Vous devez implanter les booléens sous forme de fonctions. Pour vous donner un indice, le type des deux fonctions qui représentent le boolean `true` et `false` est le suivant :

```
type Bool a = a -> a -> a
```

Vous devez :

1. Écrire la fonction `true` et `false` de type `Bool a`. En fait, il n'y a que deux fonctions possibles avec ce type.
2. Écrire la fonction `if2` qui a pour type `Bool a -> a -> a -> a`. C'est à dire quelle prend un booléen et deux valeur de type `a` et retourne la première valeur si le booléen est `true` et la deuxième sinon.

Par exemple, dans le code suivant `testIfTrue` vaut 1

```
testIfTrue = if2 true 1 2
```

et dans le code suivant `testIfFalse` vaut 2

```
testIfFalse = if2 false 1 2
```

Partie II — Nombres naturels

Vous devez implanter les nombres naturel 0 1 2 3... à l'aide de fonctions. Le type d'un nombre naturel est :

```
type Number t = (t -> t) -> t -> t
```

La logique de l'encodage est la suivante : un nombre est représenté par une fonction qui attend une valeur de base de type `t` et une fonction de type `t -> t`. Remarquez que la fonction retourne une valeur de type `t`. L'idée est que 0 correspond à la valeur de base, 1 correspond à *une* application de la fonction, 2 correspond à *deux* applications de la fonction, 3 correspond à *trois* applications de la fonction, etc.

Vous devez :

1. Écrire la fonction `zero` de type `Number t` qui correspond au nombre 0.
2. Écrire la fonction `succ` de type `Number t -> Number t` qui retourne le successeur du nombre reçu en argument. Pour vous aider, voici une définition partielle de `succ`

```
succ  :: Number t -> Number t
succ n = \f z ->
```

3. Écrire la fonction `plus` de type `Number t -> Number t -> Number t` qui prend deux entiers et retourne un entier qui représente l'addition des deux.

Par exemple, dans le code suivant `myOne` vaut 1

```
one  :: Number t
one = succ zero

myOne = one (+ 1) 0
```

et dans le code suivant `myTwo` vaut 10

```
two  :: Number t
two = plus (succ zero) (succ zero)

myTwo = two (+ 5) 0
```

Partie III — Paires

Vous devez implanter le concept d'une paire à l'aide de fonctions. Le type d'une paire est :

```
type Pair a b t = (a -> b -> t) -> t
```

Vous devez :

1. Écrire la fonction `mkPair` de type `a -> b -> Pair a b t` qui construit une paire. Pour vous aider, voici une définition partielle de `mkPair`

```
mkPair :: a -> b -> Pair a b t
mkPair a b = \f ->
```

2. Écrire la fonction `fst` de type `Pair a b a -> a` qui retourne le premier élément de la paire.
3. Écrire la fonction `snd` de type `Pair a b b -> b` qui retourne le deuxième élément de la paire.

Par exemple, dans le code suivant `x` vaut 0

```
x = fst (mkPair 0 "c")
```

et dans le code suivant `x` vaut "c"

```
x = snd (mkPair 0 "c")
```

Partie IV — Liste

Finalement, vous devez implanter le concept d'une liste à l'aide de fonctions. Le type d'une liste est :

```
type List a t = (a->t->t)->t->t
```

Ce type est très similaire à celui des entiers. Vous pouvez réutiliser la même logique. Vous devez :

1. Écrire la fonction `cons` de type `a -> List a t -> List a t`. Pour vous aider, voici une définition partielle de `cons`
- ```
cons :: a -> List a t -> List a t
cons x l = \f z ->
```
2. Écrire la fonction `nil` de type `List a t` qui représente la liste vide.
  3. Écrire la fonction `isNil` de type `List a (Bool t2) -> Bool t2` qui retourne vrai ou faux si la liste est vide ou non. Il s'agit bien du type `Bool a` vu à la section I
  4. Écrire la fonction `head` de type `List a a -> a -> a` qui retourne le premier élément de la liste. Le deuxième paramètre est une valeur par défaut qu'il faut retourner si la liste est vide.

Par exemple, dans le code suivant `x` vaut "vrai"

```
x = isNil nil "vrai" "faux"
```

et dans le code suivant `x` vaut "faux"



```
x = isNil (cons 1 nil) "vrai" "faux"
```

et dans le code suivant x vaut 1

```
x = head (cons 1 nil) 0
```

et dans le code suivant x vaut 0

```
x = head nil 0
```

## Pour les curieux

Maintenant vous savez qu'en présence des fermetures il n'est pas nécessaire (en théorie) d'avoir des structures de données. C'est pourquoi le lambda calcul, qui est un langage minimaliste composé uniquement de fonctions, est souvent utilisé pour modéliser les langages fonctionnels.

### Exercice 6.2

Définir en Haskell des fonctions pour gérer des tables associatives sans utiliser de constructeur. Plus précisément, définir :

```
-- Le type des tables
type T a b = ...

-- Une table vide
empty :: T a b

-- Ajoute une valeur de type b liée à la clé de type a
-- dans une table existante
add :: (Eq a) => a -> b -> T a b -> T a b

-- Renvoie la valeur de type b liée à la clé de type a
-- dans la table ou sinon une valeur par défaut (3e paramètre)
lookup :: (Eq a) => a -> T a b -> b -> b
```

Vu que les constructeurs de données ne peuvent pas être utilisés, les tables seront nécessairement représentées par des fermetures.

## 7 Types

### Exercice 7.1

Inférer les types des expressions Haskell ci-dessous. Vous pouvez tenir pour acquis que les nombres sont de type `Int`.

1. `5`
2. `'c'`
3. `"Hello World"`
4. `[1, 2]`
5. `['c', 'b']`
6. `\x -> x + 1`
7. `let x = 5 in x + 1`
8. `\f x -> f x x`
9. `\f f2 y -> f (f2 y)`
10. `\f -> let x = 5; y = "hello" in f x y`
11. `[\x -> x + 1, \y -> y]`

### Exercice 7.2

Donner le type des expressions Haskell ci-dessous. Par exemple, pour la question 0, la réponse pourrait être : `(Int, Int)`. Le type donné devrait être aussi polymorphe que possible.

0. `(2, 3)`
1. `\a b -> a`
2. `['a', 'b'] ++ ['c', 'd']`
3. `\f -> map f [1, 2, 3]`
4. `map (\x -> x) [1, 2, 3]`

```

5. let chain f1 f2 = \x -> f1 (f2 x) in chain fst fst
6. \x y -> x + y
7. [(1, 2), (3, 4)]
8. let f eval exp = case eval exp of
 Left a -> Left a
 Right x -> Right (x + 1)
 in f
9. let x = y x
 y a b = b
 in x
10. let f x = x (x 5) in f

```

Pour rappel, `\` sert à définir une fonction anonyme. Par exemple `\x -> x` est la fonction identité. De plus, les fonctions `map` et `fst` ont pour définition :

```

map f [] = []
map f (x : xs) = f x : map f xs

fst (a, b) = a

```

## 8 Programmation fonctionnelle

### Exercice 8.1

Soit un arbre binaire défini par le datatype ci-dessous

```
data Tree = Leaf Int
 | Node Tree Int Tree
```

Écrire le code des fonctions suivantes :

**nbrLeaf** Retourne le nombre de feuilles dans l'arbre

**nbrNodes** Retourne le nombre de noeuds dans l'arbre (la fonction ne compte pas les feuilles)

**applyFunction** Prend une fonction  $f$  de type `Int -> Int` en paramètre ainsi qu'un arbre  $t$  et applique la fonction  $f$  à l'entier de chaque **Node** et **Leaf** de  $t$ .

La signature de `applyFunction` est :

```
applyFunction :: (Int -> Int) -> Tree -> Tree.
```

### Exercice 8.2

Écrire une petite calculatrice Haskell qui fait des additions. Le langage arithmétique de cette calculatrice correspond au datatype suivant :

```
data Exp = Enum Int
 | Eplus Exp Exp
```

Il faut écrire la fonction `eval` qui convertit ces expressions en nombre. Cette fonction a la signature suivante :

```
eval :: Exp -> Int
```

Voici trois exemples :

- `eval (Enum 1) == 1`
- `eval (Eplus (Enum 1) (Enum 1)) == 2`
- `eval (Eplus (Eplus (Enum 1) (Enum 1)) (Enum 1)) == 3`

### Exercice 8.3

Implanter en Haskell une variante de quicksort pour des listes d'entiers. En clair, trier une liste comme suit :

1. choisir un élément, que l'on nommera le pivot.
2. partitionner la liste en deux sous-listes d'éléments plus petits et respectivement plus grands que le pivot.
3. trier les deux sous-listes.
4. combiner ces sous-listes triées et le pivot en une liste triée.

Le type sera : `quicksort :: [Int] -> [Int]`. Il faudra peut-être définir une ou plusieurs fonctions auxiliaires. L'opération de concaténation de deux listes s'écrit `++` en Haskell.

Finalement, généraliser la fonction de tri précédente pour pouvoir l'appliquer à des listes quelconques (pas seulement `Int`), en passant un argument supplémentaire qui indique l'opération de comparaison à utiliser.

Donner aussi le type de cette fonction plus générale et de toutes les fonctions auxiliaires que vous avez définies.

### Exercice 8.4

Soit le type suivant en Haskell qui définit un arbre binaire que l'on peut utiliser pour représenter une table associative (qui associe des *clés* de type `Int` à des valeurs de type `b`) :

```
data TreeMap b = Empty | Node Int b (TreeMap b) (TreeMap b)
```

L'exercice est de définir les opérations typiques sur une telle structure de donnée. Bien sûr, pour être utile l'arbre doit être maintenu dans l'ordre : toutes les clés dans la branche de gauche d'un `Node` doivent être plus petites que la clé du noeud, et vice versa pour la branche de droite.

Il y a trois opérations :

- `tmLookup` : rechercher la valeur associée à une clé passée en paramètre.
- `tmInsert` : ajouter une entrée (donnée sous la forme d'une clé et de sa valeur) dans la table.
- `tmRemove` : enlever une entrée (dont la clé est passée en paramètre).

Ces fonctions ne doivent jamais signaler d'erreur.

1. Donner le type de ces trois fonctions.
2. Donner une liste, aussi concise et complète que possible, d'axiomes formels auxquels ces opérations doivent obéir. E.g. un de ces axiomes formalisera le fait qu'un `tmLookup` d'une clé  $x$  juste après un `tmInsert` de la même clé avec une valeur  $v$  devrait trouver  $v$ .

3. Donner le code des trois fonctions.

Pour rendre l'exercice plus utile, il est important de faire ces étapes dans l'ordre : i.e. ne pas écrire le code avant d'avoir décidé du type des fonctions et de leurs spécifications.

### Exercice 8.5

Vous devez écrire le vérificateur de types et l'évaluateur pour un langage uniquement composé d'opérateur arithmétique. Les opérateurs disponibles sont `+`, `-`, `==`, `<`, `if` et `not`. Ils peuvent être encodés par le datatype ci-dessous :

```
data Operateur = OPlus
 | OMinus
 | OEqual
 | OLessThan
 | OIf
 | ONot
 deriving (Show)
```

Le langage est composé d'applications préfixes d'opérateurs, de nombres entiers et de booléens. Le datatype `Exp` ci-dessous peut être utilisé comme arbre de syntaxe abstraite.

```
data Exp = EInt Int
 | EBool Bool
 | EOp Operateur
 | EApp [Exp]
 deriving (Show)
```

Par exemple, l'ASA ci-dessous est équivalent à `(if (< 1 2) True False)`

```
EApp [EOp OIf, (EApp [EOp OLessThan, EInt 1, EInt 2]),
 EBool True,
 EBool False]
```

Le résultat d'une évaluation est une valeur représentée par le type `Value`.

```
data Value = VInt Int
 | VBool Bool
 deriving (Show)
```

Vous devez écrire la fonction `typeCheck` qui doit vérifier que les expressions sont valides. Chaque expression peut uniquement être de type entier ou booléen.

```
data Type = TInt
 | TBool
 deriving (Eq, Show)
```

Les applications partielles d'opérateur ne sont pas permises. Évidemment une application avec trop de paramètres est également une erreur. Les opérateurs acceptent soit des entiers ou des booléens selon leur sémantique habituelle. La branche vraie et la branche alternative du `if` doivent être du même type. Votre fonction `typeCheck` doit avoir la signature suivante :

```
typeCheck :: Exp -> Either String Type
```

Une fois la fonction `typeCheck` écrite, vous pouvez écrire la fonction `eval` sans avoir à gérer les erreurs. Ainsi, les cas impossibles lorsqu'une expression est bien typée, une application de `+` avec un seul argument par exemple, n'ont pas à être gérés. Votre fonction `eval` aura pour signature :

```
eval :: Exp -> Value
```

## Exercice 8.6

Les listes infinies sont souvent appelées *streams*. En Haskell, l'ordre d'évaluation utilisé permet d'utiliser n'importe quelle structure de donnée infinie sans effort particulier. Prenons par exemple les définitions ci-dessous :

```
zeros = 0 : zeros
uns = 1 : uns
```

De plus, Haskell prédéfinit les opérations suivantes :

```
(x : _) !! 0 = x
(_ : xs) !! n = xs !! (n - 1)

take 0 _ = []
take _ [] = []
take n (x : xs) = x : take (n - 1) xs

zipWith :: (a->b->c) -> [a]->[b]->[c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (a : as) (b : bs) = f a b : zipWith f as bs
```

1. Définir la liste de nombres :  
(1 2 3 4 5 ...)
2. Définir la liste des nombres de Fibonacci :  
(1 1 2 3 5 8 13 ...)
3. Définir la liste de nombres :  
(1 1/2 1/6 1/24 1/120 ... 1/n! ...)

## Exercice 8.7

Soit la fonction C suivante :

```
void main (void)
{ /* Élimine les caractères répétés et
 * stoppe après la première ligne vide. */
 int c;
 int last = EOF;
 while ((c = getchar ()) != EOF) {
 if (c == last) {
 if (c == '\n') {
 break;
 } else {
 continue;
 }
 }
 putchar (last = c);
 }
}
```

D'abord, réécrire le code dans un style de programmation structurée stricte, c'est à dire sans utiliser de `continue`, `break`, ou `goto`.

Ensuite, réécrire le code à nouveau mais cette fois dans un style fonctionnel, c'est à dire sans opération d'affectation (sauf bien sûr dans les initialisations, e.g. `int last = EOF`). Il faudra introduire une fonction récursive auxiliaire et éliminer `while`, `break`, et `continue`.



## 9 Programmation structurée

### Exercice 9.1

Vous devez aider un ami à traduire dans le langage C un programme écrit en Haskell.

1. Convertir les datatypes Haskell ci-dessous dans le langage C :

```
data Personne = Personne Int String -- Age et Prénom
```

```
data SuperPouvoir = SauterHaut Int
 | SauterLoin Int
 | GrimperAuMur Int
 | Imiter Personne
```

```
data SuperHero = SuperHero Personne SuperPouvoir
```

2. Écrire les fonctions Haskell ci-dessous dans le langage C dans un style de programmation structurée (pas de goto).

Vous pouvez considérer que les listes Haskell sont des tableaux en C et que la taille du tableau vous est fournie en paramètre.

```
plusVieux :: [SuperHero] -> [SuperHero]
```

```
plusVieux ls =
```

```
 map (\(SuperHero (Personne age nom) sp) ->
 SuperHero (Personne (age + 1) nom) sp)
 ls
```

```
auMoinsUnSuperHero :: [SuperHero] -> Bool
```

```
auMoinsUnSuperHero ls =
```

```
 let foo (SuperHero (Personne _ "Spiderman") _) _ = True
 foo (SuperHero (Personne _ "Batman") _) _ = True
 foo (SuperHero (Personne _ "Capitaine Bobette") _) _ = True
 foo _ b = b
 in foldr foo False ls
```

La fonction foldr a pour définition :

```
foldr f z [] = z
```

```
foldr f z (x :xs) = f x (foldr f z xs)
```

## 10 Gestion mémoire

### Exercice 10.1

1. Nommer et expliquer deux types de problèmes qui peuvent survenir lorsque le programmeur gère lui-même la mémoire avec `malloc` et `free`
2. Expliquer en quoi consiste la technique de comptage de références pour gérer la mémoire. Nommer et expliquer une faiblesse importante de cette méthode.
3. Nous avons vu en cours deux algorithmes pour la gestion automatique de la mémoire : Mark and Sweep et Stop and Copy. Expliquer à l'aide d'un pseudo-code ou d'un schéma le fonctionnement d'un de ces deux algorithmes. Nommez un avantage de l'algorithme choisi par rapport à l'autre.

### Exercice 10.2

Soit une librairie de gestion de listes simplement chaînées en C :

```
typedef struct list list;
struct list {
 int refcount;
 void *head;
 list *tail;
}
list *list_cons (void *head, list *tail);
void *list_head (list *l);
list *list_tail (list *l);
/* Copie un pointeur (pas la liste elle même). */
list *list_copy (list *l);
/* Libère un pointeur (et la liste si c'est le dernier). */
void list_free (list *l);
```

1. Écrire le code des fonctions proposées.
2. Compléter en ajoutant une opération `list_map`.
3. Justifiez pourquoi les incréments et décréments que vous avez judicieusement placés sont suffisants pour garantir que le comportement sera correct.
4. En extraire une convention spécifiant pour les programmeurs qui utilisent votre librairie où doivent être ajoutés les appels à `list_copy` et `list_free`.

5. Que se passe-t-il si vous voulez manipuler des listes de listes ?

### Exercice 10.3

Écrire une librairie de listes simplement chaînées en C.

```
typedef struct list_elem list_elem;
struct list_elem {
 void *value;
 list_elem *next;
}
typedef struct list list;
struct list { list_elem *head; }

list *list_alloc (void);
void list_insert (list *l, void *v);
void *list_get (list *l, int n);
```

1. Écrire le code des trois fonctions proposées. Compléter en ajoutant les opérations suivantes : `list_remove`, `list_free`.
2. Décrire précisément les conditions nécessaires (que l'utilisateur de la librairie doit suivre) pour qu'il n'y ait pas de dérèférérence de pointeur fou, ni de fuite.
3. Expliquer pourquoi ces conditions sont nécessaires et suffisantes pour garantir l'absence d'erreurs de type pointeur fou ou fuite.

### Exercice 10.4

1. Que se passe-t-il si vous passez à la fonction `free` un pointeur qui n'a jamais été retourné par la fonction `malloc` ? Que se passe-t-il si vous passez un pointeur de valeur `NULL` ?
2. Que fait la fonction `mmap` qui se trouve dans la librairie `<sys/mman.h>` sous Unix ?
3. Quelle est la différence entre `mmap` et `malloc` ?

# 11 Continuation

## Exercice 11.1

Réécrire les fonctions suivantes en mode CPS (continuation passing style). C'est à dire que chaque fonction doit prendre une continuation et envoyer son résultat à cette continuation.

```
import Prelude hiding (length)

length :: [a] -> Int
length [] = 0
length (_ : xs) = 1 + length xs

applatir :: [[a]] -> [a]
applatir [] = []
applatir (xs : xss) = xs ++ applatir xss

fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n - 1)

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

## 12 Macros

### Exercice 12.1

1. Définir une macro qui démontre que les macros implémentent le passage par nom. Dans quel cas cela fait-il une différence par rapport au passage par valeur ?
2. Indiquer comment obtenir un passage par valeur.
3. Votre solution pour le passage par valeur a fort probablement introduit un nouveau problème, lequel et comment le régler ?

### Exercice 12.2

Soit une macro **case** en Scheme qui peut s'utiliser comme suit :

```
(case <exp>
 ((1 3 5) <exp1>)
 ((4) <exp2>)
 (else <exp3>))
```

qui signifierait exécuter **exp1** si **exp** vaut 1, 3, ou 5 ; exécuter **exp2** si **exp** vaut 4 et exécuter **exp3** sinon.

- Écrire une définition (naïve) de cette macro en Scheme avec **define-macro**.
- Discuter des différents problèmes qui peuvent apparaître lors de l'usage de cette macro dûs à son implémentation naïve. Montrer des exemples concrets d'usage où la macro ne fait pas ce que le programmeur attendait.
- Écrire une implémentation plus raffinée qui fait très attention à ce que le code donne une sémantique propre, sans mauvaises surprises pour l'utilisateur de cette macro.

### Exercice 12.3

Définir la macro **postfix** qui prend une expression sous forme postfixée :

```
(let ((x 5))
 (postfix 1 x + 3 * 2 /)) ; retourne 9
```

Il suffira d'accepter les opérateurs `+`, `-`, `*`, `/`, `not`, `≥`, et `if`.

Montrer les étapes de la compilation et de l'évaluation de `(postfix 1 x + 3 * 2 /)` ci-dessus, jusqu'à l'obtention du résultat 9, en indiquant clairement quelles parties ont lieu lors de la compilation et quelles parties ont lieu à l'exécution.

### Exercice 12.4

Faire une macro pour la notation infixe des opérateurs `+`, `-`, `*`, `/`, `>`, `<`, `=`.

Par exemple :

```
(let ((x 5))
 (infix (1 + x) * 3 / 2)); 9
```

```
(let ((x 5)
 (y 10))
 (infix x * 2 < y - 1)); #f
```

```
(let ((x 5)
 (y 10))
 (infix x * 2 = y)); #t
```

Avec la forme `quasiquote` écrire une macro qui permet de faire des boucles `while` avec la syntaxe suivante.

```
(while <condition> <expression>)
```

Utiliser `gensym` pour tout nouvel identificateur introduit par la macro pour éviter la capture de variable. Voici un exemple d'utilisation :

```
(let ((i 1))
 (while (< i 10)
 (begin
 (write i)
 (set! i (* i 2)))))
```

### Exercice 12.5

À l'aide de `define-macro`, écrire la macro `define-avec-trace` qui a le comportement suivant :

```
> (define-avec-trace f (lambda (n) (if (= n 0) 1 (* n (f (- n 1))))))
> (f 5)
(fonction : f parametres : (n = 5))
(fonction : f parametres : (n = 4))
(fonction : f parametres : (n = 3))
(fonction : f parametres : (n = 2))
(fonction : f parametres : (n = 1))
(fonction : f parametres : (n = 0))
120
```

### Exercice 12.6

Après vous être renseigné sur la définition de `let` et `let*` de Scheme, écrire une macro `mylet` et une macro `mylet*` qui fournissent les mêmes fonctionnalités et mais dont la définition n'utilisent que des fonctions anonymes (`(lambda ... ...)`) et l'application de fonction dans leur expansion.

# 13 Programmation logique

## Exercice 13.1

Soit `add(X,Y,Z)` une relation qui dit que `Z` est la somme de `X` et `Y`. Définir la relation `mult(X,Y,Z)` qui fait la même chose pour la multiplication en utilisant `add`.

## Exercice 13.2

La relation `axe(X,Y,Z)` telle que définie ci-après peut donner des solutions redondantes.

```
axe(_,0,0).
axe(0,_,0).
```

Donner d'abord un exemple d'interaction avec le système GNU Prolog qui montre cette redondance.

Utiliser ensuite l'opérateur prédéfini `T1 \== T2` pour corriger la définition précédente de manière à éviter les solutions redondantes produites par la relation `axe(X,Y,Z)`. L'opérateur `T1 \== T2` permet de tester si les termes `clos` et `T2` sont différents. C'est un opérateur impur car il ne permet pas, par exemple, d'énumérer tous les termes `T2` possibles qui sont différents de `T1`.

## Exercice 13.3

Soit le code Prolog suivant qui interprète un mini langage typé dynamiquement et composé seulement d'entiers `int(N)`, de variables `id(X)`, de fonctions `lambda(X,E)`, et d'applications de fonctions `app(E1,E2)` :

```
%% subst(+E1, +E2, +X, -V)
%% indique que la substitution de X par V dans E1 renvoie E2.
%% On présume que V est fermé!
subst(int(N), int(N), _, _).
subst(id(X), V, id(X), V).
subst(id(Y), id(Y), id(X), _) :- X \== Y.
subst(lambda(X, E), lambda(X, E), X, _).
subst(lambda(Y, Ea), lambda(Y, Eb), X, V) :-
 X \== Y, subst(Ea, Eb, X, V).
subst(app(E1a, E2a), app(E1b, E2b), X, V) :-
 subst(E1a, E1b, X, V), subst(E2a, E2b, X, V).

%% reduce(+E, -V)
```



```

%% indique que l'évaluation de E renvoie V.
reduce(int(N), int(N)).
reduce(lambda(X, B), lambda(X, B))
reduce(app(E1, E2), V) :-
 reduce(E1, lambda(X, B)),
 reduce(E2, V2),
 subst(B, E, X, V2),
 reduce(E, V).

```

1. Quel mode de passage d'arguments l'interpréteur ci-dessus plante-t-il ?
2. Modifier le code Prolog de sorte à implanter l'autre mode de passage d'arguments.
3. Est-ce que la portée des variables est statique ou dynamique ? Donner un morceau de code prolog qui fait la différence.

### Exercice 13.4

Supposons qu'on a déjà défini les relations suivantes :

```

pere(X,Y) X est le père de Y
mere(X,Y) X est la mère de Y

```

Définissez les relations suivantes, en évitant autant que possible les solutions redondantes et les boucles infinies. Vous pouvez supposer qu'il n'y a pas d'enfants issus d'un couple consanguin.

```

parent(X,Y) X est un parent de Y
grandpere(X,Y) X est un grand-père de Y
grandmere(X,Y) X est une grand-mère de Y
freresoeur(X,Y) X est un frère ou une soeur de Y
oncletante(X,Y) X est un oncle ou une tante de Y

```

### Exercice 13.5

Soit la relation `membre` qui définit quand un élément est dans une liste :

```

membre(E, [E | _]).
membre(E, [_ | L]) :- membre(E, L).

```

1. Montrer l'arbre de preuve des étapes importantes par lesquelles passe le système Prolog pour essayer de satisfaire la requête :

```
membre(X, [1, 1]), X == 2.
```

Où `A == B` est le prédicat qui demande l'unification de `A` et de `B`.

Les étapes importantes sont celles où la recherche tombe sur une impossibilité (et doit donc faire un retour-arrière).

2. Corriger la relation avec l'aide de l'inégalité `\==` de sorte qu'elle évite la redondance.
3. En utilisant votre nouvelle définition, dessiner l'arbre de preuve qui satisfait la requête `membre(1, [2, 1, 3])`.
4. De même montrer les arbres de preuve importants construits pour essayer de satisfaire la requête `membre(1, [2, 3, 4])`.

### Exercice 13.6

Définir la règle de tri `quicksort(X,Y)` qui dit que `Y` contient les mêmes éléments que la liste `X`, mais triés par ordre croissant. Cette version utilisera un opérateur de comparaison fixe, la relation `'<'`.

Une règle auxiliaire `partition(X,L,S,G)` sera nécessaire qui dit que la liste `S` contient les éléments de la liste `L` qui sont plus petits que `X`, alors que `G` contient ceux qui sont plus grand.

Utiliser la règle prédéfinie `append(X,Y,Z)` qui dit que `Z` est la concaténation des listes `X` et `Y`.

### Exercice 13.7

En Prolog, tout comme en Haskell, les chaînes de caractères sont représentées par des *listes* de caractères.

Soit la définition suivante du prédicat de filtrage `match(RE,Str,Tail)` qui dit que l'expression régulière `RE` filtre la chaîne de caractères `Str` avec un résidu `Tail`.

```
match(RE, Str, Tail) :- append(RE, Tail, Str).
match(any, [_|Tail], Tail).
match(concat(RE1, RE2), Str, Tail) :-
 match(RE1, Str, Tail1), match(RE2, Tail1, Tail).
match(repeat(RE), Str, Tail) :-
 match(RE, Str, Tail1), match(repeat(RE), Tail1, Tail).
match(repeat(_), Str, Str).
```

Par exemple :

```
| ?- match("hello", "hello world", X).
X = " world"
| ?- match(concat("hel", "lo "), X, "").
X = "hello "
| ?- match(concat(repeat(any), concat("or", repeat(any))),
 "hello world", X).
X = ""
X = "d"
X = "ld"
```

1. Montrer le (les) arbres de recherche de la requête :  
`match(concat(repeat("a"), "ab"), "abc", X)`
2. Que se passe-t-il avec la requête :  
`match(concat(repeat("a"), "ab"), X, "")`
3. Changer le code de manière à éviter ce problème.
4. Ajouter du code pour l'expression régulière `or(RE1, RE2)`, de sorte à pouvoir faire des choses telles que :

```
| ?- match(or("hello", "world"), X, "").
X = "hello"
X = "world"
| ?- match(repeat(or("a", "b")), "abac", X).
X = "c"
X = "ac"
X = "bac"
X = "abac"
```

5. Montrer le (les) arbres de recherche de la requête :  
`match(repeat(or("a", "b")), "abac", X)`
6. Ajouter du code pour l'expression régulière `and(RE1, RE2)`, de sorte à pouvoir faire des choses telles que :

```
| ?- match(concat(and(concat(repeat(any),
 concat("ol", repeat(any))),
 concat(repeat(any),
 concat("ll", repeat(any)))),
 "o"),
 "bell hollow world", X).
X = "rld"
X = "rld"
X = "w world"
X = "w world"
```

# 14 Solutions

## Solution de l'exercice 2.1

1. **préfixe**  $++abc$   
**postfixe**  $ab+c+$
2. **préfixe**  $+a+bc$   
**postfixe**  $abc++$
3. **préfixe**  $+ \cdot ab \cdot cd$   
**postfixe**  $ab \cdot cd \cdot +$
4. **préfixe**  $< +ab \cdot a + cd$   
**postfixe**  $ab + acd + \cdot <$
5. **préfixe**  $\sqrt{-} \cdot bb \cdot \cdot 4ac$   
**postfixe**  $bb \cdot 4a \cdot c \cdot - \sqrt{}$

## Solution de l'exercice 2.2

1. 1. **préfixe**  $* - + a b c d$   
**infixe**  $(a + b - c) * d$   
2. **préfixe**  $|| \&\& a b == c d$   
**infixe**  $(a \&\& b) || (c == d)$   
3. **préfixe**  $\text{not} == + a b * c d$   
**infixe**  $\text{not}(a + b == c * d)$
2. Il faut évaluer à l'aide d'une pile l'expression  $1\ 2\ +\ 3\ 4\ *\ ==\ \text{not}$ . La pile sera représenté par une liste, l'élément au sommet de la pile est à gauche de la liste. Voici les étapes d'évaluation
  1. Lecture du chiffre 1. Lorsque c'est un nombre, on l'ajoute dans la pile. Elle vaut  $[1]$
  2. Lecture du chiffre 2. La pile vaut  $[2, 1]$

3. Lecture du symbole `+`. On pop les deux premiers éléments de la pile, calcule le résultat et l'ajoute dans la pile. La pile vaut `[3]`.
4. Lecture du symbole `3`. La pile vaut `[3, 3]`
5. Lecture du symbole `4`. La pile vaut `[4, 3, 3]`
6. Lecture du symbole `*`. Même chose que pour l'étape 3. La pile vaut `[12, 3]`
7. Lecture du symbole `==`. Même chose que pour l'étape 3. La pile vaut `[False]`
8. Lecture du symbole `not`. On pop le premier élément de la pile, calcule le résultat et l'ajoute sur la pile. La pile vaut `[True]`
9. Fin de l'expression. La valeur sur la pile donne le résultat, soit `True`

### Solution de l'exercice 2.4

Voici une phrase ambiguë : `if E1 then if E2 then E3 else E4`. Elle peut s'interpréter des deux façons ci-dessous.

- `if E1 then (if E2 then E3 else E4)`
- `if E1 then (if E2 then E3) else E4`.

Pour lever l'ambiguïté, il faut par exemple arbitrairement décider que les `else` se réfèrent aux `if` les plus proches.

```
<S> ::= <X>
 | 'if' <E> 'then' <S>
 | 'if' <E> 'then' <SElse> 'else' <S>

<SElse> ::=
 | 'if' <E> 'then' <SElse> 'else' <SElse>
```

### Solution de l'exercice 3.2

`(x -> 5 ; y -> 6 ; ...)` `exp` dénote l'évaluation de `exp` dans l'environnement où `x` vaut 5, `y` vaut 6, etc. On peut imaginer l'environnement comme une pile. Lorsqu'on cherche la valeur d'un identificateur, on prend la première variable sur le dessus qui correspond.

Pour rappel, `\x -> body` est une fonction anonyme dont le paramètre est `x`.

### Évaluation avec la portée dynamique

```
-- L'environnement avant d'évaluer (f3 5)
{f3 -> \z = f2 (z + 2) ;
 f2 -> \x = f1 (x + 1) ;
 f1 -> \y = z + x + y avec {z = 1 ; x = 2}} ;
x -> 2 ;
```

```

z -> 1} (f3 5)

-- Recherche la valeur de f3 dans l'environnement
{...} ((\z = f2 (z + 2)) 5)

-- Application de fonction, on ajoute le
-- paramètre et sa valeur dans l'environnement
{z -> 5; ...} (f2 (z + 2))

-- Recherche la valeur de z dans l'environnement
{z -> 5; ...} (f2 (5 + 2))

-- Calcul du paramètre
{z -> 5; ...} (f2 7)

-- Recherche la valeur de f2 dans l'environnement
{z -> 5; ...} ((\x = f1 (x + 1)) 7)

-- Application de fonction
{x -> 7; z -> 5; ...} (f1 (x + 1))

-- Recherche la valeur de x dans l'environnement
{x -> 7; z -> 5; ...} (f1 (7 + 1))

-- Calcul du paramètre
{x -> 7; z -> 5; ...} (f1 8)

-- Recherche la valeur de f1 dans l'environnement
{x -> 7; z -> 5; ...} ((\y = z + x + y) 8)

-- Application de fonction
-- Ici x fait référence au x le plus récent
-- Même chose pour z
{y -> 8; x -> 7; z -> 5; ...} (z + x + y) z

-- Recherche la valeur de z, x et y
{y -> 8; x -> 7; z -> 5; ...} (5 + 7 + 8)

-- Calcul du résultat
{y -> 8; x -> 7; z -> 5; ...} 20

```

## Évaluation avec la portée statique

Pour implanter la portée statique, il faut pour chaque expression mémoriser la définition des variables libres. Ainsi, si d'autres variables avec le même identifiant sont

ajoutées dans l'environnement, cela n'affectera pas le résultat car on utilisera la définition qui a été mémorisée.

Ainsi, on peut imaginer que chaque fonction a son propre environnement pour ses variables libres. Pour simplifier, dans cet exercice nous mémoriserons seulement la définition des variables  $x$  et  $y$ .

```
-- L'environnement avant d'évaluer (f3 5)
{f3 -> \z = f2 (z + 2) ;
 f2 -> \x = f1 (x + 1) ;
 f1 -> \y = z + x + y ;
 x -> 2 ;
 z -> 1} (f3 5)

-- Recherche la valeur de f3 dans l'environnement
{...} ((\z = f2 (z + 2)) 5)

-- Application de fonction, on ajoute le
-- paramètre et sa valeur dans l'environnement
{z -> 5 ; ...} (f2 (z + 2))

-- Recherche la valeur de z dans l'environnement
{z -> 5 ; ...} (f2 (5 + 2))

-- Calcul du paramètre
{z -> 5 ; ...} (f2 7)

-- Recherche la valeur de f2 dans l'environnement
{z -> 5 ; ...} ((\x = f1 (x + 1)) 7)

-- Application de fonction
{x -> 7 ; z -> 5 ; ...} (f1 (x + 1))

-- Recherche la valeur de x dans l'environnement
{x -> 7 ; z -> 5 ; ...} (f1 (7 + 1))

-- Calcul du paramètre
{x -> 7 ; z -> 5 ; ...} (f1 8)

-- Recherche la valeur de f1 dans l'environnement
-- f1 a mémorisé les définitions pour x et z
-- Cela revient à dire que f1 s'évalue dans cet
-- environnement spécial
{x -> 2 ; z -> 1} ((\y = z + x + y) 8)

-- Application de fonction
```

```

{y -> 8; x -> 2; z -> 1} (z + x + y) z

-- Recherche la valeur de z, x et y
{y -> 8; x -> 2; z -> 1} (8 + 2 + 1)

-- Calcul du résultat
{y -> 8; x -> 2; z -> 1} 11

```

### Solution de l'exercice 3.3

```

data Exp = EInt Int
 | Var String
 | Lambda String Exp
 | App Exp Exp
 deriving (Show)

type Env a = [(String, a)]

data ValueD = VVar String
 | VInt Int
 | VDLambda String Exp
 deriving (Show)

evalD :: Env ValueD -> Exp -> Either String ValueD
evalD _ (EInt x) = Right $ VInt x
evalD env (App e1 e2) = do
 e1' <- evalD env e1
 e2' <- evalD env e2
 case e1' of
 VDLambda var body -> evalD ((var, e2') : env) body
 _ -> Left "Operand must be a lambda"
evalD env (Lambda x body) = Right $ VDLambda x body
evalD env (Var x) =
 case lookup x env of
 Nothing -> Left "Variable not found"
 Just x -> Right x

-- Remarquer que les fonctions portent un environnement pour retenir
-- la valeur de leur variable libre et que c'est dans cet
-- environnement que le corps de la fonction est exécuté.
data Value = VVar String
 | VInt Int
 | VLambda String Exp (Env Value)

```



```

 deriving (Show)

evalL :: Env Value -> Exp -> Either String Value
evalL _ (EInt x) = Right $ VInt x
evalL env (App e1 e2) = do
 e1' <- evalL env e1
 e2' <- evalL env e2
 case e1' of
 VLambda var body closure -> evalL ((var, e2') : closure) body
 _ -> Left "Operand must be a lambda"
-- Normalement il faudrait retenir seulement les variables libres de
-- la fonction, mais pour la démo il est suffisant
-- de retenir tout l'environnement et tant pis pour la fuite mémoire
evalL env (Lambda x body) = Right $ VLambda x body env
evalL env (Var x) =
 case lookup x env of
 Nothing -> Left "Variable not found"
 Just x -> Right x

-- Le code suivant démontre la différence entre les deux évaluateurs
-- Il est l'équivalent de :
-- let x = 4
-- f = \y -> x Ici x doit être 4 en portée lexicale
-- x = 3
-- in f 3

e1 = App (Var "f") (EInt 3)
e2 = App (Lambda "x" e1) (EInt 3)
e3 = App (Lambda "f" e2) (Lambda "y" (Var "x"))
e4 = App (Lambda "x" e3) (EInt 4)

eDynamique = evalD [] e4 -- Vaut VInt 3
eLexicale = evalL [] e4 -- Vaut VInt 4

```

## Solution de l'exercice 4.1

Il est plus facile de scinder le problème en deux. Une fonction qui teste la portée et l'autre le passage des paramètres. Voici un programme C qui permet de faire la différence.

```

#include <stdio.h>

int nom = 0;
int valeurResultat = 1;
int reference = 2;

```

```

int portee = 0;

// Test de portée
int bar(){
 return portee;
};

int foo(int testRef, int testNom){
 // Si y est passé par nom, son effet de bord sera dupliqué
 int b1 = testNom + testNom;

 // On modifie le paramètre
 // Cela affectera l'appelant avec passage par référence
 // et valeur-résultat
 // Avec passage par nom, on peut considérer que cela va aussi le
 // modifier si l'utilisation du paramètre a gauche du = est permise
 testRef = testRef + 1;

 // Avec un passage par référence, référence est immédiatement modifié
 // Avec un passage par valeurRésultat, référence est modifié seulement
 // a la fin de la fonction et non lors de l'exécution de cette ligne
 valeurResultat = reference;

 // On cache la variable portee
 int portee = 1;

 return bar();
}

void main(){
 int p = foo(reference, nom++); // Appel à foo pour effectuer le test
 printf("(%d, %d, %d, %d)", nom, valeurResultat, reference, p);
 return;
}

// Portée statique et :
// Par valeur : (1, 2, 2, 0)
// Par référence : (1, 3, 3, 0)
// Par nom : (2, 3, 3, 0)
// Par valeur résultat : (1, 2, 3, 0)

// Portée dynamique et :
// Par valeur : (1, 2, 2, 1)
// Par référence : (1, 3, 3, 1)
// Par nom : (2, 3, 3, 1)

```

```
// Par valeur résultat : (1, 2, 3, 1)
```

## Solution de l'exercice 5.1

```
mylength :: [Int] -> Int
mylength [] = 0
mylength (_ : xs) = 1 + mylength xs

myconcat :: [Int] -> [Int] -> [Int]
myconcat [] xs = xs
myconcat (x : xs) ys = x : (myconcat xs ys)

mymember :: Int -> [Int] -> Bool
mymember _ [] = False
mymember x (y : ys) = x == y || mymember x ys

myreverse :: [Int] -> [Int]
myreverse xs =
 let helper [] acc = acc
 helper (x : xs) acc = helper xs (x : acc)
 in helper xs []

mysubList :: [Int] -> [Int] -> Bool
mysubList [] _ = True
mysubList _ [] = False
mysubList xList yList@(_ : ys) =
 let matchInFront [] _ = True
 matchInFront (x : xs) (y : ys) =
 if x == y
 then matchInFront xs ys
 else False
 matchInFront _ _ = False
 in matchInFront xList yList || mysubList xList ys
```

## Solution de l'exercice 6.1

```
import Prelude hiding (Bool, succ, fst, snd, head, tail)

-- Booléen et If
type Bool a = a -> a -> a

false :: Bool a
```

```

false a b = b

true :: Bool a
true a b = a

if2 :: Bool a -> a -> a -> a
if2 test a b = test a b

testIfTrue = if2 true 1 2
testIfFalse = if2 false 1 2

-- Nombre entier
type Number t = (t -> t) -> t -> t

zero :: Number t
zero = \f z -> z

succ :: Number t -> Number t
succ n = \f z -> f (n f z)

one :: Number t
one = succ zero

plus :: Number t -> Number t -> Number t
plus n m = \f z -> n f (m f z)

-- Pair
type Pair a b t = (a -> b -> t) -> t

mkPair :: a -> b -> Pair a b t
mkPair a b = \f -> f a b

fst :: Pair a b a -> a
fst p = p (\a b -> a)

snd :: Pair a b b -> b
snd p = p (\a b -> b)

-- Liste
type List a t = (a -> t -> t) -> t -> t

cons :: a -> List a t -> List a t
cons x l = \f z -> f x (l f z)

nil :: List a t

```

```

nil = \f z -> z

isNil :: List a (Bool t2) -> Bool t2
isNil l = l (_ _ -> false) true

head :: List a a -> a -> a
head l ifEmpty = l (\x _ -> x) ifEmpty

```

## Solution de l'exercice 6.2

```

-- Première version
-- On peut pour démarrer utiliser le datatype Maybe comme
-- valeur de retour. Ainsi, notre table est fabriquée à
-- l'aide des fermetures.
type T a b = (a -> Maybe b)

empty :: T a b
empty = \x -> Nothing

add :: (Eq a) => a -> b -> T a b -> T a b
add key value oldtable = \k -> if k == key
 then Just value
 else oldtable k

lookup :: (Eq a) => a -> T a b -> b -> b
lookup key table def = case table key of
 Nothing -> def
 Just x -> x

-- Deuxième version
-- Sans le type Maybe
-- Le type T2 comporte un b extra par rapport à T pour
-- la valeur de défaut
type T2 a b = a -> b -> b

empty2 :: T2 a b
empty2 x y = y -- On retourne la valeur de défaut

add2 :: (Eq a) => a -> b -> T2 a b -> T2 a b
add2 key value oldtable = \k def -> if k == key
 then value
 else oldtable k def

```

```
lookup2 :: (Eq a) => a -> T2 a b -> b -> b
lookup2 key table def = table key def
```

## Solution de l'exercice 7.1

1. `Int`
2. `Char`
3. `[Char]`
4. `[Int]`
5. `[Char]`
6. `Int -> Int`
7. `Int`
8. `(a -> a -> b) -> a -> b`
9. `(a -> b) -> (c -> a) -> c -> b`
10. `(Int -> [Char] -> b) -> b`
11. `[Int -> Int]`

## Solution de l'exercice 8.1

```
data Tree = Leaf Int
 | Node Tree Int Tree

nbrLeaf (Leaf _) = 1
nbrLeaf (Node tg _ td) = nbrLeaf tg + nbrLeaf td

nbrNodes (Leaf _) = 0
nbrNodes (Node tg _ td) = 1 + nbrLeaf tg + nbrLeaf td

applyFunction :: (Int -> Int) -> Tree -> Tree
applyFunction f (Leaf n) = Leaf (f n)
applyFunction f (Node tg n td) =
 let tg' = applyFunction f tg
 td' = applyFunction f td
 in Node tg' (f n) td'
```

## Solution de l'exercice 8.3

```
-- Version spécialisée pour les liste d'entiers
quicksortInt :: [Int] -> [Int]
quicksortInt [] = []
quicksortInt (x :xs) =
 let (smaller, bigger) = partition (\y -> y <= x) xs
 in quicksortInt smaller ++ [x] ++ quicksortInt bigger

where partition :: (Int -> Bool) -> [Int] -> ([Int],[Int])
partition _ [] = ([],[])
partition test (z :zs) =
 let (a, b) = partition test zs
 in if (test z)
 then (z : a, b)
 else (a, z : b)

-- Version générale. Pour une liste arbitraire et un
-- opérateur de comparaison arbitraire.
quicksort :: (a -> a -> Bool) -> [a] -> [a]
quicksort _ [] = []
quicksort compare (x :xs) =
 let (smaller, bigger) = partition (\y -> compare y x) xs
 in quicksort compare smaller ++ [x] ++ quicksort compare bigger

where partition :: (a -> Bool) -> [a] -> ([a],[a])
partition _ [] = ([],[])
partition test (z :zs) =
 let (a, b) = partition test zs
 in if (test z)
 then (z : a, b)
 else (a, z : b)
```

## Solution de l'exercice 8.4

```
data TreeMap b = Empty | Node (TreeMap b) Int b (TreeMap b)
 deriving (Show, Eq)

tmLookup :: TreeMap b -> Int -> Maybe b
tmLookup Empty _ = Nothing
tmLookup (Node tl k v tr) k1 | k1 == k = Just v
tmLookup (Node tl k v tr) k1 | k1 < k = tmLookup tl k1
```

```

tmLookup (Node tl k v tr) k1 | k1 > k = tmLookup tr k1

tmInsert :: TreeMap b -> Int -> b -> TreeMap b
tmInsert Empty k v = Node Empty k v Empty
tmInsert (Node tl k v tr) k1 v1 | k1 == k =
 (Node tl k v1 tr)
tmInsert (Node tl k v tr) k1 v1 | k1 < k =
 (Node (tmInsert tl k1 v1) k v tr)
tmInsert (Node tl k v tr) k1 v1 | k1 > k =
 (Node tl k v (tmInsert tr k1 v1))

tmRemove :: TreeMap b -> Int -> TreeMap b
tmRemove Empty _ = Empty
tmRemove (Node tl k1 v1 tr) k
 | k < k1 = (Node (tmRemove tl k) k1 v1 tr)
tmRemove (Node tl k1 v1 tr) k
 | k > k1 = (Node tl k1 v1 (tmRemove tr k))
tmRemove (Node tl k1 v1 tr) k
 | k == k1 = (merge tl tr)
 where merge :: TreeMap b -> TreeMap b -> TreeMap b
 merge Empty y = y
 merge x Empty = x
 merge x (Node tl2 k2 v2 tr2) =
 (Node (merge x tl2) k2 v2 tr2)

```

## Solution de l'exercice 8.5

```

-- Une mini calculatrice avec type

-- Les 5 opérateurs de notre calculatrice
-- +, -, ==, <, if, not
data Operateur = OPlus
 | OMinus
 | OEqual
 | OLessThan
 | OIf
 | ONot
 deriving (Show)

-- Le langage est composé d'application préfixe d'opérateurs
-- ou de constante Int ou Bool
-- Il n'y a pas d'application partielle possible

```



```

data Exp = EInt Int
 | EBool Bool
 | EOp Operateur
 | EApp [Exp]
 deriving (Show)

-- ex : EApp [EOp OPlus, EInt 1, EInt 2] -> (+ 1 2)
-- ex : EApp [EOp OIf, (EApp [EOp OLessThan, EInt 1, EInt 2]),
-- EBool True, EBool False]
-- -> (if (< 1 2) True False)

data Value = VInt Int
 | VBool Bool
 deriving (Show)

data Type = TInt
 | TBool
 deriving (Eq, Show)

typeCheck :: Exp -> Either String Type
typeCheck (EInt _) = Right $ TInt
typeCheck (EBool _) = Right $ TBool
typeCheck (EOp _) = Left $ "Opérateur sans arguments"
typeCheck (EApp (EOp op : args)) =
 case op of
 OPlus -> do
 binaryInt args
 return TInt -- Pour Either, return Int <-> Right Int
 OMinus -> do
 binaryInt args
 return TInt
 OEqual -> do
 binaryInt args
 return TBool
 OLessThan -> do
 binaryInt args
 return TBool
 OIf -> do
 if length args /= 3 then Left "Expecting 3 arguments" else Right ()
 unaryBool [head args]
 t1 <- typeCheck (args !! 1)
 t2 <- typeCheck (args !! 2)
 if t1 == t2
 then Right t2
 else Left "Opérateur If must have the same type in both branches"

```

```

 ONot -> do
 unaryBool args
 return TBool

where binaryInt (e1 : e2 : []) = do
 t <- typeCheck e1
 t2 <- typeCheck e2
 if t == TInt && t2 == TInt
 then return ()
 else Left "Expecting integers"
 binaryInt _ = Left "Expecting 2 arguments"

 unaryBool (e1 : []) = do
 t <- typeCheck e1
 if t == TBool
 then return ()
 else Left "Expecting integers"
 unaryBool _ = Left "Expecting 1 argument1"

-- Fonction eval comme si le code était bien écrit
eval :: Exp -> Value
eval (EInt x) = VInt x
eval (EBool x) = (VBool x)
eval (EApp (EOp op : args)) =
 case op of
 OPlus -> binaryInt (+) args
 OMinus -> binaryInt(-) args
 OEqual -> binaryBool (==) args
 OLessThan -> binaryBool (<=) args
 OIf -> case args of
 [e1, e2, e3] ->
 let (VBool r1) = eval e1
 in if r1 then eval e2 else eval e3
 ONot -> let (VBool r1) = oneArg args
 in (VBool $ not r1)

where
 oneArg :: [Exp] -> Value
 oneArg (e1 : []) = eval e1

 twoArgs :: [Exp] -> (Value, Value)
 twoArgs (e1 : e2 : []) =
 let v1 = eval e1
 v2 = eval e2

```

```

 in (v1 , v2)

binaryInt :: (Int -> Int -> Int) -> [Exp] -> Value
binaryInt op args =
 let (VInt r1, VInt r2) = twoArgs args
 in VInt (r1 `op` r2)

binaryBool :: (Int -> Int -> Bool) -> [Exp] -> Value
binaryBool op args =
 let (VInt r1, VInt r2) = twoArgs args
 in VBool (r1 `op` r2)

```

## Solution de l'exercice 10.2

```

#include <stdio.h>
#include <stdlib.h>

typedef struct list list;
struct list {
 int refcount; // 4 bytes
 // 4 bytes
 void *head; // 8 bytes
 list *tail; // 8 bytes
};

// list_cons
list* list_cons (void *head, list *tail) {
 list *newlst = malloc(sizeof(list));
 if (newlst != NULL) {
 newlst->refcount = 1;
 newlst->head = head;
 newlst->tail = tail;
 }
 return newlst;
}

// list_head
void* list_head(list *l) {
 return l != NULL ? l->head : NULL;
}

// list_tail
list* list_tail(list *l) {
 list *cdr = l != NULL ? l->tail : NULL;
}

```

```

 if(cdr) {
 cdr->refcount++;
 }
 return cdr;
}

// Copie un pointeur (pas la list elle même)
list* list_copy (list *l) {
 if (l != NULL) {
 l->refcount++;
 }
 return l;
}

// list_free
void list_free (list *l) {
 list *next;
 while(l && l->refcount == 1) {
 next = l->tail;
 free(l);
 l = next;
 }
 if (l != NULL) {
 l->refcount --;
 }
}

void list_map (void (*f)(list*), list *l) {
 while(l) {
 f(l);
 l = l->tail;
 }
}

// Exemple utilisation.
void add2(list* l) {
 int *i = (int*) l->head;
 *i = *i + 2;
}

void showElem(list* l) {
 int *i = (int*) l->head;
 printf(" %d", *i);
}

```

```

int main(int argc, char *argv) {
 int a = 1, b = 2;
 list *lst = list_cons(&a, list_cons(&b, NULL)); // 1 : 2 : []
 list *cp = list_copy(lst);
 printf("list before map :"); list_map(&showElem, lst);
 list_map(&add2, lst);
 printf("list after map :"); list_map(&showElem, lst);
 putchar('\n');
 list_free(lst);
 list_free(cp);
 return 0;
}

// Copy incrémente de 1 le compteur et free décrémente de 1 le compteur
// Lorsque le compteur arrive à 0, on libère la tête de liste

// Il faut absolument utiliser list_copy pour copier une liste
// et non pas une déclaration du type list newlist = old;
// Le premier élément doit être créé avec NULL pour cdr.

// Liste de liste : Le compteur des listes dans la liste ne sont pas incrémentés
// si on copie la liste

```

### Solution de l'exercice 10.3

```

// Gestion mémoire manuel
#include <stdio.h>
#include <stdlib.h>

typedef struct list_elem list_elem;
struct list_elem {
 void *value;
 list_elem *next;
};

typedef struct list list;
struct list { list_elem *head; };

list* list_alloc () {
 list *newlst = malloc(sizeof(list));
 if (newlst) { /* newlst != NULL */
 newlst->head = NULL;
 }
}

```

```

 return newlst;
}

void list_insert (list *l, void *v) {
 if(l) {
 list_elem *elem = malloc(sizeof(list_elem));
 if (elem) { // elem != NULL
 elem->next = l->head;
 l->head = elem;
 elem->value = v;
 }
 }
}

void *list_get (list *l, int n) {
 if(!l) return NULL;
 list_elem *elem = l->head;
 while(elem && n > 0) { // elem != NULL
 elem = elem->next;
 n--;
 }
 return n == 0 && elem != NULL ? elem->value : NULL;
}

void* list_remove (list *l) {
 void *elem = NULL;
 if (l != NULL) {
 list_elem *head = l->head;
 elem = head->value;
 l->head = head != NULL ? head->next : NULL;
 free(head);
 }
 return elem;
}

void list_free(list *l) {
 if(l) {
 while(l->head) {
 list_remove(l);
 }
 free(l);
 }
}

// Exemple utilisation

```

```

int main(int argc, char **argv) {
 list *lst = list_alloc();
 const int size = 5;
 for(long i = 0; i < size; ++i) {
 list_insert(lst, (void*)i);
 }
 int i = 0;
 while(i < size) {
 printf("list_get(lst, %i) : %ld\n", i, (long)list_get(lst, i));
 i++;
 }
 list_free(lst);
 return 0;
}

```