TD Programmation Fonctionnelle
Année universitaire 96-97.

Exercice 1.1 Pourquoi les deux commandes suivantes ne fournissent-elles pas nécessairement le même résultat ?

```
#let x = 3
 in let y = x + 1
    in x + y;;

#let x = 3 and y = x + 1 in x + y;;
```

Exercice 1.2 Expliquer le comprtement suivant.

```
#let x=3;;
x : int = 3
#let f=function y->y+x;;
f : int -> int = <fun>
#f 2;;
- : int = 5
#let x=0;;
x : int = 0
#f 2;;
- : int = 5
```

Exercice 1.3 Evaluer l'expression

```
# (function x -> x+1) 3;;
```

Donner le type de `F` et `G` définies comme suit :

```
#let F = function f -> 2*f(function x->x+1);;
#let G = function f -> (function x -> f(function y -> x+y));;
```

Expliquer comment le type de cette fonctionnelle `G` a été inféré par le compilateur. Que vaut l'expression `G(function f->f(f 5))` ? Comment est calculée cette valeur ?

Exercice 1.4 Écrire une fonction qui renvoie le maximum de deux entiers. En donner une version prenant un couple en argument et une version curryfiée. Quel est le type de ces fonctions ?

Exercice 2.1 Definir la fonction $n \mapsto 2^n$ en utilisant seulement l'addition et en faisant en sorte que le nombre d'additions effectuées soit linéaire.

Exercice 2.2 La suite de Fibonnacci est une suite de mots sur l'alphabet $\{a, b\}$ définie par récurrence par

$$\begin{aligned} f_0 &= b, \\ f_1 &= a, \\ f_{n+2} &= f_{n+1} f_n \quad \text{pour } n \geqslant 2 \end{aligned}$$

Définir la suite de Fibonnacci de plusieurs facons differentes. En donner une version qui applique un nombre linéaire de fois l'opérateur de concaténation sur les chaînes.

Exercice 2.3 Définir une fonction `Fibonnacci` qui calcule le terme d'ordre `n` de la suite de Fibonnacci usuelle. Pourquoi la séquence suivante, où l'on essaie de définir la suite directement ne donne-t-elle pas le résultat ? Que produit-elle ?

```
# let Fibonnacci n =
let rec H n =
if n=0 then (1,1) else (x+y,x) where (x,y) = H (n-1);;
in fst (H n);;
```

Exercice 2.4 Écrire et tester une fonction `pgcd` naïve, de type `int * int -> int`, puis une fonction `pgcd` qui utilise la fonction `mod` de Caml. Écrire ensuite une fonction qui calcule le pgcd de son argument et de 60.

Exercice 2.5 Écrire une fonction d'ordre supérieur `dicho` telle que `dicho (f,a,b,epsilon)` retourne une paire `(u,v)` telle que l'intervalle `[u,v]` est de longueur inférieure à `epsilon` et contient un zéro de f (sous l'hypothèse que `f` est de type `float -> float`, continue, monotone, et $\mathtt{f(a)f(b)} < 0$).

Exercice 2.6 Écrire une fonction `root` telle que `root (a, ε, essai)` retourne une approximation `e` de la racine carrée de `a` vérifiant $|e^2 - a| < \varepsilon$. On utilisera la suite définie par récurrence : $e_{n+1} = \frac{1}{2}(e_n + \frac{a}{e_n})$.

Exercice 2.7 Ecrire la fonction `deriv` définie par : $(\mathtt{deriv(f, dx)})(\mathtt{x}) = \frac{\mathtt{f(x+dx)-f(x)}}{\mathtt{dx}}$.

Utiliser `deriv` pour chercher les zéros d'une fonction `f` selon la méthode de Newton. On supposera que `f` a un unique zéro dans l'intervalle $[\mathtt{a}, \mathtt{b}]$, que $\mathtt{f(a)f(b)} < 0$, et que `f` possède des dérivées première et seconde de signes constants). La fonction `zero` prendra en argument (f,dx,epsilon,essai) et on utilisera la suite $\mathtt{e_{n+1}} = \mathtt{e_n} - \frac{\mathtt{f(e_n)}}{\mathtt{f'(e_n)}}$ où $\mathtt{f'} = \mathtt{deriv(f, dx)}$.

Exercice 2.8 Écrire une fonction de sommation qui prend en argument une fonction `f` et un entier `n` et renvoie $\sum_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{f(i)}$.

Exercice 2.9 Définir un opérateur infixe `o` qui prend en argument deux fonctions `f` et `g` que l'on peut composer et renvoie la fonction qui a `x` associe `f(g(x))`.

Définir une fonction prenant en argument une fonction `f` et un entier `n` et qui renvoie l'itérée `n` fois de `f`.

Exercice 2.10 Écrire une fonction `curry` de curryfication. Quel est son type ? Même question pour une fonction `uncurry`. Réécrire la fonction qui calcule le pgcd de son argument et de 60.

Exercice 3.1 Pour commencer... Donner rapidement les types des expressions suivantes, s'ils sont légaux.

```
fun x -> x x;;
fun f x ->
if (fst (f (fst x)))=(snd x)
then (snd (f (fst x)))
else (snd (f (fst x)));;
function f -> fun g h -> function x -> h f ((g x+1),true);;
```

Exercice 3.2 Donner le type des fonctions suivantes.

```
let F = function
0 -> (function f -> f 0)
| x -> (function f -> 1 + f (x-1));;

let G f x = (F x) f;;

let rec H n = match n with
0 -> (function x -> 0)
| x -> G (H (x-1));;

let I (x,y) = H x y;;
```

Que fait la fonction `I` ?

Exercice 3.3 Définir une fonction `boucle(p,f,x)` où `p` est un prédicat, `f` une fonction s'appliquant à `x`, et qui renvoie la première valeur $\texttt{f}^{\texttt{n}}(\texttt{x})$ satisfaisant le prédicat. Quel est le type de `boucle` ? En déduire une fonction qui prend en argument deux entiers `x` et `n` et qui renvoie la plus petite puissance de `x` supérieure à `n`.

Exercice 3.4 Définir un type `complexe` permettant de calculer en coordonnées cartésiennes ou polaires. Écrire des fonctions retournant abscisse et ordonnée d'un point d'affixe passé en argument. Écrire une fonction `rotation` qui prend un `angle` en argument et retourne une fonction qui à tout point associe son image par la rotation. Écrire de même une fonction `translation` prenant un `complexe` en argument et une fonction `homothétie`. Définir un type matrice et une application qui donne la matrice associée à une homothétie ou à une rotation.

Exercice 3.5 Définir un type abstrait `Entier` pour les entiers relatifs. Définir ensuite l'addition `addition_Entier` de ce type. On écrira une fonction de conversion vers le type `int`, mais on ne s'en servira pas dans le reste de l'exercice. Définir une fonction dse comparaison de deux entiers de type `Entier`. On pourra définir une fonction de simplification `simplifie` qui prend en argument un `Entier` et renvoie un `Entier` représenté sans utiliser le constructeur `successeur` s'il est négatif et sans le constructeur `prédecesseur` s'il est positif.
Question subsidiaire : Définir un type entier de Gauss représentant les éléments de $\mathbb{Z}[i]$ et les opérations associées, y compris une division euclidienne.

Exercice 4.1 Programmer en Caml Light les fonctions suivantes sur les listes :

- `length` : calcule la longueur d'une liste. Exemple : `length [1;2;3] = 3`.
- `append` : concaténe deux listes. Exemple : `append [1;2;3] [4;5] = [1;2;3;4;5]`.
- `reverse` : inverse l'ordre des éléments d'une liste. Exemple : `reverse [1;2;3] = [3;2;1]`.
- `sigma` : calcule la somme d'une liste d'entiers. Exemple : `sigma [1;2;3] = 6`.
- `accumuler` : Exemple : `accumuler add_int 0 = sigma`.
- `map` : la fonction telle que `map f [a1 ;...;an] = [f (a1);...;f (an)]`.
- `flat` : la fonction «fusionnant» un niveau de liste. Exemple : `flat [[2];[];[3;4;5]] = [2;3;4;5]`.

Exercice 4.2 Les ensembles représentés par des listes. On considère un type `'a` tel que l'ensemble des valeurs de type `'a` est muni d'un ordre total, implémenté par une fonction `inf : 'a * 'a -> bool`.

- Écrire une fonction `insert` qui insère un élément `x` à la bonne place dans une `'a list` supposée triée par ordre strictement croissant (selon `inf`). Si `x` est déjà dans la liste, ne pas l'insérer.
- Écrire une fonction sort qui trie une `'a list` quelconque par ordre croissant selon `inf`, sans laisser d'éléments redondants.
- Écrire une fonction `member` qui détermine si l'élément `x` est dans la liste `l` , d'abord dans le cas où la liste n'est pas triée, puis dans le cas où elle l'est.
- Écrire de même les opérations d'union, intersection, et différence symétrique (`union, inter, delta`) de deux listes triées.

Exercice 5.1 Codage de Huffman. Le codage de Huffman peut être utilisé pour compresser une suite d'éléments (par exemple un texte, qui est une suite de caractères) en remplaçant chaque élément par son code, qui est une séquence de valeurs booléennes.

- Déclarer le type `direction` contenant les constantes `Gauche` et `Droite`.
- Déclarer le type `'a arbre` des arbres binaires non vides, dont les feuilles (`Leaf`) contiennent des éléments de type `'a`. Les nœuds internes (`Node`) ne contiendront aucune valeur.

Étant donné un `'a arbre` dont toutes les feuilles sont différentes, on peut coder les éléments de type `'a` qu'il contient de la manière suivante : le chemin qui va de la racine de l'arbre à une feuille peut être représenté par la liste des choix *sous-arbre de gauche* ou *sous-arbre de droite* effectués à chaque noeud interne. Ceci définit une injection de l'ensemble des feuilles de l'arbre dans l'ensemble des `direction list`. Appelons *ensemble des codes valides* l'image de cette injection.

- Écrire une fonction `decode` qui, étant donnés un `'a arbre` et une `direction list L` qui commence par un code valide, retourne le couple formé de l'élément codé et du reste de la liste (le suffixe de `L` qui n'a pas été utilisé pour trouver l'élément).
- Écrire une fonction `decode_list` qui, étant donnés un `'a arbre` et une `direction list` obtenue par concaténation de codes corrects, retourne la liste des éléments de type `'a` qu'elle code.

Le codage de Huffman est efficace si l'arbre de départ est bien choisi : il faut placer les éléments souvent rencontrés le plus proches possible de la racine. Supposons qu'on dispose d'une `('a * int) list` de la forme `[(elem`$_1$`,poids`$_1$`);...;(elem`$_n$`,poids`$_n$`)]`, où les `elem`$_i$ sont les éléments qui doivent apparaître dans l'arbre de Huffman, et les `poids`$_i$ représentent leur fréquence probable dans les textes à compresser. On contstruit un arbre de Huffman optimal contenant les `elem`$_i$.

- Commencer par écrire une fonction `coerce` qui prend une `('a * int) list` en argument et retourne la `('a arbre * int) list` où chaque élément `elem`$_i$ est remplacé par l'arbre `Leaf (elem`$_i$`)` et où les couples sont triées par poids croissants. L'algorithme suivant permet de construire un arbre optimal :
  - On choisit dans la liste deux arbres `a1` et `a2` de poids `p1` et `p2` minimaux.
  - On construit l'arbre `A` ayant `a1` pour fils gauche et `a2` pour fils droit.
  - On lui donne le poids `p1 + p2`, et on considère alors la liste de départ, moins les couples `(a1,p1)` et `(a2,p2)`, plus `(A, p1+p2)`. Cette nouvelle liste contient un élément de moins que la précédente. On applique à nouveau la transformation jusqu'à obtenir une liste réduite à un seul couple. Elle contient l'arbre optimal.
- Programmer une fonction `make_huff` qui prend une `('a * int) list` en argument, et qui retourne un arbre de Huffman optimal pour cette liste.
- Écrire une fonction qui prend un arbre de Huffman en argument, et qui retourne la fonction de codage associée. (C'est-à-dire la fonction qui prend un argument de type `'a` et renvoie la `direction list` qui mène à cet élément dans l'arbre donné).

Cet exemple est inspiré d'une compétition organisée par l'armée américaine opposant des programmeurs expérimentés utilisant des langages divers. Les concurrents utilisant le seul langage fonctionnel présenté (Haskell) remportèrent facilement l'épreuve et l'élégance de leur solution impressionna le jury.

Le but du programme est de visualiser des avions (mobiles) se déplaçant dans un environnement contenant des zones jugées sensibles. L'affichage montrera les zones traversées et signalera la trop grande proximité de deux avions. Pour simplifier le problème, on se place dans un univers à deux dimensions.

Pour la mise au point, on utilisera un toplevel interactif contenant le module `graphics`. On lancera Caml par la commande `camllight camlgraph`. On ouvrira ce module par

```
open graphics
```

et on initialisera l'écran par

```
open_graph "";;
```

Si vous avez un problème à ce niveau, vérifiez la valeur de votre variable `DISPLAY`.

1 Déplacements et mobiles

- Commencez par la définition d'un point du plan (type `point`).
- Définissez le type `mobile`. Un mobile est défini par la donnée de la position courante et d'un déplacement, un déplacement est une fonction qui à partir de la position courante rend le mobile suivant.
- Définir la fonction `translate: int -> int -> point -> point`.
- Définir le déplacement qui correspond à un point immobile `immobile: déplacement`.
- Définir le déplacement qui correspond à un mouvement uniforme défini par deux incréments `dx` et `dy` : `uniforme: int -> int -> déplacement`.
- Définir un déplacement pseudo-circulaire défini par le centre du cercle, l'angle de rotation, et un coefficient multiplicatif destiné à corriger l'approximation entière (le mobile décrit en fait une spirale) :

  ```
  pseudo_circulaire : point -> float -> float -> d\'eplacement
  ```
- Définir un déplacement uniforme sur un nombre de pas `n` défini à l'avance. Ce déplacement sera suivi d'un autre déplacement `cont` que l'on passera en argument :

  ```
  avance: int (* dx *) -> int (* dy *) -> int (* n *)
                                       -> deplacement (* cont *) -> deplacement
  ```
- Définir une fonctionnelle `compose` sur les déplacements qui prend un entier `n` et deux déplacements en arguments, effectue le premier durant `n` pas puis effectue le second. Redéfinir le déplacement précédent à partir de cette fonction et de `uniforme`.
- Définir une fonction affichant un point `affiche: point -> unit`.
- Définir une fonction qui étant donné un mobile, affiche la position courante et calcule le mobile suivant (la séquence s'écrit `e1; e2`).
- Généraliser cette fonction à une liste de mobiles.
- Écrire une fonction `play` itérant la fonction précédente `n` fois. On procédera à l'itération suivante :

  ```
  for i = 0 to 10000 do () done
  ```

pour ralentir l'affichage. On peut aussi utiliser `Unix.sleep 1` mais c'est un peu lent.

- Tester avec les deux mobiles suivants :

Mobile 1 :
- position initiale (10,10),
- avancer 20 fois de `dx` $= 5$, `dy` $= 0$,
- avancer 75 fois de `dx` $= 1$, `dy` $= 8$,
- rester immobile.

Mobile 2 :
- position initiale (500,300),
- avancer 10 fois de `dx` $= -5$, `dy` $= 0$,
- avancer indéfiniment de `dx` $= -6$, `dy` $= -3$

Mobile 3 :
- position initiale (270,350),
- avancer 15 fois de `dx` $= 4$, `dy` $= 0$,
- tourner indéfiniment autour de la position initiale, par angle de $0,1$ à $0,7$ radians, avec un coefficient multiplicatif de $1,01$ à $1,03$, au choix.

2 Interaction avec des zones

Les zones sont définies abstraitement par une fonction d'affichage, une fonction testant si un point est à l'intérieur de la zone, et un état dont l'utilisation est décrite plus tard. C'est un booléen qui vaudra initialement `true`. Écrire le type correspondant. Coder des formes simple :

- le cercle défini par son centre et son rayon,
- le rectangle défini par son point inférieur gauche et son coin supérieur droit,
- l'union de deux formes.

L'état servira à coder la présence d'un objet dans la forme. Écrire la fonction

- qui teste si un ensemble de mobiles est dans une forme
- qui affiche le contour d'une forme en rouge si l'état vaut `true` et en noir sinon.
- qui étant données une liste de mobiles et une forme rend une forme avec l'état modifié et réaffiche celle-ci si celui-ci a changé : `modif_forme: mobile_list -> forme -> forme`

3 Tests de proximité

Écrire une fonction `trop_près: int -> mobile -> mobile -> bool` qui vérifie si deux mobiles sont à une distance inférieure à la valeur donnée en premier argument.

Écrire une fonction qui, étant données une distance `d` et une liste de mobiles calcule la liste des paires de mobiles de la liste qui se trouvent à une distance inférieure à `d` l'un de l'autre.

Enfin, écrire une fonction qui étant donnés deux mobiles affiche un trait bleu entre leurs positions courantes.

4 Tout réunir

Écrire une nouvelle version de `play` qui prend en argument un nombre d'itétations, une liste de mobiles et une liste de formes, qui affiche les déplacements des mobiles et les formes dans l'état correspondant à chaque itération. Cette fonction devra aussi afficher un trait de liaison entre deux mobiles quand ceux-ci seront trop près l'un de l'autre (on définira la distance de danger comme étant 150 pixels). Noter que l'on ne cherchera pas à effacer les traits des positions précédentes.

Tester cette fonction avec les mobiles précédemment définis, le mobile immobile à la position $(40, 300)$, et les formes suivantes :

- un cercle centré en $(60, 20)$ et de rayon 15,
- un carré défini par les points $(100, 100)$ et $(200, 200)$.

Dans tout le TD, on note $\mathrm{VL}(M)$ l'ensemble des variables libres d'un λ-terme M.

1 Préliminaires

Exercice 7.1 Pour chaque occurrence de variable dans les λ-termes suivants, dire si elle est libre ou liée.

$$y(zt), \qquad \lambda x \cdot (x\lambda y \cdot (xy)), \qquad (\lambda x \cdot (\lambda y \cdot yx)z)t,$$
$$(\lambda x \cdot (\lambda y \cdot yx)z)ty, \qquad \lambda z \cdot \lambda x \cdot \lambda y \cdot (zxy), \qquad (\lambda y \cdot yx(\lambda x \cdot y(\lambda y \cdot z)x))zt$$

Quelles sont les variables libres dans ces λ-termes ? Quelle est la longueur de ces λ-termes ? Le λ-terme $y(zt)$ a-t-il une occurrence dans le λ-terme $xy(zt)$?

Exercice 7.2 Définir un type Caml pour représenter les termes du λ-calcul pur suivant la syntaxe concrète. Programmer une fonction `longueur` qui rend la longueur d'un terme.

Exercice 7.3 Écrire une fonction prenant en argument un nom de variable (sous forme de chaîne de caractères) et un terme et qui renvoie un booléen testant si la variable apparaît libre dans le terme.

Exercice 7.4 Un λ-terme de la forme $\lambda y \cdot \lambda x \cdot (y(y(\cdots(yx))))$ est appelé *entier de Church*. Écrire une fonction qui teste si un λ-terme représente un entier de Church.

2 Notation de de Bruijn

On considère dans cette section des termes n'ayant pas de variable libre (termes fermés ou clos). De Bruijn a proposé une notation abstraite pour les λ-termes, ne faisant pas intervenir les noms des variables. Chaque occurrence de variable liée est représentée par un entier. Cet entier est la profondeur à laquelle se trouve l'occurrence de la variable par rapport à son lieur (le λ qui lui est associé).

Exercice 7.5 Donner les termes en notation de de Bruijn pour les termes fermés de l'exercice 7.1. Proposer un type Caml pour représenter les termes en syntaxe abstraite.

Exercice 7.6 Sans utiliser l'exercice 7.3, et en travaillant sur la syntaxe abstraite, proposer une fonction testant si son argument représente un λ-terme fermé.

Exercice 7.7 Définir une fonction qui prend en argument un λ-terme fermé en syntaxe concrète et qui rend ce terme en syntaxe abstraite.

3 η-réduction

Soit M un terme et x une variable telle que $x \notin \mathrm{VL}(M)$. Si $N = \lambda x \cdot (Mx)$, alors lorsqu'on applique N à un argument P, on obtient après une β-réduction le terme MP comme si on avait appliqué directement M à P. Ainsi, $\lambda x \cdot (Mx)$ et M se comportant similairement vis-à-vis de l'application. Ceci motive la considération d'une règle supplémentaire de réduction, la *η-réduction*, qui réduit $\lambda x \cdot (Mx)$ en M :

On appelle *η-redex* tout terme de la forme $\lambda x \cdot (Mx)$ tel que $x \notin \mathrm{VL}(M)$ On appelle *η-règle* la règle qui dans un terme donné remplace un η-redex $\lambda x \cdot (Mx)$ par sa forme réduite M. On appelle *η-normal* un terme qui n'a pas de η-redex.

Exercice 7.8 Le terme $\lambda y \cdot \lambda x \cdot (yx)$ est-il η-normal? En donner le cas échéant une forme normale. Même question pour $\lambda y \cdot \lambda x \cdot (xyx)$, puis pour $\lambda b \cdot \lambda x \cdot \lambda y \cdot (bxy)$.

Exercice 7.9 Écrire une fonction `eta_normalise` qui réduit jusqu'à épuisement tous les η-redex d'un terme. On admettra que l'application de la η-règle termine toujours quelque soit l'ordre des réductions, et que la η-réduction est confluente.

1 Calcul sur les entiers de Church

Exercice 8.1 On veut montrer que l'on peut rerésenter certaines fonctions numériques définies sur les entiers à l'aide du λ-calcul. On reprend la définition d'un entier de Church de l'exercice 7.4 du TD précédent : on pose $c_n = \lambda f \cdot \lambda x \cdot \underbrace{(f(f(\cdots(fx))))}_{n \text{ occurrences de } f}$. On définit par ailleurs les termes :

$$\begin{aligned} s &= \lambda nfx \cdot f(nfx) \\ z &= \lambda nxy \cdot n(\mathbf{K}x)y \qquad \text{où } \mathbf{K} = \lambda xy \cdot x \\ a &= \lambda mnfx(mf)(nfx) \\ m &= \lambda mnf \cdot m(nf) \\ e &= \lambda mn \cdot nm \end{aligned}$$

En calculant sc_n, zc_n, ac_nc_m, mc_nc_m, et ec_nc_m, montrer que l'on peut calculer tester si un entier et nul, définir le successur, l'addition, la multiplication et la puissance d'un entier par un autre.

2 Réécriture

Exercice 8.2 Une relation de réécriture R sur un ensemble de termes T est donnée par un ensemble de couples $\{(M,N) \mid M,N \in T\}$. On écrit $M \longrightarrow N$ si $(M,N) \in R$, et on note $\xrightarrow{*}$ la fermeture réflexive transitive de $\longrightarrow$. On écrit $M \xrightarrow{=} N$ si $M \longrightarrow N$ ou si $M = N$. On note $\sim$ l'équivalence engendrée par R, c'est-à-dire la ferméture réflexive transitive symétrique de $\longrightarrow$. On dit que R

- est confluente si $M' \xleftarrow{*} M \xrightarrow{*} M''$ implique qu'il existe N tel que $M' \xrightarrow{*} N \xleftarrow{*} M''$.
- est fortement confluente si $M' \longleftarrow M \longrightarrow M''$ implique qu'il existe N tel que $M' \xrightarrow{=} N \xleftarrow{=} M''$.
- est Church-Rosser si $M' \sim M''$ implique qu'il existe N tel que $M' \xrightarrow{*} N \xleftarrow{*} M''$.
- a la propriété du parallélogramme si $M' \longleftarrow M \xrightarrow{*} M''$ implique qu'il existe N tel que $M' \xrightarrow{*} N \xleftarrow{*} M''$.
- est localement confluente si $M' \longleftarrow M \longrightarrow M''$ implique qu'il existe N tel que $M' \xrightarrow{*} N \xleftarrow{*} M''$.
- est nœthérienne s'il n'y a pas de suite infinie M_i telle que $M_i \longrightarrow M_{i+1}$.

Toute relation localement confluente et nœthérienne est confluente. Montrer que si la relation n'est plus nœthérienne, le résultat est faux. Donner un exemple pour lequel la relation n'a pas de cycle. Montrer :

- que toute relation fortement confluente est confluente ;
- que toute relation qui a la propriété du parallélogramme est confluente.
- qu'une relation est confluente si et seulement si elle est Church-Rosser.

Exercice 8.3 On se place sur un alphabet A contenant au moins 3 lettres. On considère la relation de réecriture $\longrightarrow$ sur le monoïde libre composée de toutes les règles de la forme $u \longrightarrow v$ telles qu'il existe deux factorisations $u = sx^2t$ et $v = sxt$.

On admettra qu'il existe un nombre fini de classes de la relation d'équivalence $\sim$, mais qu'il existe une infinité de mots réduits.

Montrer que cette relation est nœthérienne. En déduire qu'elle n'est pas localement confluente. Redémontrer ce fait à la main.

On peut montrer que la règle $\longleftarrow$ est confluente, en s'inspirant de la preuve du théorème de Church-Rosser.

Exercice 8.4 Montrer que la β-réduction n'est pas nœthérienne en considèrant le λ-terme $(\lambda x \cdot xxy)(\lambda x \cdot xxy)$. Quelles sont les suites de réécritures possibles à partir de ce terme par β-réduction ? Ce terme a-t-il une forme normale ?

On considère le terme $(\lambda z.t)((\lambda x \cdot xxy)(\lambda x \cdot xxy))$. Montrer que ce terme a une forme normale mais a aussi une réduction infinie.

1 Réécriture, stratégies d'évaluation

Exercice 9.1 On dit qu'une partie P d'un ensemble E muni d'une relation $\prec$ est *progressive* si elle contient un élément dès qu'elle contient ses prédécesseurs, c'est-à-dire si :

$$\forall x\ (\forall y\ (y \prec x \Longrightarrow y \in P)) \Longrightarrow x \in P$$

1. Montrer que si $\prec$ est bien fondée sur E et si $P \subseteq E$ est progressive, alors $P = E$.
2. En déduire qu'une relation $\longrightarrow$ nœthérienne et localement confluente est confluente. On pourra utiliser la relation $\prec$ définie par $x \prec y$ si et seulement si $y \xrightarrow{+} x$.

Exercice 9.2 Soit la fonction `if_then_else` définie comme suit :

```
let if_then_else c e1 e2 = if c then e1 else e2;;
```

L'expression `if_then_else c e1 e2` est-elle équivalente à `if c then e1 else e2`? Expliquer ce qui se passe sur l'exemple suivant.

```
type Num = Infini | Num of int;;
let Num_div x y = if_then_else (y=0) Infini (Num (x/y));;
Num_div 1 0;;
```

Exercice 9.3 On admet que les réductions de tête commutent avec les réductions internes. Montrer que Si un λ-terme M se réécrit par β-réduction en N, alors, il existe une réécriture de M en N comportant une suite de réductions de tête suivie d'une suite de réductions internes. En déduire que la stratégie de réduction à gauche est normalisante.

Exercice 9.4 L'implémentation actuelle de Caml évalue les composantes des n-uplets toujours dans le même ordre. Écrire une expression pour déterminer lequel. L'expression devra s'évaluer en une chaîne de caractères, `"Gauche -> droite"` ou `"Droite -> gauche"` suivant l'ordre d'évaluation.

2 Preuves de programmes

Exercice 9.5 On définit `inserer` par

```
let rec inserer x = function
[]     -> [x]
| (h::t) -> if x <= h then x::h::t else h::(inserer x t);;
```

Montrer qu'on définit ainsi une fonction totale, et que si `l` est triée, alors `inserer x l` aussi. Combien de fois un entier donné a-t-il d'occurrences dans `inserer x l`?

Exercice 9.6 On considère la fonction f suivante :

```
let rec f = function
0,y -> y+1
| x,0 -> f (x-1,1)
| x,y -> f (x-1, f (x,y-1));;
```

1. Montrer que cette fonction est définie pour tout couple $(x, y) \in \mathbb{N}^2$ et qu'elle prend des valeurs entières strictement positives.
2. Montrer que $f(1, y) = y + 2$, que $f(2, y) = 2y + 3$ et que $f(3, y) = 2^{y+3} - 3$.
3. On définit f_1 et f_2 par

```
let f1 x y = f(x,y) and f2 x y = f(y,x);;
```

   Montrer que pour tout $x \in \mathbb{N}$, les fonctions $f_1(x)$ et $f_2(x)$ sont croissantes de $\mathbb{N}$ dans $\mathbb{N}^*$.

1 Références

Exercice 10.1 Évaluer la dernière expression des trois programmes suivants. Expliquer les résultats obtenus.

1er programme

```
let a = ref 0;;
let b = ref(!a);;
a := 1;;
(!a,!b);;
```

2ème programme

```
let a = ref 0;;
let b = a;;
a := 1;;
(!a,!b);;
```

3ème programme

```
type t = {mutable x: int;};;
let a = {x = 0;};;
let b = a;;
a.x<-1;;
(a.x,b.x);;
```

Exercice 10.2 Même question pour :

1er programme

```
let inc =
let num = ref 0 in
num := !num + 1;
!num;;
inc;;
inc;;
```

2ème programme

```
let inc =
let num = ref 0 in
(function () -> num := !num + 1;
!num);;
inc();;
inc();;
```

Exercice 10.3 Donner l'exemple d'une expression e à laquelle on peut appliquer un argument et qui n'est pas équivalente à `(function x ->` e `x)`. À rapprocher de l'exercice sur la η-réduction du TD 7.

Exercice 10.4 Définir un type `'a arbre` d'arbres binaires dont les feuilles sont étiquetées sur le type `'a`, en utilisant des références. Écrire une fonction qui compare si deux arbres ont même frontière (sans calculer *toute* la frontière des deux arbres mais juste le minimum !).

2 Pour se reposer...

Exercice 10.5 Soit T le λ-terme suivant :

$$T = \lambda abcdefghijklmnopqrstuvwyzx \cdot x(jesuisunoperateurdepointfix)$$

et $C = TTTTTTTTTTTTTTTTTTTTTTTTTT$. Montrer que C est un opérateur de point fixe.

Exercice 10.6 Définir une fonction de type `'a -> 'b -> 'c`.

Exercice 11.1 Définir un type `'a arbre` d'arbres binaires dont les feuilles sont étiquetées sur le type `'a`. Écrire une fonction de type `'a arbre -> 'a list` qui calcule la frontière d'un arbre, c'est-à-dire la liste des éléments qui étiquettent les feuilles rencontrées lorsqu'on parcourt l'arbre en profondeur gauche. En déduire une fonction qui teste si deux arbres ont même frontière.

Réécrire cette dernière fonction sans calculer *toute* la frontière des deux arbres mais juste ce qui est nécessaire.

Exercice 11.2 Soit T le λ-terme suivant :

$$T = \lambda abcdef \cdot f(abadfef)$$

Montrer que $C = TTTTTT$ est un opérateur de point fixe, c'est-à-dire que pour tout λ-terme M, on a $CM \xrightarrow{\beta *} M(CM)$.

Exercice 11.3 On rappelle que l'on peut donner une traduction `T` d'un programme Caml en λ-calcul non typé de la manière suivante :

- On commence par transformer toutes les déclarations `let f = e1;;R`, où `R` est le reste du programme, en `let f = e1 in R`. Si `R` est vide, c'est-à-dire si la déclaration est la dernière instruction du programme, on la transforme en `let f = e1 in f`.
- On donne alors la règle de transformation `T` de la construction `let ...in...` :

$$T(\texttt{let x = e1 in e2}) = (\lambda x \cdot T(e2))T(e1)$$

Donner la traduction de

```
#let x = 1;;
#let z = 8;;
#let val = let x = 3 in let  y = x + z in x + y;;
```

Comment s'évalue cette dernière expression ?

Problème Soit X l'alphabet (infini) $\{x_0, x_1, x_2, \ldots\}$. On veut écrire une fonction qui prend en argument un λ-terme dont aucune variable libre n'est dans X, et qui renomme tous les lieurs et les occurrences liées correspondantes par des symboles de X. Cette fonction sera ensuite utilisée pour implémenter la β-réduction.

Au cours du renommage, la lettre x_i ne sera utilisée que si toutes les lettres précédentes $x_0, \ldots, x_{i-1}$ le sont aussi. Les occurrences libres ne seront pas renommées. Par exemple, le λ-terme $(\lambda x \cdot (\lambda y \cdot yx)z)ty$ pourra être renommé en $(\lambda x_0 \cdot (\lambda x_1 \cdot x_1 x_0)z)ty$, ou bien aussi en $(\lambda x_1 \cdot (\lambda x_0 \cdot x_0 x_1)z)ty$.

- Définir un type `terme` pour représenter les λ-termes en syntaxe concrète. Les variables seront représentées par des chaînes (type `string`).
- Écrire une fonction `print_term` de type `terme -> unit` permettant d'afficher un terme (afficher `L` pour λ et bien expliciter les parenthèses).
- Définir une fonction `assoc` de type `'a -> ('a * 'b) list -> 'b`, telle que
  - `assoc x l` déclenche l'exception `Not_found` si `l` ne contient pas d'élément dont la première composante est `x`,

- `assoc x [...;(x,y);...]` s'évalue en `y` si `(x,y)` est le couple le plus à gauche de la liste ayant comme première composante `x`.

– Définir une fonction `renomme` de type `terme -> terme` qui effectue le renommage. On ne demande pas de vérifier que les variables libres du λ-terme argument ne sont pas dans X, ceci est supposé vrai. On pourra utiliser :
  - une référence sur un entier correctement initialisé pour générer les noms de variables `x0`, `x1`, etc.
  - une fonction auxiliaire de type `(string * string) list -> terme -> terme`. Son premier argument est un environnement permettant de retenir la correspondance entre ancien et nouveau nom de variable, c'est-à-dire une liste de couples qui sont de la forme `(ancien_nom,nouveau_nom)`.

  Est-il assuré que pour tout `t`, les indices des variables liées de `renomme t` apparaissent dans l'ordre croissant lorsqu'on lit le terme de gauche à droite ? Justifier. Sinon, modifier la fonction `renomme` pour qu'il en soit ainsi.
– On s'intéresse maintenant à la β-réduction. Écrire une fonction de type `terme -> bool` testant si un λ-terme est normal.
– Écrire une fonction `substitue` de type `terme -> string -> terme -> terme`, qui substitue dans le premier terme toutes les occurrences de la variable donnée en deuxième argument, qui seront supposées libres, par le terme donné en troisième argument.
– Écrire une fonction `beta_direct` de type `terme -> terme` déclenchant l'exception `Failure "Terme normal"` si son argument est normal, et telle que `beta_direct t` s'évalue en le terme obtenu par β-réduction du rédex le plus à gauche de `t` sinon.
– Écrire enfin une fonction `beta_n_pas` de type `terme -> int -> terme` qui effectue n pas successifs de β-réduction gauche.

Exercice 11.4 Soit X l'alphabet (infini) $\{x_0, x_1, x_2, \ldots\}$. On veut écrire une fonction qui prend en argument un λ-terme dont aucune variable libre n'est dans X, et qui renomme toutes les variables liées par des symboles de X. La lettre x_i ne sera utilisée que si toutes les lettres précédentes $x_0, \ldots, x_{i-1}$ sont aussi utilisées. Par exemple, le λ-terme $(\lambda x \cdot (\lambda y \cdot yx)z)ty$ pourra être renommé en $(\lambda x_0 \cdot (\lambda x_1 \cdot x_1 x_0)z)ty$, ou bien aussi en $(\lambda x_1 \cdot (\lambda x_0 \cdot x_0 x_1)z)ty$.

– Définir un type `terme` pour représenter les λ-termes en syntaxe concrète, et une fonction `print_term` de type `terme -> unit` permettant d'afficher un terme (afficher `L` au lieu de λ et bien expliciter les parenthèses).
– Définir une fonction `assoc` de type `'a -> ('a * 'b) list -> 'b`, telle que
  - `assoc x l` déclenche l'exception `Not_found` si `l` ne contient pas d'élément dont la première composante est `x`,
  - `assoc x [...;(x,y);...]` s'évalue en `y` si `(x,y)` est le couple le plus à gauche de la liste ayant comme première composante `x`.
– Définir une fonction `renomme` de type `terme -> terme` qui effectue le renommage. On ne demande pas de vérifier que les variables libres du λ-terme argument ne sont pas dans X, ceci est supposé vrai. On pourra utiliser :
  - une référence sur un entier correctement initialisé pour générer les noms de variables `x0`, `x1`, etc.
  - une fonction auxiliaire de type `(string * string) list -> terme -> terme`. Son premier argument est un environnement permettant de retenir la correspondance entre ancien et nouveau nom de variable, c'est-à-dire une liste de couples qui sont de la forme `(ancien_nom,nouveau_nom)`.

Est-il assuré que pour tout `t`, les indices des variables liées de `renomme t` apparaissent dans l'ordre croissant lorsqu'on lit le terme de gauche à droite? Justifier. Sinon, modifier la fonction `renomme` pour qu'il en soit ainsi.