L1, informatique 2^{nd} semestre

TD d'Informatique No. 1

RÉVISIONS

Exercice 1 : On considère la session ML ci-dessous ; indiquez la réponse que donnerait l'ordinateur pour chacune des commandes successives :

1. `50 + 2 * 2;;`
   .
2. `bool & true;;`
   .
3.
```
let nom = "gaston" and age = 22
    in if age < (string_length nom)
        then nom ^ " est un enfant."    else nom ^ " est un adulte.";;
```
   .
4.
```
let hypotenuse (x,y,z) = let max (a,b) = if a < b then b else a
                             in let d = max(max(x,y),z)
                                in if x*.x +. y*.y +. z*.z = 2.*.d*.d
                                    then d else failwith "pas rectangle!";;
```
   .
5. `hypotenuse (3.,4.,5.);;`
   .
6. `hypotenuse (4.,5.,6.);;`
   .
7. `hypotenuse (3,4,5);;`
   .
8. `let h x =  if  x <> 1.5  then  1. /. (x -. 1.5);;`
   .

Exercice 2 : Écrivez une fonction `divisible` qui prend en entrée deux nombres entiers relatifs `a` et `b` et retourne un booléen qui dit si `a` est divisible par `b`.
Donnez ensuite le type de votre fonction.

.

.

Exercice 3 : Écrivez une fonction récursive `teste` qui prend en entrée deux entiers naturels `p` et `n` et retourne un booléen qui dit s'il existe un naturel compris entre `2` et `p` qui divise `n`.
Indication : faire échouer la fonction si `p` ou `n` ne sont pas strictement positifs, sinon faire au plus simple en testant `p`, `p-1`, `p-2`, etc, jusqu'à 2 (et si `p`=1, `teste` retournera `false`).

.

.

.

.

Exercice 4 : Écrivez une fonction `premier` qui prend en entrée un entier naturel quelconque et dit s'il est premier.
Nota : gérer proprement les cas d'erreur ; ne pas chercher à donner un algorithme efficace.

.

.

.

Exercice 5 : On suppose maintenant que l'on dispose d'une carte pour un jeu de rôle. Cette carte représente un désert et elle est quadrillée par des coordonnées cartésiennes dont l'unité est le kilomètre. Des oasis sont repérées sur cette carte par leur nom et par leurs coordonnées cartésiennes. Pour cela on utilise le type suivant :

```
type oasis = { nom : string; x : float; y : float };;
```

Écrivez la fonction `distance` qui prend en entrée deux valeurs de type `oasis` et retourne la distance (en kilomètres) entre ces deux oasis. *Rappel :* `sqrt : float→float` calcule la racine carrée.

.

.

Exercice 6 : On représente un trajet sur cette carte comme une liste d'oasis ; l'oasis la plus à gauche de la liste étant le point de départ ; celle la plus à droite de la liste étant le point d'arrivée et les oasis intermédiaires étant les haltes effectuées pour se réapprovisionner en eau.
Écrivez les deux fonctions `depart` et `arrivee` qui prennent en argument un trajet (c'est-à-dire une liste d'oasis) et retournent respectivement le nom de l'oasis de départ et d'arrivée.

.

.

.

.

.

Exercice 7 : Écrivez une fonction récursive `longueur` qui prend en entrée un trajet (c'est-à-dire une liste d'oasis) et retourne le nombre total de kilomètres parcourus de l'oasis de départ à celui d'arrivée en passant par les haltes intermédiaires.

.

.

.

Exercice 8 : Un trajet sera dit `faisable` s'il n'y a jamais plus de 50 kilomètres entre deux haltes. Écrivez la fonction booléenne correspondante.

.

.

.

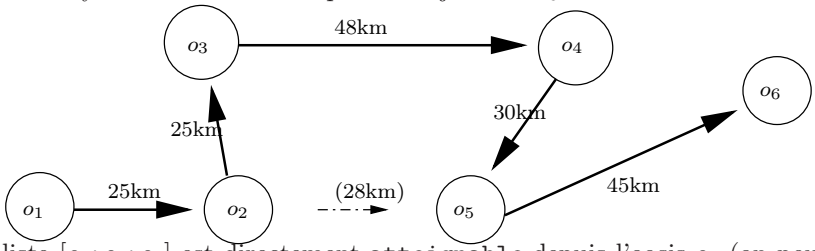
Exercice 9 : Écrivez une fonction booléenne `atteignable` qui prend en entrée une oasis `o` et une liste d'oasis `l`, et dit s'il existe une oasis dans la liste `l` qui se trouve à moins de 50 kilomètres de `o`.

.

.

Exercice 10 : Écrivez une fonction booléenne `raccourci` qui prend en entrée un trajet et telle que :
- elle échoue (avec `failwith`) si le trajet n'est pas faisable,
- sinon elle dit s'il est possible d'éviter certaines haltes en allant directement d'une oasis du trajet à une oasis ultérieure de ce trajet en évitant certaines oasis intermédiaires (mais en conservant un trajet faisable).

Ainsi `raccourci` doit renvoyer `true` dans l'exemple de trajet o_1 à o_6 suivant :



car, par exemple, la liste $[o_4; o_5; o_6]$ est directement `atteignable` depuis l'oasis o_2 (on peut éviter les haltes o_3 et o_4).

.

.

.

.

L1, informatique 2^{nd} semestre

TD d'Informatique No. 2

Exercice 1 : On considère la session ML ci-dessous ; indiquez la réponse que donnerait l'ordinateur pour chacune des commandes successives :

1. `type int = 4 ; ;`
2.
```
let il = "ça " and fait = "va " and beau = "pleuvoir" and optimiste = false
    in if optimiste then "il " ^ "fait " ^ "beau" else il ^ fait ^ beau ; ;
```
3.
```
let rec rectangles l =
          let ordre (a,b,c) = if a < b
                              then if b < c then(c,a,b) else (b,a,c)
                              else if a < c then(c,a,b) else (a,b,c)
          in match l with []   ->   true
             | x::y::z::r ->   let (u,v,w) = ordre(x,y,z)
                               in (u *. u = v *. v +. w *. w) && (rectangles r)
             |   l'    ->   failwith "Valeurs trois par trois SVP" ; ;
```
4. `rectangles [3. ;4. ;5. ;4. ;5. ;3.] ; ;`
5. `rectangles [2. ;3. ;4.] ; ;`
6. `rectangles [3. ;4.] ; ;`

.

Exercice 2 : Écrivez une fonction `val_absolue` qui prend en entrée un nombre réel `x` retourne sa valeur absolue. Donnez ensuite le type de votre fonction.

Exercice 3 : Écrivez une fonction `longueur_moyenne` qui prend en entrées deux chaînes de caractères `s` et `t`, et retourne un nombre entier qui est la moyenne des longueurs de `s` et `t`. Attention, cette fonction doit échouer si ces deux longueurs ne sont pas de même parité (i.e. si leur somme est impaire).

Exercice 4 : Écrivez une fonction récursive `carre` qui prend en entrée deux entiers relatifs `n` et `p` et retourne un booléen qui dit si `n` est le carré d'un entier supérieur ou égal à `p` (i.e. s'il existe un entier q supérieur ou égal à `p` tel que `n`$=q^2$). Attention, on prendra soin d'examiner tous les cas, y-compris si `n` est négatif, et de renvoyer un booléen correct à chaque fois. Si `p` est négatif en revanche, on fera échouer la fonction. INDICATION : on peut par exemple comparer `n` à `p*p` pour lancer un appel récursif éventuel.

Exercice 5 : Un voyage touristique est caractérisé par le nom de la ville de `depart`, celui de la ville de `destination`, le prix du `billet` d'avion aller-retour, le prix d'une `nuit` d'hotel et la `duree` du séjour en nombre de nuits passées à l'hotel.
Déclarez le type `voyage` qui permet de représenter ces informations.

Exercice 6 : En reprenant le type `voyage` de l'exercice précédent, écrivez une fonction `correct` qui prend en entrée une valeur de type `voyage` et retourne un booléen qui dit si cette valeur est correcte c'est-à-dire : les noms des villes de départ et de destination sont non vides, les prix du billet et de chaque nuit sont positifs, et le nombre de nuits est au moins égal à 1.

Exercice 7 : Toujours avec le même type `voyage`, écrivez une fonction `prix` qui calcule le prix d'un voyage comme la somme des coûts de l'avion et de l'hotel (le prix dépend donc du nombre de nuits d'hotel). On fera échouer la fonction si le voyage n'est pas correct. RAPPEL : `float_of_int` transforme un nombre entier en nombre réel.

Exercice 8 : Écrivez une fonction récursive `nb_positifs` qui prend en entrée une liste d'entiers relatifs et retourne le nombre d'entiers strictement positifs qu'elle contient.
Donnez le type de votre fonction.

Exercice 9 : Écrivez une fonction récursive `repetition` qui prend en entrée une liste quelconque et rend un booléen qui dit s'il existe deux éléments consécutifs dans la liste qui sont égaux.

Exercice 10 : Écrivez une fonction récursive `debute` qui prend en entrée deux listes d'éléments de même type, notées `l1` et `l2`, et qui renvoie un booléen : `true` si la liste `l2` commence par `l1`, et `false` sinon.
- Par exemple `debute(['A';'T';'G'],['A';'T';'G';'C';'G';'G'])` vaut `true`.
  `debute([],[1;2;3;4])` vaut également `true`, de même que `debute([6;6],[6;6;6;7;8])`.
- Par contre, `debute(['A';'T';'G'],['A';'G';'C';'G';'G'])` vaut `false`.
  `debute(['A';'T';'G'],['A';'T'])` vaut également `false`, de même que `debute([6;6],[6;7;8;9])`.

Exercice 11 : Écrivez une fonction récursive `contient` qui prend en entrée, comme dans l'exercice précédent, deux listes de même type `l1` et `l2` et qui renvoie un booléen qui dit si la liste `l2` contient la liste `l1` (en un seul bloc mais pas nécessairement en son début).

- Par exemple `contient(['A';'G'],['G';'C';'A';'G';'C';'G';'T'])` vaut `true`, de même que `contient([2;1],[3;2;1])`.
- Par contre, `contient(['A';'G';'T'],['G';'C';'A';'G';'C';'G';'T'])` vaut `false`, de même que `contient(['A';'T';'G'],['A';'T'])`.

Indication : on pourra utiliser la fonction `debute` de l'exercice précédent.

On considère dans toute la suite les types `substrat` et `enzyme` suivants :

```
type substrat = S0 | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9 ;;
type enzyme = { nom: string ; entree: substrat ;  sortie: substrat } ;;
```

L'idée intuitive de ces deux types de données est la suivante :

- On se donne un ensemble fixé de substrats qui nous intéressent pour une voie donnée. Ici on considère 10 substrats (de `S0` à `S9`) et l'ensemble de ces substrats constitue le type `substrat`.
- Un enzyme est considéré comme un simple catalyseur qui transforme un substrat en un autre. Par exemple ci-dessous `e1` représente l'enzyme dont le nom est `"E1"` et qui transforme `S0` en `S1`. Un enzyme donné peut avoir plusieurs rôles, il suffit alors de considérer plusieurs valeurs de type `enzyme` avec le même `nom`. Par exemple en considérant `e3` et `e3bis` qui partagent le même nom d'enzyme `"E3"` et inversent leurs entrées et sorties, on représente une transformation totalement réversible :

```
let e1 = { nom="E1"; entree=S0; sortie=S1 }
     and e2 = { nom="E2"; entree=S1; sortie=S2 }
     and e3 = { nom="E3"; entree=S2; sortie=S3 }
     and e3bis = { nom="E3"; entree=S3; sortie=S2 };;
```

Exercice 12 : Écrivez une fonction `enzymeCoherent` qui prend en entrée un enzyme et retourne un booléen qui dit s'il est cohérent c'est-à-dire : son nom est non vide et le substrat d'entrée n'est pas égal au substrat de sortie.

Exercice 13 : Écrivez une fonction `produit` qui prend en arguments un enzyme `e` et un substrat `s`, et qui retourne le substrat produit c'est-à-dire : le substrat de sortie de `e` si `s` est égal au substrat d'entrée de `e`, et `s` lui-même sinon. On écrira de préférence une fonction à 2 arguments plutôt qu'une fonction prenant un couple en unique argument.

Exercice 14 : Écrivez une fonction récursive `sequence` qui prend en arguments une liste d'enzymes `l` et un substrat `s`, et qui retourne le substrat obtenu en supposant que `s` rencontre successivement les enzymes de la liste `l`, dans l'ordre de `l`. Par exemple avec les enzymes définis dans la première question, `(sequence [e1;e2;e3] S0)` retourne le substrat `S3` et `(sequence [e2;e1;e3] S0)` retourne `S1` (car `e2` laisse `S0` tel quel, `e1` le transforme en `S1` et enfin `e3` laisse `S1` tel quel).

TD d'Informatique No. 3

1 Arbres Binaires de Recherche

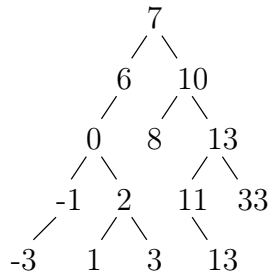
Dans toute la suite, on considère le type des arbres binaires suivant :

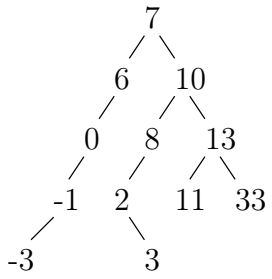
```
type 'a ABin = empty
             | node of 'a * ('a ABin) * ('a ABin) ;;
```

Un arbre binaire est dit *de recherche* si en tout nœud de l'arbre :
- la valeur qui est en ce nœud est supérieure ou égale à toutes les valeurs contenues dans son sous-arbre de gauche
- et cette valeur est strictement inférieure à toutes les valeurs contenues dans le sous-arbre de droite

Un arbre binaire est dit *complet* si tout nœud de l'arbre possède 0 ou 2 fils (i.e. jamais un seul fils non vide).

Exercice 1 : Pour chacun des deux arbres ci-dessous, dites s'il est de recherche ou non :





Justifiez vos réponses.

Exercice 2 : Écrivez en ML le sous-arbre du premier arbre de l'exercice précédent dont la racine est 0.

Exercice 3 : Tracez graphiquement chacun des trois arbres ci-dessous et dites s'il est complet puis s'il est de recherche.

```
let a1 = node(7,node(6,node(0,empty,empty),empty),
                node(33,empty,empty)) ;;
let a2 = node(10,node(8,empty,empty),
                 node(13,node(11,empty,empty),
                         node(33,empty,empty))) ;;
let a3 = node(2,node(-3,empty,empty),
                node(2,node(-1,empty,empty),
                       node(3,empty,empty))) ;;
```

2 Opérations sur les A.B.R.

Exercice 4 : Écrivez une fonction `maximum` qui prend en entrée un arbre binaire `a` *que l'on suppose de recherche*, et fournit en sortie la valeur maximale contenue dans cet arbre. Écrivez de même une fonction `minimum`. Ces fonctions devront échouer sur les arbres vides. On prendra soin de produire des algorithmes qui exploitent efficacement le fait que `a` soit de recherche.

Exercice 5 : Écrivez une fonction `deRecherche` qui prend un arbre binaire quelconque en entrée et retourne un booléen qui dit si l'arbre est de recherche.

Exercice 6 : Développez à la main les calculs de la fonction `deRecherche` sur les arbres `a1` et `a3` précédents.

Exercice 7 : Écrivez une fonction `appartient` qui dit si une valeur `v` appartient à un arbre binaire `a` *supposé de recherche*. On prendra soin de produire un algorithme qui exploite efficacement le fait que `a` soit de recherche.

Exercice 8 : Écrivez une fonction `insere` qui insère une valeur `v` dans un arbre binaire `a` de telle sorte que si `a` est de recherche, alors le résultat aussi. *Indication* : faire l'insertion en une feuille de l'arbre.

Exercice 9 : Réécrire la fonction `insere` pour qu'elle insère l'élément `v` à la racine de l'arbre. *Indication* : il faut d'abord écrire une fonction de `coupe` de l'arbre, qui produit un couple d'A.B.R., l'un ne contenant que des éléments inférieurs ou égaux à `v` et l'autre des éléments strictement supérieurs.

3 Stratégies de suppression dans un A.B.R.

Il s'agit d'écrire une fonction `supprime` qui supprime une valeur `v` d'un arbre binaire `a` *supposé de recherche*, de telle sorte que le résultat est encore de recherche.

Si `v` n'est pas dans l'arbre, le résultat est conventionnellement l'arbre `a` tel quel plutôt que de produire une erreur. Sinon, on est confronté au problème de supprimer un nœud d'arbre puis de « raccrocher » d'une manière ou d'une autre les deux sous-arbres qui sont ses fils. Ce n'est pas si simple et plusieurs stratégies sont possibles.

Pour chacune des stratégies proposées ci-dessous, il est conseillé de se faire un dessin des opérations à effectuer avant de programmer.

Exercice 10 : On peut remplacer le nœud par la valeur maximale du sous-arbre de gauche. *Indication* : on peut pour cela écrire d'abord une fonction `extraitMax` qui fournit à la fois la valeur maximale d'un A.B.R. et l'arbre obtenu en la supprimant.

Exercice 11 : On peut faire de même avec la valeur minimale du sous-arbre de droite.

Exercice 12 : On peut raccrocher le sous-arbre de gauche sous la feuille la plus à gauche du sous-arbre de droite. *Indication* : écrire d'abord une fonction `raccrocheGauche`.

Exercice 13 : On peut raccrocher le sous-arbre de droite sous la feuille la plus à droite du sous-arbre de gauche...

Exercice 14 : Enfin on peut choisir l'une des stratégies précédentes en fonction de la forme du sous-arbre de la valeur à supprimer, afin d'équilibrer au mieux l'arbre résultat. Ceci est une question subsidiaire.

TD d'Informatique No. 4

1 Recherche efficace du i-ième élément d'un ABR

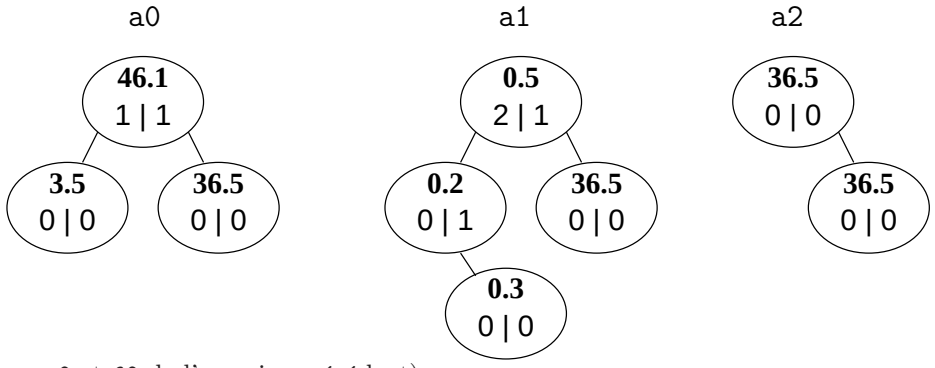
Dans cette partie, on considère une version des arbres binaires de recherche dans laquelle chaque nœud d'un arbre contient trois informations :

- la valeur `v` mémorisée par ce nœud, qui est supposée comme en cours être d'un type de données `'a` sur lequel les comparaisons du langage ML s'appliquent,
- le nombre de valeurs inférieures ou égales à `v` mémorisées dans l'arbre, c'est-à-dire la taille du sous-arbre de gauche puisque l'arbre est supposé de recherche,
- le nombre de valeurs strictement supérieures à `v` mémorisées dans l'arbre, c'est-à-dire la taille du sous-arbre de droite pour les mêmes raisons.

Exercice 1 : Déclarez deux types en ML, l'un appelé `'a cellule` qui contiendra les informations utiles dans un nœud d'arbre, l'autre appelé `'a arbre` qui définit ces arbres.

Exercice 2 : Déclarez en ML une cellule `c0` de valeur `36.5` où les champs `nbinf` et `nbsup` sont nuls. Déclarez ensuite une feuille `f0` de type `'a arbre` ne contenant que cette cellule.

Exercice 3 : Déclarez en ML les arbres `a0`, `a1` et `a2` suivants dans lesquels la valeur est écrite en gras, et les champs `nbinf` et `nbsup` en italique séparés par une barre verticale :



(on pourra utiliser `c0` et `f0` de l'exercice précédent).

Exercice 4 : Un arbre sera dit `correct` si c'est un arbre binaire de recherche et si en chaque nœud la valeur de `nbinf` est égale à la taille (i.e. le nombre de nœuds) du sous-arbre de gauche et celle de `nbsup` est égale à la taille du sous-arbre de droite.
Pour chacun des trois arbres de l'exercice précédent, dites s'il est correct, et s'il ne l'est pas dites pourquoi.

Exercice 5 : En supposant qu'elle ne s'applique qu'à des arbres corrects, programmez la fonction `taille` sur le type `'a arbre`. INDICATION : si `a` est supposé correct alors sa taille, s'il est non vide, est égale à la somme des champs `nbinf` et `nbsup` de sa racine plus 1.

Exercice 6 : En supposant qu'elles ne s'appliquent qu'à des arbres corrects, programmez les fonctions `minima` et `maxima` sur le type `'a arbre`, qui fournissent respectivement la plus petite et la plus grande valeur contenue dans l'arbre. INDICATION : un arbre correct est en particulier de recherche...

Exercice 7 : Programmez la fonction booléenne `correct` sur le type `'a arbre`.
INDICATION : attention de traiter correctement tous les cas de sous-arbres vides.

Exercice 8 : Programmez une fonction `ieme` qui prend en entrée un `'a arbre` supposé correct et un entier `n`, et retourne la `n`-ième plus petite valeur de l'arbre. Par exemple, la première plus petite valeur de `a1` est `0.2` (donc `ieme(a1,1)` est égal à `0.2`), la deuxième plus petite valeur est `0.3` (donc `ieme(a1,2)` est égal à `0.3`), la troisième plus petite est celle de la racine `0.3`, et la quatrième est `36.5`. Si le nombre `n` est inférieur ou égal à 0, ou s'il est supérieur au nombre de nœuds, alors la fonction `ieme` doit échouer.

2 Chemins dans un arbre binaire

Dans la suite, on considère le type énuméré suivant :

```
type direction = gauche | droite ;;
```

et l'on utilise des `direction list` pour identifier un nœud dans un arbre. Par exemple le nœud de valeur `0.3` dans l'arbre `a1` est identifié par la liste `[gauche;droite]` parce que, partant de la racine, on suit d'abord le sous-arbre de gauche puis le sous-arbre de droite pour arriver à ce nœud. Le nœud de valeur `0.2` est de même identifié par la liste `[gauche]` dans l'arbre `a1`. En revanche la liste `[droite;gauche]` ne représente aucune valeur de `a1` puisqu'après avoir suivi le sous-arbre de droite, on tombe sur un sous-arbre vide à gauche. Il en est de même pour `[droite;droite;droite]`. La liste vide `[]` identifie toujours la racine de l'arbre s'il est non vide.

Exercice 9 : Programmez une fonction `valeur` qui prend en entrée un `'a arbre` et une `direction list`, et retourne la valeur du nœud identifié par la liste. Cette fonction doit échouer si le nœud n'existe pas.

Exercice 10 : Programmez une fonction `identifie` qui prend en entrée un `'a arbre` supposé correct et un entier `n`, et retourne la liste de directions qui identifie la `n`-ième plus petite valeur de l'arbre. Par exemple, la première plus petite valeur de `a1` est identifiée par `[gauche]` (donc `identifie(a1,1)` est égal à `[gauche]`), la deuxième plus petite valeur est identifiée par `[gauche;droite]`, la troisième plus petite (celle de la racine) par `[]`, et la quatrième par `[droite]`. Si le nombre `n` est inférieur ou égal à 0, ou s'il est supérieur au nombre de nœuds, alors la fonction `identifie` doit échouer.

Exercice 11 : Programmez une fonction `tousChemins` qui prend en entrée un `'a arbre` supposé correct et retourne la liste de toutes les listes de directions qui identifient un élément de l'arbre.

Exercice 12 : En ignorant l'ordre d'énumération des listes de directions, quelle est la condition nécessaire et suffisante pour qu'un ensemble de listes de direction caractérise une forme d'arbre binaire ? (« forme » signifie qu'on ne considère pas le contenu des cellules).

L1, informatique 2^{nd} semestre

TD d'Informatique No. 5

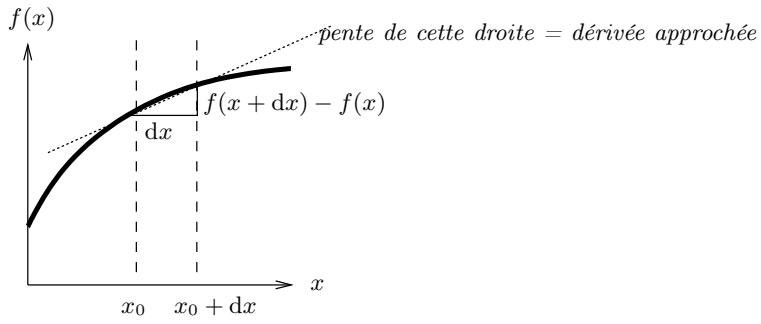
Exercice 1 : On considère le type des arbres binaires suivant :

```
type 'a ABin = empty
             | node of 'a * ('a ABin) * ('a ABin) ;;
```

Écrivez une fonction `partout` qui prend en argument une fonction `f` d'un type $(\alpha \to \beta)$ et un arbre binaire dont les éléments sont de type α, et retourne l'arbre binaire de même forme dont les éléments sont les images par `f` des précédents.
Donnez le type de cette fonction.

Dérivées et intégrales

Étant donnée une fonction $f : I\!R \to I\!R$ (c'est-à-dire de type `float→float` pour le langage ML), la dérivée de f en une valeur x_0, si elle existe, est la valeur vers laquelle tend $\frac{f(x_0+\mathrm{d}x)-f(x_0)}{\mathrm{d}x}$ lorsque $\mathrm{d}x$ tend vers 0. La phrase précédente signifie que l'on peut calculer la pente *locale* de la fonction f de manière de plus en plus précise en prenant une valeur de $\mathrm{d}x$ de plus en plus petite, comme le montre la figure ci-dessous.



Par exemple, on peut choisir $\mathrm{d}x = 0,01$ mais une approximation plus précise sera probablement obtenue avec $\mathrm{d}x = 0,001$, ou encore mieux $0,0001$, etc.

La dérivée d'une fonction f est elle même une fonction : celle qui associe à chaque valeur x_0 la dérivée de f en x_0. On la note souvent $f' : I\!R \to I\!R$.

Exercice 2 : Écrivez selon le principe décrit au-dessus la fonction d'ordre supérieur `derivee` qui prend en entrée une valeur d'approximation $\mathrm{d}x$ (de type `float`) et une fonction f (de type `float→float`, supposée dérivable), et retourne la fonction f' correspondante.
Calculez ensuite son type en faisant l'arbre de l'expression qui la définit.

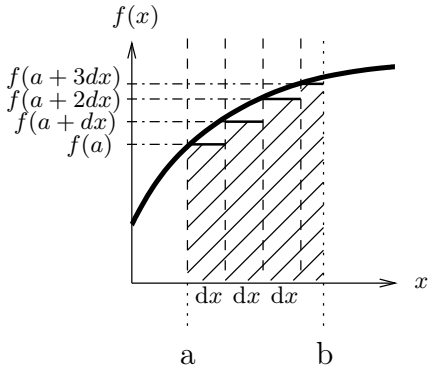
Exercice 3 : Écrivez une fonction `derivee_au_millieme` qui prend en entrée une fonction de type `float→float` et retourne sa dérivée approximée par un $\mathrm{d}x$ égal à 10^{-3}. On utilisera la question précédente. Calculez ensuite le type de cette fonction.

Exercice 4 : Testez votre fonction sur un polynome de degré 2. On constatera que l'approximation au millième en $\mathrm{d}x$ induit une petite erreur qui dépasse le millième : $\mathrm{d}x$ n'est pas l'erreur admise mais la largeur utilisée pour mesurer la pente.

Étant donnée une fonction $f : I\!R \to I\!R$ (c'est-à-dire de type `float→float` pour le langage ML), l'intégrale de f d'une valeur a à une valeur b, si elle existe, est la surface contenue sous la courbe de f au-dessus de l'intervalle $[a, b]$. On peut calculer cette intégrale, si elle existe, comme la surface vers laquelle tend la somme

$$\mathrm{d}x \times f(a) + \mathrm{d}x \times f(a + \mathrm{d}x) + \mathrm{d}x \times f(a + 2\mathrm{d}x) + \cdots + \mathrm{d}x \times f(a + (n-1)\mathrm{d}x) + (b - (a + n\mathrm{d}x)) \times f(a + n\mathrm{d}x)$$

(où $a + n\mathrm{d}x < b$ et $a + (n+1)\mathrm{d}x > b$) lorsque $\mathrm{d}x$ tend vers 0. La phrase précédente signifie que l'on peut calculer la surface sous la courbe de la fonction f de manière de plus en plus précise en prenant une valeur de $\mathrm{d}x$ de plus en plus petite, comme le montre la figure ci-dessous.



L'intégrale de la fonction f est elle même une fonction qui prend en arguments les deux réels a et b et retourne la surface qui est elle même un réel.

Exercice 5 : Écrivez selon le principe décrit au-dessus la fonction d'ordre supérieur `integrale` qui prend en entrée une valeur d'approximation dx (de type `float`) et une fonction f (de type `float→float`, supposée intégrable), et retourne la fonction intégrale de f correspondante. Cette intégrale prend elle même en entrée a et b et devra donc être soit de type `float*float→float`, soit de type `float→float→float` : programmez ces deux possibilités. Calculez ensuite le type de la fonction `integrale` en traçant l'arbre de l'expression qui la définit.

Exercice 6 : Écrivez une fonction `integrale_au_millieme` qui prend en entrée une fonction de type `float→float` et retourne son intégrale approximée par un dx égal à 10^{-3}. On utilisera la question précédente. Calculez ensuite le type de cette fonction.

Exercice 7 : Testez votre fonction sur le polynome $p(x) = x$ entre 0 et 1, puis entre 1 et 2, puis entre 2 et 3. Remarquer les approximations obtenues.

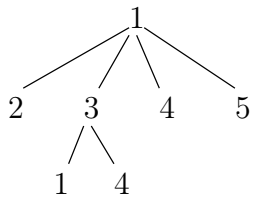
1 Les arbres généraux (dits planaires)

On considère le type suivant :

```
type 'a arbre = noeud of 'a * (('a arbre) list) ;;
```

Exercice 8 : Quel type de données peut-il intuitivement représenter ?

Exercice 9 : Déclarez `a0` de type `int arbre` représentant l'arbre suivant :



Dans toute la suite, on utilisera `a0` pour tester les fonctions écrites.

Exercice 10 : Écrivez une fonction `sup` qui prend en entrée un arbre planaire et fournit en sortie le plus grand élément de cet arbre. *Rappel :* la fonction `max` : $\alpha\to\alpha\to\alpha$ retourne le plus grand de ses deux arguments.

Exercice 11 : Écrivez une fonction `hauteur` qui calcule la hauteur d'un arbre planaire.

Exercice 12 : Écrivez une fonction `taille` qui calcule le nombre de nœuds d'un arbre planaire.

L1, informatique 2^{nd} semestre

TD d'Informatique No. 6

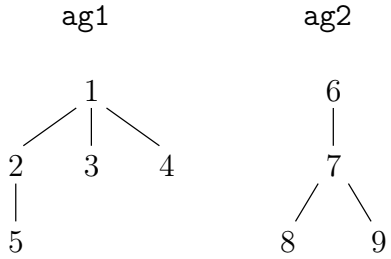
Arbres généraux

Un arbre « général » est un arbre dont chaque nœud interne peut avoir un nombre quelconque de sous-arbres fils. Ainsi, on représente les arbres généraux avec un constructeur de nœud interne qui prend en argument une liste de sous-arbres (plutôt que seulement deux sous-arbres comme dans le cas des arbres binaires).

Dans toute la suite, on considère le type des arbres suivant (qui n'est pas le même qu'au précédent TD) :

```
type 'a arbre = feuille of 'a
              | interne of 'a * (('a arbre) list) ;;
```

Exercice 1 : Déclarez chacun des deux arbres suivants et donnez leur type :



Est-il possible d'avoir un arbre vide de type α `arbre` ?

Par convention on appelle « forêt » une liste d'arbres. Un arbre est donc soit réduit à une feuille, soit construit par une racine et une forêt de fils.

Exercice 2 : Écrivez une fonction `negations` qui prend en entrée un arbre de booléens et fournit en sortie l'arbre des négations de chacune des valeurs de l'arbre.

Exercice 3 : Écrivez une fonction `maximum` qui prend en entrée une liste d'éléments et fournit en sortie le plus grand de ces éléments. On utilisera un itérateur pour éviter de faire une programmation récursive.
INDICATION : `max` : $\alpha \to \alpha \to \alpha$ calcule le maximum de deux éléments pour l'ordre standard de ML.

Exercice 4 : Écrivez une fonction `plusGrand` qui prend en entrée un arbre général `a` et fournit en sortie la valeur maximale contenue dans cet arbre.

Exercice 5 : Écrivez une fonction `hauteur` qui calcule la hauteur d'un arbre.

Exercice 6 : L'arbre `ag3 = interne(10,[])` a-t-il un sens ?
Écrivez un prédicat `correct` qui vérifie si une valeur de type α `arbre` a un sens.
Pourquoi n'aurait-il pas été plus simple d'accepter un constructeur de base `vide` plutôt que `feuille` ?

Exercice 7 : Réécrivez toutes les fonctions des exercices 4 à 6 précédents en vous interdisant l'usage de `map`.

Exercice 8 : Récrivez `plusGrand` en prenant en argument supplémentaire comme en cours une fonction d'ordre stricte qui compare deux valeurs de l'arbre, au lieu d'utiliser la fonction `max` qui a le désavantage d'être uniquement fondé sur les comparaisons natives ($<$) de ML.

Exercice 9 : Écrivez une fonction `taille` qui calcule le nombre de valeurs contenues dans une arbre, selon les deux versions (avec et sans `map`).

L1, informatique 2^{nd} semestre

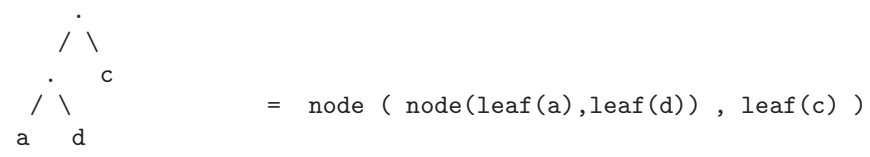
TD d'Informatique No. 7

(Incluant quelques révisions)

1 Codage de Huffman

Exercice 1 : Déclarez en ML le type α `huffman` des arbres binaires non vides et tels que :
- les feuilles (`leaf`) contiennent des éléments de type α
- les nœuds internes (`node`) ne contiennent aucune valeur (contrairement au type d'arbre binaire classique, où les nœuds internes contiennent aussi des éléments de type α).

Par exemple le dessin suivant représente un `char huffman` constitué de trois feuilles et deux nœuds internes :



Le *codage de Huffman* utilise le type α `huffman` pour coder une liste d'éléments de type α (par exemple un texte qui peut être représenté par une liste de caractères) en remplaçant chaque élément par son code, le code en question étant simplement une liste de « directions ».

```
type direction = gauche | droite ;;
```

Exercice 2 : Etant donné un α `huffman` non redondant, on peut coder les éléments de type α qu'il contient de la manière suivante : le chemin qui va de la racine de l'arbre à une feuille est représenté par la liste des choix « sous-arbre de gauche » ou « sous-arbre de droite » effectués en chaque nœud interne. Ceci fournit donc un codage de l'ensemble des feuilles de l'arbre dans l'ensemble des listes de type `direction list`. Appelons « ensemble des codes valides » l'ensemble des listes qui dénotent un chemin valide de l'arbre.
Écrivez une fonction booléenne `valide` qui prend en arguments un α `huffman` et une `direction list` et indique si cette liste est un code valide. Pour l'arbre dessiné plus haut, la liste `[gauche ;droite]` est un code valide (qui mène à la feuille contenant `d` et code donc le caractère `d`). Par contre les listes `[gauche]` ou `[droite ;gauche]` ne sont pas des codes valides.

Dans toute la suite, on pourra supposer que les arbres considérés ne sont pas réduits à une feuille.

Exercice 3 : Écrivez une fonction `decode` qui, étant donnés un α `huffman a`, une `direction list l` et un élément `e` de type α, retourne un triplet formé d'un booléen indiquant si la liste `l` commence par un code valide, de l'élément codé dans le cas où il existe (sinon l'élément `e` par convention) et du reste de la liste, c'est-à-dire le suffixe de `l` qui n'a pas été utilisé pour trouver l'élément (n'importe quelle liste si le booléen est faux).
Par exemple :
`decode` appliquée à l'arbre précédent et la liste `[gauche ;droite ;gauche ;gauche ;gauche]` retourne le triplet `(true,d,[gauche ;gauche ;gauche])`. Par contre, sur la liste `[gauche]` le booléen de la première composante vaudrait `false`.
On prendra soin de ne parcourir la liste qu'une fois.

Exercice 4 : Écrire une fonction `decodetout` qui, étant donnés un α `huffman`, une `direction list` et un élément de type α, retourne un couple formé : d'un booléen qui indique si la liste est une concaténation de codes corrects, et de la liste des éléments de type α qu'elle code (n'importe quelle liste si le booléen est faux).
Par exemple :
`decodetout` appliquée à l'arbre précédent et la liste `[droite ;droite ;gauche ;gauche ;gauche ;droite]` retourne le couple `(true,[c ;c ;a ;d])`, alors que la liste `[droite ;droite ;gauche ;gauche ;gauche]` conduirait à un booléen valant `false`.
On prendra soin de ne parcourir la liste qu'une fois.

2 Itérateurs sur les listes

Exercice 5 : Écrivez avec un itérateur en ML une fonction `extraction` qui prend en arguments une liste l'éléments `l` et deux prédicats `p` et `q` et retourne la liste des éléments de `l` qui vérifient `p` et ne vérifient pas `q`.

Exercice 6 : Que fait la fonction définie ci-dessous ?

```
let toto x = let rec mkl n = if n > 0 then n::(mkl (n - 1))
                                      else []
             in list_it (prefix *) (mkl x) 1 ;;
```

Exercice 7 : Étant donnée une α `list` quelconque `l`, on est souvent confronté au problème de compter les éléments de `l` selon leur appartenance à diverses catégories. Par exemple, pour une liste de livres d'une bibliothèque, on peut vouloir faire un inventaire qui compte pour chaque auteur le nombre de livres de cet auteur ; le résultat est alors une liste de couples (*nom_ auteur*,*nbre*), classée alphabétiquement sur les noms d'auteurs. Par exemple encore, pour une liste d'achats dans un supermarché, on peut vouloir compter le nombre d'article dans chacune des catégories `"électroménager"`, `"épicerie"`, `"liquides"`, etc. et le résultat est donc une liste de couples (*nom_ catégorie*,*nbre*) classée par ordre alphabétique sur les noms de catégories. Dans tous les cas, une catégorie comptée 0 fois dans la liste n'apparaîtra pas dans la liste des bilans.
Écrivez une fonction très générale de `bilan` :

1. quels sont les arguments pertinents de `bilan` ?
2. quel sera le type du résultat ?
3. on utilisera un itérateur pour programmer `bilan` ; que faut-il accumuler au cours de l'itération et en quoi consistera la mise à jour de l'accumulateur à chaque rencontre d'un élément de type α ?

Une fonction de mise à jour déclarée localement pourra sans doute améliorer la lisibilité du programme.