Arbres

Table des matières

Première partie

Cours

1 Création de type énuméré en Caml

On peut définir un type en énumérant simplement les valeurs possibles :

```
type jour = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi | Dimanche ;;
```

Penser que le « | » signifie un « ou ».

L'utilisation d'un tel type se fait par filtrage. Par exemple :

```
let demain j =
    match j with
        | Lundi -> Mardi
        | Mardi -> Mercredi
        etc...
;;

```

Mais chacune des valeurs de base peut aussi être d'un certain type :
`type Rbarre = Infini` Minfini | Fini of float ; ;|
Exemple de fonction :

```
let addition x y=
    match (x,y) with
        | Infini, Minfini -> failwith "nan"
        | Minfini, Infini -> failwith "nan"
        | Infini,_ -> Infini
        | _ ,Infini-> Infini
        | Minfini,_ -> Minfini
        | _ ,Minfini-> Minfini
        | Fini(a), Fini(b) -> Fini (a+.b)
;;
```

Remarque : « nan » signifie « not a number », c'est ce qu'on obtient dans la plupart des langages lorsqu'on effectue une opération qui correspond à une forme indéterminée.

Exemple important : Reconnaissez-vous le type suivant ?
`type a' mystere = Vide` Cons of 'a * mystere ; ;|

N.B. C'est un type récursif !

Vocabulaire : les éléments de base dans un type somme (dans les exemples précédents il y avait `Lundi, Infini, Fini, Vide` s'appellent les « constructeurs » du type. Ils peuvent être constants ou être des fonctions. En Ocaml, les constructeurs prennent une majuscule.
Important : Les constructeurs sont les seules fonctions qui peuvent être utilisés dans un motif (un objet d'un type construit peut être « déconstruits »).

cf exercice : 1

2 Vocabulaire des arbres

Un graphe est un ensemble de *sommets* reliés par des *arêtes*.

Un arbre est un graphe connexe, ce qui signifie qu'entre deux points existe toujours un chemin, et acyclique, ce qui signifie qu'il n'existe pas de chemin dans l'arbre qui revienne à son point de départ (pas de boucle). De manière équivalente, un arbre est un graphe tel que deux points sont toujours reliés par un unique chemin.

Dans un arbre, les sommets sont généralement appelés des *nœud*. Un nœud qui est relié à un seul autre nœud est souvent appelé une *feuille*. On peut considérer que c'est l'extrémité d'une branche.

En informatique, on choisit un sommet qu'on appelle la *racine*. Lorsqu'on dessine un arbre, on place la racine en haut, puis une ligne en dessous tous les sommets reliés à la racine, la ligne suivante on dessine les sommets reliés à ces sommets, etc... Les feuilles sont donc en bas.
Remarques :

- On peut préférer appeler « arbre déraciné » un arbre pour lequel on n'a pas fixé de racine. En informatique, nous n'en utiliserons presque jamais.
- Un graphe non connexe, mais sans sans cycle est donc une réunion de plusieurs arbres. On dit parfois qu'il s'agit d'une « forêt ».

La *hauteur* d'un arbre est le nombre maximal d'arêtes entre une feuille et la racine.
Remarque : Certains sujets de concours préfèrent compter le nombre maximal de nœuds entre la racien et une feuille. C'est alors un de plus.

Les sous-arbres situés en dessous d'un nœud sont appelés ses *fils*.

En informatique, on utilise des arbres pour stocker des informations. Selon la situation, on peut attacher une donnée à chaque nœud, ou à chaque feuille. La donnée enregistrée dans un nœud s'appelle son *étiquette*.

Par exemple, soit A un arbre. Supposons que chaque nœud ait soit deux soit aucun fils (c'est-à-dire un nœud qui n'est pas une feuille a deux fils). Soit h la hauteur de A. Alors le nombre de feuilles est au maximum 2^h. Et le nombre total de nœuds est au maximum $2^{h+1} - 1$. Ce maximum est atteint lorsque toutes les branches sont de longueur h.
Remarque : Un tel arbre s'appelle arbre *binaire*, ou parfois binaire entier, si on veut insister sur le fait qu'un nœud ne peut avoir un seul fils.
Ainsi un arbre binaire de hauteur h permet de stocker $2^{h+1} - 1$ données. D'un autre point de vue, pour enregistrer N données, il faudra un arbre dont la hauteur sera de l'ordre de $\log_2(N)$.

3 Type Caml

En pratique pour enregistrer un arbre, le principe est exactement le même que pour les listes chaînées, la seule différence est que pour chaque valeur, on enregistre plusieurs pointeurs : un vers chaque fils.
La manipulation d'arbre sera donc très proche de la manipulation de liste, en particulier adaptée à la programmation récursive.

Signalons enfin que le programme de MPSI limite l'étude des arbres *persistants*.

3.1 Définition du type

En Caml on peut procéder ainsi pour définir un arbre : un arbre peut être

- vide ;
- ou un nœud pouvant avoir plusieurs fils.

Selon l'implémentation choisie, une feuille sera un nœud qui n'a aucun fils, ou un nœud qui n'a que des fils vides.
La manière de créer le type dépend des étiquettes qu'on veut mettre à chaque nœud et du nombre de fils de chaque nœud. Quelques exemples :

- Arbre binaire, où tous les nœuds ont une étiquette :

```
type 'a arbre = Vide  | Noeud of ('a arbre * 'a * 'a arbre);;
```

Remarque : Ce sera le type le plus fréquemment utilisé dans les exemples à suivre.

- Arbre binaire, où seules les feuilles ont une étiquette, contenant un élément de type `'a` :

```
type 'a arbref = Feuille of 'a  | Noeud of ('a arbref * 'a arbref);;
```

Remarque : Ce type ne permet pas de créer un arbre vide, ce qui peut poser problème dans certaines situations.

- Arbre binaire où les arêtes sont étiquetées :

```
type 'a arbrea = Feuille | Noeud of ('a * 'a arbrea * 'a * 'a arbrea
```

Dans l'expression `Noeud(e1,f1, e2,f2)`, on pense que `e1` est l'étiquette de l'arête menant à `f1`.

- Arbres ternaires :

```
type 'a arbre3 = Vide  | Noeud of ( 'a * 'a arbre3 * 'a arbre3 * 'a arbre3 );;
```

- Arbres où chaque nœud peut avoir un nombre quelconque de fils, et contient une étiquette :

```
type 'a arbreg = Noeud of 'a * ('a arbreg) list;;
```

Il n'est pas besoin de prévoir un cas de base (arbre vide ou feuille) car le type `list` en contient un. En pratique, une feuille est représentée par `Noeud (x, [])`, où `x` est l'étiquette de la feuille. Ce type ne permet pas d'enregistrer un arbre vide : si besoin rajouter un constructeur `Vide` exprès.

3.2 Manipulation

Les types d'arbres que nous avons présentés sont tous des types somme (c'est-à-dire « construits »). On va donc les manipuler par filtrage (c'est-à-dire en les « déconstruisant »).

Rappel : un motif de filtrage peut être construit avec, et uniquement avec :

- des constructeurs ;
- des identificateurs libres (c'est-à-dire non utilisées jusque là).

Dans les exemples précédents, les constructeurs étaient `Noeud, Vide, Feuille`.

Remarque : Dans les filtrages utilisés depuis le début de l'année, on avait utilisé principalement les constructeurs des listes `::` et `[]`, mais aussi la virgule qu'on peut voir comme le constructeur des couples (ou n-uplets plus généralement).

Traitons quelques exemples :

- Calculer le nombre de nœuds d'un arbre ;
- Calculer le nombre de feuilles d'un arbre ;
- Calculer la hauteur d'un arbre ;
- Calculer la somme des étiquettes d'un arbre ;
- Tester si un élément est un étiquette d'un arbre.

4 Nombre de nœuds et de feuilles

Pour tout arbre a, notons $NI(a)$ sont nombre de nœuds internes (c'est-à-dire pas des feuilles) et $NF(a)$ son nombre de feuilles.

Nous dirons qu'un arbre a est «binaire strict» lorsque chacun de ses nœud a zéro ou deux fils (jamais un).

Remarque : Le vocabulaire n'est malheureusement pas fixé et peut varier d'un sujet de concours à un autre.

Proposition 4.1. *Pour tout arbre binaire strict non vide a, $NF(a) = NI(a) + 1$.*

Remarque : Le nombre total de nœuds est donc $N = NI + NF = 2NI + 1$.

On prouve ceci par récurrence sur la hauteur de l'arbre. Prenez cet exemple comme référence pour une démonstration concernant les arbres : ce type de démonstration revient régulièrement.

Démonstration : Pour tout $n \in \mathbb{N}$, soit $P(n)$: « Pour tout arbre a de hauteur n, $NF(a) = NI(a) + 1$.»

- **Initialisation :** Soit a un arbre de hauteur 0. C'est donc une feuille. Donc $NI(a) = 0$ et $NF(a) = 1$. La formule fonctionne. Donc $P(0)$.
- **Hérédité (forte) :** Soit $n \in \mathbb{N}$. Supposons $\forall k \in [\![1, n]\!]$, $P(k)$. Soit a binaire strict de hauteur $n+1$. Soient f_g et f_d les deux fils de sa racine, qui sont non vides car a est binaire strict de hauteur $\geqslant 1$. Ils sont de hauteur $\in [\![0, n]\!]$, on peut donc leur appliquer l'hypothèse de récurrence : $NF(f_g) = NI(f_g) + 1$ et $NF(f_d) = NI(f_d) + 1$.
  Comptons alors le nombre de nœuds internes de a : $NI(a) = NI(f_g) + NI(f_d) + 1$ (le $+1$ c'est la racine).
  Et le nombre de feuilles : $NF(a) = NF(f_g) + NF(f_d)$.

On rassemble le tout :

$$\begin{aligned} NF(a) =& NF(f_g) + NF(f_d) \\ =& NI(f_g) + 1 + NI(f_d) + 1 \\ =& NI(a) + 1 \end{aligned}$$

D'où $P(n+1)$.

En conclusion, pour tout $n \in \mathbb{N}$, $P(n)$. □

5 Arbre binaire de recherche

Dans cette partie, nous considérons des arbres binaires définis par le type

```
type 'a arbre = Vide | Noeud of ('a arbre * 'a * 'a arbre) ;;
```

On suppose que le type `'a` est muni d'une relation d'ordre.
Remarque : La notion de relation d'ordre est très utile en informatique. Le fait de pouvoir utiliser un ABR est un exemple très parlant. En conséquence, presque tous les types sont munis de relation d'ordre, notamment grâce à l'ordre lexicographique.

On notera $\mathcal{B}$ l'ensemble des arbres de ce type.

5.1 Description et complexité

Définition 5.1. *Soit $a \in \mathcal{B}$.*
On dit que a est un arbre binaire de recherche (ABR) lorsque pour tout nœud n,
les étiquettes du fils gauche de n sont inférieures à l'étiquette de n elle même inférieure aux étiquettes de son fils droit.

Exercice : Écrire une fonction qui indique si un arbre est un ABR.
Pour programmer ceci de manière efficace, il y a deux possibilités :

- **Argument supplémentaire :** Fonction auxiliaire prenant en arguments supplémentaires le min et le max autorisés pour les étiquettes ;
- **Valeur renvoyée supplémentaire :** Fonction auxiliaire qui renvoie en outre le min et le max de l'arbre.

5.2 Programmation

La fonction la plus importante est celle permettant de tester si un élément est présent dans un ABR.

```
let rec appartient x a=
  (* entrée : un ABR a
               un élément x
     sortie : le booléen « x est une étiquette de a » *)
  match a with
  | Vide -> false
  | Noeud(fg, e, fd) when e=x -> true
  | Noeud(fg, e, fd) when e < x ->  (*Poursuivre la recherche à droite *)
     appartient x fd
  | Noeud(fg, e, fd) -> (* e > x : poursuivre la recherche à gauche *)
     appartient x fg
;;
```

Programmons aussi une fonction permettant d'ajouter un élément dans un ABR.

```

let rec insertion x a =
    (* Entree : un ABR a
        Sortie : un ABR contenant les elements de a ainsi que x. Si x y etait deja, on y met
    ↪ un nouvel exemplaire*)
    match a with
    |Vide -> Noeud(Vide,x,Vide)
    |Noeud(fg,e,fd) when x>=e -> Noeud(fg, e, insertion x fd)
    |Noeud(fg,e,fd) -> Noeud(insertion x fg, e, fd)
;;
```

On déduit immédiatement, une fonction pour ranger le contenu d'une liste dans un ABR :

```
let rec abr_of_list = function
  |[] -> Vide
  |t::q -> insertion t (abr_of_list q)
;;
```

Version fonctionnelle en une ligne :

```
let abr_of_list l=
  List.fold_right insertion l Vide ;;
```

Remarque : Pour la différente entre `fold_left` et `fold_right` :

- `List.fold_right op [a0…;;an] e` renvoie $a_0 op ....(a_{n-2} op(a_{n-1} op(a_n ope)))$
- Alors que :
  `List.fold_left op e [a0…;;an]` renvoie $(((eopa_0)opa_1*)*a_2)....$
  Noter que l'ordre des arguments n'est pas le même...

Pour les autres fonctions élémentaires, voir l'exercice 5.

5.3 Complexité

On constate que la complexité de cette fonction est en $O(h_a)$. Par conséquent, on aura intérêt à ce que la hauteur de nos arbres soit la plus faible possible.

Définition 5.2. *Soit a un arbre binaire. On dit que c'est un peigne lorsque chaque nœud de a a au monis un fils vide.*

Notons h_a la hauteur de a et N_a son nombre total de nœuds. On vérifie que $\log_2(N_a+1)-1 \leq h_a \leq N_a - 1$. (Dessiner un exemple où le minimum est atteint, et un exemple où le maximum est atteint.)

Proposition 5.3. *Soit a un arbre binaire. On a $\left\lfloor \log_2(N_a) \right\rfloor \leqslant h_a \leqslant N_a - 1$. Le minimum est atteint lorsque tous les niveaux, sauf éventuellement le dernier sont complets. Le maximum est ateint lorsque a est un peigne.*

Définition 5.4. *On dit que a est équilibré lorsque pour tout nœud n, en notant g le fils gauche et d le fils droit de n, on a $|h_g - h_d| \leq 1$ (les hauteurs des fils gauche et droit diffèrent au plus de 1).*

Remarque : Il est bien sûr impossible en général d'avoir toujours $h_g = h_d$: il faudrait que toutes les branches aient la même longueur, ce qui n'est possible que lorsque le nombre de données est de la forme $2^n - 1$. Par contre, il est toujours possible de ranger un nombre quelconque de données dans un arbre équilibré.

On prouve que si a est équilibré alors : $h_a \leq \frac{3}{2}\log_2(N_a+1)$.
bonus : Calculer la constante optimale pouvant remplacer le $\frac{3}{2}$ dans la formule ci-dessus.
Ainsi, lorsque l'arbre est équilibré, la complexité de la recherche d'un élément est logarithmique. C'est un sujet d'étude classique que de voir comment créer et maintenir des arbres équilibrés.
On peut considérer que l'utilisation d'un arbre de recherche est l'équivalent récursif de l'utilisation du tri dichotomique dans un vecteur trié. Comparons les avantages et inconvénients des deux :

- L'arbre est plus facile à manipuler, moins de risque d'erreur ;
- On peut rajouter un élément dans un ABR en $O(\log h)$ contre $O(n)$ pour un tableau trié (programmer l'insertion si pas déjà fait) ;
- Lorsque l'arbre est parfait, la complexité de la recherche est la même que dans un tableau. Lorsqu'il est seulement équilibré, il y a un facteur multiplicatif. Lorsqu'on n'a pas de certitude sur son squelette, on ne peut pas savoir.
- Il n'est pas évident de maintenir le caractère équilibré d'un ABR (il existe plusieurs méthodes, qui ne sont pas au programme : arbres bicolores, arbres AVL, B-arbres (pas binaires, qui combinent en fait tableaux triés et ABR)...)

cf exercice : 5.

6 Une application : complexité maximale d'un tri

Fixons $n \in \mathbb{N}^*$. Le but est d'estimer la complexité minimale pour trier une suite de n éléments en les comparant deux à deux.
Soit $\mathcal{A}$ un algorithme de tri. On suppose que $\mathcal{A}$ est basé sur des comparaisons deux à deux des éléments à trier. On compare deux éléments, selon le résultat on en compare deux autres, etc. jusqu'à ce qu'on puisse renvoyer le résultat.
Considérons l'arbre de décision a associé : chaque nœud correspond un test de type "si $u_i < u_j$", chaque feuille est le résultat final (la liste triée), donc une permutation des éléments de départ.
Il y a $n!$ manière d'ordonner n les n éléments de départ, donc $n!$ résultats possible, donc au moins $n!$ feuilles.
Chaque exécution de l'algorithme consiste à suivre un chemin dans l'arbre.

6.1 Complexité au pire

La hauteur de a est donc le nombre maximal de comparaisons effectuées (c'est-à-dire la complexité au pire). Notons-la h_n.

Enfin, a est un arbre binaire. On a donc $n! \leq 2^h$, d'où :

$$h_n \geq \log_2(n!).$$

Or, $\log_2(n!) \geqslant \log_2\left(\left(\frac{n}{2}\right)^{n/2}\right) = \frac{n}{2}\log_2(\frac{n}{2})$.

On en déduit que :

$$n\log(n) = O_{n\to\infty}(h_n)$$

Remarque : Si vous l'avez vu en maths, vous pouvez aussi utiliser que $\ln(n!) \underset{n\to\infty}{\sim} n\ln(n)$.

Ce qui signifie que la complexité au pire est au moins de l'ordre de $n\log(n)$.

En particulier, la méthode de tri fusion est (asymptotiquement) optimale. On peut éventuellement l'améliorer un peu, mais on ne peut pas trouver de méthode de complexité négligeable devant la sienne.

Remarque : On peut aussi concevoir des tris plus adaptés à telle ou telle situation : si les données sont déjà presque triées, si on veut un tri "en place" (dans un vecteur, sans avoir à utiliser d'autre vecteur)...

6.2 Complexité moyenne

Menons la même étude pour la complexité moyenne. Celle-ci est égale à la moyenne des hauteurs des feuilles de a. (En considérant que les feuilles d'arrivées sont équiprobables.)

Pour tout feuille F, on notera $h(F)$ sa hauteur. On note aussi NF le nombre de feuilles. Enfin, on notera $M(a)$ la hauteur moyenne des feuilles, c'est-à-dire la complexité en moyenne de $\mathcal{A}$.

Supposons qu'il existe une feuille F_1 de hauteur $\leq h_a - 2$. Alors on peut améliorer l'algorithme ! En effet : soient F_2 et F_3 deux feuilles "soeurs" de hauteur h_a. On les déplace et les colle sous F_1 (cf dessin...). On obtient un nouvel arbre qu'on note a', qui correspond donc à un nouvel algorithme $\mathcal{A}'$.

Dans le calcul de sa complexité moyenne, il y aura $h_a - 1 + 2.(h(F_1)+1)$, et dans celle de $\mathcal{A}$ $2.h_a + h(F_1)$. Comme $h(F_1) \leq h_a - 2$, la complexité moyenne de $\mathcal{A}'$ est meilleure.

Soit b l'arbre obtenu après toutes les optimisations possibles. Donc $M(a) \geq M(b)$, et dans b toutes les feuilles sont de hauteur h_b ou $h_b - 1$.
Remarque : Donc b est équilibré, mais même mieux : il est complet.

Alors $M(b) \geq h_b - 1 \geq \log_2(NF) - 1$.

Puis comme avant...

7 Arbres de valence non bornée

Voyons maintenant comment utiliser des arbres dont chaque nœud peut avoir un nombre quelconque de fils.

On enregistrera alors pour chaque nœud la *liste* de ses fils. Dans le type suivant, on décide en outre de munir chaque nœud d'une étiquette. Cependant on peut aussi munir chaque arête d'une étiquette : dans ce cas on enregistre pour chaque nœud la liste des couples (fils, étiquette de l'arête y menant).

```
1 type 'a arbre3 =   Noeud of 'a * ('a arbre3 list);;
```

N.B. Ici une feuille est un nœud dont la liste des fils est vide. Ce type ne permet pas un arbre vide.

On rappelle qu'une liste d'arbres est parfois appelée une «forêt». Ainsi, chaque nœud est muni d'une étiquette et d'une forêt : la liste de ses fils.

Pour parcourir un tel arbre, on écrit en général deux fonctions mutuellement récursives :

1. Une fonction conçue pour balayer un arbre ;
2. et une fonction conçue pour balayer une forêt.

Il est souvent possible de s'épargner la seconde fonction, à l'aide de programmation fonctionnelle (typiquement, on pourra mapper la premier fonction à la liste des fils).

cf exercice : 9

Deuxième partie

Exercices

MPSI, option informatique
Arbres

1 Création d'un type somme

Exercice 1. ** Les entiers

Voici la définition mathématiques des entiers positifs (en théorie des ensembles) :

```
type entier = Zero | Succ of entier;;
```

1. Définir les entiers 2 et 3.
2. Programmer le test d'égalité et la relation d'ordre sur les entiers, puis l'addition et la multiplication. Vérifier à titre d'exemple que $2 + 3 = 3 + 2$.

2 Arbres binaires

Exercice 2. *! Fonctions élémentaires sur les arbres binaires

Dans cet exercice, on utilise le type suivant :

```
type 'a arbreBin = Vide | Noeud of 'a arbreBin * 'a * 'a arbreBin
```

1. Tester l'égalité de deux arbres.
2. Tester si deux arbres ont le même « squelette », c'est-à-dire sont égaux aux contenus des étiquettes près. On fera la distinction entre le fils gauche et le fils droit.
3. Écrire une fonction `mapArbre` qui prend en entrée une fonction f et un arbre a et qui renvoie l'arbre obtenu en appliquant f à chaque étiquette de a.
4. De même écrire une fonction `fold` prenant en entrée un arbre a, une lci o et son neutre e et qui compose toutes les étiquettes de a.
5. (**!) Tester si un arbre est « équilibré » , c'est-à-dire si pour chaque nœud, les hauteurs des deux fils différent d'au plus 1.
6. Prendre une liste et en ranger le contenu dans un « peigne », c'est-à-dire un arbre dont chaque nœud a au moins un fils vide.
7. (**) Prendre une liste et en ranger le contenu dans un arbre équilibré. Plusieurs méthodes sont envisageables.

Exercice 3. ** Arbre des appels pour le tri fusion

Écrire une fonction qui prend en entrée une liste, la trie selon l'algorithme du tri fusion, et renvoie en même temps l'arbre des appels effectués lors de ce tri.

On utilisera le type suivant :

```
type arbreAppels = Feuille of int list | Noeud of arbreAppels * int list * arbreAppels
```

L'expression `Feuille l`, où `l` est singleton ou vide représente un cas de base. L'expression `Noeud fg l fd` , représente un appel pour trier la liste `l`, `fg` et `fd` étant les arbres des deux appels récursifs effectués.

Exercice 4. * Dessiner un arbre binaire**

On souhaite écrire une fonction pour dessiner un arbre d'entiers. La fonction prendra en argument, outre l'arbre à représenter, un entier représentant la taille que prendra chaque feuille et la hauteur entre deux lignes, ainsi que la fonction d'affichage d'une étiquette (`draw_string` pour un arbre de chaînes, `fun x-> draw_string (string_of_int x)` pour un arbre d'entiers...)

Pour simplifier, on considérera des arbres binaires.

On pourra créer une fonction auxiliaire prenant comme arguments supplémentaires abscisse et ordonnée du coin haut gauche du rectangle où il faudra tracer l'arbre, et renvoyant comme valeur supplémentaire l'abscisse de l'extrémité droite de ce rectangle.

3 Arbres binaires de recherche

Exercice 5. *! Fonctions élémentaires sur les ABR

Gardez un fichier `abr.ml` *facilement accessible en cas de besoin l'an prochain.*

On garde le même type Caml qu'à l'exercice 2.

Pour tout arbre a, on notera h_a sa hauteur et N_a son nombre de nœuds (feuilles incluses).

1. Dessiner un arbre de 7 nœuds de hauteur minimale, puis de hauteur maximale.
2. Rechercher le minimum.
3. Renvoyer l'arbre privé de son minimum.
4. Ranger le contenu d'une liste dans un ABR.
5. Renvoyer la liste des étiquettes contenues dans un ABR, dans l'ordre croissant.
6. Combiner les fonctions précédentes pour obtenir un tri. Quelle est sa complexité au pire ? Dans quel cas le pire se produit-il ?

   Ce tri est isomorphe au « tri par segmentation » qu'on étudiera, mais pour les tableaux, en seconde année.
7. (**) Ranger le contenu d'une liste triée dans un ABR.
8. Renvoyer l'arbre privé de tous les éléments inférieurs à un élément donné dans un ABR.
9. (**!) Écrire une fonction pour tester si un arbre binaire est un arbre de recherche.
10. (**) Renvoyer l'arbre privé d'un élément.

Exercice 6. ** Une application des ABR

Écrire une fonction pour dédoublonner une liste, en utilisant un ABR pour enregistrer les éléments déjà rencontrés.

N.B. On utilise ici un ABR pour enregistrer un ensemble. En effet, cette structure permet d'implémenter efficacement l'ajout d'un élément et le test d'appartenance, qui sont les deux opérations élémentaires sur les ensembles. On peut donc dire que les ABR sont une manière d'implanter les ensembles.

4 Variantes sur les arbres binaires

Exercice 7. * Arbre syntaxique d'une formule

Il est très pratique de représenter une formule à l'aide d'un arbre. A chaque nœud on place une opération, et à chaque feuille un nombre.

1. Dessiner l'arbre correspondant à la formule $-(2+1)*3+2/4$.
2. On propose le type suivant pour implémenter un arbre de formule :

```
type formule =
   Feuille of int
  | Somme of formule*formule
  | Produit of formule*formule
  | Quotient of formule*formule
  | Oppose of formule
;;
```

   Écrire une fonction `calcule` prenant en entrée une formule et calculant son résultat.
3. (Avec le cours sur les piles) Écrire une fonction prenant une expression algébrique postfixée et renvoyant l'arbre de cette formule.

Exercice 8. * Analyse syntaxique**

Cet exercice est la suite logique de 7 : le but est de lire une chaîne de caractères représentant une formule, et de créer l'arbre correspondant pour pouvoir calculer le résultat. Vous aurez alors programmé votre propre calculette !

Pour simplifier, on ne prendra pas en compte la priorité des opérations : on supposera que l'utilisateur mets des parenthèses partout.

Remarque : En comparaison avec la notation polonaise inversée, nous avons là une méthode plus compliquée à programmer, mais plus agréable pour un utilisateur.

1. Écrire une fonction `parentheseCorrespondante` qui prend en entrée une chaîne m, un indice i tel que $M.[i]$ soit une parenthèse ouvrante, et qui renvoie l'indice de la parenthèse fermante correspondante.

   On lèvera une erreur si au cours du procédé on constate que m était mal parenthésé.
2. Écrire une fonction `arbre_of_string` qui prend la chaîne et la traduit en une formule selon l'exercice 7.

   En déduire une fonction pour évaluer une formule tapée par l'utilisateur. Lui donner si possible un nom d'animal, serpents et chameaux étant déjà pris.

5 Arbres de valence non bornée

Exercice 9. **! Fonctions élémentaires pour un arbre quelconque

On utilise ici le type suivant :

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

1. (*) Écrire une fonction pour tester si un arbre est une feuille.
2. Calculer la somme des étiquettes
3. Tester la présence d'un élément x parmi les étiquettes.
4. Calculer la maximum des étiquettes.
5. Appliquer une fonction `f int -> int` à toutes les étiquettes d'un arbre.

Exercice 10. ** Modèle de Galton-Watson

Le modèle de Galton-Watson étudie la probabilité d'extinction d'une espèce animale. On considère que chaque individu a la probabilité 1/8 d'avoir 3 descendants, 3/8 d'avoir 2 descendants, 3/8 d'avoir 1 descendant, et 1/8 de n'avoir aucun descendant.

1. Écrire une fonction prenant en entrée un entier n et qui génère l'arbre généalogique issu d'un individu sur au plus n générations.
2. Modifier la fonction pour qu'elle renvoie en plus de l'arbre un booléen indiquant si la lignée s'est éteinte, c'est-à-dire si à la n-ème génération, il n'y a plus aucun descendant.
3. On peut démontrer que la probabilité d'extinction de la lignée est $\sqrt{5}-2$. Écrire une fonction permettant de vérifier ce chiffre.

Exercice 11. * Arbre généalogique**

Dans un arbre généalogique, on enregistre pour chaque personne son nom, ses dates de naissance et décès, et la liste de ses fils. Le type est défini ainsi :

`type arbreGenealogique = noeud of string * int * int * (arbreGenealogique list);;`

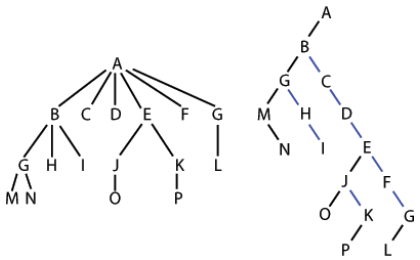
On conviendra de plus de donner les enfants dans l'ordre de leur naissance.

On considère une famille royale. Les règles de succession sont les suivante : au décès du roi, son fils aîné prend sa place. Mais s'il n'a pas d'enfant, ou qu'ils sont déjà morts, on va chercher dans l'ordre : son frère, ses neveux, ses cousins, ses petits neveux, etc... Entre deux frères, c'est toujours l'aîné qui sera prioritaire.

Écrire une fonction prenant en entrée l'arbre généalogique d'une lignée royale, et renvoyant la liste des rois qui se sont succédé, ainsi que les dates de leurs règnes.

Exercice 12. * Transformer un arbre quelconque en un arbre binaire**

Écrire une fonction pour transformer un arbre quelconque en un arbre binaire, en suivant le principe illustré ci-dessous :



Que penser de la complexité pour atteindre un nœud dans la version binarisé de l'arbre comparée à celle dans l'arbre initial ?

Exercice 13. * Arbre des permutations d'une liste**

Étant donné une liste l on demande de construire un arbre dont les feuilles sont toutes les permutations de l.

Remarque : On reprend le principe utilisé dans le chapitre des piles pour résoudre le jeu des chiffres et des lettres sans les lettres.

On pourra utiliser une fonction auxiliaire prenant deux arguments supplémentaires :

- Une liste `dejaPlaces` contenant les éléments de l déjà placés.
- Une liste `elementsRestants` contenant les éléments de j pas encore placés

A chaque étape, on parcourt la liste des éléments restant, pour chacun d'eux on le retire de `elementsRestants`, on l'ajoute dans `dejaPlaces`, et on recommence.

Le parcours de `elementsRestants` pourra être effectué par une autre fonction auxiliaire, ou bien à l'aide de `List.flatten (List.map ...)`.

Quelques solutions

1

2

3

4 Attention : ceci est encore du Caml light...

```
#load "graphics.cma";;

#open "Graphics";; (* Pour éviter de devoir précéder les commandes du nom du module

let rec dessinAux a deb hauteur taille=
    match a with
        |feuille n  ->  lineto (hauteur*taille) deb;
                        draw_string (string_of_int n);
                        deb + taille
        |noeudBin( n, g,d) ->
            begin
              let milieu = dessinAux g deb (hauteur +1) taille in
               (* on dessine le fils gauche, et on recupère l abscisse de sa fin*)
                lineto (hauteur * taille) milieu;
                draw_string (string_of_int n);
                let fin = dessinAux d milieu (hauteur +1) taille in
                 (*on dessine le fils droit et on recupère l abscisse de sa fin*)
                    lineto (hauteur*taille) milieu; (*retour au noeud de départ*)
                    fin
            end
;;


let dessin a taille = dessinAux a 0 0 taille;;

open_graph " 800x600";;
dessin exempleBin 100;;
```

5 6) : $\log_2(\frac{\sqrt{5}-1}{2})$.

6

7

8

9

10

11

12

13

6 Autres exercices

Exercices non présents dans la feuille de TD.

Exercice 14. On dispose de 12 boules numérotées et d'une balance Roberval. Les boules sont toutes de même poids sauf une.

1. Proposer un algorithme pour déterminer quelle boule a un poids différent des autres en trois pesées.
2. En déduire que $12 \leqslant 27$.
3. Proposer une méthode analogue pour démontrer que $27 \leqslant 27$.

1.
2. Compter les feuilles.
3. Prendre plus de boules, mais en sachant à l'avance que la boule différente des autres est plus lourde.

Exercice 15. * Numérotation Sosa**

Le numérotation Sosa binaire d'un arbre binaire a consiste à attribuer un entier à chaque nœud de la manière suivante :

- La racine est numérotée 1
- Si un nœud porte le numéro k, son fils gauche (si il existe) est numéroté $2k$ et son fils droit (si il existe) $2k+1$.

1. Écrire une fonction prenant un arbre binaire (sans étiquette pour simplifier) et renvoyant l'arbre ayant le même squelette mais étiqueté selon la numérotation Sosa.
2. (***) Écrire une fonction prenant en entrée la liste des étiquettes de Sosa d'un arbre et reconstruisant cet arbre.

Exercice 16. ** Arbres quaternaires (d'après CCP 2014)

On va représenter des images à l'aide d'arbres quaternaires. Ceci permettra en particulier d'enregistrer de manière compacte une zone monochrome, et ainsi de gagner un peu d'espace. Pour simplifier, les images enregistrée seront des carrés, de largeur puissance de 2.

Le principe est le suivant : l'image est découpée en 4 :

4	3
1	2

Chacun des quatre carrés ainsi découpés sera enregistré dans un des quatre fils, dans l'ordre indiqué ci-dessus. Et ainsi de suite récursivement, jusqu'à obtenir un carré monochrome. Alors ce carré sera représenté juste par une feuille, dont l'étiquette contiendra la couleur.

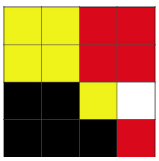
L'ouverture du module "graphics" charge automatiquement un type "color" chargé de représenter les couleurs. On crée ensuite le type suivant :

```
type arbreQ = Feuille of color | Noeud of arbreQ *arbreQ *arbreQ *arbreQ ;;
```

Commandes graphiques utiles : (Caml light ! À modifier)

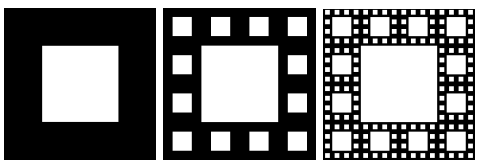
- ouvrir une fenêtre graphique `open_grpah "800*600"`. Aller la chercher dans display/graphics
- colorier un rectangle : `fill_rect x y largeur hauteur`
- changer la couleur : `set_color couleur`

1. Dessiner l'arbre représentant l'image 4×4 ci-contre, puis enregistrer cet arbre sous l'identifiant `arbreExemple`.



2. Écrire une fonction prenant un entier n et renvoyant un damier de format $2^n \times 2^n$.
3. Écrire une fonction pour dessiner l'image représentée par un arbre quaternaire. On prendra en argument l'arbre mais aussi les coordonnées du carré dans lequel on veut dessiner l'image, sous la forme x, y, l, x, y désignant les coordonnées du coin en base à gauche, et l la largeur du carré.
4. Écrire une fonction pour tourner d'un quart de tour dans le sens trigonométrique une image. Puis une fonction pour tourner d'un nombre quelconque (rentré par l'utilisateur) de quarts de tours.

5. Écrire une fonction prenant en argument un entier n et calculant le $n$ème carré de Sierpiński. Voir ci-contre les trois première étapes.



6. Écrire une fonction `consulter` prenant en argument trois entiers x, y, n et un arbre a et renvoyant la couleur du pixel de coordonnées x, y dans l'image représentée par a. L'entier n est celui tel que la taille de l'image est $2^n \times 2^n$.

7. Modifier la fonction précédente pour renvoyer en outre la taille de la plage monochrome contenant le pixel de coordonnées (x, y).
8. Écrire une fonction `fusion` qui rassemble quatre arbres quaternaire. On prendra garde à ce qu'un carré monochrome soit bien représenté par seulement une feuille.
9. Écrire une fonction `changePixel` qui prend un arbre a, les coordonnées xy d'un pixel, l'entier n tel que la taille de l'image soit 2^n, une couleur c, et qui peint le pixel de la couleur c.
10. Écrire une fonction de type `couleur vect vect -> arbreQ` qui prend une image donnée par un simple tableau de couleur (image bitmap) et la transforme en un arbre quaternaire.