Algorithmique des structures de données arborescentes
Feuille d'exercices 1

1 OCaml : Types paramétrés

Certaines fonctions peuvent manipuler des objets de types quelconques. On dit qu'elles sont *polymorphes*. Par exemple les fonctions `fst` et `snd`, qui renvoient respectivement le premier et le deuxième élément d'un couple, s'appliquent à des couples d'éléments de types quelconques :

```
# fst ('a',2);;
- : char = 'a'
# snd (2, 3.5);;
- : float = 3.5
```

La fonction `fst` est de type : `'a * 'b -> 'a`. La fonction `snd` est de type : `'a * 'b -> 'b`. Les lettres précédées d'apostrophes, `'a` et `'b`, désignent des types quelconques.

Exercice 1.1 Ecrire les fonctions `triplet_fst`, `triplet_snd` et `triplet_trd` qui renvoient respectivement le premier, le deuxième et le troisième élément d'un triplet de valeurs quelconques.
Quels sont les types de ces fonctions ?

Certains types, appelés *types paramétrés*, peuvent aussi prendre en paramètre un ou des types quelconques (`int`, `char`, ...). La déclaration d'un type paramétré prendra la forme `type 'a type1 = ...` ou, s'il y a plusieurs types quelconques, `type ('a,'b,...) type2 = ...`.
Par exemple les listes d'éléments de type quelconque sont de type `'a list`. Tous les éléments de la liste doivent être du même type `'a`.
Un autre exemple : pour un dictionnaire qui associe des clés de type `'k` et des valeurs de type `'v`, on définira un type `type ('k, 'v) dict = ...`.

On rappelle qu'une *liste* `l` est

- soit la liste vide, notée `[]` en OCaml,
- soit constituée d'un premier élément, noté par exemple `x`, suivi du reste de la liste, noté par exemple `tail`. Cela s'écrit alors `x::tail` en OCaml.

Une fonction ayant en paramètre une liste pourra donc faire un match de la liste avec ces deux cas.
Attention : En OCaml, les différents éléments d'une liste sont séparés par des **points-virgules**.

Exercice 1.2

- Écrire une fonction récursive `list_length` (sans utiliser le module List d'OCaml) de type `'a list -> int`, qui renvoie la longueur de la liste passée en argument.

  Exemple : `list_length [ 2; 3; 1; 4; 7]` doit renvoyer 5.
- Écrire une fonction récursive `list_member` (sans utiliser le module List d'OCaml) de type `'a -> 'a list -> int`, qui prend en argument un élément de type `'a` et une liste, et qui renvoie `true` si et seulement si l'élément est présent dans la liste.

  Exemple : `list_member 4 [ 2; 3; 1; 4; 7]` doit renvoyer `true`.
- Écrire une fonction `list_nb_occ` qui prend en paramètre un élément `e` et une liste `l` et qui renvoie le nombre d'occurrences de cet élément dans la liste. Exemple : `list_nb_occ 2 [5;2;4;2;3;7;2]` doit renvoyer 3.

Par la suite nous utiliserons le module List d'OCaml et les fonctions qu'il contient :
https ://v2.ocaml.org/api/List.html

2 Arbres binaires

2.1 Définition, vocabulaire

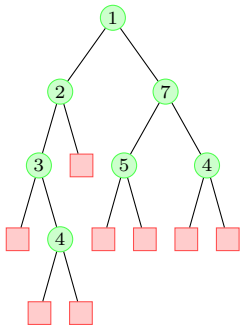
Les *arbres binaires* sont des structures de données assez courante en informatique. Ce sont des structures récursives. De même qu'une liste non vide est constituée d'un premier élément suivi d'une liste plus courte, un arbre non vide est constitué d'un premier élément suivi de deux arbres plus petits.
Plus formellement un ***arbre binaire*** `t` est

- soit l'arbre vide,
- soit constitué :
  - d'un *nœud* `r`, appelé la ***racine*** de l'arbre binaire, contenant une information, appelée ***étiquette*** du nœud,
  - et de deux arbres binaires, `G` et `D`, appelés ***sous-arbres gauche*** et ***droit***.

Le mot *sommet* est synonyme du mot *nœud*.

La ***taille*** d'un arbre est son nombre de nœuds.
Si l'arbre `G` n'est pas vide, sa racine est appelée ***fils gauche*** du nœud `r`. Si l'arbre `D` n'est pas vide, sa racine est appelée ***fils droit*** du nœud `r`. Si un sommet `s` a pour fils (gauche ou droit) un sommet `t`, on dit que `s` est le ***père*** de `t`. Chaque nœud est relié à son père par une ***arête***.
le ***niveau*** (ou profondeur) d'un nœud est le nombre d'arêtes qui séparent le nœud de la racine.
L'***arité*** d'un nœud est son nombre de fils. Une ***feuille*** est un nœud d'arité `0`. Un nœud interne est un nœud qui n'est pas une feuille, càd qui a au moins un fils. Il est donc d'arité `1` ou `2`.

Par exemple, l'arbre suivant comporte 7 nœuds dont 3 feuilles et 4 nœuds internes. Notez que deux feuilles portent la même étiquette : 4. Le fils gauche de la racine est le nœud étiqueté 2. Son fils droit est le nœud étiqueté 7. Les nœuds à profondeur 2 sont les nœuds étiquettés 3, 5 et 4.



Une ***branche*** d'un arbre t est une suite de nœuds allant de la racine à une feuille (sans "remonter"). Formellement, c'est une suite de nœuds $n_0, n_1, \ldots, n_k$ où n_0 est la racine de t, n_k est une feuille de t, et pour chaque i, n_{i+1} est un fils (droit ou gauche) de n_i. La ***longueur*** d'une branche est le nombre d'arêtes qu'elle contient.
La ***hauteur*** de t est le nombre de nœuds de sa plus longue branche, moins 1. La hauteur de l'arbre vide est par convention -1.

Exercice 1.3 1. Soit les arbres binaires illustrés sur la Fig.1.1. Pour chaque arbre calculer sa taille, sa hauteur, son nombre de feuilles, sont nombre de noeuds internes.

On appellera ***squelette*** d'un arbre binaire la structure obtenue en supprimant les étiquettes des nœuds.

Exercice 1.4 Le nombre de squelette d'arbres binaires de taille n est $\frac{1}{n+1} \cdot \binom{2n}{n}$

1. Énumerer les squelettes d'arbres binaires de taille $1, 2, 3$
2. Quelle est la hauteur minimale d'un arbre binaire de taille 4 ? Donner un exemple.
3. Quelle est la hauteur maximale d'un arbre binaire de taille 4 ? Donner un exemple.

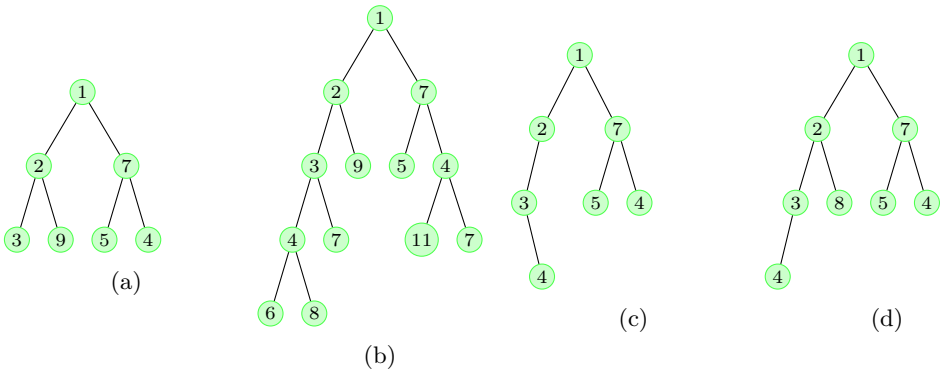


FIG. 1.1 : Arbres binaires

2.2 Premières propriétés

Comme les arbres binaires sont des structures récursives, les fonctions mesurant les arbres binaires (hauteur, taille, ...) vérifient des relations de récurrence simples.

Exercice 1.5 Trouver des relations satisfaites par les fonctions de hauteur, taille, nombre de nœuds internes et nombre de feuilles, entre un arbre binaire et ses deux sous-arbres gauche et droit.

Exercice 1.6

- Dans un arbre binaire, quel est le nombre maximal de nœuds à profondeur d ? Justifier.
  En déduire la taille maximale d'un arbre binaire de hauteur h.
- Quelle est la taille minimale d'un arbre binaire de hauteur h. Justifiez

2.3 Implémentation en OCaml

En OCaml, nous représenterons un arbre binaire avec étiquettes de type `'a` par le type paramétré suivant :

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

Voici un exemple d'arbre binaire de type `int btree` :

```
let t = Node(7, Node(2, Node(5,Empty,Empty), Empty),
                Node(1, Node(5,Empty,Empty), Node(3,Empty,Empty)))
```

On utilisera le filtrage pour écrire des fonctions sur les arbres binaires. Par exemple la fonction qui renvoie l'étiquette de la racine de l'arbre s'écrit :

```
let btree_root t =
  match t with
| Empty -> failwith "l'arbre est vide"
| Node(x,l,r) -> x
```

Le deuxième motif de ce filtrage créé trois variables locales : `x` l'étiquette de la racine, `l` le sous-arbre gauche de la racine et `r` le sous-arbre droit de la racine. Ces trois variables peuvent ensuite être utilisées dans l'expression après la flèche. On remarque qu'ici les variables `l` et `r` ne sont pas utiles pour le résultat de la fonction, on aurait alors pu écrire le deuxième motif `Node(x, _, _)`.

Exercice 1.7 Écrire une fonction `btree_mem` de type `'a -> 'a btree -> bool` qui prend en paramètre un élément de type `'a` et un arbre binaire, et qui teste si cet élément apparaît dans l'arbre.

Exercice 1.8 Écrire une fonction `btree_nb_occ` de type `'a -> 'a btree -> int` qui prend en paramètre un élément de type `'a` et un arbre binaire, et qui compte combien de fois cet élément apparaît dans l'arbre.

Exercice 1.9 Écrire des fonctions récursives en OCaml pour calculer la hauteur, la taille, le nombre de nœuds internes et le nombre de feuilles d'un arbre binaire donné en argument. On utilisera pour cela les relations vues dans le paragraphe 2.2.

Exercice 1.10 En reprenant le type récursif `path` du dernier exercice de la feuille précédente, qui modélise tous les chemins d'un labyrinthe avec des murs, des sorties et des embranchement avec un chemin à gauche et un à droite, écrire une fonction `nb_min_junctions` de type `path -> int` qui prend en paramètre un chemin et qui renvoie le nombre minimum d'embranchements jusqu'à la sortie la plus proche. Si le chemin n'a pas de sortie (cas des murs) la fonction renverra -1. Exemple :

```
# let p = Junction( Junction( Junction (Wall, Exit),
                              Junction(Wall, Wall)),
                    Junction( Junction (Wall, Wall),
                              Junction(Junction (Wall, Exit),
                              Junction(Wall, Wall))));;
val p : path = ...

# let _ = nb_min_junctions p;;
- : int = 3
```