Algorithmique des structures de données arborescentes
Feuille d'exercices 1

1 Introduction à OCaml

Nous nous baserons sur le polycopié de Marc Zeitoun :
https ://moodle1.u-bordeaux.fr/pluginfile.php/1481587/mod_resource/content/1/main.pdf

1.1 Les expressions.

Une expression est un morceau de code que l'ordinateur peut interpréter, calculer, et dont il peut trouver le type et la valeur. En OCaml, tout ou presque est une expression et nous programmerons uniquement en construisant des expressions (voir polycopié page 3).
Ecrire différents types d'expressions : expressions entières, flottantes, booléennes,...

1.2 Les liaisons.

Le concept de variable n'existe pas réellement en OCaml. On peut définir des "liaisons" entre un nom et une expression (dont OCaml saura calculer la valeur) avec le mot-clé `let`.
De plus, on peut définir des liaisons de façon locale, en limitant leur portée à une expression. Pour celà, après la déclaration de la liaison, on ajoute le mot-clé `in`, et une expression dans laquelle on pourra utiliser la liaison que l'on vient de définir (voir polycopié page 5).

Exercice 1.1
- Déterminer la valeur de l'expression `let x = 5.0 in x *. x +. 30.0` .
- Déterminer la valeur de l'expression `let x = 5 in let y = 6 in y + 2 * x` .

1.3 Conditions et filtrage

Comme dans d'autres langages de programmation, OCaml dispose d'une structure conditionnelle, qui utilise les mot-clés `if`, `then` et `else`.
Attention : en OCaml toute expression doit avoir un type uniforme et bien défini. La structure if/then/else étant une expression, elle ne peut avoir qu'un seul type, le même pour la partie then et pour la partie else. À cause de cette même règle, la partie else est obligatoire.

En plus du if/then/else OCaml propose une façon plus puissante de tester une valeur : le filtrage par cas. L'idée générale est d'associer à un cas à une expression, en décrivant tous les cas possibles. Le filtrage permet de reconnaître des "motifs" et pas seulement des égalités.
La syntaxe du pattern-matching en OCaml utilise les mots-clés `match` et `with` (voir polycopié, page 15 et 16).

Exercice 1.2 Ecrire une fonction `helloworld` qui prend en paramètre une chaine de charactère indiquant une langue et renvoie "Bonjour" écrit dans cette langue. Cette fonction traitera le cas de plusieurs langues différentes. Si la langue passée en paramètre ne fait pas partie des cas prévus, la fonction renverra "Je ne parle pas" concaténé avec la langue demandée. ■

1.4 Les fonctions, les appels de fonctions, les fonctions récursives

De même on définit une fonction en déclarant une liaison entre un nom de fonction et le code de la fonction. Plusieurs syntaxes sont possibles (voir polycopié, page 6 et page 17).

Exercice 1.3
- Déterminer la valeur de l'expression `let f = fun x y -> x - y in f (f 1 2) 3` .
- Déterminer la valeur de l'expression `let cube x = x * x * x in (cube 2) + (cube 3)` .

Dans ce cours, pour répéter un traitement, nous n'utiliserons pas de boucles, mais des fonctions récursives. Une fonction *récursive* est appelée dans sa propre définition. Pour écrire une fonction récursive, il est nécessaire de nommer la fonction et d'indiquer explicitement qu'elle est récursive par le mot-clé **rec**.

Exercice 1.4 Soit la fonction `fact` définie comme suit :

```
let rec fact n =
if n = 0 then 1 else n * fact (n-1)
```

Dessiner la pile des appels pour `fact 5`. ■

Exercice 1.5 Écrire une fonction `somme` qui calcule récursivement la somme des entiers de 0 à n (n étant supposé positif). ■

Exercice 1.6 Soient n, p deux entiers. On rappelle que $\text{pgcd}(n, 0) = n$ et que $\text{pgcd}(n, p) = \text{pgcd}(p, n \bmod p)$. Écrire une fonction `pgcd` calculant le pgcd de deux entiers. ■

Exercice 1.7 La suite de Fibonacci est définie par $u_0 = u_1 = 1$ et pour tout $n \geqslant 2$, $u_n = u_{n-1} + u_{n-2}$.

1. Écrire une fonction `fib` calculant le n-ème terme de cette suite.

■

1.5 Les types, types produits, types sommes, types récursifs.

En OCaml on peut définir un nouveau type à l'aide du mot clé `type` suivi du nom du nouveau type. On peut définir un type produit cartésien de types existants en les séparant par des étoiles. Les éléments d'un type produit cartésien sont des n-uplets. Voir polycopié page 10.
Les types sommes permettent de définir des types avec un nombre fini de formes possibles. Les différentes formes possibles sont désignées par des noms commençant obligatoirement par une majuscule et on les sépare par des barres verticales. Voir polycopié, pages 11 et 12.

Exercice 1.8 On définit le type `vect` (pour vecteur) suivant :

```
type vect = Vint of int * int * int | Vfloat of float * float * float;;
```

Donner deux exemples de valeurs du type vect, en utilisant chacun des constructeurs. ■

Exercice 1.9 Définir des types pour représenter les valeurs suivantes :

1. Un type contenant trois valeurs, nommées Vrai, Faux et Peut_etre.
2. L'ensemble des entiers, auquel on a ajouté une constante symbolique Pi.
3. Les points à coordonnées entières, en dimension 2 ou 3.

■

On peut définir des types récursifs. Par exemple on peut définir les entiers naturels avec un type récursif, en utilisant le fait qu'un entier naturel est soit l'entier zéro, soit le successeur d'un entier naturel. Ce type s'écrit `type nat = Zero | Succ of nat`

Exercice 1.10

1. Définir les constantes `one`, `two`, `three` de type `nat`.
2. Ecrire une fonction `is_zero` qui teste si un entier naturel `n` est l'entier zero.
3. Ecrire une fonction `pred` qui renvoie le prédecesseur d'un entier naturel `n` si ce prédecesseur existe et qui renvoie `failwith "pred undefined"` sinon.

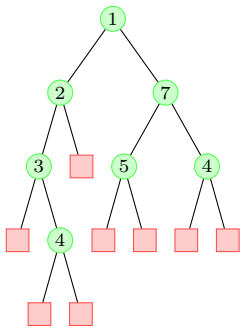
■

2 Arbres binaires : définitions

Un ***arbre binaire*** `T` est

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

On voit que la définition est récursive, puisqu'un arbre non vide est constitué de deux sous-arbres (et de sa racine).
On appellera ***squelette*** d'un arbre binaire la structure obtenue en supprimant les étiquettes des nœuds.
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***.
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. 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.



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.
Le ***niveau*** (ou profondeur) d'un nœud est le nombre d'arêtes qui séparent le nœud de la racine.
On dit qu'un arbre est ***complet*** si tout nœud interne a exactement deux fils.
On dit qu'un arbre est ***parfait*** s'il est complet et toutes les feuilles ont le même niveau.
On dit qu'un arbre est ***quasi-parfait*** s'il est complet jusqu'au niveau $h-1$, et ses feuilles sont de niveau h ou $h-1$, et les feuilles de niveau h sont "le plus à gauche possible".

En OCaml, on représente un arbre binaire avec étiquettes de type `'a` par le type suivant :

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

Exercice 1.11
1. Soit les arbres binaires illustrés sur la Fig.1.1. Pour chaque arbre calculer sa taille et son hauteur, indiquer s'il s'agit d'un arbre binaire particulier : complet, parfait ou quasi-parfait.
2. Dessiner des squelettes d'arbres binaires complet, parfait et quasi-parfait de taille 7.
3. Dessiner des squelettes d'arbres binaires complet, parfait et quasi-parfait de hauteur 3.

3 Propriétés importantes des arbres binaires

Exercice 1.12
1. 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.
2. En déduire des fonctions récursives OCaml pour calculer la hauteur, la taille, le nombre de nœuds internes et de feuilles d'un arbre binaire donné en argument.

Exercice 1.13 Dans un arbre binaire, quel est le nombre maximal de nœuds à profondeur d ? Justifier.

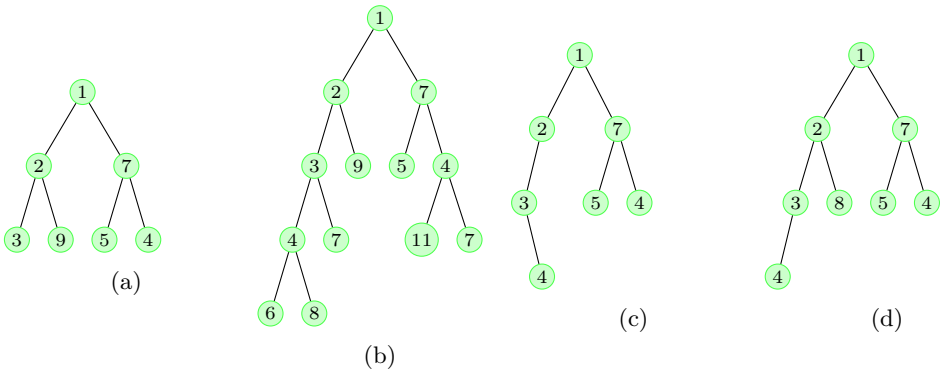


FIG. 1.1 : Arbres binaires

3.1 Preuves par induction structurelle

En informatique on utilise très souvent des structures de données récursives (par exemple les listes ou les arbres). Les preuves de propriétés par *induction structurelle* s'appliquent à tous les ensembles construits récursivement et se basent comme le nom l'indique sur la structure récursive des éléments de l'ensemble.

Soit un ensemble $\mathcal{E}$ défini récursivement par un cas de base qui énumère un nombre fini d'éléments de $\mathcal{E}$ et un cas récursif qui indique comment construire un nouvel élément `X` de $\mathcal{E}$ en appliquant une opération `op` à des éléments de $\mathcal{E}$ `X_1, X_2, ...X_k` déjà construits : `X = op(X_1, X_2, ...X_k)` . La preuve par *induction structurelle* que la propriété P est vraie pour tous les éléments de $\mathcal{E}$ comporte deux étapes :

- prouver `P` pour le ou les cas de base (en nombre fini)
- prouver que si `X = op(X_1, X_2, ...X_k)` et si `P(X_1), P(X_2), ...P(X_k)` sont vraies alors `P(X)` est vraie.

Lorsque l'on a fait la preuve dans le cas de base et dans le cas récursif, on peut conclure que la propriété `P` est vraie pour tous les éléments de l'ensemble $\mathcal{E}$.

Par exemple dans le cas des listes, prouver par *induction structurelle* qu'une propiété `P` est vraie pour toutes les listes comporte deux étapes :

- prouver que `P` est vraie pour la liste vide
- prouver que si la liste `l` est composée d'un premier élément `a` suivi du reste `l'` (en OCaml cela s'écrit `l=a::l'`) et si `P(l')` est vraie, alors `P(l)` est vraie.

Dans le cas des arbres binaires, prouver par *induction structurelle* qu'une propiété `P` est vraie pour tous les arbres binaires comporte deux étapes :

- prouver que `P` est vraie pour l'arbre vide
- prouver que si l'arbre `T` est composé d'une racine `r` et de deux sous-arbres gauche et droit `G` et `D` (en OCaml cela s'écrit `T=Node(r,G,D)`) et si `P(G)` et `P(D)` sont vraies alors `P(T)` est vraie.

Il faudra adapter le cas de base selon la propriété à prouver : par exemple si la propriété doit être démontrée pour tous les arbres binaires non vides, il faudra prendre pour cas de base l'arbre réduit à un seul nœud.

Exercice 1.14 Pour un arbre binaire complet t non vide avec $n(t)$ nœuds, $l(t)$ feuilles et $i(t)$ nœuds internes, montrer les deux relations suivantes :

$$l(t) = 1 + i(t) \tag{1.1}$$

$$n(t) = 2 \cdot l(t) - 1 \quad et \quad n(t) = 2 \cdot i(t) + 1 \tag{1.2}$$

Sont-elles vraies pour tous les arbres binaires ?

Exercice 1.15 On note $h(t)$ la hauteur et $n(t)$ le nombre de nœuds d'un arbre binaire t. Montrer :

a) Qu'on a $h(t) + 1 \leqslant n(t) \leqslant 2^{h(t)+1} - 1$.

b) Que l'arité de tous les nœuds internes de t est 1 si et seulement si $h(t) + 1 = n(t)$.

c) Que l'arbre t est parfait si et seulement si $n(t) = 2^{h(t)+1} - 1$.