



## TD4 – Types sommes – Arbres binaires

Dans les trois premiers exercices de ce TD, on considère le type des arbres binaires défini par :

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

Les exemples sont donnés en considérant l'arbre `t_ex` de la figure 1.

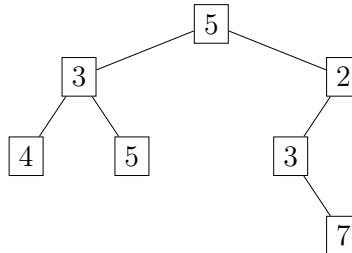


FIGURE 1 – exemple : arbre binaire `t_ex`

**Exercice 4.1** (Taille d'un arbre binaire).

Définir une fonction de signature `size (t:'a btree) : int` qui calcule la taille (c-à-d le nombre de nœuds) de l'arbre binaire `t`.

```
# size Empty;;           # size t_ex;;
- : int = 0              - : int = 7
```

**Exercice 4.2** (Branches d'un arbre binaire).

Une branche d'un arbre binaire est une liste d'étiquettes rencontrées sur les nœuds de l'arbre lors d'un parcours de l'arbre depuis la racine jusqu'à une feuille. Par exemple, la liste `[5;3;5]` est une branche de l'arbre `t_ex` de la figure 1.

1. Définir une fonction de signature `max_length_branch (t:'a btree) : 'a list` qui construit la branche la plus longue de l'arbre `t` (si l'arbre contient plusieurs branches de mêmes longueurs, on choisit la branche la plus à gauche).

*Indication.* Utiliser la fonction prédéfinie `List.length`.

```
# (max_length_branch t_ex);;
- : int list = [5; 2; 3; 7]
```

2. L'utilisation de la fonction `List.length` dans la fonction `max_length_branch` conduit à effectuer plusieurs fois le même parcours de liste. En définissant une fonction locale permettant de calculer à la fois la branche de longueur maximale d'un arbre et sa longueur, définir la fonction `max_length_branch` sans utiliser la fonction `List.length`.

Une branche de flot maximal d'un arbre est une branche obtenue en sélectionnant à chaque nœud le sous-arbre dont l'étiquette de la racine est le plus grand.

3. Définir une fonction de signature `max_flow_branch (t:'a btree) : 'a list` qui construit une branche de flot maximal de l'arbre `t` (on utilise ici l'opérateur de comparaison générique `<`).

```
# (max_flow_branch t_ex);;
- : int list = [5; 3; 5]
```

**Exercice 4.3** (Profondeur d’une étiquette).

On appelle *profondeur* d’une étiquette  $e$  dans un arbre binaire  $t$  la distance entre la racine de  $t$  et le nœud étiqueté par  $e$ ; la *distance* entre deux nœuds  $n_1$  et  $n_2$  est le nombre de nœuds à traverser pour atteindre  $n_2$  à partir de  $n_1$ . L’étiquette de la racine est de profondeur 0 et sur l’exemple de l’arbre binaire  $t\_ex$  de la figure 1, l’étiquette 5 est à la fois à la profondeur 0 et à la profondeur 2.

1. Définir la fonction de signature `at_prof (n:int) (x:'a) (t:'a btree) : bool` qui retourne `true` si et seulement si l’étiquette  $x$  est présente à la profondeur  $n$  dans l’arbre binaire  $t$ .

```
# (at_prof 0 5 t_ex);;      # (at_prof 2 5 t_ex);;      # (at_prof 1 5 t_ex);;
- : bool = true           - : bool = true           - : bool = false
```

2. Définir la fonction de signature `at_prof_list (n:int) (t:'a btree) : 'a list` qui construit la liste des étiquettes présentes à la profondeur  $n$  dans  $t$ .

```
# (at_prof_list 2 t_ex);;
- : int list = [4; 5; 3]
```

3. Définir la fonction de signature `etiq_prof_list (x:'a) (t:'a btree) : int list` qui construit la liste des profondeurs auxquelles est présente l’étiquette  $x$  dans  $t$ .

```
# (etiq_prof_list 3 t_ex);;
- : int list = [1;2]
```

*Indication.* Dans l’arbre `Node(e,g,d)` la liste des profondeurs d’une étiquette  $x$  peut s’obtenir en additionnant 1 aux profondeurs de  $x$  dans les sous-arbres  $g$  et  $d$ , et en comparant  $e$  et  $x$ .

4. Définir la fonction de signature `prof_max (t:'a btree) : int` qui calcule la profondeur maximale des étiquettes de l’arbre  $t$ . Par convention, la fonction retourne -1 si l’arbre est vide.

```
# (prof_max t_ex);;
- : int list = 3
```

5. Définir la fonction de signature `max_prof_etiq (x:'a) (t:'a btree) : int` qui calcule la profondeur maximale de l’étiquette  $x$  dans  $t$ . Par convention, la fonction retourne -1 si  $x$  n’apparaît pas dans  $t$ .

```
# (max_prof_etiq 5 t_ex);;      # (max_prof_etiq 7 t_ex);;
- : int list = 2              - : int list = 3
```

**Exercice 4.4** (Types sommes et exceptions).

1. Définir un type somme `value` permettant de regrouper au sein d’un même type les booléens (constructeur `B`) et les entiers (constructeur `I`).
2. On définit l’exception :

```
exception TYPE_ERROR of int
```

Définir une fonction de signature `not1 (v:value) : value` qui renvoie la valeur correspondant à la négation de  $v$  lorsque  $v$  représente une valeur booléenne et lève l’exception `TYPE_ERROR`  $n$  lorsque  $v$  représente un entier  $n$ .

```
# not1 (B true);;      # not1 (I 4);;
- : value = B false    Exception: TYPE_ERROR 4.
```

3. Lorsque  $v$  représente une valeur entière non nulle, on peut « interpréter »  $v$  par le booléen `true` et si  $v$  représente la valeur entière 0, on peut « interpréter »  $v$  par le booléen `false`. En utilisant cette interprétation et la fonction `not1`, définir une fonction de signature `not2 (v:value) : value` qui renvoie la valeur correspondant à la négation du booléen  $b$ . La fonction `not2` devra utiliser la

fonction `not1` et rattraper l'exception levée par la fonction `not1` lorsque `v` représente une valeur entière.

```
# not2 (B true);;           # not2 (I 4);;           # not2 (I 0);;
- : value = B false        - : value = B false        - : value = B true
```

4. On définit l'exception :

```
exception DIV_BY_0 of int
```

Définir une fonction de signature `div1 (v1:value) (v2:value) : value` qui :

- calcule la valeur représentant le quotient (entier) `n1/n2` lorsque `n1` est la valeur entière représentée par `v1` et `n2` est la valeur entière non nulle représentée par `v2`
- lève l'exception `DIV_BY_0 n1` lorsque `v1` représente une valeur entière `n1` et `v2` représente l'entier 0
- lève l'exception `Invalid_argument` lorsqu'au moins une des deux valeurs `v1` et `v2` représente un booléen

```
# div1 (I 6) (I 3);;           # div1 (B false) (I 4);;
- : value = I 2                Exception: Invalid_argument "div".
# div1 (I 6) (I 0);;           # div1 (B false) (I 0);;
Exception: DIV_BY_0 6.         Exception: Invalid_argument "div".
```

5. En utilisant la fonction `div1`, définir une fonction de signature :

```
div2 (v1:value) (v2:value) : value option
```

- qui calcule la valeur représentant le quotient (entier) `n1/n2` lorsque `n1` est la valeur entière représentée par `v1` et `n2` est la valeur entière non nulle représentée par `v2`
- qui renvoie `None` lorsque `v1` représente une valeur entière et `v2` représente l'entier 0
- qui lève l'exception `Invalid_argument` lorsqu'au moins une des deux valeurs `v1` et `v2` représente un booléen

La fonction `div2` devra utiliser la fonction `div1` et rattraper l'exception levée par la fonction `div1` dans le cas d'une division par 0.

```
# div2 (I 6) (I 3);;           # div2 (B false) (I 4);;
- : value option = Some (I 2)   Exception: Invalid_argument "div".
# div2 (I 6) (I 0);;           # div2 (B false) (I 0);;
- : value option = None        Exception: Invalid_argument "div".
```