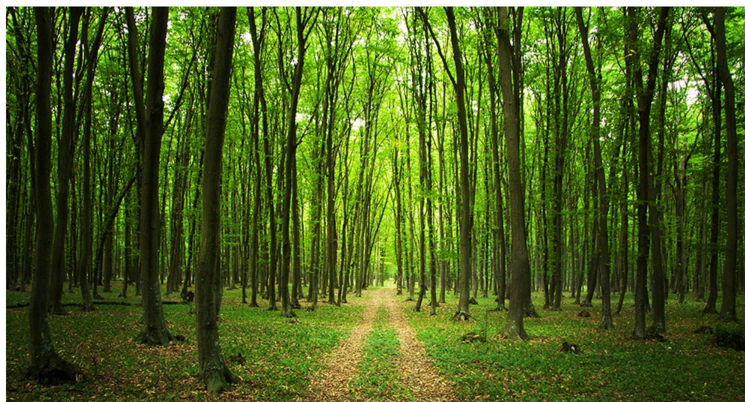
TP nº 7 : Types enregistrements et arbres *n*-aires



1 Types personnalisés

1.1 Alias de types

En CAML on peut donner un nouveau nom à un type existant :

OCAML
```
type complexe = float * float;;
```

Il s'agit d'une abréviation — d'un surnom — pour un type déjà existant et non de la définition d'un *nouveau* type, contrairement aux types sommes.

Cependant, CAML continue d'interpréter les couples de flottants avec le type de base et non avec notre nouveau type pour les complexes :

OCAML
```
(3., 4.);;
- : float * float = 3.0, 4.0
```

Il est possible de forcer le type à l'aide de la syntaxe suivante :

OCAML
```
((3, 4) : complexe);;
- : complexe = 3.0, 4.0

let somme (z1 : complexe) (z2 : complexe) =
  let (re1, im1), (re2, im2) = z1, z2 in
  ((re1 +. re2, im1 +. im2) : complexe)
;;
```

Question 1

Définir la fonction `produit : complexe -> complexe -> complexe`.

Néanmoins forcer les types n'offre pour nous que peu d'intérêt[1] et alourdit l'écriture. Après tout il s'agit de synonymes exacts, alors oublions l'esthétique et laissons l'inférence de type de CAML faire son travail ! On ne cherchera donc pas à respecter au mot près la signature des fonctions (mais les types doivent bien sûr être compatibles, c'est-à-dire équivalents ou plus généraux).

1.2 Types enregistrements

Lorsque l'on souhaite représenter une structure de données comportant plusieurs champs, on peut utiliser en CAML un type produit.

Imaginons par exemple que l'on souhaite représenter un élève ainsi que les notes qu'il a obtenues dans différentes matières. On pourrait définir un type :

OCAML
```
type bulletin = string * int * int * int * int * int;;
```

Mais il faut alors se souvenir exactement quelle position correspond à quelle matière ! Ce n'est pas pratique du tout. Il existe en CAML une structure de données appelée *enregistrement* ou *type produit nommé*.

En revenant sur l'exemple précédent, on pourrait alors définir :

OCAML
```
type bulletin = {
  etudiant : string;
  note_maths : int;
  note_physique : int;
  note_philo : int;
  note_info : int;
  note_langues : int
};;
```

Pour définir les nombres complexes[2] on peut prendre un enregistrement à deux *champs* :

OCAML
```
type complexe = {re : float; im : float};;
```

Pour « créer » le nombre complexe $z = 2 + i$ on écrira :

OCAML
```
let z = {re = 2.0; im = 1.0};;
z : complexe = {re = 2.0; im = 1.0}
```

Question 2

Pourquoi CAML a-t-il reconnu qu'il s'agissait d'une valeur de type `complexe` ici ?

z est une structure contenant deux « cases » appelées `re` et `im` dont le contenu est accessible à l'aide du symbole « . » :

OCAML
```
z.re;;
- : float = 2.0
z.im;;
- : float = 1.0
```

1. Sauf éventuellement pour des TIPE, mais il existe alors d'autres moyens plus intéressants.
2. Pas exactement les nombres complexes bien sûr. Pourquoi ?

On a, en particulier, un accès direct à tous les champs d'un élément.
On peut par exemple définir la fonction de conjugaison :

OCaml
```
let conjuge z = {re = z.re; im = -.z.im};;
conjuge : complexe -> complexe = <fun>
```

ou encore :

OCaml
```
let conjuge {re = a; im = b} = {re = a; im = -.b};;
```

Question 3

Écrire les fonctions `somme : complexe -> complexe -> complexe` et `produit : complexe -> complexe -> complexe` avec ce nouveau type.

1.3 Polymorphie

Qu'il s'agisse de type somme ou de type enregistrement, il est possible de paramétrer les types comme dans les exemples suivants :

OCaml
```
type 'a boite = {taille : int; contenu : 'a list};;
```

OCaml
```
type 'a option =
  | None
  | Some of 'a
;;
```

☞ **Remarque 1**

Ce dernier type est prédéfini en Caml et est très utile lorsque dans certains cas on ne veut pas renvoyer de valeur dans une fonction, par exemple lors de la recherche d'un élément si l'élément n'est pas présent. La sortie doit toujours être de même type et le type `'a option` permet donc d'encapsuler les deux cas.

Question 4

Écrire une fonction `indice 'a -> 'a list -> int option` qui recherche le premier indice d'un élément dans une liste (on renvoie donc `None` si et seulement si l'élément n'est pas présent).

2 Arbres n-aires et forêts

Dans cette section on s'intéresse à une généralisation possible des arbres binaires vus en cours. Chaque nœud peut avoir un nombre illimité (mais fini !) de fils. Autrement dit, le degré d'un arbre n-aires n'est plus nécessairement 2, mais peut être quelconque.

Définition 2.1

Un arbre n-aire sur un ensemble de valeurs de nœuds $\mathcal{X}$ est soit l'arbre vide, soit un couple :

$$(x, (a_1, \ldots, a_k))$$

avec

- $x \in \mathcal{X}$ l'étiquette du nœud (racine)
- $k \in \mathbb{N}$ l'*arité* du nœud
- $a_1, \ldots, a_k$ des arbres n-aires non-vides

Remarque 2

L'équivalent d'une feuille est donc un nœud d'arité 0.

Définition 2.2

Une *forêt* est un ensemble fini (éventuellement vide) d'arbres non vides sans nœuds communs.

Remarque 3

Les fils d'un nœuds forment donc une forêt.

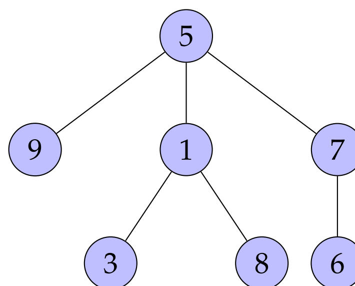
En Caml on peut définir les arbres n-aires ainsi :

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

On peut vouloir donner un nom plus explicite à une liste d'arbres qui est une forêt ! Mais alors pour définir une forêt il faut savoir définir un arbre et réciproquement... De la même manière que l'on peut définir des fonctions mutuellement récursives, on peut définir des types mutuellement récursifs :

OCaml
```
type 'a arbre = Vide | Noeud of 'a * 'a foret
  and 'a foret = 'a arbre list;;
```

OCaml
```
let exemple = Noeud (5,
   [Noeud (9, []);
    Noeud (1, [Noeud (3, []); Noeud (8, [])]);
    Noeud (7, [Noeud (6, [])])
  ]);;
```



Les notions de taille et profondeur existent toujours sur les arbres n-aires. On peut les programmer en Caml à l'aide de deux fonctions mutuellement récursives :

OCaml
```
let rec taille arbre =
  match arbre with
  | Vide -> 0
  | Noeud (_, fils) -> 1 + taille_foret fils
and taille_foret foret =
  match foret with
  | [] -> 0
  | arbre :: autres -> (taille arbre) + (taille_foret autres)
;;
```

Question 5

Écrire la fonction `hauteur : 'a arbre -> int`. On pourra définir la fonction mutuellement récursive `max_hauteur_foret : 'a arbre list -> int`.

De même, les parcours préfixe et suffixe (mais pas infixe) peuvent être définis sur les arbres n-aires. Par exemple :

OCaml
```
let rec parcours_prefixe arbre =
  match arbre with
  | Vide -> ()
  | Noeud (etiquette, fils) ->
     print_int etiquette;
     parcours_foret fils
and parcours_foret foret =
  match foret with
  | [] -> ()
  | arbre :: autres ->
     parcours_prefixe arbre;
     parcours_foret autres
;;
```

Question 6

Écrire de même le parcours suffixe d'un arbre `parcours_suffixe : int arbre -> unit`.

Question 7

Écrire les fonctions `parcours_prefixe : int arbre -> int list` et `parcours_suffixe : int arbre -> int list` qui renvoient les listes des nœuds parcours.

Question 8

Écrire une fonction `degre : 'a arbre -> int` qui donne le degré d'un arbre, c'est-à-dire le plus grand degré de ses nœuds.

Si on suppose que l'on a pas besoin de l'arbre vide (ce qui est assez naturel dans certaines études) on peut se passer du constructeur de type. Pour bien insister sur le fait que l'on manipule un arbre et non un couple quelconque, on propose le type suivant :

OCaml
```
type 'a arbre = {etiquette : 'a; fils : 'a arbre list};;
```

Question 9

Réécrire les fonctions `taille` et `hauteur` avec ce nouveau type.

Question 10

Peut-on écrire une fonction qui renverse l'ordre des fils de chaque nœud d'un arbre (du *même* arbre, sans en recréer un nouveau) ? Pourquoi ?

Question 11

Écrire une fonction `nb_feuilles : 'a arbre -> int` qui renvoie le nombre de feuilles d'un arbre.

Question 12

Écrire une fonction `max_etiquettes : 'a arbre -> 'a` qui renvoie le maximum des étiquettes des nœuds de l'arbre.

Question 13

Une branche est un chemin de la racine vers une feuille. Le poids d'une branche est la somme des étiquettes des nœuds qui la composent. Écrire une fonction `max_somme_branche : int arbre -> int` qui renvoie le poids de la branche de poids maximal.