TP12 : Corrigé

1. On obtient l'erreur suivante :

```
Error: This expression has type char arbre/1
but an expression was expected of type 'a arbre/2
Hint: The type arbre has been defined multiple times in this toplevel
session. Some toplevel values still refer to old versions of this
type. Did you try to redefine them?
```

   À la question : Did you try to redefine them ? on répond bien sûr oui. C'est d'ailleurs là qu'est le problème : on a définit deux types `'a arbre` différents portant le même nom. Pour résoudre le problème, on ferme et ré-ouvre le fichier (le top-level suffit parfois).

2. La fonction `appartient` indique a priori si l'élément `x` est présent dans l'arbre `a`. Quand on l'évalue, on obtient le warning (qui témoigne en fait d'une erreur) suivant :

```
Line 4, characters 3-15:
4 |   |Noeud(_,g,d) -> (appartient g x) || (appartient d x);;
       ^^^^^^^^^^^^
Warning 11: this match case is unused.
```

   C'est normal : le cas `Noeud(x,g,d)` filtre n'importe quel arbre non vide quelle que soit la valeur de sa racine car le `x` qui y est présent est indépendant de l'argument `x` de la fonction. Par conséquent, le dernier cas n'est jamais utilisé. On modifie le deuxième cas comme suit :

```
let rec appartient a x = match a with
  |Vide -> false
  |Noeud(y,g,d) when y = x -> true
  |Noeud(_,g,d) -> (appartient g x) || (appartient d x)
```

   La fonction `nb_differents` détermine le nombre de noeuds dans un arbre portant une étiquette égale (le nom de la fonction est donc très peu pertinent !) au deuxième argument. Ici, même pas besoin d'évaluer pour se rendre compte du problème : `val` est en gras car c'est un mot clé du langage. Il ne peut donc pas être utilisé comme identifiant.

   Une fois cette erreur corrigée, on obtient après évaluation un warning de pattern-matching non exhaustif. Le filtrage est en fait exhaustif mais comme toutes les clauses sont gardées, ce fait ne peut pas être automatiquement détecté. Pour supprimer le warning on corrige ainsi :

```
let rec nb_differents a x = match a with
  |Vide -> 0
  |Noeud(r,g,d) when r = x -> 1 + nb_differents g x + nb_differents d x
  |Noeud(r,g,d) (* when r <> x *) -> nb_differents g x + nb_differents d x
```

   La fonction `hauteur` détermine la hauteur de l'arbre en entrée. Son évaluation provoque l'erreur :

```
Error: Unbound value hauteur
Hint: If this is a recursive definition, you should add the 'rec' keyword on line 1
```

Le problème est effectivement l'oubli du mot clé `rec`.

La fonction `inserer` cherche à insérer un élément dans un arbre ayant la propriété d'un tas min de telle sorte à conserver cette propriété et augmenter la hauteur du sous-arbre le moins haut dès que possible. La première erreur obtenue est :

```
Line 4, characters 38-39:
4 |     |Noeud(_,_,_) -> 1 + max (hauteur g) (hauteur d)
                                          ^

Error: Unbound value g
```

Elle signifie que l'identifiant `g` n'a pas été lié dans cette ligne et c'est effectivement le cas. C'est aussi le cas pour `d` dans la même ligne. Après correction, l'erreur suivante est :

```
Line 8, characters 39-74:
8 |                       if hd <= hg then min(r, x), g, inserer_1 d max(r, x)
                                           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Error: This expression has type 'a * 'b * 'c
       but an expression was expected of type 'd arbre
```

C'est normal : il manque le constructeur `Noeud` pour encapsuler le triplet souligné par les chevrons. Après correction aux deux endroits impliqués on obtient l'erreur de type :

```
Line 8, characters 45-54:
8 |                   if hd <= hg then Noeud(min(r, x), g, inserer_1 d max(r, x))
                                             ^^^^^^^^^

Error: This expression has type 'a * 'b -> 'a * 'b
       but an expression was expected of type 'b
       The type variable 'b occurs inside 'a * 'b -> 'a * 'b
```

Le problème est qu'on a donné à `min` le couple `(r,x)`. Grâce au polymorphisme, la fonction `min` peut prendre en entrée deux couples mais comme on ne lui en donne qu'un, `min (r,x)` est en fait une fonction partielle qui prend en entrée un couple et renvoie le plus petit entre ce couple et le couple `(r,x)`. Ainsi, `min (r,x)` est de type `'a*'b -> 'a*'b` comme indiqué dans l'erreur, alors qu'on attend un élément comme racine de l'arbre. Il faut remplacer `min (r,x)` par `min r x` pour calculer le minimum voulu. Le même problème survient à la ligne d'après et pour `max`. Après correction on obtient :

```
Line 8, characters 57-66:
8 |                       if hd <= hg then Noeud(min r x, g, inserer_1 d (max r x))
                                                             ^^^^^^^^^

Error: Unbound value inserer_1
Hint: Did you mean inserer?
```

La lecture de l'erreur donne la solution. À ce stade, si on a corrigé uniquement les erreurs indiquées la fonction `inserer` est :

```
let rec inserer a x =
  let rec hauteur a = match a with
    |Vide -> 0
    |Noeud(_,g,d) -> 1 + max (hauteur g) (hauteur d)
  in match a with
```

```
    |Vide -> Noeud(x, Vide, Vide)
    |Noeud(r,g,d) -> let hg = hauteur g and hd = hauteur d in
                     if hd <= hg then Noeud(min r x, g, inserer d (max r x))
                     else Noeud(min r x, inserer (max r x), d)
```

Il reste encore une erreur :

```
Line 9, characters 50-59:
9 |                       else Noeud(min r x, inserer (max r x), d);;
                                                      ^^^^^^^^^
Error: This expression has type 'a but an expression was expected of type
         'a arbre
       The type variable 'a occurs inside 'a arbre
```

La fonction `inserer` prend en entrée dans cet ordre un `'a arbre` et un élément de type `'a`. Or, l'erreur indique qu'il y a une occurrence où on a oublié de donner l'arbre qui devrait être le premier argument. Vu la spécification de la fonction, il faut remplacer `inserer (max r x)` par `inserer g (max r x)` et cette fois la fonction s'évalue sans erreur.

3. La fonction `transposer` devrait transposer un graphe donné par listes d'adjacence. Après évaluation, on obtient l'erreur suivante :

```
Line 4, characters 18-20:
4 |     |t::q -> g[t] <- s::g[t] ; renverser_aretes g q
                      ^^
Error: Syntax error
```

L'erreur de syntaxe n'est pas exactement celle pointée par les chevrons mais elle est proche : `g` est un tableau donc `g[t]` n'est pas syntaxiquement correct : il faut écrire `g.(t)`. Après avoir corrigé les trois occurrences de cette erreur on obtient l'erreur suivante :

```
Line 11, characters 4-6:
11 |     gt;;
         ^^
Error: Syntax error
```

La localisation de l'erreur est un peu vague. Le réflexe à avoir lorsqu'il y a une erreur de syntaxe vague est d'utiliser l'indentation automatique ; souvent, il y aura un défaut dans cette indentation qui précisera l'endroit où se trouve l'erreur de syntaxe. Ici, lorsqu'on indente automatiquement on obtient :

```
let transposer G =
let rec renverser_aretes g s l = match l with
  |[] -> ()
  |t::q -> g.(t) <- s::g.(t) ; renverser_aretes g q
in
let n = Array.length G;
        let gt = Array.make n [] in
        for i = 0 to n-1 do
          renverser_aretes gt i G.(i)
        done
          gt
```

Les lignes 7 et suivantes ne sont pas indentées comme on l'attendrait et c'est normal car entre la ligne 6 et la ligne 7 se trouve une erreur de syntaxe : la fin de la ligne 6 devrait être un `in` et pas un `;`. De même, la ligne 11 est mal indentée par rapport à la ligne 10 (elles devraient être alignées). C'est normal : il manque un `;` à la fin de la ligne 10. Après correction on a :

```
Line 1, characters 15-16:
1 | let transposer G =
                   ^
Error: Unbound constructor G
```

L'erreur signale que, comme `G` commence par une majuscule, ce devrait être un constructeur (seuls identifiants autorisés à commencer par une majuscule). Or, rien dans le code ne définit un constructeur qui s'appelle `G`. Il suffit de changer le nom du graphe pour corriger. Erreur suivante :

```
4 |     |t::q -> g.(t) <- s::g.(t) ; renverser_aretes g q
                                     ^^^^^^^^^^^^^^^^^^^

Error: This expression has type int list -> unit
       but an expression was expected of type unit
```

La fonction `renverser_aretes` prend en entrée trois arguments et on ne lui en a donné que deux donc `renverser_aretes g q` est une fonction prenant en entrée une liste et renvoyant `unit` (car le premier cas du filtrage renvoie `()`). Après correction on obtient enfin :

```
let transposer g =
  let rec renverser_aretes g s l = match l with
    |[] -> ()
    |t::q -> g.(t) <- s::g.(t) ; renverser_aretes g s q
  in
  let n = Array.length g in
  let gt = Array.make n [] in
  for i = 0 to n-1 do
    renverser_aretes gt i g.(i)
  done;
  gt
```

On peut remplacer la fonction auxiliaire `renverser_aretes` par l'utilisation de `List.iter` ainsi :

```
let transposer (g:graphe) :graphe =
  let n = Array.length g in
  let gt = Array.make n [] in
  for i = 0 to n-1 do
    List.iter (fun s -> gt.(s) <- i::gt.(s)) g.(i)
  done;
  gt
```

La fonction `parcours` effectue un parcours en profondeur du graphe `g` à partir du sommet `s` tout en stockant les sommets dans l'ordre dans lequel ils sont parcouru. Première erreur :

```
Line 11, characters 16-18:
11 |   List.rev ordre;;
                    ^^
```

```
Error: Syntax error
```

L'erreur est peu explicite : on indente pour chercher un indice. Après indentation on obtient :

```
1  let parcours g s =
2    let n = Array.length g in
3    let vus = Array.make n false in
4    let ordre = ref [] in
5    let rec visite u =
6      if not vus.(u) then
7        vus.(u) = true;
8      ordre = u::ordre;
9      List.iter visite g.(u)
10       visite s;
11     List.rev ordre
```

Les lignes 8 et 9 devraient être indentées comme la 7. Le problème est la faible priorité de `;` : si on veut faire trois instructions dans le `if`, il faut les délimiter par des parenthèses ou `begin ... end`. On constate aussi que les lignes 10 et 11 sont mal indentées, ce à cause d'un `in` manquant permettant de définir localement la fonction `visite`. On corrige pour obtenir un code correctement indenté :

```
let parcours g s =
  let n = Array.length g in
  let vus = Array.make n false in
  let ordre = ref [] in
  let rec visite u =
    if not vus.(u) then
      begin
        vus.(u) = true;
        ordre = u::ordre;
        List.iter visite g.(u)
      end
  in visite s;
     List.rev ordre
```

Deux erreurs pour le prix d'une à l'itération suivante :

```
Line 8, characters 8-22:
8 |         vus.(u) = true;
            ^^^^^^^^^^^^^^

Warning 10: this expression should have type unit.
Line 9, characters 16-24:
9 |         ordre = u::ordre;
                    ^^^^^^^^

Error: This expression has type 'a list
       but an expression was expected of type 'b list ref
```

Dans une suite d'instruction `e1 ; e2 ; ... ; ek`, toutes les instructions sauf la dernière doivent être de type `unit`. Or ce n'est pas le cas de `vus.(u) = true` qui est de type `bool`. Le problème est qu'on a utilisé `=` (qui sert à tester l'égalité) plutôt que `<-` pour modifier le contenu de `vus.(u)`. La deuxième

est due à une mauvaise façon de modifier la référence `ordre`. Cette erreur est également présente à la dernière ligne du code. Le code complètement corrigé est :

```
let parcours g s =
  let n = Array.length g in
  let vus = Array.make n false in
  let ordre = ref [] in
  let rec visite u =
    if not vus.(u) then
      begin
        vus.(u) <- true;
        ordre := u::(!ordre);
        List.iter visite g.(u)
      end
  in visite s;
     List.rev !ordre
```

Pour visiter tous les sommets du graphe en profondeur et pas uniquement ceux accessibles depuis un sommet donné , on modifie en :

```
let parcours (g:graphe) :int list =
  let n = Array.length g in
  let vus = Array.make n false in
  let ordre = ref [] in
  let rec visite (u:int) :unit =
    if not vus.(u) then
      begin
        vus.(u) <- true;
        ordre := u::(!ordre);
        List.iter visite g.(u)
      end
  in for s = 0 to n-1 do
       visite s;
     done;
     List.rev !ordre
```

4. Pour la fonction `decompte`, l'évaluation ne cause pas de problème. En revanche, lorsqu'on évalue `let _ = decompte 8`, on obtient l'information suivante :

```
Stack overflow during evaluation (looping recursion?).
```

C'est normal : l'appel récursif déclenché par `decompte m` se fait aussi sur l'entrée `m` et donc cette fonction ne termine pas. Une fonction qui termine et respectant la spécification est :

```
let rec decompte n =
  if n = 0 then [0]
  else n::(decompte (n-1))
```

Pour la fonction `quotients`, pas d'erreur non plus à l'évaluation mais lors de l'évaluation du test :

```
Exception: Invalid_argument "index out of bounds".
```

Le problème n'est pas évident. Un bon réflexe à avoir ici est d'aller consulter la documentation de `Array.iter`. On constate alors que cette fonction applique un traitement aux **éléments** d'un tableau et pas aux **indices** d'un tableau. Observons donc ce qu'il se passe lorsqu'on calcule `quotients 3 7`.

On commence par construire le tableau `q = [|7;7;7|]`. Puis, on considère le premier élément de ce tableau : 7. Il n'est pas nul donc on tente de remplacer `q.(7)` par `q.(3)` ; or, ces deux cases sont en dehors du tableau `q` et on comprend maintenant l'exception.

Après avoir corrigé ce problème avec `Array.iteri`, ou en modifiant complètement la fonction à l'aide d'une boucle, on constate qu'il reste un problème lors de l'évaluation du test : contrairement à ce qui est indiqué dans la spécification, `quotients` ne renvoie rien. On corrige pour obtenir :

```
let quotients (n:int) (k:int) :int array =
  let q = Array.make n k in
  Array.iteri (fun i e -> if (i=0) then ()
                          else q.(i) <- (q.(i-1))/2) q;
  q
```

5. A priori, `filtrer` renvoie la liste en entrée privée des éléments strictement plus petits que le seuil. La première erreur est une erreur de syntaxe désignant le premier `else` : c'est normal, juste avant se trouve un `;` parasite qui fait que le premier `else` n'est rattaché à aucun `if`. Après correction on obtient :

```
Line 6, characters 20-22:
6 |     then (List.hd l)::(filtrer seuil List.tl l)
                        ^^
Error: This variant expression is expected to have type unit
       The constructor :: does not belong to type unit
```

Comportons nous comme un inféreur de type : quand la liste est vide, on évalue `Printf.printf "Fini"` qui est de type `unit`. On en déduit que `filtrer` renvoie un type `unit`. Donc en particulier, `filtrer seuil List.tl l` devrait être de type `unit` ; or on l'utilise comme une liste vu qu'on le fait précéder du constructeur `::`. Vu la spécification, on corrige en renvoyant `[]` dans le cas où `l` est vide. Puis :

```
Line 6, characters 23-30:
6 |     then (List.hd l)::(filtrer seuil List.tl l)
                           ^^^^^^^
Error: This function has type 'a -> 'a list -> 'a list
       It is applied to too many arguments; maybe you forgot a `;'
```

L'erreur est explicite : en l'état, `filtrer` est appliqué à 3 arguments au lieu de 2 (`seuil`, `List.tl` et `l`). On corrige les occurrences de cette erreur à l'aide de parenthèses puis :

```
Line 7, characters 9-16:
7 |     else filtrer seuil (List.tl l)
             ^^^^^^^
Error: This function has type 'a -> 'a list -> 'a list
       It is applied to too many arguments; maybe you forgot a `;'.
```

On a toujours un problème de nombre d'arguments pour `filtrer` dans le dernier appel fait à cette fonction. C'est du au fait qu'à la dernière ligne, `filtrer` est appliqué à `seuil`, `List.tl l`, `filtrer`, `5` et `[1;4;5;1;2;5;4;21;2;54;1;2;5;6]` car les espaces et passages à la ligne ne sont pas la syntaxe permettant de définir une expression suite à une autre expression. On corrige en :

```
let rec filtrer seuil l =
  if l = []  then []
  else if List.hd l >= seuil then (List.hd l)::(filtrer seuil (List.tl l))
  else filtrer seuil (List.tl l)

let _ = filtrer 5 [1;4;5;1;2;5;4;21;2;54;1;2;5;6]
```

6. La fonction `lire_lignes` devrait permettre la lecture des lignes d'un fichier et leur stockage dans une liste. On obtient l'erreur :

```
Lines 5-7, characters 4-8:
5 | ....while true do
6 |       lignes := input_line flux::(!lignes)
7 |     done
Error: This expression has type unit but an expression was expected of type
         string list
```

C'est normal : l'expression dans le `try` et les expressions rattrapant les erreurs doivent avoir le même type ; or l'expression dans le `try` est une boucle `while` donc de type `unit` alors que le rattrapage de `End_of_file` renvoie une liste. Une correction possible :

```
let lire_lignes (fichier:string) : string list =
  let lignes = ref [] in
  let flux = open_in fichier in
  begin
    try
      while true do
        lignes := input_line flux::(!lignes)
      done
    with End_of_file -> close_in flux
  end;
  List.rev !lignes
```

La fonction s'évalue et se comporte comme attendu sur l'exemple. Une version récursive :

```
let lire_lignes (fichier:string) :string list =
  let flux = open_in fichier in
  let rec lire (acc:string list) :string list =
    try
      let l = input_line flux in lire (l::acc)
    with
    |End_of_file -> List.rev acc
  in let lignes = lire [] in
     close_in flux;
     lignes
```

7. A priori, `creer_file` devrait créer une file de priorité vide. On obtient une erreur de syntaxe provoquée par le fait qu'on essaie d'initialiser le champ `fin` avec `<-` plutôt que `=`. Puis on obtient une erreur de type car le tableau des priorités doit contenir des flottants. Après correction :

```
let creer_file n = {fin = 0; priorites = Array.make n 0.; cles = Array.make n 0}
```

La fonction `remonte` tamise vers le haut un élément (c'est-à-dire l'échange avec son père tant qu'il est plus petit que ce père). La fonction semble s'évaluer correctement mais le test sur `f1` échoue :

```
Line 1, characters 8-20:
1 | let _ = remonte f1 2; f1;;
            ^^^^^^^^^^^^
Warning 21: this statement never returns (or has an unsound type.)
```

Le problème vient d'un mauvais parenthésage dans `remonte` : on veut faire deux instructions dans le `then` mais comme elle ne sont pas parenthésées, `remonte f j` est en fait systématiquement exécuté et par conséquent les appels récursifs à `remonte` ne s'arrêtent jamais. On corrige ainsi :

```
let rec remonte f i =
  let j = parent i in
  if i > 0 && f.priorites.(i) < f.priorites.(j)
  then (echange f i j; remonte f j)
```

La fonction `inserer_file` vise à insérer un nouvel élément dans une file de priorité min. La première erreur est tout à fait explicite :

```
Line 5, characters 2-12:
5 |   f.fin <- i;
      ^^^^^^^^^^
Error: The record field fin is not mutable
```

On modifie donc la définition de l'enregistrement `file_prio` pour faire de `fin` un champ mutable de sorte à pouvoir modifier la partie active de la file :

```
type file_prio = {mutable fin : int; priorites : float array; cles : int array}
```

Après cette correction, la fonction s'évalue sans erreur mais lors du dernier test on obtient :

```
Line 1, characters 8-20:
1 | let _ = inserer_file f2 (8,2.3) f2;;
            ^^^^^^^^^^^^
Error: This function has type file_prio -> int * float -> unit
       It is applied to too many arguments; maybe you forgot a `;'.
```

Lire l'erreur permet de corriger le problème : il manque un `;` entre l'appel à `inserer` et l'observation de `f2` une fois modifiée. Après l'avoir ajoutée, relancer les tests produit :

```
Exception: Invalid_argument "index out of bounds".
```

En observant plus attentivement `f2` on constate qu'initialement c'est une file pouvant contenir au plus 5 éléments, qu'elle en contient initialement 4 et qu'on en ajoute 2 : la fonction `inserer` oublie de prévoir le cas d'une insertion dans une file pleine. On corrige :

```
let inserer_file f (elem, prio) =
  let nb_elem = f.fin+1 and nb_elem_max = Array.length f.cles in
  if nb_elem = nb_elem_max then failwith "la file est pleine"
  else
    begin
      let i = f.fin + 1 in
      f.cles.(i) <- elem;
      f.priorites.(i) <- prio;
      f.fin <- i;
      remonte f i
    end
```

Pour écrire la fonction `extraire_min` on peut commencer par écrire une fonction récursive `entasser` qui entasse un tas min à partir d'une certaine position (en comparant l'élément à cette position à ces éventuels fils et en l'échangeant avec son plus petit fils le cas échéant) puis écrire :

```
let extraire_min (f:file_prio) :int =
  let m = f.cles.(0) in (* qui existe car la file est non vide *)
  echange f 0 f.fin;
  f.fin <- f.fin -1;
  entasser f 0;
  m
```