TP : Erreurs en Ocaml

Cours : Galerie de messages d'erreurs

L'objectif de cette partie est d'expliquer les messages d'erreurs fréquents que vous pouvez lire dans l'interpréteur Ocaml et de donner des idées qui permettent fréquemment de les corriger.

Les messages en question sont répartis dans trois grandes catégories (les messages signalent d'ailleurs à quelle catégorie ils appartiennent) : erreurs (Error), avertissements (Warning) et exceptions (Exception). Les exceptions sont de nature différentes des erreurs et warnings : il peut être tout à fait voulu qu'un programme provoque une exception dans certaines conditions. Les erreurs et les warnings quant à eux dénotent **toujours** un problème, de syntaxe ou de conception : il ne faut **jamais** en laisser.

Erreurs de type

```
Error: This expression has type <type1> but an expression was expected of type <type2>
```

Cette erreur survient lorsqu'un conflit de types intervient. L'expression incriminée est désignée par des chevrons. Par exemple, le code suivant déclenche une telle erreur en désignant l'expression `2.5` : cette dernière est de type `float` mais devrait être de type `int` car `+` ne permet d'additionner que des entiers.

```
let a = 2.5 + 8
```

Cette erreur peut facilement apparaître :

- Lorsqu'on mélange des flottants et des entiers, auquel cas il faut bien vérifier que les opérations effectuées concernent bien le bon type. Il n'y a pas d'exponentiation sur les `int`.
- Lorsqu'on utilise des références ou des types option : les types `'a`, `'a option` et `'a ref` sont différents. Au passage, attention à ne pas mélanger `=` (test d'égalité), `:=` (modification du contenu d'une référence) et `<-` (modification d'une case d'un tableau ou d'un champ d'un enregistrement).
- Dans un `if e1 then e2 else e3` : les expressions `e2` et `e3` doivent avoir même type. Même principe dans un `try ... with ...` : les expressions dans le `try` et rattrapant les exceptions doivent avoir même type.
- Dans une expression de la forme `e1 ; e2 ; ... ; en` : tous les `ei` sauf éventuellement la dernière doivent avoir type `unit`. Dans ce cas, on n'aura en fait qu'un warning mais a priori le code est faux :

  ```
  Warning 10: this expression should have type unit.
  ```

- Lorsqu'on utilise incorrectement une fonction (mauvais nombre / types d'arguments).
- A cause de virgules parasites. Par exemple : virgules au lieu de point-virgules comme séparateurs dans une liste ou tableau / appel d'une fonction `f` à plusieurs arguments avec `f (x,y)` au lieu de `f x y`.

Pour résoudre le problème, il faut analyser le type des objets manipulés pour identifier quelle expression modifier. Introduire des annotations de type sur les arguments des fonctions peut aider.

Cas particulier : conflit entre un type et lui même

On peut parfois obtenir l'erreur déstabilisante suivante, indiquant un conflit entre un type et lui même :

```
Error:This expression has type <type>/1 but an expression was expected of type <type>/2
```

Cette erreur est parfois accompagnée du message suivant, qui explicite le problème :

```
Hint: The type <type> 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?
```

Cette erreur apparaît lorsqu'on a défini deux types avec le même nom ou qu'on a modifié la définition d'un type en cours de route. Par exemple on obtient cette erreur avec le code suivant :

```
type toto = A | B;;
let a = A;;
type toto = A | B;;
let b = B;;
a = b;;
```

En effet, la première définition du type toto est indépendante de la deuxième définition du type toto. Ainsi, a est de type toto selon la ligne 1 et b est de type toto selon la ligne 3 et ces deux types ont beau être sémantiquement les mêmes, pour l'inférence de types ce n'est pas la même chose et on est donc en train de mettre un égal entre deux expressions de types différents ce qui est interdit. Une solution est de sauvegarder et fermer son fichier et de le relancer dans Emacs.

Cas particulier : erreurs de type impliquant des fonctions

Les erreurs suivantes sont des erreurs de types fréquentes lorsqu'on utilise des fonctions :

- ```
  Error: This expression has type 'a -> 'b but an expression was expected of type 'b
  ```

  C'est exactement celle présentée dans le cas général. Généralement, c'est le signe qu'on a oublié un argument dans une fonction, comme par exemple pour :

  ```
  let rec jongler u v w =
    if u = 0 then ()
    else jongler v (u-1)
  ```

- A l'inverse, lorsqu'on donne trop d'arguments à une fonction, on peut produire l'erreur :

  ```
  Error: This function has type <type_fonctionnel>
    It is applied to too many arguments; maybe you forgot a `;'.
  ```

  Cela peut arriver en particulier lorsqu'on oublie de séparer divers blocs à l'aide ;; Par exemple dans le code suivant affiche 7 est interprété comme faisant partie de la définition de affiche :

  ```
  let affiche i = print_int i
  affiche 7
  ```

  Pour régler ce problème, on peut soit séparer les blocs par un double point-virgule, soit faire de la deuxième ligne une déclaration top-level via let _ = affiche 7.

- Une dernière erreur fréquente vis-à-vis des fonctions est :

  ```
  Error: This expression has type <type_non_fonctionnel>
      This is not a function; it cannot be applied.
  ```

  Elle survient lorsque l'inférence de types a déduit que le type d'un identifiant n'est pas fonctionnel mais que cet identifiant est utilisé (souvent à cause d'une erreur de syntaxe) comme une fonction. Exemple :
  ```

```
let ploum t =
 let n = Array.length t in
 let i = Random.int n in
 t(i)
```

L'oubli du point dans la dernière ligne fait que cette dernière est interprétée comme étant l'application de la fonction t à l'argument i (entre des parenthèses inutiles). Mais comme on a appliqué Array.length à t, on sait que t n'est pas une fonction, d'où l'erreur.

De manière générale, quand on utilise une fonction native et qu'il y a une erreur, ne pas hésiter à relire la documentation de façon à retrouver l'ordre et le nombre des arguments ainsi que la spécification.

Erreurs de syntaxe

```
Error: Syntax error
```

Cette erreur n'est pas toujours facile à corriger car elle peut apparaître pour de nombreuses raisons et la zone désignée par les chevrons comme étant cause de l'erreur n'est pas toujours celle causant le problème à première vue. Quelques causes classiques (non exhaustives) générant cette erreur :

- Utilisation d'un mot clé réservé par le langage comme nom de variable. Sous Emacs cette situation est facile à détecter car la coloration syntaxique met les mots clés réservés en gras.
- Oubli ou surplus de parenthèses.
- Oubli du in allant avec un let lors d'une définition locale.
- Oubli d'un ; séparant deux expressions.
- Mauvaise délimitation des expressions dans un if ... then ... else. On rappelle que

```
let toto t i j =
  if t.(i) < t.(j) then t.(i) <- 0;  t.(j) <- 1
  else t.(i) <- 5
```

est interprété comme suit : si t.(i) < t.(j) alors la valeur de t.(i) est remplacée par 0. Puis, dans tous les cas, t.(j) est remplacé par 1. Puis, on rencontre un else qui n'est rattaché à aucun if, d'où l'erreur. Remarquez que si il n'y avait pas de else, il n'y aurait pas d'erreur de syntaxe, ce qui serait bien embêtant si l'objectif était de n'écraser t.(j) que lorsqu'il est plus grand que t.(i) !

Utiliser la touche tab sur chacune des lignes du code permet sous Emacs de l'indenter automatiquement : les défauts d'indentation permettent très souvent de repérer plus précisément où se situent les erreurs de syntaxe. Dans le cas d'une mauvaise délimitation dans un if ... then .. else, on rappelle qu'il faut utiliser begin ... end ou un couple de parenthèses pour modifier les priorités.

Warnings relatifs aux filtrages

Il y en a deux principaux : cas non utilisé et filtrage non exhaustif. Le premier est signalé via :

```
Warning 11: this match case is unused.
```

C'est le symptôme d'un code au mieux mal conçu, au pire qui ne fait pas ce qu'on veut. Par exemple :

```
let egal x y = match y with
 |x -> true
 |_ -> false
```

signale que le deuxième cas du filtrage ne sera jamais utilisé. En effet, la variable x en ligne 2 est une variable fraîche qui masque le x de la ligne 1. En particulier, y peut toujours être filtré par le x en ligne 2 et cela provoque le comportement a priori non désiré suivant : egal 5 6 renvoie true.

Le second est signalé par :

```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched: <exemple>
(However, some guarded clause may match this value.)
```

Il signale qu'un cas n'est pas couvert et précise un cas qui ne l'est pas pour aider à la correction. Il peut apparaître parfois alors même que tous les cas semblent couverts mais chacun des cas du filtrage est "gardé" par un when. Par exemple il n'y a pas de problème sémantique avec le code suivant

```
let est_pair n = match n with
 |n when n mod 2 = 0 -> true
 |n when n mod 2 <> 0 -> false
```

mais un warning de pattern-matching non exhaustif sera tout de même signalé. Pour corriger la chose, il suffit de ne pas garder la dernière clause qui de toutes façons correspond bien à tous les cas restants si le filtrage est exhaustif. Remarque : pour est_pair, la bonne chose à faire est de toutes façons d'utiliser un if ... then ... else ; un filtrage est tout à fait inutile.

Erreurs de définition

Sont rangés dans cette catégorie toutes les erreurs du type :

```
Error: Unbound <quelque chose> <nom du quelque chose>
```

Elle signifie qu'un identifiant utilisé dans le code n'a pas été lié à une expression. Elle intervient :

- Lorsqu'on a oublié d'évaluer une fonction f et qu'on cherche à l'utiliser. On obtient alors l'erreur :

  ```
  Error: Unbound value f
  ```

  accompagnée parfois de l'information suivante :

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

  Solution : corriger la fonction f (souvent l'évaluation a échoué à cause d'une autre erreur et c'est pour ça que f n'est pas définie) puis l'évaluer. Ajouter rec le cas échéant.

- Lorsqu'une variable n'est pas définie. C'est souvent le signe qu'on a oublié un in.

- Lorsqu'on utilise un constructeur non défini auquel cas on obtient l'erreur :

  ```
  Error: Unbound constructor <nom>
  ```

  Souvent, cette erreur se produit car on a donné un nom à quelque chose commençant par une majuscule (à part les noms de module et les identifiants des constructeurs RIEN ne commence par une majuscule

en Ocaml). Elle peut aussi être détectée via la coloration syntaxique. Le fix est alors simple : changer les identifiants utilisés !

Autres erreurs

Des erreurs de parsing peuvent également être signalées :

```
Error: This '(' might be unmatched
```

La parenthèse peut être remplacée par un autre délimiteur, par exemple une accolade. Cette erreur peut parfois ne même pas être affichée car Emacs refusera peut être d'évaluer votre fonction en vous prévenant que : `The expression after the point is not well braced`.

Une dernière erreur relativement transparente peut apparaître à l'exécution :

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

C'est le signe que votre code évalue une fonction sur une entrée pour laquelle elle ne termine pas. Il faut vérifier en particulier la correction des fonctions récursives (et leurs cas d'arrêts) et la terminaison des boucles while (par exemple, vérifier que l'indice d'une boucle while est mis à jour dans celle ci).

Exceptions classiques

Ces dernières sont responsables de messages lors de l'exécution de vos fonctions sur certaines entrées :

- L'exception Failure est paramétrée par un type string. Un exemple natif classique intervient lorsqu'on tente d'accéder à la tête d'une liste vide (exception similaire pour la queue) :

  ```
  Exception: Failure "tl"
  ```

  On rappelle que ce constructeur peut être utilisé pour afficher le message que vous voulez. Par exemple, Failure "toto" est une expression de type exception tout à fait valide. Pour lever cette exception, on peut faire comme pour toutes les exceptions :

  ```
  raise (Failure "toto")
  ```

  ou utiliser le mot clé failwith :

  ```
  failwith "toto"
  ```

- L'exception Not_found signale qu'un élément n'a pas été trouvé. La fonction Hashtbl.find par exemple est suceptible de la lever. Il existe souvent des versions des fonctions levant Not_found permettant de renvoyer None lorsque l'élément n'est pas trouvé et Some v sinon (Hashtb.find_opt par exemple).

- L'accès ou la modification en dehors des bornes d'un tableau provoque l'exception :

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

- L'exception End_of_file est levée par les fonctions de lecture (input_line par exemple) dans un fichier lorsque la fin du fichier est atteinte. Généralement on rattrape cette exception à l'aide de try ... with pour pouvoir fermer le fichier en fin de parcours et exploiter le résultat des lectures.

Exercices : Identification et correction d'erreurs

Copier le répertoire TP_erreurs_Ocaml dans votre espace personnel.

1. Evaluer une à une les lignes de 1_arbres_types.ml. Modifier le type 'a arbre et l'arbre a de sorte à remplacer le constructeur Vide par V. Réévaluer, commenter et résoudre le problème.
2. Pour chacune des fonctions dans 2_arbres.ml :
   a) Inférer sa spécification à l'aide des éléments à disposition.
   b) Tant qu'elle ne s'évalue pas sans erreur ni warning : l'évaluer, lire et comprendre le message d'erreur, corriger **uniquement** l'erreur indiquée.
3. On considère à présent les fonctions du fichier 3_graphes.ml :
   a) Leur appliquer le même traitement qu'à la question 2.
   b) Réécrire transposer en utilisant List.iter.
   c) Modifier parcours de sorte à effectuer un parcours en profondeur de l'ensemble du graphe.
4. Pour chacune des fonctions du fichier 4_evaluations.ml :
   a) Evaluer la fonction. Y a-t-il un problème ?
   b) Evaluer le test accompagnant la fonction. Expliquer et corriger.
5. a) Appliquer le même traitement à la fonction de 5_filtre.ml qu'à la question 2.
   b) Réécrire la fonction filtrer à l'aide de List.filter.
6. a) Appliquer le traitement de la question 2 à la fonction de 6_lecture.ml.
   b) Vérifier le bon fonctionnement de la fonction lire_lignes corrigée. Est-on satisfait ?
   c) Réécrire lire_lignes de manière récursive.
7. On s'intéresse à présent aux fonctions du fichier 7_file_prio.ml. Une file de priorité y est implémentée à l'aide d'un tas min (décomposé en deux morceaux : les clés et leurs priorités ; si une clé est en position i dans cles, sa priorité est en position i dans priorites) dont la partie "active" est située entre les indices 0 et fin inclus. Les tableaux cles et priorites sont de même taille, fixée à la création de la file.
   a) Pour chacune des fonctions, appliquer le traitement décrit en question 2 et vérifier que le comportement des fonctions corrigées est cohérent avec les spécifications inférées sur l'exemple proposé.
   b) (bonus) Ecrire une fonction extraire_min renvoyant l'élément de priorité minimal dans une file et modifiant cette dernière de sorte à le supprimer.
8. (bonus) On cherche à implémenter l'algorithme de Dijkstra sur un graphe $G = (S, A)$ représenté par listes d'adjacence et dont les sommets sont numérotés de 0 à $|S| - 1$. Pour ce faire, on reprend le fichier file_prio.ml et on modifie le type file_prio en y ajoutant un champ emplacements tel que :
   - Si la clé (sommet ici) i n'est pas dans la file, emplacements.(i) = -1.
   - Sinon, cles.(emplacements.(i)) = i et la priorité correspondante est dans priorites.(emplacements.(i)).

   Autrement dit, emplacements permet de retrouver l'emplacement d'une clé dans le tas ce qui est nécessaire pour pouvoir en modifier facilement la priorité.
   a) Ecrire une fonction diminuer_prio : file_prio -> int -> float -> unit telle que diminuer_prio f s p modifie la priorité de s dans f en la remplaçant par p si p est inférieure à la priorité actuelle de s.
   b) En déduire une fonction dijkstra prenant en entrée un graphe G et un sommet s de G et déterminant pour tout sommet u la distance minimale de s vers u.