Syntaxe : Erreurs en OCaml

L'objectif de ce document 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 généralement 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
```

Causes fréquentes

Les causes classiques de cette erreur sont :

- Le mélange de flottants et d'entiers, auquel cas il faut vérifier que les opérations effectuées concernent bien le bon type et faire des conversions au besoin. 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).
- La non concordance des types dans une expression conditionnelle. Dans `if e1 then e2 else e3` les expressions `e2` et `e3` doivent avoir même type. Même chose dans un `try ... with ...` : les expressions dans le `try` et rattrapant les exceptions doivent avoir même type.
- Une expression qui devrait être une instruction n'est pas de type `unit`. Dans une expression de la forme `e1 ; e2 ; ... ; en` toutes les `ei` sauf éventuellement la dernière doivent avoir type `unit`. Si ce n'est pas le cas, on aura en fait le warning suivant (et a priori, le code est faux) :

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

- L'utilisation incorrecte d'une fonction (mauvais nombre / types d'arguments).
- Un mauvais ou une absence de parenthésage.
- Des virgules parasites. Par exemple : virgules au lieu de point-virgules comme séparateurs dans une liste ou tableau. Autre exemple : appel d'une fonction `f` 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
let _ = (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. 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 puis de le relancer.

Cas particulier : erreurs de type impliquant des fonctions

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

- L'erreur générique de type se décline fréquemment en :

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

  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 arrive en particulier lorsqu'on oublie de séparer deux expressions. Par exemple, bien qu'ils soient sur deux lignes, les arguments de `print_int` sont `i`, `affiche` et `7` dans ce code :

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

  Pour régler ce problème, il suffit de faire de la deuxième ligne une phrase 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. En évaluant directement une fonction dans l'interpréteur, ce dernier fournit l'ordre et le type de ses arguments ce qui permet d'aller encore plus vite qu'avec la documentation pour ces informations.

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 ainsi : 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` : erreur. Remarque : si on supprime la ligne 3, il n'y a pas d'erreur de syntaxe. Mais, si l'objectif du code était de n'écraser `t.(j)` que lorsqu'il est plus grand que `t.(i)`, le code est alors faux de manière silencieuse.

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.

Erreurs de définition

Sont rangées dans cette catégorie toutes les erreurs du type :

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

Cela signifie qu'un identifiant utilisé dans le code n'a pas été lié à une expression. Cela arrive notamment :

- 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
  ```

  Le debug passe généralement par la correction de 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). Ajouter **`rec`** le cas échéant.

- Lorsqu'un identifiant n'est pas défini. 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 écrit un identifiant commençant par une majuscule (à part les noms de module et les identifiants des constructeurs RIEN ne commence par une majuscule en OCaml). On peut la détecter via la coloration syntaxique. Pour la corriger, il suffit de changer le nom des identifiants utilisés de sorte à ce qu'ils ne commencent pas par une majuscule.

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 dans le minibuffer 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 ?)
```

Ce peut être le signe que votre code évalue une fonction récursive sur une entrée pour laquelle elle ne termine pas. Parfois cette erreur survient alors même que le code est correct : c'est juste que vous utilisez une fonction récursive sur une entrée pour laquelle la pile d'appels récursifs est trop grande. Pour corriger le problème dans ce cas, il faut soit rendre votre fonction récursive terminale (les méthodes pour ce faire ne sont pas au programme mais si vous savez le faire, faites), soit utiliser une autre stratégie.

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. Parfois, il apparaît alors même que tous les cas semblent couverts : cela arrive dès que 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 = 1 -> 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 d'écrire : `let est_pair n = (n mod 2) = 0`.

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 `exn` 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.