Programmation fonctionnelle

TP 2

Giron David thor@epitech.net

Résumé: Ce deuxième TP de programmation fonctionnelle est très important. Nous allons pratiquer ensemble les différentes structures de données d'OCaml et également découvrir les exceptions. À la fin de ce TP, vous serez capables de combler les derniers trous de votre projet `MyList`.

Table des matières

Chapitre I

TP 2

I.1 Les tuples

On rappelle qu'un tuple (ou "n-uplet") est une liste non-mutable. Une fois créé, un tuple ne peut en aucune manière être modifié. C'est l'ordre et le type des élements d'un tuple qui définit le type de ce tuple.

Prenons par exemple, le tuple suivant :

```
(42, "epitech", true)
```

Ce tuple est de type :

```
(int * string * bool)
```

Ce type se lit :

```
"int croix string croix bool"
```

I.1.1 Exercice 1

Considérons un étudiant représenté sous forme d'un tuple de type :

```
(string * string * int * bool)
```

Les élements composant ce tuple sont respectivement des types suivants :

- Le prénom (string)
- Le nom (string)
- La promotion (int)

- Si l'étudiant est close-compte ou pas (bool)

Écrivez les fonctions suivantes permettant de manipuler notre tuple :

- `get_first_name : ('a * 'b * 'c * 'd) -> 'a`
- `get_last_name : ('a * 'b * 'c * 'd) -> 'b`
- `get_promotion : ('a * 'b * 'c * 'd) -> 'c`
- `is_close : ('a * 'b * 'c * 'd) -> 'd`

Vos fonctions auront les comportements suivants :

```
# let etudiant = ("David", "GIRON", 2009, false);;
val etudiant : string * string * int * bool = ("David", "GIRON", 2009, false)
# get_first_name etudiant;;
- : string = "David"
# get_last_name etudiant;;
- : string = "GIRON"
# get_promotion etudiant;;
- : int = 2009
# is_close etudiant;;
- : bool = false
```

Pourquoi la fonction get_first_name est-elle de type :

('a * 'b * 'c * 'd) -> 'a

au lieu d'être de type :

(string * string * int * bool) -> string

?

- Définissez la fonction suivante :

`is_in_promo : ('a * 'b * 'c * 'd) -> 'c -> bool`

Cette fonction doit avoir le comportement suivant :

```
# let etudiant = ("David", "GIRON", 2009, false);;
val etudiant : string * string * int * bool = ("David", "GIRON", 2009, false)
# is_in_promo etudiant 2009;;
- : bool = true
# is_in_promo etudiant 2014;;
- : bool = false
```

- En utilisant la fonction `is_in_promo` que vous venez d'écrire, définissez la fonction récursive suivante :

  `is_in_promo_range : ('a * 'b * int * 'c) -> int -> int -> bool.`

Cette fonction doit avoir le comportement suivant :

```
# is_in_promo_range etudiant 2000 2010;;
- : bool = true
# is_in_promo_range etudiant 2000 2003;;
- : bool = false
# is_in_promo_range etudiant 2000 2000;;
- : bool = false
# is_in_promo_range etudiant 2010 2010;;
- : bool = false
# is_in_promo_range etudiant 2009 2009;;
- : bool = true
# is_in_promo_range etudiant 2010 2002;;
- : bool = false
```

I.2 Les records

On a vu que les tuples souffraient d'un inconvénient notable : Le compilateur tente d'inférer les types les plus généraux possibles, les tuples sont donc souvent victimes d'ambiguïté dans leur utilisation simple. En attendant de savoir réstreindre les types d'une fonction, une solution à ce problème pourrait être d'utiliser des records (ou "enregistrements" en français).

I.2.1 Exercice 2

Nous allons reprendre l'exercice précédent en utilisant cette fois des records au lieu des tuples. Commencez par définir un nouveau type record nommé "`etudiant`" contenant les champs suivants :

- first_name : string
- last_name : string
- promotion : int
- is_close : bool

Si vous êtes un peu perdus, relancez la video n° 3 du cours n° 2.
Pour atteindre un champ d'un record, on utilise la même syntaxe qu'en C :

```
1  # type point = {x : int; y : int};;
2  type point = { x : int; y : int; }
3  # let a = {x = 4; y = 12};;
4  val a : point = {x = 4; y = 12}
5  # a.x;;
6  - : int = 4
7  # a.y;;
8  - : int = 12
```

Réécrivez les fonctions de l'exercice précédent dont le type a été modifié en conséquence. Vous pouvez bien entendu modifier leurs noms si vous écrivez dans le même fichier que pour l'exercice précédent afin de ne pas avoir de conflits de symboles.

- `get_first_name : etudiant -> string`
- `get_last_name : etudiant -> string`
- `get_promotion : etudiant -> int`
- `is_close : etudiant -> bool`
- `is_in_promo : etudiant -> int -> bool`
- `is_in_promo_range : etudiant -> int -> int -> bool`.

I.3 Les variants

Les variants sont extrêmement répandus en OCaml. Il est donc essentiel que vous sachiez les utiliser. Je vous rappelle qu'un variant permet de définir un type et d'exprimer son domaine de définition (de la même manière que les enums en C) mais également de paramétrer les valeurs de ce domaine. De plus, contrairement au C, les valeurs (ou constructeurs) d'un variant s'évaluent vers eux-mêmes au lieu de s'évaluer vers des entiers.

I.3.1 Exercice 3

Nous allons réaliser un petit calendrier et quelques fonctions outils allant avec. Rappelez-vous que chaque valeur d'un variant est appelé un "constructeur" (pouvant être paramétré) dont l'identifiant doit obligatoirement commencer avec une majuscule.

Cet exercice peut vous paraître long à première vue, mais en réalité, il est très détaillé pour vous aider à mieux comprendre la notion. Au final, il n'y a que très peu de code à produire. Laissez-vous guider !

Je vous conseille d'écrire le code de cet exercice dans un fichier et de charger celui-ci dans l'interprète avec la directive `#use` que vous connaissez depuis le premier TP.

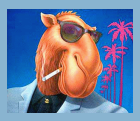
Jours et mois

- Écrivez un type variant nommé `day` énumerant les jours de la semaine
- Écrivez un type variant nommé `month` énumerant les mois de l'année

Nous représenterons nos dates sous forme d'un tuple ayant le type suivant :

```
(day * int * month * int)
```

Pour les deux fonctions ci-dessous, vous devez faire un filtrage introduit par le mot clef `function`. Référez-vous au cours si vous ne vous en souvenez plus.



La syntaxe du filtrage en début de fonction avec match est un peu plus simple à lire au debut, mais la forme avec "function" est plus proche du concept mathématique sous-jacent. Elle met mieux en avant qu'un filtrage sur un paramètre (en l'occurence le dernier pour des raisons que je ne nommerai pas ici) est avant tout la compartimentation du domaine de définition du type de ce paramètre. Chaque cas d'un filtrage correspond à une fonction définie pour un segment du domaine de depart.
C'est pourquoi j'encourage la syntaxe utilisant le mot clé "function".
Pour les plus curieux, étudiez le filtrage dans le langage Haskell, cela vous semblera limpide.

Le suivant

- Écrivez une fonction `next_day` de type `day -> day` qui renvoit le jour suivant celui passé en paramètre.
- Écrivez une fonction `next_month` de type `month -> month` qui renvoit le mois suivant celui passé en paramètre.

Les sorties pourront être similaires aux suivantes :

```
1  # next_day Sunday;;
2  - : day = Monday
3  # next_month December;;
4  - : month = January
```

Années bissextiles

Nous allons maintenant écrire une fonction qui détermine si une année est bissextile ou non. Je vous rappelle qu'une année est bissextile si et seulement si :

- Elle est divisible par 4 et non divisible par 100

ou bien

- Elle est divisible par 400.

L'operateur de modulo en OCaml est "mod".
Exemple : 42 mod 4 est egal a 2.

- Écrivez une fonction `is_bissextile` de type `int -> bool` qui renvoit vrai si l'année passée en paramètre est bissextile, et qui renvoit faux sinon.

Par exemple :

```
# is_bissextile 2010;;
- : bool = false
# is_bissextile 2000;;
- : bool = true
# is_bissextile 2012;;
- : bool = true
```

Jours du mois

Maintenant que nous savons déterminer si une année est bissextile ou non, nous allons écrire une fonction renvoyant le nombre de jours par mois. Le premier paramètre est l'année et le deuxième est le mois. Cela vous permet de déclarer un filtrage sur les mois en utilisant le mot clef `function`.

- Écrivez une fonction `nb_days` de type `int -> month -> int` qui renvoit le nombre de jours que dure le mois de l'année passée en paramètre.

Par exemple :

```
# nb_days 2011 January;;
- : int = 31
# nb_days 2011 February;;
- : int = 28
# nb_days 2012 February;;
- : int = 29
```

Les jours passent

Nous avons également besoin d'une fonction incrémentant le numéro du jour en tenant compte du nombre de jours dans un mois grâce à la fonction précédente.

- Écrivez une fonction `next_nday` de type :

  ```
  int -> month -> int -> (int * bool)
  ```

  Elle prend en paramètre, le jour (en chiffres), le mois et l'année et renvoit un couple (tuple à 2 éléments) composé du jour incrementé de 1 ainsi qu'un booléen représentant la retenue si l'incrémentation fait passer au 1er du mois suivant.

  Vous devez utiliser la fonction `nb_days` définie précédement pour déterminer si l'incrémentation du jour fait passer au mois suivant.

  Vous devez bien sûr gérer les annees bissextiles.

Par exemple :

```
# next_nday 22 January 2011;;
- : int * bool = (23, false)
# next_nday 31 January 2011;;
- : int * bool = (1, true)
# next_nday 28 February 2012;;
- : int * bool = (29, false)
# next_nday 28 February 2011;;
- : int * bool = (1, true)
```

Le calendrier complet

Grâce à toutes les fonctions précédentes, nous pouvons maintenant incrémenter une date complète en prenant en compte les fins de semaines, de mois, d'années et les années bissextiles.

- Écrivez une fonction `next` de type :

  ```
  (day * int * month * int) -> (day * int * month * int)
  ```

  Elle prend en paramètre une date sous forme du tuple décrit au début de cet exercice et renvoit cette date incrementée de 1 jour sous la même forme en ayant calculé les retenues nécéssaires.

Par exemple :

```
# next (Monday, 27, December, 2010);;
- : day * int * month * int = (Tuesday, 28, December, 2010)
# next (Friday, 31, December, 2010);;
- : day * int * month * int = (Saturday, 1, January, 2011)
# next (Sunday, 31, January, 2010);;
- : day * int * month * int = (Monday, 1, February, 2010)
# next (Sunday, 29, February, 2004);;
- : day * int * month * int = (Monday, 1, March, 2004)
# next (Saturday, 28, February, 2004);;
- : day * int * month * int = (Sunday, 29, February, 2004)
# next (Monday, 28, February, 2005);;
- : day * int * month * int = (Tuesday, 1, March, 2005)
```

I.4 Les listes

Le type `list` est un type natif en OCaml et reste la structure de données la plus courante de ce langage. Il existe également des tableaux, mais étant de nature impérative, les tableaux seront abordés dans le 4ème cours de ce module et restent rarement utilisés. Toutefois, les avantages et inconvénients des listes et des tableaux en C restent vrais en OCaml, en tenant compte que la mémoire est gérée automatiquement et de manière optimisée (données partagées, garbage collector, ...).

La bibliothèque standard d'OCaml propose un module de manipulation de listes très complet et très pratique appelé `List`. Le sujet du premier projet du module consiste à réécrire quelques unes de ces fonctions, ce qui constitue un excellent exercice technique et vous inspirera certainement pour écrire votre propre module en C si ce n'est déjà fait.

Dans certains cas particuliers, les fonctions du module `List` peuvent lever une exception. Nous verrons donc en fin de ce TP comment manipuler les exceptions en OCaml afin que vous puissiez terminer votre projet. N'hésitez pas à revoir la vidéo sur les listes avant d'entamer les exercices qui suivent.

I.4.1 Exercice 4

Je vous propose de traiter ensemble une fonction simple du projet :

`iter` de type `('a -> unit) -> 'a list -> unit`

La quasi-totalité des fonctions du projet reposent sur le parcours d'une liste, la fonction `iter` est donc un candidat idéal pour commencer. Cette fonction, comme son nom l'indique, consiste à parcourir chaque élément d'une liste et d'appliquer un effet de bord dessus, comme par exemple l'afficher.

Décodons ensemble le type de cette fonction :

- `'a -> unit` : Le premier paramètre est une fonction prenant en paramètre un type `'a` et renvoyant `unit`. Cette fonction sera celle à laquelle on appliquera chaque élément de la liste.
- `'a list` : Le second paramètre est une liste d'éléments d'un type `'a`. Il s'agit bien sûr du même type `'a` qu'au dessus.
- `unit` : Le type de retour de `iter` est `unit`, donc elle ne renvoit "rien". Elle ne consiste qu'en une suite d'applications d'une fonction s'évaluant elle-même vers `unit`.

Vous aurez probablement remarqué que cette fonction est très similaire à l'algorithme `for_each` de la `STL` de `C++` que vous avez rencontré à la fin de votre piscine C++.

Pour commencer la réalisation de cette fonction, je vous propose dans un premier temps d'écrire `iter` en ne faisant que parcourir la liste sans se soucier de la fonction passée en paramètre.

Rappelons ici le **pseudo code** du parcours d'une liste de manière récursive :

```
1  Parcours(liste) :
2      Si liste == vide (* Cas de base *)
3         Return;
4      Sinon (* Cas recursif *)
5          tete = Premier_element(liste);
6          reste = Suite_liste(liste);
7          Parcours(reste);
8      Return;
```

Grâce au filtrage et à la déconstruction à la volée d'OCaml, ce pseudo code peut être adapté pour être plus dans l'esprit de ce language :

```
1  Parcours(liste) :
2      Filtrage sur liste :
3        .Cas liste_vide -> Return
4        .Cas tete::reste -> Parcours(reste)
```

À votre tour d'écrire en OCaml cette fonction à partir du pseudo code. Le travail est déjà bien mâché !

- Écrivez la fonction `iter` de type `'a list -> unit` parcourant la liste passée en paramètre.

Maintenant que nous sommes capables d'itérer une liste, nous allons insérer dans notre fonction `iter` un traitement quelconque. Nous pouvons donc adapter notre pseudo code précédent pour prendre en compte ce traitement :

```
1  Parcours(Fonction_traitement, liste) :
2      Si liste == vide (* Cas de base *)
3         Return;
4      Sinon (* Cas recursif *)
5          tete = Premier_element(liste);
6          reste = Suite_liste(liste);
7          Func_traitement(tete);
8          Parcours(Func_traitement, reste);
9      Return;
```

Comme précédement, ce pseudo code peut être adapté pour être plus dans l'esprit du language OCaml :

En Ocaml, une séquence d'instructions s'exprime dans un bloc délimité par les mots clefs "begin" et "end". Les instructions sont separées par un ';'.

```
1  Parcours(Fonction_traitement, liste) :
2      Filtrage sur liste :
3        .Cas liste_vide -> Return
4        .Cas tete::reste -> Begin
5                              Func_traitement(tete);
6                              Parcours(reste)
7                            end
```

En Ocaml, les instructions d'une séquence d'instructions sont separées par un ';' (point-virgule). Il n'y en a donc PAS après la dernière insctruction de la séquence ! Nous verrons en détails les aspects impératifs d'OCaml dans le 4ème cours de ce module.

Et voila, vous êtes maintenant capables d'écrire la fonction `iter` complète et générique telle que presentée en début de cet exercice.

- Écrivez la fonction `iter` de type `('a -> unit) -> 'a list -> unit` appliquant la fonction passée en premier paramètre sur chaque élément de la liste passée en second paramètre.

Votre fonction devra avoir un comportement similaire à ces exemples :

```
# iter print_endline ["Bonjour"; "les"; "tech2 !"];;
Bonjour
les
tech2 !
- : unit = ()
# iter (fun n -> print_int n) [0; 1; 2; 3];;
0123- : unit = ()
```

I.5 Les exceptions

Pour terminer votre premier projet, vous aurez besoin d'utiliser les exceptions d'OCaml. En effet, certaines fonctions du projet sont succeptibles de lever une exception et nous allons profiter de cet exercice de TP pour découvrir leur syntaxe en OCaml.

Vous avez déjà découvert le mécanisme d'exceptions pendant votre piscine C++ et vous serez heureux d'apprendre que les exeptions en OCaml fonctionnent de manière assez similaire.

Un certain nombre d'exceptions standards existent mais vous pouvez naturellement déclarer les vôtres grâce à la syntaxe suivante :

```
1   exception <Nom> [of <type>]
```

Le nom de l'exception devant obligatoirement prendre une majuscule et pouvant être suivi par un type. On reconnaît très facilement la syntaxe d'un variant ! Nous verrons que ce n'est pas un hasard et que les exceptions sont un type variant spécial qu'on pourrait étendre dynamiquement, contrairement aux variants classiques.

Voyons un exemple de déclaration d'une nouvelle exception :

```
1   # exception My_excep;;
2   exception My_excep
3   # My_excep;;
4   - : exn = My_excep
5   # exception Other of int;;
6   exception Other of int
7   # Other 42;;
8   - : exn = Other 42
```

On constate que toutes les exceptions sont de type `exn`. Donc à chaque fois qu'on déclare une nouvelle exception, on ne fait en réalité qu'étendre le variant `exn` avec un nouveau constructeur. Étendre un variant quelconque est impossible, mais `exn` est un variant spécial qui possède cette particularité.

Pour lever une exception, on utilise le mot clef `raise` suivi du constructeur de l'exception à lever. La syntaxe est désarmante de simplicité :

```
# exception My_excep;;
exception My_excep
# raise My_excep;;
Exception: My_excep.
```

L'exception remontera alors la pile d'appels jusqu'à être rattrappée. Si elle n'est jamais rattrapée, le programme se terminera de façon... Brutale...
Pour éviter cette terminaison brûtale et gérer une exception (qu'elle soit définie dans la bibliothèque standard ou définie manuellement comme vu plus haut), on mettra le mot clef `try`, devant l'expression succeptible de lever une exception, suivi d'un filtrage sur les exceptions possibles pour les traiter.

L'exemple suivant illustre la gestion d'exceptions :

```
# exception My_excep_1;;
exception My_excep_1

# exception My_excep_2;;
exception My_excep_2

# try
    raise My_excep_1
  with
    | My_excep_1 -> print_endline "[exception 1 !]"
    | _ -> print_endline "exception non geree"
  ;;
[exception 1 !]
- : unit = ()

# try
    raise My_excep_2
  with
    | My_excep_1 -> print_endline "[exception 1 !]"
    | _ -> print_endline "exception non geree"
  ;;
exception non geree
- : unit = ()
```

Je vous propose le petit exercice suivant pour mettre en pratique :

- Écrivez la fonction `list_mul` de type `int list -> int` qui prend en paramètre une liste d'entiers et renvoit leur produit. Si un des éléments est égal a `0`, lever une exception de votre choix qui interrompt le parcours de la liste et renvoit directement la valeur `0`.

Chapitre II

Conclusion

Et voilà, bonne fin de projet !

Nous éspérons que vous avez apprécié ce sujet autant que nous avons apprécié le rédiger pour vous.

Vos avis sont très importants pour nous et nous permettent chaque jour d'améliorer nos contenus. C'est pourquoi nous comptons beaucoup sur vous pour nous apporter vos retours.

Si vous trouvez que certains points du sujet sont obscurs, pas assez bien expliqués ou tout simplement contiennent des fautes d'orthographe, signalez-le nous. Pour cela, il vous suffit de nous envoyer un mail à l'adresse koala@epitech.eu.

Pour aller plus loin dans votre apprentissage de la programmation fonctionnelle, nous vous conseillons le livre "Développement d'applications avec Objective CAML" : http://bibliotech.epitech.eu/?page=book&id=156 ou sa version en anglais dont le PDF est disponible : http://bibliotech.epitech.eu/?page=book&id=155.

Il existe bien sûr d'autres livres sur la programmation fonctionnelle : http://bibliotech.epitech.eu/?page=search&categ=15. N'hésitez pas à vous renseigner auprès des koalas, ils seront ravis de vous conseiller. Certains livres sont disponibles en version PDF si vous êtes connectés sur le site.