TP8: Premiers pas en OCaml

MP2I Lycée Pierre de Fermat

Récapitulatif

Pour faire des commentaires en OCaml, on les entoure par `(* *)`. Par exemple :

```
(* multiplie x et y *)
let mul x y = x * y;;
```

Liste des types et des opérateurs de base :

Type t	Opérateurs pour t	Type de l'expression
`int`	`+`, `-`, `*`, `/`, `mod`	`int`
`float`	`+.`, `-.`, `*.`, `/.`, `**` (puissance)	`float`
Tous sauf fonctions	`>`, `>=`, `=`, `<`, `<=`, `<>` ("différent de")	`bool`
`bool`	`&&`, `||`, `not`	`bool`
`string`	`^` (concaténation)	`string`

Nous avons également vu les types produits `'a * 'b`. Par exemple, (2, 3.6, "coucou") sera de type `int * float * string`.

On peut ajouter une variable au contexte global avec `let x = ... ;;` et ajouter une variable dans un contexte local avec `let x = ... in ...`. Les définitions récursives utilisent `let rec`.

On peut définir et appliquer des fonctions avec la syntaxe suivante :

```
(* définition *)
let f x1 x2 ... xn =
  ... ;;
```

```
(* application *)
f e1 e2 ... ek ;;
```

où `e1, e2, ... ek` sont des expressions ayant des types compatibles avec la signature de la fonction.

Voici des exemples de programmes mettant en oeuvre ces différentes notions :

```
(* Renvoie la somme de x et y *)
let somme (x, y) = x + y;;

(* Renvoie le couple diagonal (x, x) *)
let dedoubler x = (x, x);;

let f x = somme (dedoubler x);;

let a = f 3;;
```

```
(* factorielle de n *)
let fact n =
  if n <= 1 then 1
  else n * factorielle (n-1) ;;

let a = factorielle 5;;
```

Notions de base

Exercice 1.

Écrivez les réponses pour cet exercice dans un fichier "exercice1.ml". On rappelle que dans utop / ocaml, vous pouvez exécuter un fichier avec la directive "***#use***". Pour chaque fonction que vous écrivez, testez la sur plusieurs arguments

Question 1. Écrivez les fonctions suivantes :

a) Une fonction `double: int -> int` renvoyant le double de son entrée

b) Deux fonctions `first` et `second` prenant en entrée un tuple `'a * 'b` et renvoyant respectivement la première et la deuxième composante.

c) Une fonction `ajouter: float * float * float -> float` qui ajoute les trois composantes du tuple donné en entrée

d) Une fonction `est_pair ; int -> bool` déterminant si un entier est pair

e) Une fonction `divise: int -> int -> bool` qui détermine si sa première entrée est un diviseur de sa première

Question 2. Peut-on écrire `divise 3` ? Que représente cette expression ?

Question 3. Écrire une fonction `ajouteur: int -> (int -> int)` telle que `ajouteur k` est une fonction ajoutant k à son entrée.

Question 4. Écrire une fonction `est_racine: (int -> int) -> int -> bool` prenant en entrée une fonction f et un entier n, déterminant si $f(n) = 0$.

Question 5. Écrire la fonction identité `id` telle que `id x` vaut `x` pour tout `x`. Quel est le type de cette fonction ?

Question 6. Écrire une fonction `composee: ('a -> 'b) -> ('c -> 'a) -> ('c -> 'b)` prenant en entrée deux fonctions f et g et renvoyant leur composée $f \circ g$.

Exercice 2.

Étudions deux éléments importants de la syntaxe du OCaml : le if-then-else et le match-with.

OCaml possède une syntaxe conditionnelle : le ***if-then-else***. La syntaxe est la suivante :

```
1 if b then e1 else e2;;
```

où :

— `b` est une expression de type `bool`

— `e1` et `e2` sont des expressions de même type

Par exemple :

```
1 let valeur_absolue x =
2   if x < 0 then -x else x;;
```

Question 1. Écrivez une fonction `n_roots: (float * float * float) -> int` qui prend en entrée un triplet (a, b, c) et calcule le nombre de solutions réelles distinctes du polynôme $aX^2 + bX + c$.

Question 2. Écrivez une fonction `nom_chiffre: int -> string` qui prend en entrée un entier n et :

- si n est un chiffre entre 2 et 5 inclus, renvoie son nom en toutes lettres ("trois" pour $n = 3$ par exemple)
- sinon, renvoie la chaîne vide "".

On veut écrire cette fonction de manière plus concise. En OCaml, il existe une généralisation du if-else appelée le ***match with***, ou ***pattern matching***. Pour la fonction précédente, on peut écrire en OCaml :

```
let nom_chiffre n = match n with
  | 2 -> "deux"
  | 3 -> "trois"
  | 4 -> "quatre"
  | 5 -> "cinq"
  | _ -> "" ;;    (* Tous les autres cas *)
```

Pour évaluer un ***match with***, on évalue l'expression à matcher (ici, n lors de l'appel de la fonction), et on compare avec chaque motif possible : 2, 3, 4, 5, _. Dès que l'on en trouve un qui correspond, on évalue l'expression associée. Par exemple, si l'on appelle `nom_chiffre` avec $n = 4$, on va comparer 4 avec 2, puis avec 3, puis avec 4. On renverra donc "quatre".

Cette syntaxe est particulièrement puissante. Voyons un exemple plus poussé :

Question 3. Lisez le code suivant et tentez de deviner ce qu'il affiche. La fonction `print_int: int -> unit` sert à afficher un entier, et `print_newline: unit -> unit` affiche un retour à la ligne.

```
let f x y = match (x-1, y) with
| (0, 0) -> 0
| (_, 0) -> x + 1
| (0, _) -> y + 1
| _ -> x * y;;

print_int (f 3 5); print_newline ();;
print_int (f 1 2); print_newline ();;
print_int (f 0 0); print_newline ();;
print_int (f 1 0); print_newline ();;
```

Question 4. Recopiez le code précédent pour vérifier vos suppositions

Question 5. Écrivez une fonction prenant en entrée deux entiers x et y et :

- Si x vaut $\pm y$, renvoie 0
- Si $x \in \{y + 1, y - 1\}$, renvoie $(x + y)^2 + 1$
- Si $x + y \in \{1, -1\}$, renvoie $(x - y)^2 - 1$
- Sinon, renvoie $x * y$

Question 6. Écrivez une fonction prenant en entrée un entier n et renvoyant :

- Si n est multiple de 3, "gou"
- Si n est multiple de 5, "ba"
- Si n est multiple des deux, "meu"
- Sinon, n sous forme de string.

Exercice 3. Le type `unit` n'a qu'une seule valeur : `()`. Il sert à représenter le type des fonctions qui ne "renvoient rien". Par exemple, les fonctions suivantes prédéfinies en OCaml servent à ***afficher*** du texte, elles sont l'équivalent de printf en C :

```
print_int ;;
print_float ;;
print_string ;;
print_newline ;;
```

Question 1. Vérifiez le type de ces fonctions, et utilisez les pour afficher un entier, un flottant, une chaîne de caractère, et un retour à ligne.

Le type unit sert ainsi à représenter des ***commandes***, et servira plus tard à faire des programmes impératifs en OCaml.

Le type unit possède une syntaxe particulière : le point-virgule ";" permet d'enchaîner plusieurs commandes :

```
print_int 5 ; print_string " bonjour " ; print_float 9.8 ; print_newline () ;;
```

Question 2. Tapez l'expressions précédente. Quel est son type ?

";" n'est ni une fonction ni un opérateur, mais on peut informellement le voir comme une fonction de type `unit * unit -> unit` qui permet de combiner séquentiellement deux commandes.

Question 3. Créez une fonction `print_case: int*int -> int -> int -> unit` qui prend en entrée un couple (x, y), un entier m et un entier n, et affiche (x, y) si et seulement si $0 \leq x < m$ et $0 \leq y < n$.

Question 4. Créez une fonction `print_voisins: int*int -> int -> int -> unit` qui prend en entrée une position (x, y) et les dimensions n et m d'une grille, et qui affiche la liste des cases voisines de (x, y) dans cette grille. Par exemple, pour $(0, 0)$, les cases voisines sont $(1, 0)$ et $(0, 1)$, et pour $(2, 2)$ les cases voisines sont $(1, 2), (3, 2), (2, 1), (2, 3)$.

Question 5. Créez une fonction `print_retour: string -> unit` qui prend en entrée un string s et affiche s, suivi d'un retour à la ligne.

Remarque 1. La dernière fonction existe en OCaml : elle s'appelle `print_endline: string -> unit` !

Remarque 2. En Ocaml, lorsque l'on écrit `if a then b else ()`, autrement dit si l'on veut effectuer une commande `b` de type `unit` si une condition `a` booléenne est vérifiée, et ne rien faire sinon, on peut ne pas écrire le `else` :

```
let affiche_si_pair x =
  if x mod 2 = 0 then print_int x;;
```

En revanche ça ne marche pas pour les autres types !

On peut aussi utiliser le point virgule pour effectuer une commande avant de calculer une valeur :

```
let x = 5 ;;
print_int x; print_newline () ; x + 3;;
```

Récursivité

En OCaml, l'outil principal de programmation est la ***récursivité***, c'est à dire le fait qu'une fonction peut s'appeler elle-même.

Prenons la fonction factorielle. On a vu en C comment la calculer de manière impérative, avec une boucle for. En OCaml, pour écrire la fonction factorielle, il faudra trouver une relation de récursivité permettant de ***définir*** la factorielle. On remarque :

$$\begin{aligned} \textbf{factorielle}(0) &= 1 \\ \textbf{factorielle}(n) &= n \times \textbf{factorielle}(n-1) \quad \text{pour } n > 1 \end{aligned}$$

Ces formules permettent de ***définir récursivement*** ce qu'est la factorielle d'un entier. En OCaml, on voudrait donc écrire :

```
1 let factorielle n = match n with
2 | 0 -> 1
3 | _ -> n * factorielle (n-1) ;;
```

Cette expression n'est pas acceptée par OCaml, car on tente d'assigner une valeur à l'identifiant `factorielle` en utilisant ce même identifiant.

La syntaxe `let rec` permet de définir des fonctions récursives :

```
1 let rec factorielle n =
2   if n = 0 then 1
3   else n * factorielle (n-1) ;;
4 print_int ( factorielle 5); print_newline () ;;
```

Remarquons que si $n < 0$, cette fonction va s'appeler à l'infini. En effet, on n'est sensé calculer la factorielle que pour les entiers positifs. La fonction `failwith` en OCaml permet de renvoyer un message d'erreur et d'arrêter le programme. Par exemple, pour la factorielle :

```
1 let rec factorielle n =
2   if n < 0 then failwith " Factorielle d'un entier négatif"
3   else if n = 0 then 1
4   else n * factorielle (n-1) ;;
```

Si l'on évalue `factorielle (-3)`, ocaml affichera une ***exception*** :

```
Exception: Failure "Factorielle d'un entier négatif".
```

La programmation OCaml va donc principalement consister à trouver des définitions récursives pour les objets et les fonctions que l'on manipule.

Exercice 4.

Définir de manière récursives les fonctions suivantes, puis les implémenter en OCaml :

Question 1. `puiss: int -> int -> int` qui calcule de manière naïve la puissance d'un entier par un autre

Question 2. `reste: int -> int -> int` qui calcule le reste de la division euclidenne d'un entier a par un autre b, sans utiliser `mod`. On supposera $a \geq 0$ et $b > 0$.

Question 3. `pgcd: int -> int -> int` qui calcule le PGCD de deux entiers avec l'algorithme d'Euclide

Question 4. (*difficile*) `puiss_rapide: int -> int -> int` qui calcule de manière rapide la puissance d'un entier par un autre

Question 5. (*difficile*) `div_eucl: int -> int -> (int * int)` qui calcule le couple (quotient, reste) de la division euclidenne d'un entier a par un autre b, sans utiliser les opérateurs `/` et `mod`. On supposera $a \geq 0$ et $b > 0$.