TP10: Premiers pas en OCaml

MP2I Lycée Pierre de Fermat

Fichiers sources OCaml

L'extension des fichiers OCaml est `.ml`. Lorsque vous écrivez du code dans un fichier OCaml `bla.ml`, vous pouvez le compiler avec `ocamlc bla.ml -o nom_executable` comme avec GCC en C.

On peut aussi importer des fichiers `.ml` dans l'interpréteur utop avec la directive `#use`. Par exemple, si dans un fichier `bla.ml` vous avez écrit :

```
1 let ajouter x y = x + y;;
2 let echanger (x, y) = (y, x);;
```

alors dans utop vous pouvez écrire :

```
1 #use "bla.ml" ;;
2 let a = ajouter 2 3;;
```

La première ligne va exécuter tout le code de `bla.ml`, et donc rajouter les deux fonctions `ajouter` et `echanger` au contexte global, ce qui fait que la deuxième ligne ne pose pas de problème.

Notions de base

Exercice 1.

Écrivez les réponses pour cet exercice dans un fichier "exercice1.ml".

Q1. Écrivez les fonctions suivantes (veillez à bien respecter les types demandés) :

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 deuxième entrée.

Q2. Tapez `fst` et `snd` : que sont ces fonctions ?

Q3. Peut-on écrire `divise 3` ? Que représente cette expression ?

Q4. Écrire une fonction `ajouteur: int -> (int -> int)` telle que `ajouteur k` est une fonction ajoutant k à son entrée.

Q5. Écrire une fonction `est_racine: ('a -> int)-> 'a -> bool` prenant en entrée une fonction f et un élément x, déterminant si $f(x) = 0$.

Q6. Écrire la fonction identité `id` telle que `id x` vaut `x` pour tout `x`. Quel est le type de cette fonction ? Que veut-il dire ?

Q7. 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 a =
2    if 3 = 5 then "lapin"
3    else "hibou";;
4 (* [résultat] a: int = "hibou" *)
5
6 let valeur_absolue x =
7   if x < 0 then -x else x;;
```

Q1. Écrivez une fonction `n_roots: (float * float * float)-> int` qui prend en entrée un triplet (a, b, c) et calcule le nombre de racines réelles distinctes du polynôme aX^2+bX+c.

Q2. 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 :

```
1 let nom_chiffre n = match n with
2   | 2 -> "deux"
3   | 3 -> "trois"
4   | 4 -> "quatre"
5   | 5 -> "cinq"
6   | _ -> "" (* _ veut dire "Tous les cas" *)
7 ;;
```

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". Le motif `_` est un attrape-tout : toutes les valeurs lui correspondront.

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

Q3. 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. Le point virgule simple sert ici à exécuter plusieurs print d'affilées.

```
let f x y = match (x-1, y) with
  | (0, 0) -> 0
  | (0, _) -> y + 1
  | (z, 0) -> z + 100
  | _ -> x * y
;;

print_int (f 3 5); print_newline ();;
print_int (f 1 0); print_newline ();;
print_int (f 1 3); print_newline ();;
print_int (f 6 0); print_newline ();;
```

Q4. Recopiez le code précédent pour vérifier vos suppositions

Q5. On veut écrire 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 $x * y$

Complétez le code suivant pour implémenter la fonction décrite :

```
let g x y = match (x-y, x+y) with
  | 0, _ -> 0
  | _, 0 ->
  | 1, _ ->
  | (* A COMPLETER *)
  | (* A COMPLETER *)
  | (* A COMPLETER *)
  | _    -> (* A COMPLETER *)
```

Testez sur quelques exemples pour vérifier.

Q6. Écrivez une fonction prenant en entrée un entier n et renvoyant :

- Si n est multiple de 3 mais pas de 5, "gou"
- Si n est multiple de 5 mais pas de 3, "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 mais font quelque chose". Par exemple, les fonctions suivantes prédéfinies en OCaml servent à ***afficher*** des valeurs de différents types :

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

Q1. 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 possède une syntaxe particulière : le point-virgule ";" permet d'enchaîner plusieurs expressions de type unit :

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

Q2. Tapez l'expression précédente. Quel est son type ?

";" n'est ni une fonction ni un opérateur, mais on peut informellement le voir comme un opérateur binaire sur les `unit`. On peut donc voir une expression de type unit comme une instruction impérative, et le point-virgule sert à combiner séquentiellement deux instructions. [1]

Q3. Créez une fonction `print_pair: int -> unit` qui prend en entrée un entier et l'affiche si et seulement si il est pair.

Q4. Créez une fonction `print_pairs: (int*int*int)-> unit` qui prend en entrée un triplet d'entiers, et affiche uniquement ceux qui sont pairs.

Q5. 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, ce qui est logique : il serait impossible de donner un type à `1 + (if b then 5);;`

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

```
let x = 5 ;;
let y =
  print_int x;
  print_newline ();
  x + 3;;
(* y vaut 8, et l'exécution a affiché 3 suivi d'un retour ligne *)
```

1. Il existe même des façons de faire des boucles for et while en OCaml, mais pour l'instant c'est interdit !

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 :

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

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

```
let factorielle n = match n with
| 0 -> 1
| _ -> 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. Il faut dire à OCaml que la définition est récursive, et pour signifier cela on utilise le mot clé `let rec` au lieu de `let` :

```
let rec factorielle n =
  if n = 0 then 1
  else n * factorielle (n-1) ;;
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 :

```
let rec factorielle n =
  if n < 0 then failwith "Factorielle d'un entier négatif"
  else if n = 0 then 1
  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.

Exemple 1. On remarque :

$$\begin{aligned} x \times 0 &= 0 && \text{pour tout } x \in \mathbb{N} \\ x \times y &= x \times (y-1) + x && \text{pour } y > 0 \end{aligned}$$

On peut en déduit la fonction suivante (très inefficace) qui calcule le produit de deux entiers :

```
let rec produit x y =
  match y with
  | 0 -> 0
  | _ -> produit x (y-1) + x
;;
```

Exercice 4.

Trouver une définition récursive des fonctions suivantes, puis les implémenter en OCaml :

Q1. `puiss: int -> int -> int` qui calcule de manière naïve la puissance d'un entier par un autre

Q2. `reste: int -> int -> int` qui calcule le reste de la division euclidienne d'un entier a par un autre b, sans utiliser `mod`. On supposera $a \geq 0$ et $b > 0$.

Q3. `pgcd: int -> int -> int` qui calcule le PGCD de deux entiers avec l'algorithme d'Euclide

Q4. (*difficile*) `puiss_rapide: int -> int -> int` qui calcule de manière rapide la puissance d'un entier par un autre

Q5. (*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 $b > 0$.

Q6. (*difficile*) `a_racine: (int -> int)-> int -> int -> bool` telle que `a_racine f a b` indique si f admet une racine dans $[\![a, b]\!]$.