Éléments de base d'OCaml
Aide-mémoire

MP2I Lycée Pierre de Fermat

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

```
(* produit de x et y *)
let mul x y = x * y;;
```

Les types de base sont int, float, bool, char, string, unit.

Les opérations sur ces types sont presque les même qu'en C. Le ***non*** booléen se note `not` et les opérateurs flottants nécessitent un point : `+.` au lieu de `+`, etc...

Type t	Opérateurs binaires pour t	Opérateurs unaires pour t	Type de l'expression composée
`int`	`+`, `-`, `*`, `/`, `mod`	`-`	`int`
`float`	`+.`, `-.`, `*.`, `/.`, `**` (puissance)	`-.`	`float`
Tous	`>`, `>=`, `=`, `<`, `<=`, `<>` (“différent de”)		`bool`
`bool`	`&&`, `||`	`not`	`bool`
`char`			
`string`	`^` (concaténation)		`string`

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

On peut définir des fonctions avec la syntaxe suivante :

```
let f x1 x2 ... xn =
  ...
```

Le type d'une telle fonction est `t1 -> t2 -> ... -> tn -> t`, avec t1, ... tn les types attendus pour les paramètres de la fonction, et t le type de la valeur renvoyée. On peut comprendre ça de plusieurs manières :

- La fonction prend en entrée un seul argument de type t1, et renvoie une fonction de type `t2 -> ... -> tn -> t`
- La fonction prend en entrée deux arguments de types t1 et t2, et renvoie une fonction de type `t3 -> ... -> tn -> t`
- ...
- La fonction prend en entrée n arguments de types t1, ..., tn et renvoie une valeur de type t.

Tous ces points de vues sont équivalents, mais le premier est le plus proche de la manière dont OCaml représente f en interne.

On applique une fonction avec :

```
f e1 e2 ... ek
```

où `e1, e2, ... ek` sont des expressions ayant des types compatibles avec la signature de la fonction `f`. Si $k < n$ alors c'est une application partielle, et on obtient une fonction de type `t(k+1) -> ... -> tn -> t`, prenant donc $n - k$ arguments.

Les parenthèses servent à encadrer les sous-expressions **et pas à appliquer les fonctions**.

Voici un exemple de programme mettant en oeuvre ces différentes notions :

```
let x = 3;;
let y = 5;;
let z = 3 * 5 ;;

(* somme de x et y *)
let sum x y = x + y;;
let t = sum 1 2 ;; (* vaut 3 *)

(* FONCTIONS D'ORDRE SUPERIEUR *)

(* renvoie une approximation de la dérivée de f *)
let derivee f = fun x -> (f (x +. 0.0001) -. f x) /. 0.0001;;
(* on pourrait aussi écrire let derivee f x = (f (x +. 0.0001) -. f x) /. 0.0001 *)

let carre x = x ** 2.0 ;;
let derivee_carre = derivee carre ;;
derivee_carre 3.0;; (* affiche environ 6, logique car d/dx (x^2) = 2x *)


(* composée de f et g, i.e. une fonction h
   telle que h(x) = f(g(x)) *)
let composition f g =
  fun x -> f (g x);;
(* on pourrait aussi écrire let composition f g x = f (g x) *)

(* double de x *)
let double x = 2*x;;

let u = composition double int_of_string ;;
let a = u "12";; (* vaut 24 *)

let quadrupler = composition double double;;
let b = quadrupler 5;; (* vaut 20 *)
```

Quiz Pour chacune des expressions suivantes, prédire son type et sa valeur :

```
(* Q1 *)
let g x = 2*x in
let f x = g x + 1 in
f 5 + f 3;;

(* Q2 *)
let double x = 2*x in
let triple x = 3*x in
double (triple 5);;

(* Q3 *)
let s = "bonjour " in
let saluer x = s ^ x ^ " !!" in
let s = "salut " in
saluer "Jérémy";;

(* Q4 *)
let f x y z = (x z) (y z) in
let g x = let x = x - 1 in x * x in
let h x = fun y -> let x = y in x+1 in
f h g 3;;

(* Q5 *)
let u f (x, y) = f x y in
let g = u (fun x -> (fun y -> x y)) in
g ((fun a -> a+3), 5);;
```