TP 1 : Débuts en Caml. Programmation impérative

1 Déclaration, premières fonctions

Le but de cette section est de maîtriser les concepts introduits dans le premier cours. On rappelle que :

— les opérations usuelles sur les entiers sont `+`, `-`, `*`, `/` et `mod` ;

— les opérations usuelles sur les flottants sont `+.`, `-.`, `*.`, `/.`, `**` ainsi que toutes les fonctions mathématiques (`exp`, `sqrt`, `atan`...)

— on déclare une « variable » avec `let x = ...`

— le mot clé `in` sert à l'affectation locale, `and` pour la déclaration simultanée.

— un calcul Caml (déclaration de fonction, par exemple) se termine par `;;`.

— pour déclarer une fonction curryfiée à n arguments on écrira `let f x1 x2 ... xn= ....`

Exercice 1. Calculez $\frac{1+\sqrt{2}+\sqrt{2}^3}{1+e^{\sqrt{2}}}$ à l'aide d'une affectation locale.

Exercice 2. Définir une fonction qui calcule le carré d'un entier.

Exercice 3. Définir des fonctions prenant en entrée une fonction $f : \mathbb{R} \to \mathbb{R}$ (dont le type est `float -> float =<fun>` a priori) et renvoyant :

— la valeur $\frac{f(0)+f(1)}{2}$,

— la fonction f^2 (carré de f).

— la fonction $f^{(2)}$, c'est à dire $x \mapsto f(f(x))$.

— la fonction $x \mapsto f(x+1)$.

C'est un bon exercice d'essayer de deviner le type de la fonction avant de l'écrire : il peut y avoir polymorphisme.

Exercice 4. *composition de fonctions*. Écrire une fonction `c` telle que `c f g` soit la fonction $f \circ g$. Avant cela donner son type. Objectif : 20 caractères.

```
#c (function x -> x+1) (function x-> x*x) 4 ;;
- : int = 17
```

Exercice 5. *Complexes, un peu long, ne faites pas tout tout de suite*. On rappelle que dans le cas particulier des couples (tuples de taille 2), il existe les fonctions `fst` et `snd` pour accéder aux composantes, mais elles ne se généralisent pas à de plus grands tuples. En assimilant un complexe à un couple de floats, écrire les fonctions de manipulation des complexes : partie réelle, partie imaginaire, conjugué, module. Puis rajoutez l'addition et la multiplication. Vous pouvez traiter l'argument également, le plus simple est d'utiliser que $\arg(z) = 2\frac{\Im(z)}{\Re(z)+|z|}$, sauf si z est un réel négatif. Sinon, vous pourrez avoir besoin de $\pi = 4\arctan(1)$, qui s'obtient via `atan` en Caml. Remarque : 0 ne possède pas d'argument, on pourra utiliser `failwith "zero"` pour produire une erreur si le complexe passé en argument est zéro. Voici les types que j'obtiens :

```
pr : 'a * 'b -> 'a
pi : 'a * 'b -> 'b
cj : 'a * float -> 'a * float
modu : float * float -> float
add : float * float -> float * float -> float * float
mult : float * float -> float * float -> float * float
arg : float * float -> float
```

Pour la dernière fonction, on rappelle que `if cond then ... else ...` est la syntaxe d'une instructions conditionnelles, les deux blocs ayant le même type.

2 Boucles et références

On rappelle que :

— pour déclarer une référence vers un entier, on utilise par exemple `ref 0`;

— l'accès à la valeur pointée par une référence `r` se fait avec `!r`, la modification de la valeur pointée avec `r:= ...`

— dans une boucle `for` ou `while` ne se trouvent que des instructions de type `unit`, séparées par des point-virgules.

Exercice 6. *boucle for*. En utilisant une boucle `for`, définir une fonction `puiss` de calcul de puissance sur les entiers.

```
#puiss 3 7 ;;
- : int = 2187
```

Exercice 7. *boucle while*. En utilisant une boucle `while`, définir une fonction `sdc` qui effectue la somme des chiffres d'un entier. On utilisera des divisions euclidiennes par 10.

```
#sdc 4948353 ;;
- : int = 36
```

Exercice 8. *autre boucle while*. Écrire une fonction `miroir` qui renvoie le « symétrique » d'un entier. Par exemple, le symétrique de 123 est 321.

```
#miroir 16163223 ;;
- : int = 32236161
```

Exercice 9. Écrire une fonction `racine x epsilon` qui retourne $\sqrt{\texttt{x}}$ avec une précision de ε, en utilisant l'algorithme suivant, appelé algorithme de Babylone :

$$\begin{cases} u_0 = 1 \\ u_{n+1} = \dfrac{1}{2}\left(u_n + \dfrac{x}{u_n}\right) \end{cases}$$

qui s'arrête lorsque $\left|u_n - \dfrac{x}{u_n}\right| < \varepsilon$. Cette fonction aura pour type `float -> float -> float`. On pourra utiliser la fonction `abs_float`.

3 Fonctions sur les tableaux

Un tableau est essentiellement une portion de mémoire contiguë. Comme les listes Python, on peut accéder à un élément du tableau ou le modifier en temps constant. Par contre, contrairement aux listes Python [1], une fois le tableau créé il n'est pas possible de changer sa taille : celle-ci est immuable.

Voici plein de fonctions sur les tableaux [2], qui se trouvent dans le module Array. N'apprenez pas tout par cœur, retenez comment on construit un tableau par la données de ses éléments, comment on accède à ou modifie une entrée d'un tableau, `Array.make` et `Array.length`, on peut recoder assez facilement toutes les autres avec !

1. qui sont en fait des tableaux redimensionnables
2. Liste complète ici : `https://caml.inria.fr/pub/docs/manual-ocaml/libref/Array.html`.

commande	effet
`[|0;1;7;8;9|]`	construit un tableau par donnée explicite des éléments.
`t.(i)`	le i-eme élément de `t` ($0 \leq i < n$ avec n la taille de `t`)
`t.(i) <- x`	remplacer le i-eme élément de t par x.
`Array.length t`	renvoie la longueur de t.
`Array.make n x`	construit un tableau de longueur n contenant des x (attention les éléments sont physiquement tous égaux !)
`Array.init n f`	construit un tableau de longueur n contenant les $f(i)$ pour i entre 0 et $n-1$
`Array.copy t`	renvoie une copie `t`
`Array.sub t i k`	renvoie un tableau de longueur k, égal à la portion de `t` qui démarre à l'indice i
`Array.append t1 t2`	concatène deux tableaux (ne pas confondre avec Python !)
`Array.concat q`	concatène une liste de tableaux (les listes seront vues ultérieurement.)
`Array.make_matrix n m x`	construit une matrice de taille n, m contenant des x (c'est-à-dire un tableau de tableaux. Un élément est accessible par `t.(i).(j)`).
`Array.map f t`	crée un tableau dont les éléments sont les $f(x)$ pour x dans t.
`Array.iter f t`	applique f sur chaque x de t (f est de type `'a -> unit`).
`Array.sort f t`	trie le tableau `t` en place, avec fonction de comparaison `f`. `f x y` renvoie un entier, nul si les éléments sont égaux, strictement positif si $x > y$, et strictement négatif sinon.

Exercice 10. Écrire une fonction faisant la somme des éléments d'un tableau d'entiers.

Exercice 11. Écrire une fonction `inverse t` prenant en entrée un tableau `t` et produisant un nouveau tableau dont les éléments sont dans l'ordre inverse.

Exercice 12. Même question avec une fonction qui modifie le tableau passé en entrée, c'est-à-dire une fonction de type `'a array -> unit`.

Exercice 13. Écrire une fonction de tri (en place) d'un tableau, de type `'a array -> unit`. Évidemment on n'utilisera pas `Array.sort`.

4 Optimisation

Les exercices de cette section sont plus relevés, et moins guidés. Ce n'est pas grave si vous ne les traitez pas aujourd'hui, ils sont là pour occuper les plus rapides.

Exercice 14. Étant donné un tableau d'entiers (positifs et négatifs) `t`, on veut trouver le sous-tableau de somme maximale : il s'agit donc de trouver un couple d'indice (i, j) tel que $i \leq j$, et $\sum_{k=i}^{j}$ `t.(k)` maximal.

1. Écrire une fonction qui trouve ce sous-tableau de somme maximale, et qui renvoie la somme.
2. Si ce n'est pas déjà le cas, l'optimiser pour qu'elle s'effectue en temps quadratique ($O(n^2)$).
3. * Écrire une fonction qui s'effectue ne temps linéaire ($O(n)$). *Indication : on note s_j la somme maximale d'une portion terminant à l'indice j. Trouver une relation permettant de calculer les s_j en temps $O(n)$.*

Exercice 15. *Le problème des pièces de monnaies.* On se donne une collection de valeurs de pièces de monnaies, sous forme d'un tableau `t` (comme `[|1; 2; 5; 10; 20; 50; 100|]`). On souhaite calculer le nombre de façons différentes que l'on a de décomposer une quantité d'argent s avec ces pièces. Écrire une fonction `decomposition t s` résolvant ce problème. *Indication : c'est un problème difficile. On construira une matrice (tableau à deux dimensions) de taille $(s+1) \times (p+1)$, avec p le nombre de pièces. On calculera le nombre de façon que l'on a de décomposer k avec les i premières pièces du tableau de pièces, et ce pour tout $0 \leq k \leq n$ et $0 \leq i \leq p$.*