■ ■ ■

Ce TP pourrait se résumer en : « Avant de découvrir toutes les forces du langage Caml, vérifions que nous sommes capable d'y faire ce que nous faisions au premier semestre ».

I - De la syntaxe

Le manuel de Caml light se trouve là : http://caml.inria.fr/pub/docs/manual-caml-light/. Il est surement la meilleure anti-sèche mais vraissemblablement pas la plus conviviale ! Referez vous y en cas de trou de mémoire.

Pour ce début de TP, il vous suffira de savoir :

* La chose que les élèves mettent le plus de temps à assimiler (et pourtant ...) : En Caml, lorsque l'on veut appliquer la fonction `f` aux arguments `x` et `y`, on écrit `f x y` et non la notation mathématique $f(x, y)$!
* Pour stocker une valeur dans une variable, on écrit `let [ident] = [valeur] in [programme]`. Ces variables sont dites non mutable, ce ne sont que des alias immuables. Les variables que vous connaissiez du premier semestre sont appelées en Caml des *références*. Ce sont celles que l'on initialise en disant `let [ident] = ref([val]) in [prog]`, auxquelles on affecte une valeur en disant `[ident] := [val]` et dont on obtient la valeur courante en ecrivant `![ident]`.
* La boucle inconditionnelle s'écrit `for [ident] = [val] to [val] do [prog] done` il est possible d'aller dans l'ordre décroissant en remplacant `to` par `downto` mais pas de faire des pas plus exotique que de 1 en 1.
* La boucle conditionnelle s'écrit `while [cond] do [prog] done`
* La syntaxe de la condition est `if [cond] then [prog] else [prog]`. Celle du test d'égalité : un seul `=`, la différence : `<>`, `<, <=, >, >=` sont transparents.
* Enfin l'enchainement d'instruction est `[prog]; [prog]`.

Pour comprendre les erreurs que va renvoyer l'interpreteur sur les programmes mal formés, il faut légèrement anticiper la suite.

Pour rester proche de ce que vous avez entendu au premier semestre, la très brève présentation des mots clés parle de condition, d'instruction, de valeur. Ces distinctions n'existe pas en Caml. Tout est une expression, il n'y a que des expressions ! Elles ont par contre un *type* qui contraint leur comportement. Une conditionest une expression de type `bool`, une instruction, une expression qui ne renvoie rien (mais modifie l'environement) on utilise le type `unit` (dont l'unique habitant est `()`), la boucle inconditionnelle attend deux expressions de type `int` puis une de type `unit`, etc

C'est donc sous cette forme que l'interpreteur rapportera les problèmes :
`This should have type unit`,
`This expression has type float but an expression was expected of type int`, ...
En Caml, `(a, b)` représente la paire constitueée de l'expression `a` et de l'expression `b`. Donc, j'insiste, si l'interpreteur dit
`This expression has type 'a * 'b but an expression was expected of type [...]`
c'est quasiment certain que c'est que `f (x,y)` a été écrit au lieu de `f x y` et il est très mal vu de demandé de l'aide pour ça !

Pour tout autre motif, en TP ou en dehord par email à l'adresse `pierre.boutillier@pps.univ-paris-diderot.fr` il est par contre très vivement encouragé de poser n'importe quelle question.

II - Exercices divers

Exercice 1. (Autour des vecteurs)

1. Écrire une fonction `somme` qui retourne la somme des éléments d'un vecteur.
`val somme : int vect -> int`

2. Écrire une fonction `max` qui renvoie le maximum d'un vecteur.
`val max : 'a vect -> int`

3. Écrire une fonction `begaie` qui, étant donné un vecteur d'entrée $[|a_1; \ldots; a_n|]$, retourne $[|a_1; a_1; \ldots; a_n; a_n|]$.
`val begaie : 'a vect -> 'a vect`

Exercice 2. (Palindromes)

1. Écrire une fonction `est_palindrome` qui teste si un vecteur est un palindrome.
`val est_palindrome : 'a vect -> bool`

2. Écrire une fonction `est_palindrome2` qui détermine si un entier en représentation décimale est un palindrome.
`val est_palindrome2 : int -> bool`

Exercice 3. (Ordre lexicographique)

1. Écrire une fonction `ordre_N2` qui, recevant deux couples d'entiers c_1 et c_2, retourne la valeur booléenne de la comparaison lexicographique $c_1 \preceq c_1$.
`val ordre_N2 : int * int -> int * int -> bool`

2. Généraliser la fonction précédente en une fonction `ordre_lex` qui compare des vecteurs de même taille. `ordre_lex : int vect -> int vect -> bool`.

Exercice 4. (Curryfication) Donnez le type et écrivez une fonction `curry`, prenant en argument une fonction à deux arguments et renvoyant sa version currifiée. Faites de même pour sa fonction réciproque `uncurry`.
`val curry ('a * 'b -> 'c) -> 'a -> 'b -> 'c`
`val uncurry ('a -> 'b -> 'c) -> 'a * 'b -> 'c`

Exercice 5. (Crible d'Erathostène) Soit $n \in \mathbb{N}$. On désire déterminer tous les entiers premiers inférieurs ou égaux à n. On utilise pour cela la méthode du crible d'Erathostène :

- On écrit tous les nombres de 2 à n dans un vecteur.
- On cherche le premier nombre non rayé (au départ 2) et on raye tous ses multiples.
- On recommence l'étape précédente tant que toute la table n'a pas été explorée.

1. Écrire une fonction `crible_simple` qui retourne les nombres premiers inférieurs ou égaux à son argument.
`val crible_simple : int -> int vect`

2. En constatant que, mis à part 2, les nombres premiers sont impairs, et que, si p est premier, le premier multiple de p non rayé est p^2, améliorez la fonction précédente.
`val crible : int -> int vect`

III - Triangle de Sierpiński

Dans cet exercice nous allons utiliser les outils graphiques de Caml. Il faudra au préalable entrer les instructions suivantes :

```
#open "graphics";;
open_graph "";;
```

L'instruction `clear_graph()` remet à zéro l'affichage. L'instruction `plot a o` affiche un point d'abscisse `a` et d'ordonnée `o`.

3. Écrire une fonction `affiche` qui affiche un point pour chaque cellule non nulle de la matrice passée en argument.
`val affiche : int vect vect -> ()`

Nous allons considérer une matrice de largeur $2 * p + 1$ et de hauteur h, remplie de la manière suivante :

- Toutes les cellules des colonnes de gauche et de droite contiennent 0.
- Les cellules de la première ligne contiennent toutes 0, exceptée la cellule centrale qui contient 1.
- Les lignes suivantes sont déduites par itération, le contenu de la cellule $(i + 1, j)$ étant dicté par les contenus des cellules $(i, j - 1)$, (i, j) et $(i, j + 1)$ selon les règles :

111	110	101	100	011	010	001	000
0	1	0	1	1	0	1	0

4. Écrire une fonction `sierpinski` qui prend des dimensions p et h et retourne la matrice correspondante. En déduire une fonction `affiche_sierpinski` qui affiche le résultat.
`val sierpinski : int -> int -> int vect vect`
`val affiche_sierpinski : int -> int -> ()`

Les questions précédentes permettent de construire le triangle de Siepiński qui est un exemple « d'automate cellulaire » : un ensemble de cellules évoluant dans le temps en fonction de leur entourage immédiat. Chaque ligne représentait un état de notre ensemble de cellules, et la succession des lignes correspondait au passage du temps. Ainsi, chaque cellule évoluait en fonction de son propre état et des états de ses voisines directes. Vous pouvez poursuivre quelques expériences :

5. Utiliser une configuration de départ (*ie* une première ligne) aléatoire.

6. Essayer d'autres règles d'évolution.

Remarque : la règle donnant le triangle de Sierpiński s'appelle « règle 90 ». D'autres règles connues portent par exemple les numéros 30 ou 110. De manière générale, les règles d'évolution pour lesquelles le prochain état d'une cellule i dépend des cellules $i - 1$, i et $i + 1$ sont numérotées de 0 à 255.

7. À l'aide des indices précédents, proposer une manière de représenter une règle d'évolution par un entier entre 0 et 255, et écrire enfin une fonction `affiche_automate` qui reçoit des dimensions et un numéro de règle et affiche l'évolution de l'automate pour la configuration initiale de votre choix.
`val affiche_automate : int -> int -> int -> ()`

Pour d'autres exemples fameux d'automates cellulaires, vous pouvez vous renseigner sur le « jeu de la vie » de Conway. Attention, il s'agit d'un automate à deux dimensions ! Vous avez donc besoin des deux dimensions de l'écran pour représenter chaque état, et vous pouvez ensuite l'animer en affichant les états l'un après l'autre.