TD : Bases de programmation

1) Prise en main

1.a) Emacs

Pour commencer, lancez le programme `Emacs` (disponible dans la liste des Applications).

Une fenêtre s'ouvre : jetez un œil aux messages à l'écran et à la barre des menus en haut, en particulier le menu `File`. La plupart des actions sont assorties d'un code (ex : `(C-x C-f)` pour ouvrir un fichier) : il s'agit du raccourci clavier correspondant. En d'autres termes, `C-x` signifie `Ctrl+x`, et `M-y` signifie `Alt+y`

Voici la liste des raccourcis qui risquent de vous servir :

`C-w`	Couper (la fin de la ligne)
`C-c`	Copier
`C-v`	Coller
`C-z`	Annuler la dernière action
`C-x C-f`	Ouvrir un fichier
`C-x C-s`	Enregistrer le fichier courant
`C-x C-c`	Fermer Emacs
`C-x C-e`	Evaluer un bout de code Caml

Remarques :

- Pour copier, le plus simple de sélectionner un bout de code avec la souris : il est automatiquement copié.
- Si vous voulez couper une ligne, inutile de sélectionner le texte en question : placez juste le curseur au début de la ligne et appuyez sur `C-w`
- Vous pouvez aussi utiliser le menu `Edit` pour ces actions d'édition (qui d'ailleurs marchent parfois mieux que les raccourcis clavier)

En bas de la fenêtre, dans une barre grise, le nom de votre fichier est écrit en gras. Il s'agit du *buffer* courant, c'est-à-dire en gros du fichier actuellement à l'écran. Cliquez sur le texte en gras pour passer d'un buffer à un autre.

1.b) OCaml

Pour ouvrir un nouveau fichier, cliquez sur l'icône avec une page blanche en haut à gauche de l'écran, ou utilisez le raccourci `C-x C-f`. Le texte `Find file :` apparait en bas de la fenêtre. Il vous faut alors entrer le nom du fichier sur lequel vous souhaitez travailler. Si ce fichier existe déjà, son contenu sera chargé ; dans le cas contraire, un fichier du nom correspondant sera créé.

Ouvrez de cette manière un fichier se terminant par l'extension `.ml` : Emacs va deviner qu'il s'agit d'un fichier Caml, et le menu `Tuareg` va apparaître dans la barre en haut de l'écran.

Tapez un bout de code Caml (par exemple `let x = 3 ;;`), puis évaluez-le avec le raccourci `C-x C-e` (ou menu `Tuareg` → `Interactive Mode` → `Evaluate Phrase`) .

Un texte apparait tout en bas de l'écran `Caml toplevel to run: ocaml`. Contentez-vous de valider avec la touche Entrée (il s'agit en fait du "toplevel" qu'on veut utiliser pour notre code ; on ajoutera parfois des options pour utiliser certains modules particuliers, comme des outils graphiques).

La fenêtre se sépare en deux : en haut, votre fichier `.ml`, en bas, le `*caml-toplevel*`, où vient d'apparaître le résultat de votre évaluation.

Si la coloration syntaxique fonctionne mal, faites `M-x`, puis tapez `font-lock-mode` (le texte apparaît en bas de votre écran), et répétez une fois cette opération. Plus généralement, si vous tapez `C-x` ou `M-x`, vous verrez apparaître en bas de l'écran un suivi du raccourci en cours. C'est un outil très puissant pour configurer Caml, mais il arrive aussi qu'on commence une telle séquence involontairement (pour essayer, tapez juste `C-x)`: si tel est le cas, appuyez sur Esc jusqu'à l'apparition brève de `Quit` en bas de l'écran, et le retour du curseur dans votre buffer courant

2) L'interpréteur OCaml

2.a) Principe

La grande force d'Emacs est qu'il va vous permettre d'évaluer à la volée vos lignes de code. C'est à cela que sert le raccourci `C-x C-e` utilisé plus tôt.

Lorsque vous utilisez ce raccourci, vous demandez en fait à Emacs d' « interpréter » votre code, ou plus exactement la ligne de code sur laquelle votre curseur est actuellement situé.

2.b) Votre première ligne de code

Observons tout ceci sur un exemple. Commencez donc par taper la ligne de code suivante (sans oublier les points-virgules finaux) : `42;;`

Interprétez cette ligne avec le raccourci `C-x C-e`. Vous devriez voir apparaître dans l'écran inférieur les lignes suivantes :

```
# 42 ;;
- : int = 42
```

Analysons cette réponse de Caml :

- Pour commencer, vous retrouvez votre ligne de code :
- Ensuite, un simple tiret (`-`), dont nous comprendrons le sens un peu plus loin dans ce TD
- Puis le type du résultat renvoyé, ici un entier (`int`)
- Et finalement la valeur de ce résultat (`42`)

Emacs nous annonce donc qu'il a interprété notre instruction comme un entier, de valeur 42.

Faites d'autres tests avec d'autres valeurs, et tentez d'écrire une ligne de code pour chacun des types vu lors du dernier cours :

- Entier
- Nombre réel
- Caractère
- Chaîne de caractère
- Booléen
- Unit

2.c) Le point-virgule

Lors de votre découverte du langage Caml, ce sera peut-être votre meilleur ami, et sans nul doute votre pire cauchemar : le point-virgule.

Il sert en fait à indiquer à l'interpréteur la fin d'une instruction. Sauf indication contraire, on placera donc un point-virgule à la fin de chaque ligne de code.

Les plus observateurs d'entre vous auront remarqué que ces points-virgules étaient doublés dans les exemples précédents. Le double point-virgule sert en effet à indiquer à l'interpréteur qu'il doit s'arrêter et renvoyer le résultat correspondant à la dernière instruction.

Si vous oubliez un point-virgule, vous aurez généralement droit à un message d'erreur de la forme `Error: This expression is not a function; it cannot be applied`. Vérifiez alors où vous pourriez avoir oublié un point-virgule un peu plus haut dans votre code.

2.d) Commentaires

Si vous souhaitez ajouter des commentaires, la syntaxe est simple : `(* Votre commentaire ici *)`. Le contenu situé entre `(*` et `*)` sera tout simplement ignoré par l'interpréteur.

Vous pouvez placez ces commentaires où vous le souhaitez dans votre code. Testez les exemples suivants pour vous en convaincre :

```
2 + 3 ;; (* Commentaire après *),
15 + (* Commentaire au milieu *) 5 ;;
```

2.e) Espaces, passage à la ligne, et tabulation

Remarquez également que l'interpréteur ne prend pas en compte les espaces, les sauts de lignes et autres tabulations que vous pouvez utiliser pour aérer et structurer votre code.

Vous pouvez donc présenter votre code à votre convenance, mais gardez à l'esprit que plus la présentation sera simple et intuitive, plus il vous sera facile de le partager ou de le comprendre quelques semaines plus tard,

2.f) Limites de Caml

Tapez et évaluez les lignes de codes suivantes :

```
max_int;;
min_int;;
max_int + 1;;
```

Que remarquez-vous?

Testez ensuite les lignes de code suivantes :

```
max_float;;
min_float;;
max_float +. 1.;;
max_float = (max_float +. 1.);;
```

Qu'en pensez-vous ?

Nous reviendrons plus tard dans l'année sur ces bizarreries, qui sont en fait liées à la représentation de l'information dans un ordinateur.

3) Opérateurs

3.a) Opérateurs arithmétiques

Les opérations de bases sur les nombres entiers sont :

- L'addition +
- La soustraction –
- La multiplication `*`
- La division `/`
- Le modulo `mod`

Testez ces opérateurs sur quelques exemples pour voir comment ils fonctionnent. En particulier, essayez de rapprocher les opérateurs `/` et `mod` d'un outil mathématique que vous connaissez depuis des années.

3.b) Tests de comparaison et d'égalité

Il est également possible de comparer deux éléments entre eux. Pour cela, on utilise les opérateurs suivants : <, >, =, `!=`, <= et >=. Testez ces opérateurs sur les nombres entiers, en observant notamment le type du résultat renvoyé.

Essayez ensuite de comprendre comment fonctionnent ces opérateurs avec les chaînes de caractères. Quel est l'"ordre" ainsi établi ?

```
"bonjour"="bonjour";;
"bonjour"="hi";;
"bonjour"<"hi";;
"bonjour">"hi";;
"ab"<"ba";;
```

3.b) Attention au typage

Comme nous l'avons vu en cours, Caml ne vous laissera pas mettre sur le même plan des éléments qui n'ont pas le même type.

Pour vous en convaincre, testez les exemples suivants et observer le message renvoyé par l'interpréteur :

```
2 + 3.14;;
2 = "deux";;
```

3.c) Les nombres réels

Les opérations sur les nombres réels présentent en outre une autre difficulté : les opérateurs arithmétiques classiques ne fonctionnent pas. En effet, si vous tapez `1.1 + 1.2;;`, Emacs vous indiquera que vous essayez d'utiliser un nombre réel là où il attend un nombre entier.

L'explication est toute simple : les symboles `+`, `-`, `*` et `/` sont en fait liés aux nombres entiers. Si vous souhaitez faire des opérations sur les nombres réels, vous devrez utiliser la variante "pointée" de ces opérateurs : `+.`, `-.`, `*.` et `/.`. Par exemple, `1.1 +. 1.2;;`

Si l'on utilise *"des opérateurs avec des points pour les nombres avec des points"*, c'est parce que les nombre réels sont en fait des valeurs approchées, avec des méthodes de calculs différentes de celles des nombres entiers, et pour des usages différents de ces derniers.

Pour vous en convaincre, essayez d'évaluer `1.1 +. 1.3;;`.

4) Variables

Pour définir une variable en OCaml, on utilise la syntaxe suivante : `let [nom] = [valeur];;`

Entrez par exemple la ligne de code suivante : `let x = 3;;`. La réponse de l'interpréteur diffère légèrement des réponses précédentes :

```
# let x = 3 ;;
val x : int = 3
```

- Pour commencer, vous retrouvez toujours votre ligne de code :
- Puis apparaît désormais le mot clef `val` et le nom de votre variable (`x`)
- Puis le type du résultat renvoyé, ici un entier (`int`)
- Et finalement la valeur de ce résultat (`3`)

Ce mot clef `val` nous indique que nous venons de définir un nouvel élément (ce qui n'était pas le cas avec un tiret). Emacs nous annonce donc ici que nous avons défini un nouvel élément x de type entier et dont la valeur est 3.

Il vous est alors possible d'utiliser cet élément `x` dans les lignes de codes suivantes. Tapez par exemple les lignes de codes suivantes, et étudiez les réponses de l'interpréteur

```
x*4;;
let y = x*9;;
```

5) Constructions de base

5.a) Séquence d'instructions

Pour réaliser une action un peu complexe, vous aurez besoin d'exécuter plusieurs instructions. La syntaxe correspondante en Caml est la suivante :

```
instruction1;
instruction2;
…
instructionN;;
```

Pour les tests à venir, vous allez utiliser la fonction `print_endline`, qui prend en argument une chaine de caractère, affiche cette chaîne à l'écran, et ne renvoie aucun résultat. Par exemple, si vous tapez `print_endline("Hello world !");;`, l'interpréteur répondra

```
# print_endline("Hello World");;
Hello World
- : unit = ()
```

La première ligne correspond au code interprété, la deuxième au texte affiché pendant l'exécution du programme, et la troisième au résultat renvoyé (de type `unit`, c'est-à-dire rien).

Ecrivez une série d'instruction qui, lorsqu'elles seront interprétées, afficheront à l'écran le texte suivant, puis ne renverront aucun résultat :

```
Un Anneau pour les gouverner tous.
Un Anneau pour les trouver.
Un Anneau pour les amener tous et dans les ténèbres les lier.
- : unit = ()
```

5.b) Déclarer une fonction

Créer des fonctions vous sera très utile par la suite. Pour ce faire, vous pouvez utiliser la syntaxe suivante : `let [nom_fonction] [arg1] [arg2] … [argN] = [instructions] ;;`

- [nom_fonction] désigne le nom de la fonction
- [arg1], [arg2], … et [argN] sont les arguments de cette fonction
- [instructions] est la liste des instructions réalisées par cette fonction

Nous reviendrons plus en détails sur ces notions la semaine prochaine, mais sachez que les arguments sont en quelque sorte des variables utilisables dans la fonction dont la valeur n'est pas connue à l'avance. L'idée sera ensuite de pouvoir exécuter cette fonction pour plusieurs valeurs de ses arguments.

Par exemple, la fonction `oppose` définie ci-dessous prend en argument un entier `n` et renvoie l'opposé de cet entier :

```
let oppose n =
      -n;;
```

De la même façon, la fonction `somme_et_difference` ci-dessous prend en argument deux entiers `a` et `b`, affiche le somme et la différence entre ces deux nombres, et ne renvoie rien.

```
let somme_et_difference a b =
      print_int (a+b);
      print_endline("");
      print_int (a-b);;
```

On peut alors utiliser ces fonctions plus loin, pour n'importe quelles valeurs de `n`, `a` et `b` :

```
oppose(12);;
oppose(-3);;
somme_et_difference 40 2;;
somme_et_difference 10 10;;
```

Pour vous familiariser avec cette notion de fonction, créez et testez les fonctions suivantes :

- `carre`, qui prend en argument un entier `x` et renvoie le carré de `x`
- `affiche_deux_fois`, qui prend en argument une chaine de caractères `s`, qui affiche deux fois cette chaîne et ne renvoie rien.

5.c) If ... then ... else

Tapez la fonction `est_majeur` correspondant au pseudo-code suivant (`age` étant l'unique argument de cette fonction)

```
Si (age<18)
      Alors
            Renvoyer "mineur"
      Sinon
            Renvoyer "majeur"
Fin si
```

Imaginez ensuite les fonctions `minimum` et `maximum`, qui prennent chacun deux entiers en argument, et renvoient respectivement le plus petit et le plus grand de ces deux nombres.

Pour finir, imaginez une fonction `bissextile` qui prend en argument un entier `n` et indique (sous la forme d'un booléen) si l'année correspondant à cet entier est bissextile ou non.

Pour rappel, une année bissextile est

- soit divisible par 4 et non divisible par 100
- soit divisible par 400.

Remarque : `a` est divisible par `b` si et seulement si `a modulo b = 0`

5.d) Boucles `for`

Comme le suggère leur syntaxe, les boucles `for` permettent de répéter une série d'instructions un certain nombre de fois (fixé à l'avance) :

```
for [indice] = [premiere_valeur] to [derniere_valeur]
do
      instr1;
      instr2;
      …
      instrN-1;
      instrN      (La dernière instruction ne prend pas de point-virgule)
done;;
```

Utilisez une boucle `for` pour écrire une série d'instructions qui, une fois interprétée, affiche 25 fois le texte `"Les boucles for, c'est pratique"`.

Imaginez ensuite une fonction `repetition` qui prend en argument un entier `n` et permet d'afficher `n` fois le texte ci-dessus.

Tapez ensuite la série d'instruction correspondant au pseudo-code suivant (pour afficher un entier, utilisez la fonction `print_int`, et `print_endline("")` pour passer à la ligne).

```
Pour i allant de 1 à 10
      Faire
            Afficher i
      Fin faire
```

Déduisez-en une fonction qui permet d'affiche tous les nombres de 1 à `n`, puis une seconde fonction qui permet d'afficher tous les nombres de `a` à `b`.

Sachant que la fonction `print_string` permet d'afficher une chaîne de caractères sans passer à la ligne, comment pourriez-vous écrire une fonction `affiche_triangle` qui prend en argument un entier `n` qui affiche un triangle constitué de `n` lignes de `#`

Exemples :

```
affiche_triangle(3);;
#
##
###
- : unit : ()
```

```
affiche_triangle(5);;
#
##
###
####
#####
- : unit : ()
```

Comment modifier cette fonction pour que si l'entier n est négatif, le triangle soit dessiné la pointe vers le bas, comme sur l'exemple ci-dessous ?

```
affiche_triangle(-4);;
####
###
##
#
- : unit : ()
```

6) "Afficher " ou "renvoyer " ?

6.a) Attention !

Une erreur fréquente est la confusion entre :

- L'affichage d'une valeur
- Le renvoi d'un résultat

Commencez par évaluer les expressions suivantes, et examinez les réponses de Caml.

- `print_int(12);;`
- `12;;`

6.b) Quelques exemples

Testez ensuite les lignes de codes suivantes, et expliquez les éventuels messages rencontrés :

- `4 + 2;;`
- ```
  print_int(4);
  print_newline();
  print_int(2);;
  ```
- `print_int(4) + print_int(2);;`
- `let mot = "bonjour";;`
- `let word = print_string("hi");;`

Concevez ensuite les trois fonctions suivantes :

- `somme_renvoi`, telle que `somme_renvoi a b` renvoie la somme de a et b (sans rien afficher)
- `somme_affichage`, telle que `somme_affichage a b` affiche la somme de a et b mais ne renvoie rien
- `somme_affichage_et_renvoi`, telle que `somme_affichage_et_renvoi a b` affiche la somme de a et b, puis renvoie cette valeur.
  ```

Pour finir, imaginez une fonction `table_multiplication` qui prend en argument un entier `n` et qui affiche une table de multiplication "jusqu'à `n`"

Exemple :

```
table_multiplication(5);;
1     2     3     4     5
2     4     6     8     10
3     6     9     12    15
4     8     12    16    20
5     10    15    20    25
- : unit : ()
```

6.c) Résumé

Lorsqu'on **affiche une valeur** :

- L'instruction correspondante (par exemple `print_int(12);;`) est de type `unit`
- L'ordinateur ne calcule rien : il affiche "bêtement" à l'écran ce qu'on lui dit.
- On ne peut pas récupérer la valeur de ce qui est affiché (ce sont juste des pixels à l'écran)
- Si on est dans une fonction, on ne s'arrête pas forcément : on peut continuer à exécuter d'autres instructions ensuite

Lorsqu'on **renvoie un résultat** :

- L'instruction correspondante (par exemple `12;;`) est du même type que le résultat renvoyé (ici `int`)
- L'ordinateur sait ce qu'il manipule (ici un nombre entier, de valeur 12)
- On peut récupérer le résultat renvoyé (par exemple pour le stocker dans une variable)
- Si on est dans une fonction, le renvoi d'un résultat marque la fin de la fonction

7) Références et boucles `while`

7.a) Références : la théorie

Les variables permettent de stocker des valeurs, mais comme on l'a vu en cours, ces contenus sont figés : il n'est pas possible de modifier la valeur d'une variable une fois qu'on l'a déclarée.

Lorsqu'on veut pouvoir modifier le contenu d'un élément que l'on stocke, on utilise une **référence**.

La syntaxe est un peu particulière :

- Pour déclarer une référence, on utilise le mot clef `ref` après le signe égal :
  `let [nom] = ref [valeur] ;;`
  Exemple : `let x = ref 3 ;;`

- Pour obtenir la valeur actuelle de cette référence, on utilise un point d'exclamation ! :
  `![valeur] ;;`
  Exemple : `!x;;`

- Pour modifier cette valeur, on utilise le "deux points, égal" `:=`
  `nom := [nom_valeur];;`
  Exemple : `x := 4;;`

7.b) Références : la pratique

Appropriez-vous les notations précédentes en créant et en modifiant des références d'au moins deux types différents (par exemple `float` et `string`).

7.c) Remarque : déclarations à l'intérieur d'une fonction

Jusque-là, nous avons vu des syntaxes de la forme `let [nom] = [valeur];;`

Néanmoins, si vous devez déclarer une variable ou une référence à l'intérieur d'une fonction, utilisez plutôt la syntaxe `let [nom] = [valeur] in` (on a remplacé les double points-virgules par le mot clef `in`).

Un schéma classique est donc le suivant

```
let [nom_fonction] [args] =
      [debut_fonction]
      let [nom_variable] = [valeur_variable] in
      [suite_fonction]
      ;;
```

Nous verrons lors du prochain cours que cette syntaxe est liée à la notion de portée des variables.

7.d) Boucles `while`

Les références ne sont pas forcément faciles à prendre en main, mais elles se révèlent très pratiques dans le cadre d'une boucle `white`.

Comme on l'a vu en cours, et comme l'indique la syntaxe, l'idée est de répéter une série d'instructions tant qu'une condition est vérifiée :

```
while (cond)
do
      instr1;
      instr2;
      …
      instrN-1;
      instrN      (La dernière instruction ne prend pas de point-virgule)
done;;
```

Pour commencer, écrivez une série d'instructions qui affiche grâce à une boucle `while` tous les nombres de 1 à 10.

Modifiez ensuite votre code pour obtenir une fonction qui prend en argument un entier `n` et affiche, toujours grâce à une boucle `while`, tous les nombres de 1 à `n`.

7.e) Tirages aléatoires

De nombreux problèmes informatiques supposent qu'on dispose d'un générateur aléatoire, c'est-à-dire d'un outil capable de tirer des nombres au hasard.

En Caml, on utilise pour cela les lignes de code suivantes :

- `Random.self_init();;`

Exécutez cette ligne de code au début de votre fichier afin de "démarrer" le générateur aléatoire.

- `Random.int [borne_superieure]`

La fonction `Random.int` fonction prend un argument n et renvoie un nombre entier tiré au hasard entre 0 (inclus) et n (exclus).

Pour vous familiariser avec ces outils, imaginez une fonction `pile_ou_face` qui ne prend aucun argument et qui renvoie soit "pile", soit "face" (avec la même probabilité pour chaque).

Ecrivez ensuite une fonction `de_a_six_faces` qui renvoie un entier tiré au hasard entre 1 et 6.

Combinez ces techniques de tirage aléatoire avec une boucle `while` pour écrire une fonction `essaye_encore` qui :

- Tire un nombre au hasard entre 0 et 10
- Affiche ce nombre
- Recommence si ce nombre n'est pas égal à 10.

Comment pourriez-vous modifier cette fonction pour compter le nombre d'essai nécessaire avant d'obtenir un 10 ?

Sauriez-vous en outre modifier encore cette fonction afin que la valeur à atteindre (jusque-là fixée à 10) soit elle aussi tirée au hasard avant d'entrer dans la boucle `while` ?

8) Fonctions récursives

8.a) Syntaxe :

Pour définir une fonction récursive en Caml, on utilise le mot clef `rec` :

```
let rec [nom_fonction] [args] =
      [debut_fonction]
      [nom_fonctions] [args_bis]
      [suite_fonction];;
```

Attention à ne pas oublier le cas de base, sinon votre programme risque de boucler à l'infini (pour plus de sécurité, sauvegardez régulièrement votre code).

8.b) Premier exemple : la factorielle

Implémentez en Caml la fonction `factorielle` d'après le pseudo-code suivant :

```
Factorielle n =
      Si n = 0
            Alors 1
            Sinon n * Factorielle(n-1)
      Fin si
```

8.c) Suite de Fibonacci :

On définit la suite de Fibonacci de la manière suivante :

$$Fib(n) = \begin{cases} 0 \; si \; n = 0 \\ 1 \; si \; n = 1 \\ Fib(n-2) + Fib(n-1) si \; n > 1 \end{cases}$$

Ecrivez la fonction correspondante (vous aurez besoin de deux `if…then…else`).

Selon vous, peut-on écrire ce programme de façon plus intelligente ? Est-ce qu'on ne fait pas plusieurs fois les mêmes calculs ?

Imaginez (et si possible tapez) une nouvelle fonction permet de calculer *Fib(n)* en n étapes.

8.d) Binôme de Newton:

Le binôme de Newton (ou coefficients binomiaux), noté $\binom{n}{p}$, correspond au nombre de façons de choisir *p* éléments parmi *n*, sans considération d'ordre.

Que vaut $\binom{n}{0}$? Que vaut $\binom{n}{1}$?

Pour $0 < p \leq n$, on a la relation suivante : $\binom{n}{p} = \frac{n}{1} \times \frac{n-1}{2} \times \cdots \times \frac{n-p+1}{p}$

Ecrivez une fonction `binome n p` à partir de ces formules.

On a également la formule $\binom{n}{p} = \frac{n!}{p! \times (n-p)!}$ où $x! = factorielle(x)$

Pour finir, imaginez une nouvelle fonction `binome2 n p` qui utilise la fonction factorielle définie précédemment.