TME1 – Premières fonctions

Mise en place de l'éditeur de texte Vous pouvez développer en Ocaml dans n'importe quel éditeur de texte. Cependant, dans cette UE, nous vous conseillons d'utiliser l'éditeur VS Code avec l'extension Ocaml Platform, qui vous fournira un environnement de développement complet : coloration syntaxique, typage à la volée, auto-complétion, etc.

Pour cela, installez tout d'abord l'extension Ocaml Platform pour VS Code, soit en passant par le gestionnaire d'extension de VS Code, soit en ligne de commande :

```
code --install-extension ocamllabs.ocaml-platform
```

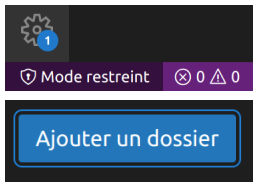
Enfin, copier et exécuter la commande suivante :

```
printf "\n. /etc/profile.d/lu2in019.sh\n\n" >> .bashrc
```

Ouvrir VS Code dans un **nouveau terminal** : celui-ci devrait maintenant reconnaître les fichiers Ocaml (en `.ml`). Pour ouvrir un fichier source Ocaml avec VS Code depuis le terminal, il suffit d'utiliser la commande :

```
code monfichier.ml
```

Il est possible que VS Code se lance en mode restreint, ce qui est indiqué par l'indication Mode restreint en bas à gauche de votre écran (voir capture d'écran à droite), ce qui empêche Ocaml Platform de fonctionner. Si cela arrive, cliquer simplement sur le bouton Mode restreint, puis sur le bouton Ajouter un dossier, et choisir votre dossier home. Enfin, relancer VS Code pour qu'il prenne en compte la nouvelle configuration.



Compilation et exécution Pour compiler un fichier source Ocaml, il suffit d'utiliser la commande :

```
ocamlc monfichier.ml -o monfichier
```

Cette commande va créer un exécutable `monfichier`, que vous pouvez alors exécuter avec la commande `./monfichier`. Il est possible de grouper ces deux étapes (compilation et exécution) en une seule étape, comme suit :

```
ocamlc monfichier.ml -o monfichier && ./monfichier
```

Utilisation d'un top-level Pour tester facilement notre code, nous utiliserons le top-level `utop`, qui permet d'écrire et d'exécuter du code Ocaml de façon interactive. `utop` peut être lancé depuis un terminal à l'aide de la commande :

```
utop
```

Il est alors possible d'écrire et d'exécuter du code Ocaml en temps réel depuis celui-ci. Contrairement au code que l'on écrit dans un fichier `.ml`, les entrés de `utop` doivent obligatoirement se terminer par deux points virgule `;;`, par exemple :

```
let x = 1;;
```

`utop` va alors évaluer cette phrase Ocaml et renvoyer :

```
val x : int = 1
```

qui indique qu'une nouvelle liaison associant le symbole `x` à la valeur `1` de type `int` a bien été déclarée.

Il est possible de demander à `utop` de charger les déclarations contenues dans un fichier `.ml`, à l'aide de la commande spéciale (attention à ne pas oublier le symbole `#`) :

```
#use "monfichier.ml";;
```

Attention, après une modification du fichier `monfichier.ml`, il est nécessaire de relancer la commande ci-dessus pour que `utop` recharge le fichier !

Utilisation des assertions Une bonne pratique consiste à ajouter après chaque définition de fonction un ensemble d'assertions permettant de tester votre code. Par exemple :

```
assert ((factorielle 3) = 6)
```

Nous vous demandons d'écrire cet ensemble d'assertions pour chacune de vos définitions de fonction. N'hésitez pas à utiliser les exemples donnés dans chaque question.

Exemple. Pour un exercice demandant de définir la fonction de signature `factorielle (n:int) : int`, il suffit d'ouvrir un fichier `ma_fact.ml` et d'y écrire votre définition accompagnée des assertions permettant de tester votre définition. Par exemple, si ce fichier contient la définition incorrecte ci-dessous :

```
let rec factorielle (n : int) : int =
  if n = 0 then 1 else 3 * (factorielle (n - 1)) ;;

assert ((factorielle 0) = 1) ;;
assert ((factorielle 3) = 6) ;;
```

la compilation de ce fichier ne produira aucune erreur (le code contenu dans ce fichier est syntaxiquement correct) :

```
$ ocamlc ma_fact.ml -o ma_fact
```

mais l'exécution de ce code produira une erreur :

```
$ ./ma_fact
Fatal error: exception Assert_failure("ma_fact.ml", 5, 0)
```

puisque l'assertion `assert ((factorielle 3) = 6)` échoue. Une fois l'erreur corrigée (en remplaçant `3` dans le corps de la fonction par `n`), la compilation et l'exécution ne produisent plus aucune erreur :

```
$ ocamlc ma_fact.ml -o ma_fact && ./ma_fact
```

Exercice 1.1 (Représentation décimale d'un entier naturel)**.**

1. Définir une fonction de signature `sum_chiffres (n : int) : int` qui calcule la somme des chiffres de la représentation décimale d'un entier n.

```
# (sum_chiffres 125);;
- : int = 8
```

2. Définir une fonction de signature `nb_chiffres (n : int) : int` qui calcule le nombre de chiffres significatifs contenus dans la représentation décimale d'un entier n.

```
# (nb_chiffres 125);;
- : int = 3
```

Indication. Combien de fois faut-il diviser 125 par 10 pour obtenir un quotient nul ?

Exercice 1.2 (Nombres premiers)**.**
Rappelons qu'un entier naturel d est un diviseur d'un entier naturel n si et seulement si le reste de la division entière de n par d est nul. Le nombre n est un nombre premier lorsqu'il n'admet pas d'autres diviseurs que 1 et lui-même (par convention 1 n'est pas un nombre premier).

1. Définir une fonction de signature `less_divider (i : int) (n : int) : int` qui calcule le plus petit diviseur de `n` compris entre `i` (inclus) et `n` (exclus) s'il existe, et 0 sinon.

```
# (less_divider 2 19);;          # (less_divider 5 21);;
- : int = 0                     - : int = 7
# (less_divider 7 21);;          # (less_divider 9 21);;
- : int = 7                     - : int = 0
```

2. Quel est le type de la fonction `less_divider` ?
3. En déduire une fonction de signature `prime (n : int) : bool` qui détermine si un entier naturel `n` est un nombre premier.

```
# (prime 1);;            # (prime 21);;            # (prime 19);;
- : bool = false         - : bool = false         - : bool = true
```

4. Définir une fonction de signature `next_prime (n : int) : int` qui calcule le plus petit nombre premier supérieur ou égal à l'entier naturel `n`. Cette fonction terminera-t-elle toujours ?

```
# (next_prime 0);; # (next_prime 15);;
- : int = 2       - : int = 17
```

5. L'ensemble des nombres premiers est dénombrable : on peut numéroter les nombres premiers c-à-d associer un entier naturel à chaque nombre premier. Les nombres premiers sont numérotés dans l'ordre croissant : 2 a le numero 0, 3 a le numero 1, 5 a le numéro 2, 7 a le numéro 3, etc. Définir une fonction de signature `nth_prime (n : int) : int` qui calcule le `n`-ième nombre premier.

```
# (nth_prime 0);;          # (nth_prime 2);;          # (nth_prime 12);;
- : int = 2               - : int = 5               - : int = 41
# (nth_prime 1);;          # (nth_prime 3);;
- : int = 3               - : int = 7
```

Exercice 1.3 (Portée statique, portée dynamique)**.**
Cet exercice est à faire sur machine avec les interprètes OCaml et Python (session interactive).

Rappel. Etant donné une fonction définie par `let` $f\ (x : t_x) : t_r = e_f$ et une expression e_a de type t_x, l'évaluation de l'expression $(f\ e_a)$ (application de la fonction f sur l'argument e_a) s'effectue comme suit :

(**1**) évaluation de l'expression e_a dans l'environnement courant pour obtenir la valeur v_a

(**2**) évaluation de l'expression e_f dans **un** environnement augmenté de la liaison (x, v_a)

Le but de cet exercice est de déterminer l'environnement d'évaluation du corps e_f de la fonction.

1. Introduire une définition pour que l'environnement courant contienne la liaison $(y, 6)$. On appelle E_1 cet environnement.
2. Dans l'environnement E_1, définir une fonction de signature `foo (x : int) : int` qui calcule `x + y`. Soit E_1' l'environnement obtenu après cette définition.
3. Que vaut `(foo 4)` dans l'environnement E_1' ?
4. Dans l'environnement E_1', introduire une (re)définition pour que l'environnement courant contienne désormais la liaison $(y, 18)$. On appelle E_2 cet environnement.
5. Que vaut `(foo 4)` dans l'environnement E_2 ?
6. (Portée statique) Dans quel environnement le corps d'une fonction OCaml est-il évalué lors de l'application de cette fonction à un argument ?
7. Soit le fichier `test.py` contenant le code Python ci-dessous :

```
y = 6
def f(x):
    return x + y
print(f(4))
y=18
print(f(4))
```

   Quels sont les affichages produits par l'exécution de ce code ?
8. (Portée dynamique) Dans quel environnement le corps d'une fonction Python est-il évalué lors de l'application de cette fonction à un argument ?