TP n°7 - Premières fonctions en OCaml

Notions abordées

- Compilation avec `ocamlc`, interprétation avec `utop`
- Jeu de test avec `assert` dans le fichier source
- Type et valeur d'expressions simples en Ocaml
- Définitions de fonctions, de fonctions récursives, de fonctions récursives terminales
- Fonctions à plusieurs arguments : type, curryfication
- Déclenchement d'exceptions avec `failwith`
- Variables de type
- Fonctions prenant en argument ou retournant des fonctions
- Définition de fonctions mutuellement récursives avec `and`

Toutes les fonctions doivent être commentées et testées. Elles doivent notamment être munies d'une description faisant suite à d'éventuelles hypothèses sur leurs arguments.
En OCaml, les commentaires s'écrivent comme suit `(*Commentaires*)`.

Pour commencer

Exercice 0 Prise en main

Question 1

Télécharger sur `cahier de prépa` le fichier nommé `exo0.ml`. Ouvrir un terminal pour le compiler grâce à la commande suivante : `ocamlc exo0.ml -o exo0`. Exécuter ensuite le fichier obtenu (*i.e.* `./exo0` comme pour un exécutable issu d'une compilation d'un code C). *Rien ne se passe, c'est normal, tout va bien.*

Question 2

Ouvrir un deuxième onglet dans le terminal (avec l'icône + ou bien dans Fichier/Nouvel onglet). Dans ce nouvel onglet lancer l'interpréteur `utop` grâce à la commande `utop`.
Copier/coller une à une les définitions dans l'interpréteur grâce aux raccourcis clavier Ctrl+C dans l'éditeur de texte, et Ctrl+Maj+V dans `utop`, puis ajouter deux points-virgules et taper Entrée pour lancer l'interprétation. Pour chaque définition, observer la réponse que fait l'interpréteur. De quoi est-elle composée ? Comprendre chaque réponse, et à défaut me demander des explications.

Question 3 🖳
Définir dans l'interpréteur une fonction `fois_6` qui prend en entrée un entier et qui le multiplie par 6 en faisant appel aux fonctions `double` et `triple`. *Plusieurs solutions possibles, au choix.*

Question 4 ✍ puis 🖳
Avant d'interpréter dans `utop` les expressions suivantes, vérifier qu'elles sont syntaxiquement correctes, et si tel est le cas, prédire leur type et leur valeur. Vérifier grâce à `utop`.

```
→ triple 2
→ triple double 5 = 30
→ triple (double 5) = double (triple 5)
→ double -5 = -10
→ triple (-5) = (-15)
→ triple (-3) = (-15)
→ texte_treize_a_la_douzaine 2
→ texte_treize_a_la_douzaine 12
→ treize_a_la_douzaine 18 + treize_a_la_douzaine 18
→ treize_a_la_douzaine 36
```

Question 5 🖳
Évaluer l'expression `"a\nb"` dans `utop`. Interpréter ensuite l'instruction `print_string "a\nb"`. Que remarque-t-on ?

Question 6 🖳 puis ✍
Fermer **utop** grâce à la commande `exit 0;;` ou au raccourci clavier Ctrl+D, puis relancer `utop`. Grâce à la flèche vers le haut, relancer les dernières évaluations. Que se passe-t-il ? Comment l'expliquer.

Question 7 🖳
Importer l'intégralité des définitions du fichier `exo0.ml` grâce à la commande `#use "exo0.ml";;`. Relancer alors les dernières évaluations.

Question 8 🖳
Modifier la fonction `texte_treize_a_la_douzaine` dans le fichier `exo0.ml` afin que le mot "trucs" soit au singulier quand la langue français l'exige. Observer les modifications en évaluant des appels bien choisis à cette fonction. *Vous veillerez à ne pas faire plusieurs appels identiques à la fonction* `treize_a_la_douzaine`.

Question 9 🖳
Créer un fichier `exo0_bis.ml` contenant une définition quelconque. Importer le fichier `exo0_bis.ml`, et évaluer une expression qui utilise la définition qu'il contient.

A priori l'import demandé à la question précédente n'a pas marché. Cela est dû au fait que l'interpréteur `utop` ne considère que les fichiers existant au moment où il est ouvert. On a donc 2 stratégies possibles : définir tous les fichiers en début de TP, avant d'ouvrir `utop`, ou bien sortir de `utop` puis le relancer à chaque fois que l'on crée un nouveau fichier. Attention dans ce cas à refaire les imports utiles, car l'environnement est perdu à chaque fois que l'on quitte l'interpréteur.

Exercice 1 Valeur absolue et jeu de test

Question 1 💻
Dans un fichier nommé `exo1.ml`, définir une fonction `abs` qui renvoie la valeur absolue d'un entier. Compiler le fichier (ce qui permet de vérifier que la définition proposée est syntaxiquement correcte).

Question 2 💻
Ouvrir un deuxième onglet dans le terminal et y lancer `utop`. Importer `exo1.ml` dans l'environnement `utop`. Tester la fonction `abs` en évaluant plusieurs appels à cette fonction.

Afin de garder une trace du jeu de test réalisé sur l'interpréteur, on utilise l'instruction `assert` dans le fichier `.ml`. Cette instruction prend en argument une expression booléenne. Lorsque cette instruction est réalisée, l'expression booléenne est évaluée. Si sa valeur est `true`, il ne se passe rien, on obtient comme valeur `()`, l'unique valeur du type `unit`. Si au contraire l'expression booléenne est évaluée à `false`, une exception appelée `"Assert_failure"` est levée.

Question 3 💻
Dans l'interpréteur, lancer les instructions `assert true` et `assert false`. Noter les réponses obtenues.

Question 4 ✍
Que dire des expressions suivantes ? *Essayez de répondre avant des les interpréter.*

```
assert (abs -3 = 3)
assert (abs (-1) = (-1))
```

Question 5 💻
Proposer un jeu de test pour la fonction `abs`, en interprétant dans `utop` plusieurs instructions `assert`.

On souhaite garder une trace des ces tests, et pour autant garder un fichier source propre, constitué d'une succession de définitions. Pour cela, on rassemble les instructions constituant le jeu de test sous une définition de valeur de type `unit`, dont le nom sera explicite. *Les instructions sont séparées par des point virgules. Attention la dernière n'est pas suivie d'un point virgule. De plus cette définition doit être placée après la définition de la fonction `abs` elle-même.*

Question 6 💻
Définir ici `test_abs` de type `unit` comme étant une suite d'instructions `assert` formant un jeu de test pertinent pour la fonction `abs`. Compiler puis exécuter. Pour s'assurer que le test a bien lieu, on peut ajouter une instruction d'affichage comme `print_string "test de abs réussi\n"`.

Exercice 2 Quelques fonctions vues en TD

Question 1 💻

À partir des définitions récursives vues en TD6, proposer, dans un fichier nommé `exo2.ml`, pour chaque fonction listée ci-dessous une définition et un jeu de tests. *Les questions qui suivent visent à améliorer le code de ces fonctions ou leur test, vous pouvez les ignorer dans un premier temps, ou au contraire les lire avant de vous lancer si vous souhaitez directement coder ces améliorations.*

- → `puiss`, calculant la puissance entière et positive d'un entier, et ce de manière naïve
- → `puiss_rt`, qui est la version récursive terminale de la fonction précédente
- → `puiss_rap`, calculant la puissance entière et positive d'un entier, et ce de manière rapide
- → `puiss_rap_rt`, qui est la version récursive terminale de la fonction précédente
- → `pgcd` calculant le PGCD de deux entiers positifs avec l'algorithme d'Euclide
- → `pgcd_rt`, qui est la version récursive terminale de la fonction précédente
- → `div_pos_rt`, calculant le couple (quotient, reste) d'un entier naturel par un entier naturel non nul, et ce manière récursive terminale
- → `div` calculant le couple (quotient, reste) d'un entier quelconque par un entier non nul.

Question 2 💻 ∗

Lorsqu'on a pour hypothèse la positivité d'un argument, on peut déclencher une exception en cas d'appel de la fonction avec un argument négatif. Le déclenchement de l'exception `"Failure"` suivie du message `m` se fait par l'instruction `failwith m`, où `m` est une expression de type `string`.
Utiliser cette façon de prévenir les erreurs d'exposant pour les fonctions qui calculent la puissance. Faire de même pour prévenir les erreurs de signe des arguments de la fonction `div_pos_rt` ou la nullité du diviseur de la fonction `div`.

On a parfois plusieurs fonctions calculant la même chose, auquel cas un jeu de test pertinent pour l'une l'est pour l'autre, et on se retrouve parfois avec des copier/coller de jeux de tests ou seul le nom de la fonction change. Afin de **factoriser ces jeux de tests**, on peut créer une fonction qui prend en argument une fonction et réalise le jeu de test avec cette fonction. Il ne reste plus qu'à appeler cette "méta-fonction-jeu-de-test" sur chaque fonction proposée pour le calcul en question.

Question 3 💻 ∗ ∗

Pour factoriser les jeux de tests des quatre fonctions calculant la puissance, définir une fonction `test_puissance_f` de type `(int -> int -> int) -> unit`, qui prend en argument une fonction `f` de type `int -> int -> int` et qui réalise les `assert` avec cette fonction `f`, par exemple :

```
assert (f 3 0 = 1)
assert (f 3 1 = 3)
```

Ensuite, pour tester la fonction `puiss`, il suffira d'appeler `test_puissance_f puiss`. Ainsi les tests des 4 fonctions calculant la puissance peuvent être rassemblés sous une même valeur de type `unit` appelée `test_puissance_4` qui consiste en 4 appels à `test_puissance_f`.

Suites récurrentes

Dans cette partie, on pourra omettre les descriptions des fonctions, qui est toujours de retourner un certain terme d'une suite définie dans l'énoncé.

Exercice 3 Suites récurrentes d'ordre 1 non paramétrées

Soient a, b et c les suites définies par :

$$\forall n \in \mathbb{N},\ a_n = \begin{cases} 8 & \text{si } n = 0 \\ 3\,a_{n-1} & \text{sinon} \end{cases} \qquad b_n = \begin{cases} 8 & \text{si } n = 0 \\ b_{n-1} + 6 & \text{sinon} \end{cases} \qquad c_n = \begin{cases} 8 & \text{si } n = 0 \\ 3\,c_{n-1} + 6 & \text{sinon} \end{cases}$$

Question 1 🖳

Dans un fichier nommé `exo3.ml` définir des fonctions `a`,`b` et `c` calculant respectivement a_n, b_n et c_n à partir d'un entier naturel n. *Ces fonctions n'ont pas besoin d'être récursives terminales.*

Question 2 🖳

Au début des trois fonctions `a`,`b` et `c` ajouter l'instruction `if n<0 then failwith "n<0" else ()` suivie d'un point virgule afin de déclencher une erreur en cas de rang négatif. Interpréter quelques expressions pour vérifier l'effet de ces ajouts.

Question 3 🖳

Définir des fonctions récursives terminales `a_rt` et `b_rt` calculant respectivement a_n, b_n à partir d'un entier naturel n. *On essayera de choisir des noms parlants pour les arguments du cœur récursif.*

Question 4 ✍ puis 🖳 $**$

En supposant que $n \geqslant 3$, exprimer c_n en fonction de c_{n-3} en procédant par substitutions successives. À chaque étape simplifier l'expression obtenue. Observer la forme de chaque expression, et comment agit une substitution sur ce type d'expression. En déduire une fonction récursive terminale `c_rt` calculant c_n à partir d'un entier naturel n.

Question 5 🖳 $*$

En repensant à la manière dont on aurait calculer un terme de la suite c en programmation impérative, définir une fonction récursive terminale `c_w` calculant c_n à partir d'un entier naturel n.

Question 6 🖳 $*$

Définir une fonction récursive, si possible récursive terminale qui prend en entrée un entier naturel n et qui calcule le terme de rang n de la suite u définie par $\forall n \in \mathbb{N}, u_n = \begin{cases} 5 & \text{si } n = 0 \\ 3\,u_{n-1} + 6 & \text{si } n \text{ est impair} \\ 4\,u_{n-1} + 5 & \text{sinon} \end{cases}$

Exercice 4 Suites récurrentes à paramètres

Question 1 💻

Dans un fichier nommé `exo4.ml` définir une fonction `v` récursive qui prend en entrée trois entiers a, b, n et qui calcule le terme de rang n de la suite v définie ci-dessous. *Cette fonction n'a pas besoin d'être récursive terminale.* $\forall n \in \mathbb{N}, v_n = \begin{cases} a & \text{si } n = 0 \\ b\, v_{n-1}^2 & \text{sinon} \end{cases}$.

Question 2 💻

Définir une fonction récursive terminale `v_rt` qui prend en entrée trois entiers a, b, n et qui calcule le terme de rang n de la suite v définie ci-avant.

Question 3 ✍

Le cœur récursif de la fonction `v_rt` doit-il prendre en argument les paramètres a et b de la suite v si ce cœur est une fonction définie en dehors de `v_rt` ? Et s'il s'agit d'une fonction définie localement à l'intérieur de `v_rt` ? En déduire un intérêt des définitions locales.

Question 4 💻

Définir une fonction qui prend en entrée 5 entiers a, b, c, d, n et qui calcule le terme d'indice n de la suite u définie par $\forall n \in \mathbb{N}, w_n = \begin{cases} a & \text{si } n = 0 \\ b & \text{si } n = 1 \\ c\, w_{n-1} + d\, w_{n-2} & \text{sinon} \end{cases}$

Exercice 5 Suites mutuellement récurrentes

Soient p et q les suites définies par :

$$\forall n \in \mathbb{N},\ p_n = \begin{cases} 3 & \text{si } n = 0 \\ 2\, q_{n-1} & \text{sinon} \end{cases} \qquad q_n = \begin{cases} 4 & \text{si } n = 0 \\ 7\, p_{n-1} + 6 & \text{sinon} \end{cases}$$

Question 1 💻

Dans un fichier nommé `exo5.ml` définir une fonction `pq` qui calcule le couple (p_n, q_n) à partir d'un entier naturel n. *Cette fonction n'a pas besoin d'être récursive terminale pour l'instant.*

Comme on le verra en cours, il est possible de définir des fonctions **mutuellement récursives**, c'est-à-dire qui peuvent s'appeler l'une l'autre sans qu'aucune ne soit définie avant l'autre. Pour cela il suffit de placer le mot clé `and` entre les deux définitions.

Question 2 💻

Dans un fichier nommé `exo5.ml` définir deux fonctions `p` et `q` calculant respectivement p_n et q_n à partir d'un entier naturel n. *Ces fonctions n'ont pas besoin d'être récursives terminales pour l'instant.*

Avec un peu plus d'abstraction...

Exercice 6 Variables de types

Question 1 ✍ puis 🖳
Selon vous, quel est le type (le plus général possible) de l'expression `fun x -> x` ? Afin de savoir quel type OCaml infère, interpréter cette expression sur `utop`.

La signature `'a -> 'a` signifie que le type de l'argument de cette fonction est quelconque (ça n'a pas de raison d'être un `int` plutôt qu'un `bool` ou un `bool*int*char`...), mais que le type de retour est nécessairement le même que celui de l'argument. Si le type de retour avait été complètement quelconque, sans lien avec le type du premier argument, il aurait été désigné par une autre variable de type.

Question 2 ✍ puis 🖳
Selon vous, quel est le type de l'expression `fun x -> (x,x)` ? De l'expression `fun x y -> x` ? Interpréter ces expressions pour vérifier.

Dans ces exemples on a laissé à OCaml le soin d'inférer les types, mais on peut aussi préciser la signature de la fonction lors de sa définition en utilisant des variables de type. En effet, n'importe quel identificateur précédé d'une apostrophe est une **variable de type**, qui désigne un type quelconque. Lors de l'appel de fonction, deux arguments dont le type est désigné par une même variable de type devront impérativement avoir le même type, tandis que deux arguments dont les types sont désignés par deux variables de type différentes pourront avoir ou non le même type. *Ça marche comme les variables en maths habituellement, un point de coordonnées (x, x) est nécessairement sur la diagonale, car son abscisse et son ordonnée sont égales, mais un point de coordonnées (x, y) peut être ou non sur la diagonale.*

Question 3 🖳 puis ✍
Interpréter les expressions suivantes, puis synthétiser le comportement observé :

```
fun (x:'a) :'a -> x
fun (x:'a) :'b -> x
fun (x:'b) :'b -> x
fun (x:'toto) :'toto -> x
fun (x:'toto) : -> x
fun (x) : 'toto-> x
```

Question 4 🖳
Définir dans un fichier nommé `exo6.ml` la fonction identité nommée `id` prenant en argument une valeur `x` et calculant cette même valeur. *On veillera à indiquer une signature complète lors de la définition.*

Question 5 ✍
Peut-on appliquer `id` sur un argument de type `int` ? de type `float` ? de type `float * int` ? de type `int -> int` ? En cas de doute interpréter une expression pour tester sur un exemple.

Question 6 🖳
Toujours dans `exo6.ml`, définir la fonction `diago` qui transforme un élément en un couple dont les deux composantes sont cet élément.

Question 7 ✍

Peut-on appliquer `diago` sur un argument de type `int` ? de type `float` ? de type `float * int` ?

Question 8 💻
Définir la fonction `couple` qui prend en argument deux éléments quelconques et qui calcule le couple composé de ces deux éléments.

Question 9 ✍
Peut-on appliquer `couple` sur deux arguments de type `int` et `int` ? de type `float` et `float` ? de type `float` et `int` ? de type `float * int` et `int` ?

Question 10 💻
Définir la fonction `comp1` (resp. `comp2`) qui calcule la première (resp. deuxième) composante d'un couple quelconque.

Exercice 7 Manipuler des fonctions

Question 1 ✍ puis 💻
Donner le type le plus général d'une fonction `application` prenant en arguments une fonction f et un élément x et calculant $f(x)$. Définir la fonction `application` dans un fichier nommé `exo7.ml`

Question 2 ✍ puis 💻
Donner le type le plus général possible pour une fonction `compose` prenant en arguments deux fonctions f et g et calculant la fonction $f \circ g$, sachant qu'il faut que la composition des fonctions prises en argument ait un sens. Définir la fonction `compose`.

Question 3 ✍ puis 💻
Donner le type le plus général possible pour une fonction `carre` prenant en argument une fonction f et calculant la fonction $f \circ f$, sachant qu'il faut que la composition ait un sens. Définir la fonction `carre` faisant appel à la fonction `compose`.

Question 4 💻
Grâce à la fonction `carre`, définir la fonction `plus2` qui ajoute 2 à l'entier qu'elle prend en argument, à partir de la `plus1` qui ajoute 1 à l'entier qu'elle prend en argument.

Question 5 💻 * *
Définir la fonction `itere` prenant en argument une fonction f et un entier n et calculant la fonction f^n, c'est-à-dire la composée de f n fois. *On attend ici une fonction récursive faisant appel à la fonction* `compose`.

Question 6 💻 *
Grâce à la fonction `itere`, définir une nouvelle fonction calculant la puissance par itération de la multiplication

Question 7 ✍ puis 💻
Donner le type le plus général possible pour une fonction `f_ou_id` prenant en arguments une fonctions f et un booléen b et calculant la fonction f si le booléen est vrai et la fonction identité sinon. Définir la fonction `f_ou_id`.

Exercice 8 Comment prendre deux données en entrée

Comme on l'a vu dans la fin de l'exercice 6, il existe deux façons pour une fonction de prendre en entrée deux données : soit elle prend en argument un couple qui rassemble ces deux données, soit elle prend en entrée deux arguments distincts. Puisqu'on a aussi vu dans l'exercice 7 que l'on peut définir des fonctions qui manipulent des fonctions, on va dans cet exercice s'attacher à définir des fonctions qui permettent de transformer une fonction de deux arguments, en une fonction équivalente qui prend en entrée un couple, ou le contraire.

Question 1 ✍
Dans cette question, on considère des fonctions prenant en entrée une donnée de type `a` et une donnée de type `b`, et calculant une donnée de type `t`. Quel est le type de la fonction si elle prend les deux données sous la forme d'un couple ? Et si elle les prend sous la forme de deux arguments ?
Quelle serait alors le type d'une fonction qui permettre de passer de l'une à l'autre ou de l'autre à l'une ?

Question 2 💻
Dans un fichier nommé `exo8.ml`, définir une fonction `curry` qui prend en argument une fonction `f` attendant une paire comme unique argument, et qui calcule une fonction à deux arguments `g` telle que pour tout `x`, `y`, on ait `(g x y) = (f (x, y))`.

Question 3 💻
Définir une fonction `uncurry` qui prend en argument une fonction `g` à deux arguments, et qui calcule une fonction `f` attendant un couple telle que pour tout `x`, `y`, on ait `(f (x, y)) = (g x y)`.

Question 4 💻
Afin de pouvoir tester les fonctions `curry` et `uncurry`, définir une fonction `somme1` (resp. `somme2`) qui prend en argument un couple d'entiers (resp. deux entiers) et qui calcule leur différence. Proposer ensuite des jeux de tests pour `curry` et `uncurry`. *On rappelle que l'opérateur `=` ne permet pas de tester l'égalité entre fonction. On se contente donc pour comparer des fonctions de vérifier si elles coïncident sur plusieurs appels bien choisis.*

Question 5 💻 ∗ ∗
Définir une fonction `transfo` qui prend en argument une fonction `h` à deux arguments le premier étant un couple et qui calcule une fonction `k` à deux arguments le deuxième étant un couple telle que pour tout `x,y,z`, on ait `(h (x, y) z) = (k x (y,z))`.