TRAVAUX PRATIQUES IV

Introduction à OCaml

OCaml est un langage :

- *fonctionnel* : il conçoit le calcul comme l'écriture d'une grosse formule mathématique dont on doit trouver la valeur.
- aux *variables immuables* : une fois défini, le contenu d'une variable ne peut plus être changé. [1]
- *impur* : on peut faire de la programmation « avec des changements de valeurs et des effets secondaires » si on le souhaite vraiment.
- *fortement typé* : une variable ne peut pas changer de types durant l'exécution, et on ne peut jamais combiner ensemble des « choses » de types différents.
- *statiquement typé* : le type de chaque variable est déterminé à la compilation. De plus, le compilateur est assez fort pour ne pas avoir besoin qu'on lui indique le type !
- *compilé* : le langage est muni d'un compilateur.
- *interpreté* : le langage est également muni d'un interpréteur.

Convention 1.
Dans cet énoncé de TP comme dans tous mes énoncés de TP, les chevrons `<` et `>` jouent le rôle de guillemets qu'il ne faut pas recopier.

A Premiers pas

Vous noterez que ces premiers pas sont assez théoriques : OCaml fonctionne différemment des langages auxquels vous êtes habitué·es et il est donc facile d'écrire un code invalide en voulant faire « comme d'habitude ». Je fais donc le choix de commencer en vous montrant les bases (un pavé de bases même), et ensuite seulement de vous donner la main.

N'hésitez pas à me poser plein (!!) de questions si vos codes ne fonctionnent pas !

A.1 Compilation

Pour compiler `file.ml`, on utilise le compilateur `ocamlopt` :

```
ocamlopt file.ml
```
Terminal

Si l'on a plusieurs fichiers, on les met les uns après les autres, en mettant en premier ceux dont les autres dépendent :

```
ocamlopt file0.ml file1.ml ... fileX.ml
```
Terminal

Le fichier compilé produit se nomme `a.out` par défaut. Pour créer un fichier nommé autrement, on utilise l'option `-o <nom_de_l_executable>`. Par exemple :

```
ocamlopt file.ml   -o toto.exe
```
Terminal

Nous verrons dans quelques semaines un interpréteur pour OCaml, mais il donne de mauvais réflexes aux débutant·es et je préfère donc vous en priver pour l'instant.

1. (NEW) Compilez le fichier `hello_world.ml`. À l'exécution, il devrait afficher le message `Hello World !`

1. Certain·es réfutent même la pertinence de parler de variable et parlent plutôt de *binding*. À titre personnel, je parlerai quand même de variable.

A.2 Commentaires, indentation

Définition 2 (Commentaires OCaml).
Les commentaires OCaml débutent par `(*` et se terminent pas `*)` . Ils peuvent tout à fait s'étendre sur plusieurs lignes.

Remarque. Les commentaires OCaml sont donc l'équivalent de `/* ... */` en C.

Définition 3.
Les retours à la ligne et l'indentation sont non significatifs.

Remarque. Conseils de présentation pour l'instant : à chaque fois que vous entre dans une « sous-structure », indentez. Cela aide à relire et à comprendre le fonctionnement.

A.3 Déclarations globales, déclarations locales

Définition 4 (Valeur, expression).

- Une **valeur** est une quantité « que l'on ne peut pas simplifier ». Par exemple, `63` ou `3.14` ou `"poitiers"` sont des valeurs.
- Une **expression** est une combinaison de valeurs. Par exemple, `5 - 2` est une expression. Notez que les valeurs sont des expressions.

Définition 5 (Structure et flot d'un programme OCaml).
Un programme OCaml est une succession de déclarations globales. Ce peut-être des déclarations de :

- variables/fonctions
- types (nous verrons cela plus tard)
- exceptions (idem)

À l'exécution, le programme effectue ces différentes déclarations de haut en bas, les unes après les autres.

Remarque.

- En fait, la syntaxe d'un programme OCaml ressemble à celle d'une preuve de maths : « Soit x = ... Soit f la fonction = ... Soit z = f(x). ... »
- En particulier, un appel de fonction ne peut pas « trainer tout seul sur une ligne ». Si on veut utiliser une fonction, on doit le faire dans une déclaration.

Définition 6 (Déclaration globale de variable).
our déclarer une variable globale nommée `var` , on utilise la syntaxe suivante (rappel : les `<` `>` jouent le rôle de guillemets à ne pas recopier) :

```
let <identifiant> =
  <expression>
```

Cela associe le nom `<identifiant>` à la valeur de `<expression>` . Le « contenu » de la « variable » `identifiant` n'est plus modifiable par la suite.[i]
Spécifités : Les noms de variables ne peuvent pas commencer par une majuscule. Le nom de variable `_` est particulier : il « oublie » immédiatement ce que l'on met dedans.

i. Rappel : les « variables » sont non-mutables !

Exemple. Le code ci-dessous déclare deux variables :

```
let x =
  5+3

let y =
  x*2
```

Proposition 7 (L).
rsque l'on effectue la déclaration ci-dessous :

```
let <identifiant> =
    <expression>
```

le compilateur infère automatiquement le type de `<expression>` et donne ce type à `<identifiant>` : il n'y a pas besoin d'indiquer le type.

Exemple. Dans l'exemple précédent, `x` est de type `int` car c'est la somme de deux entiers, et `y` aussi car c'est le produit de deux entiers.

Proposition 8 (Déclaration globale vs expression).
Une déclaration globale n'est pas une expression : elle n'a pas de valeur.
Une expression n'est pas une déclaration globale, et on ne peut donc pas écrire un code source qui soit une suite d'expressions. Un code OCaml est une suite de déclarations globales !

Exemple.

```
let x =
  let y = 3
```

Le code ci-dessus n'est pas valide car `let y = 3` est une déclaration globale et non une expression, et ne peut donc peut être la valeur donnée à `x` .

```
let x =
  5

3
```

Le code ci-dessus n'est pas valide car `3` n'est une déclaration globale, et ce code n'est donc pas une suite de déclarations globales.

Remarque. En fait, OCaml lit ce code comme `let x = 5 3` se qui signifie « la fonction `5` appliquée à l'argument `3` », ce qui n'a aucun sens (et ne compile donc pas).

Remarque. J'insiste peut-être un peu beaucoup sur ces notions, mais vous verrez qu'il s'agit de la plupart des erreurs de débutant·es.

Définition 9 (Déclarations locales).
On peut faire une déclaration locale, afin d'aider à la lisibilité :

```
(*/!\ Ceci est une expression, pas une déclaration globale !! *)
  let <identifiant> = <expr0> in
  <expr1>
```

Cette déclaration locale est une expression : d'abord on calcule la valeur de `<expr0>` et on la stocke dans `<identifiant>` , puis on calcule la valeur de `<expr1>` (en y remplaçant les occurences de `<identifiant>` par la valeur que l'on a trouvée). La déclaration locale a comme valeur (resp. comme type) la valeur (resp. le type) de `<expr1>`

Exemple.

Voici un exemple de déclaration locale en mathématiques, pour vous aider à comprendre : « En notant $a = 57831709$, considérons $3a^3 - 7a^2 + a - 3$ ». En OCaml, cela se traduit par l'expression ci-contre :

```
(* Ceci est une expression, pas une
   déclaration globale ! *)
 let a = 57831709 in
 3*a*a*a - 7*a*a + a - 3
```

Exemple. Le code ci-dessous est un code OCaml valide (enfin !) : c'est une déclaration globale, où l'expression utilisée pour définir `alpha` est un `let...in`

```
let alpha =
  let truc = 23-12 in
    truc*truc
```

2. (NEW) Prédisez la valeur de `alpha` . (Astuce de calcul : utilisez $(x+1)^2$). Solution en note de bas de page [2]

Exemple. Comme une déclaration locale est une expression, on peut « imbriquer » les déclarations locales :

```
let x =
  let y = 42 in
    let z = 33 in
      let t = y-z in
        3 + t*(2*z-y)
```

3. (NEW) Prédisez la valeur de `x` . (Astuce pour multiplier par 9 : multipliez par 10 et retranchez 1 unité.) Solution en bas de page. [3]

A.4 Types de base et opérateurs

Définition 10 (Entiers, flottants, booléens).

- Le type `int` est le type des entiers, signés. Ils sont encodés sur 63 bits. (Ce n'est pas 64 pour une raison technique hors-programme.)
  C'est le seul type d'entiers au programme.
- Le type `float` est le type des flottants, sur 64 bits (exactement comme en cours).
- Le type `bool` est le type des booléens, sur 64 bits : en fait, en OCaml, tout est sur 64bits (ce qui simplifie des choses hors-programme).
  Les deux booléens se notent `true` et `false` .

Voici les opérateurs usuels :

Opération	Opérateur
Addition	`+`
Soustraction	`-`
Multiplication	`*`
Division	`/`
Modulo	`mod`

(a) Opérateur de `int`

Opération	Opérateur
Addition	`+.`
Soustraction	`-.`
Multiplication	`*.`
Division	`/.`
Puissance	`**`

(b) Opérateur de `float`

Opération	Opérateur
Négation	`not`
Et	`&&`
Ou (inclusif)	`||`

(c) Opérateur de `bool`

FIGURE IV.1 – Opérateurs de base

2. `truc` $= 23 - 12 = 11$, donc `alpha` $= 11^2 = 121$
3. `t` $= 9$ donc `x` $= 3 + 9 * (2 * 33 - 42) = 3 + 9 * 24 = 3 + 216 = 219$

Définition 11 (Opérateurs de comparaison).

Opération	Opérateur
Égalité	`=`
Différence	`<>`
Strictement inférieur	`<`
Inférieur ou égal	`<=`
Strictement supérieur	`>`
Supérieur ou égal	`>=`

FIGURE IV.2 – Opérateurs de comparaison (valables la plupart des types)

Remarque. Notez que :

- Il n'y a pas de puissance sur les entiers.
- La puissance s'applique à deux *flottants*.
- ⚠ L'égalité est `=` et non == : ne confondez pas ![4]
  De même, la différence est `<>` et non `!=` .

A.5 Commentaires, retours à la ligne, indentation

Définition 12 (Commentaires, retours à la ligne, indentation).

- Les commentaires en OCaml débutent par `(*` et terminent par `*)` .
- Les retours à la ligne, comme en C, ne sont pas significatifs. Mais comme en C, il est conseillé de les utiliser afin d'écrire un code lisible.
- De même pour l'indentation.

Exemple.

```
let six = (* c'est une super variable !*)
  3 + (* j'avais
envie d'écrire
  des trucs ici*)              3
```

A.6 Premiers essais

Dans cette partie, vous coderez en modifiant `partie-A/debut.ml` . Les tests sont dans `partie-A/testA.ml` , et pour compiler vous ferez donc (une fois dans le bon dossier) :

```
ocamlopt debut.ml testA.ml
```
Terminal

(En ajoutant éventuellement l'option `-o` .)

4. Modifiez `partie-A/debut.ml` pour que :

- **a.** `x` contient 42
- **b.** `y` contient le reste de la division de 42 par 4 (n'écrivez pas la réponse, utilisez l'opérateur adapté)
- **c.** `z` contienne y au cube

5. De même, faites en sortie que `pi_carre` contienne 3.14^2

4. D'autant plus que `==` a un sens en OCaml : c'est l'égalité physique, c'est à dire l'égalité des valeurs *et* des adresses mémoires où sont rangées ces valeurs.

6. Relisez les deux premières définitions de ce TP. Puis relisez-les encore.[5][6]
7. (NEW) Notons P le polynôme $x \mapsto 473x^{16} - 763x^8 + 55x^2 - 6721123x - 42$. Modifiez la déclaration de `p_3` pour qu'elle contienne $P(3)$. M'appeler pour vérification.[7]

A.7 Conditionnelles

Les sauts conditionnels existent aussi en OCaml. Cependant, il y a une différence fondamentale avec C : en OCaml, les if-then-else sont des expressions : ils n'indiquent pas un embranchement du code à prendre, ils sont une valeur !

Définition 13 (if-then-else).
Voici la syntaxe d'un if-then-else :

```
if <condition> then <expr0> else <expr1>
```

Ce if-then-else est une expression dont la valeur est soit celle de `expr0` soit celle de `expr1`, selon la condition. En particulier, comme une expression ne peut avoir qu'un seul type[i], il **faut** que `expr0` et `expr1` ait le même type.

i. Je sais, c'est la première fois que je le dis de manière explicite : une expression ne peut avoir qu'un seul type (maintenant ça fait deux fois).

Exemple. Ci-dessous une version mathématique et son équivalent OCaml :

Définissons d comme ci-dessous (où $e = 5$) :

$$d = 2 + \begin{cases} e * 2 & \text{si 55 est pair} \\ e - 1 & \text{sinon} \end{cases}$$

```
let d =
  let e = 5 in
    1 + (if 55 mod 2 = 0 then e*2
     ↪  else e-1)
```

8. (NEW) Je note s la fonction qui à x entier associe $3x + 1$ si x est impair, et $x/2$ sinon. Modifiez la variable `babar` pour qu'elle vaille $s($`p_3` $- 5)$.

Remarque. Cet exemple de `if-then-else` est un peu artificiel : c'est normal, nous n'avons pas encore vues les fonctions, et c'est en faisant « varier les entrées » que les conditionnels prennent tout leur sens.[8]

B Effets secondaires : introduction

OCaml est un langage fonctionnel *impur*, et il est donc possible d'avoir des effets secondaires. Nous n'en ferons pas pendant très longtemps, mais l'un d'entre eux est utile pour déboguer : l'affichage.

B.1 Fonctions d'affichage

Les fonctions d'affichage sont les suivantes : `print_int` , `print_float` , `print_char` (pour les caractères, entre guillemets simples) et `print_string` (pour les textes, entre guillemets doubles). Chacune de ces fonctions affiche une valeur, prise en argument ; et rien de plus.

Exemple. Le code ci-dessous est une déclaration globale qui a pour effet d'afficher 4. Notez qu'en OCaml, il n'y a pas de parenthèses pour appeler une fonction :

```
let _ =
  print_int 4
```

Cette déclaration calcule l'image de `4` par `print_int` , et la stocke dans le « trou noir » `_` . En fait, cette image ne nous intéresse pas : ce qui nous intéresse, ce sont les *effets secondaires* qui ont eu lieu lors du calcul de l'image qui nous intéresse (en l'occurence, modifier des pixels à l'écran).

5. Je vous encourage par exemple fortement à finir ou refaire les TP chez vous. J'aurai toujours de quoi ~~nous amuser~~ vous occuper si vous allez trop vite.
6. Notez que je suis sûr qu'il y a des élèves de MPI qui en juin divisent par 0 en maths, ou oublient la réaction dans un bilan des forces. Les erreurs "de base" sont souvent des erreurs d'inattention, et sont communes - mais couteuses.
7. Nan, pas d'astuce de calcul qui rend cela agréable cette fois. La magie a ses limites.
8. Un pas après l'autre !

Si l'on veut enchainer les appels à des fonctions ayant des effets secondaires, l'astuce est d'utiliser des `let...in` (éventuellement imbriqués) :

```
let _ =
  let _ = print_int 3 in
    let _ = print_string " = " in
      let _ = print_float 3.14 in
        let _ = print_string " = " in
          let _ = print_int 10 in
            print_char '\n'
```

L'exemple ci-dessous affiche `3 = 3.14 = 10` , et termine par un retour à la ligne.

Remarque.

- Sur cet exemple, on commence à comprendre pourquoi il peut être agaçant d'indenter d'un cran de plus « à chaque fois que l'on entre dans une sous-structure ». Plus tard dans l'année [9], j'arrêterai d'indenter après les `in` . Cependant, pour l'instant, cela aide à comprendre.
- Le `printf` « comme en C » existe : c'est la fonction `Printf.printf` . Elle fonctionne comme en C (si ce n'est que la syntaxe pour passer des arguments n'est pas la même en C et en OCaml). Ainsi, le code C :

  ```
  printf("truc afficher%d ; autre truc : %d\n", x, y)
  ```

  correspond à l'expression OCaml :

  ```
  Printf.printf "truc afficher%d ; autre truc : %d\n" x y
  ```

- ⚠ Mettre un print tout seul sur une ligne, après une déclaration, pour la « tester comme en C/Python » est une des erreurs les plus communes quand on débute. Un programme OCaml est une suite de déclarations, si vous voulez afficher il vous faut donc une déclaration pour cela !

9. Refaites la question 6.

10. Créez un nouveau dossier `partie-B` (soit à l'aide de `mkdir` , soit à l'aide de l'explorateur de fichiers visuel). Créez-y un nouveau fichier `fun.ml` . Dedans, faites afficher 44 puis 63. Testez.
Rappelons que le retour à la ligne est le caractère `'\n'` .

B.2 Type unit

Maintenant que l'on a vues les fonctions d'affichage, une question légitime à se poser est : mais qu'est-ce donc que ce qu'elles renvoient ? La réponse est : *rien, mais pas exactement.*

En fait, dans l'idée elles ne renvoient rien. Cependant, on a besoin qu'elles renvoient une valeur ! Imagginons deux secondes que ce ne soit pas le cas. Dans ce cas, le code ci-dessous :

```
let x =
  print_int 0
```

aurait un problème fondamental : `x` n'aurait pas de valeur. C'est impossible, une variable a une valeur (non modifiable) et un type.

Ainsi, il est nécessaire que les fonctions d'affichages renvoient quelque chose : en fait, on crée une valeur spéciale pour représenter le rien.

Définition 14 (`unit`).
La valeur `()` est la valeur utilisée pour représenter « rien ». Son type est `unit` , et c'est la seule et unique valeur de ce type.

Remarque.

- C'est comme si en C, on avait crée une (et une unique) valeur dans le type `void` pour « manipuler la sortie des fonctions qui ne renvoient rien ».

9. Probablement plus tôt que tard d'ailleurs.

- C'est un peu l'équivalent OCaml du `None` de Python.[10]

Définition 15 (`else` vide).
Dans un `if-then-else`, on peut omettre le `else`. Cela équivaut à écrire `else ()`, autrement dit « Sinon : rien ».
Notez que cela oblige le `then` à être également de type `unit` puisque les deux branches doivent avoir le même type.

11. (NEW) Créez un dossier `partie-B` (à l'aide de `mkdir` - appelez moi au besoin - ou du gestionnaire de fichiers visuel). Dedans, recopiez le code ci-dessous :

```
let x = -20
let y = 23
let _ =
  if x-y > 0 then print_int (x-y) else print_int (y-x)
```

a. Expliquez ce que fait ce code.

b. Modifiez-le pour qu'il affiche également un retour à la ligne après avoir affiché ses entiers, *sans* utiliser `Printf.printf`.

C Fonctions

C.1 Syntaxes et fondamentaux

C.1.i : Fonctions anonymes

Définition 16 (Fonction anonyme).
En mathématiques, « $x \mapsto x * x$ » désigne une fonction sans lui donner de nom. On peut faire pareil en OCaml : `fun x -> x*x` est la fonction carré.

Remarque.

- Le mot clef `return` **n'existe pas** en OCaml (comme en maths). En OCaml (comme en maths), on écrit simplement la valeur que la fonction calcule.
- ⚠ C'est aussi une des erreurs usuelles quand on passe d'un langage à l'autre : mettre ou oublier des `return`. Faites attention !

Définition 17 (Application de fonctions).
i `f` désigne une fonction, alors `f x` désigne l'image de `x` par la fonction `f`.
⚠ Cette syntaxe n'est pas la même qu'en C !!

Exemple. La déclaration ci-dessous définit `demo` valant $3 * 15 + 3 = 48$.

```
let demo =
  (fun x -> 3*x+3) 15
```

C.1.ii : Fonctions pas anonymes

On utilise rarement des fonctions anonymes, car elles sont peu réutilisables : il faut recopier leur définition à chaque fois qu'on veut les utiliser. On préfère souvent leur doner un nom via une déclaration (globale ou locale).

Exemple.

10. Cette analogie a des limites, ne la poussez pas trop loin.

```ocaml
let carre =
  fun x -> x*x

let neuf =
  carre 3
```

C.1.iii : Parenthésage implicite

En cas d'ambiguïté, OCaml parenthèse à gauche d'abord. Ainsi, `x y z t` est un raccourci pour `((x y) z) t`.

12. Selon-vous, que se passe-t-il si l'on enlève les parenthèses dans le code ci-dessous ? Tester dans un fichier créé pour l'occasion pour confirmer.
[Ne pas hésiter à m'appeler !!!]

```ocaml
let quatre_vint_un =
  let carre = fun x -> x*x in
    carre (carre 3)
```

C.1.iv : Exercices

13. Créez un fichier `partie-C/fonctions.ml`. Créez-y les fonctions suivantes :

- **a.** `double` doit multiplier par 2
- **b.** `cst_0` doit être fonction constante égale à `0`.
- **c.** `id` doit être la fonction identité.
- **d.** (NEW) Déclarer la fonction `est_pair` qui prend un entier en argument et s'évalue à `true` si cet est pair et à `false` sinon.
  Attention à ne pas vous tromper dans les opérateurs de modulo / égalité / différence !
- **e.** `est_bissextile` est la fonction qui "renvoie" `true` si l'année passée en argument est bissextile [11] et `false` sinon.
- **f.** Refaites la question 6.

14. Compilez votre fichier avec `testC1.ml` pour tester, ou écrivez vos propres tests.
(Attention, `testC1.ml` appelle toutes ces fonctions et a donc besoin qu'elles soient toutes définies et du bon type.)

C.2 Fonctions d'ordre supérieur

C.2.i : Des exemples en maths

Une fonction (mathématique ou informatique) est dite *d'ordre supérieur* lorsque qu'elle travaille sur d'autres fonctions, c'est à dire qu'elles prennent en entrée ou renvoient d'autres fonction.

Pour rappel, une fonction est une « machine » qui prend en entrée un *truc* et le transforme en *autre chose*. Cet *autre chose* peut tout à fait être une autre machine.

Exemple. La fonction mathématique $\mathscr{D}$ suivante est une fonction d'ordre supérieure car son argument et son image son des fonctions :

$$\mathscr{D} : \begin{array}{c} \mathscr{C}^1(\mathbb{R}) \to \mathscr{C}^0(\mathbb{R}) \\ f \mapsto f' \end{array}$$

Cette focntion $\mathscr{D}$ est la fonction qui à une fonction f associe sa dérivée f'. Ainsi, $\mathscr{D}(x \mapsto x^2) = (x \mapsto 2x)$, et donc $\mathscr{D}(x \mapsto x^2)(3) = 6$.

15. Appelez-moi si je vous ai perdu·e.

11. Une année est bissextile si elle est divisble par 4 mais pas par 100. Par contre, les années divisibles par 400 sont quand même bissextiles.

Exemple. La fonction mathématique *douze* suivante est d'ordre supérieur car son entrée est une fonction :

$$\begin{aligned} & & C^\infty(\mathbb{R}) &\to \mathbb{R} \\ douze\ : & & f &\mapsto f(12) \end{aligned}$$

Ainsi, $douze(x \mapsto x^2) = 144$.

Exemple. La fonction mathématique *lin* suivante est d'ordre supérieur car sa sortie est une fonction :

$$\begin{aligned} & & \mathbb{R} &\to \mathscr{C}^\infty(\mathbb{R}) \\ lin\ : & & \lambda &\mapsto (x \mapsto \lambda.x) \end{aligned}$$

Ainsi, $lin(3) = (x \mapsto 3x)$ et donc $lin(3)(4) = 3.4 = 12$.

16. Appelez-moi si ce n'est toujours pas clair.

C.2.ii : Et en OCaml

Définition 18.
En OCaml, les fonctions sont des briques de bases au même titre que les entiers ou les booléens : on peut donc les prendre en argument ou les renvoyer.

Remarque. Ce n'est pas pour rien que l'on dit qu'OCaml est un langage *fonctionnel.*[12]

Exemple. La fonction *douze* précédente s'écrit en OCaml :

```
let douze =
  fun f -> f 12
```

Exemple. La fonction *lin* précédente s'écrit en OCaml :

```
let lin =
  fun lambda -> (fun x -> lambda * x)
```

Notez qu'on aurait pu/dû utiliser `*.` pour multiplier des flottants et non des entiers, puisque dans la version mathématique λ est un réel. Mais je préfère simplifier.

C.2.iii : Exercices

17. (NEW) Créez `partie-C/funSup.ml` . Dedans, codez les fonctions suivantes :

- **a.** `off_by_one` : à une fonction f associe la fonction qui à x associe $f(x+1)$.
- **b.** `divise` : à un flottant d associe la fonction qui à x associe $\frac{x}{d}$.
  On ne cherchera pas à vérifier que $d \neq 0$.
- **c.** `affine` : à un entier a associe la fonction qui à un entier b associe la fonction qui à x associe $ax+b$.
- **d.** `mini` : à une valeur x associe la fonction qui à y associe le minimum entre x et y. *On utilisera un* `if-then-else`

C.3 Fonctions à plusieurs arguments

Vous l'avez peut-être senti : avec les fonctions d'ordres supérieur, on a *presque* des fonctions à plusieurs arguments. En fait, on les a déjà - à un tour de passe-passe mathématique près.

12. Fun fact : le modèle théorique ayant inspiré tous les langages fonctionnels est le λ-calcul. C'est un modèle de calcul dans lequel il n'y a **que** des fonctions. Pas d'entiers, pas de booléens, pas de réels, **que** des fonctions. C'est spécial au début.

C.3.i : (NEW) Curryfication

Considérons une fonction à deux arguments fort sympathique, par exemple $plus : x, y \mapsto x+y$. Ainsi, $plus(2,3) = 5$.
On peut [13] fixer une des coordonnées : notons g_2 la fonction $y \mapsto plus(2, y)$. Avec cette notation, $plus(2,3) = g_2(3) = 5$. On dit que g_2 est *l'application partielle* de *plus*.
Une fois réécrit comme cela, on peut réécrire *plus* de la fonction suivante pour qu'elle prenne un seul argument :

$$plus : x \mapsto (y \mapsto x + y)$$

Autrement dit, on a transformé une fonction à deux arguments en la fonction qui à son premier argument associe l'application partielle de ce premier argument. Ce procédé s'appelle la **curryfication**. [14]

Exemple. La fonction $f : x, y, z \mapsto x^y + z$ se curryfie en $f : x \mapsto \big(y \mapsto (z \mapsto x^y + z)\big)$. Avec cette version curryfiée, $f(2)(3) = (z \mapsto 2^3 + z)$.

Autrement dit, on sait déjà faire des fonctions à plusieurs arguments !
La transformation inverse s'appelle la décurryfication. [15] Ainsi, la version décurryfiée de `off_by_one` est $(f, x) \mapsto f(x+1)$.
Pour les questions ci-dessous, écrivez les fonctions demandées dans `partie-C/funCurry.ml`

18. (NEW) Sur papier, écrivez des versions mathématiques dé-curryfiées de `divise` , `affine` et `mini` (comme pour `off_by_one` ci-dessus). Appelez-moi pour valider.

19. (NEW) Écrire une fonction `difference` (curryfiée) qui à x et y associe $|x - y|$

20. (NEW) Écrire une fonction `mini3` (curryfiée) qui renvoie le minimum de 3 valeurs.

21. (NEW) Si ce n'est pas déjà le cas, réécrivez `mini3` pour qu'il n'y ait pas de sauts conditionnels dans son code. Vous pouvez utiliser `min` (qui est la fonction `mini` précédente, présente de base dans OCaml).

22. (NEW) Écrire une fonction `est_multiple` (curryfiée) qui à un entier d et un entier m associe `true` si m est un multiple de d, et `false` sinon.

C.3.ii : Une syntaxe plus confortable

Définition 19 (Fonction anonyme à plusieurs arguments).
Plutôt que d'écrire `fun x -> fun y -> ...` , on peut écrire `fun x y -> ...`
De même avec 3 arguments : `fun x y z -> ...`
Et ainsi de suite.

Remarque. Nous verrons dans deux semaines une syntaxe encore plus agréable pour les fonctions, mais pour l'instant celle-ci nous suffit.

Proposition 20 (Toutes les fonctions OCaml sont curryfiées).
En OCaml, toutes les fonctions à plusieurs arguments sont curryfiées, et on peut donc faire de l'application partielle des premiers arguments.

Exemple. L'exemple ci-dessous se base sur l'explication de la curryfication précédente :

```
let plus =
  fun x y -> x + y

let g_2 =
  plus 2

let cinq =
  g_2 3
```

13. comme en physique
14. Du nom de H. Curry.
15. Pas très inspiré, je sais.

D Exercices finaux

Pour les exercices ci-dessous, vous devrez coder des tests vous-même.

23. (NEW)

a. Écrire une fonction `comp` qui prend en argument deux fonctions et renvoie leur fonction composée.

b. Testez-la ! Par exemple en composant `carre` et $x \mapsto x+1$, dans un sens puis dans l'autre, et vérifiant que vous avez bien le bon résultat à chaque fois.

24. Déclarer une fonction `implique` qui prend en entrée deux booléens `a` et `b` et s'évalue en `a` $\implies$ `b` .
Demandez-moi si vous ne savez pas comment écrire cette implication sans le symbole $\implies$.

25. Que devrait-être la fonction `h` suivante ? Pourquoi cela ne fonctionne-t-il pas selon vous ? Corrigez-la.

```
let h = let f = fun x -> x in let g = fun x->(-x) in min f g
```

26. Implémenter en OCaml les fonctions mathématiques suivantes [16]. Vous utiliserez des flottants pour représenter les réels :

a. `foo0` qui à $f : \mathbb{R} \to \mathbb{R}$ associe $\frac{f(0)+f(1)}{2}$

b. `foo1` qui à $f : \mathbb{R} \to \mathbb{R}$ associe $f * f$, où $f * f : x \mapsto f(x) * f(x)$

c. `foo2` qui à $f : E \to E$ associe $f \circ f$

d. `foo3` qui à $f : \mathbb{R} \to \mathbb{R}$ associe $g : x \mapsto f(x+1)$

e. `foo4` qui à $f : \mathbb{R} \to \mathbb{R}$ associe $\left[f\left(\frac{(f\circ f)(0)+(f\circ f)(1)}{2}\right)+3\right]^4$
Il s'agit bien de la puissance 4 et non, pour une fois, d'une note de bas de page.

27. (Bonus, difficile) Écrire une déclaration OCaml qui s'affiche elle-même. On appelle un tel code un programme autoreproducteur. [17]
Pseudo-indication : utilisez Printf.printf .
Indication : une fois que vous avez une déclaration presque auto-reproductrice, regardez les différents `%` *qui existent et demandez-vous si l'un d'eux peut vous aider.*

28. Retournez lire les deux premières définitions. J'insiste. Qu'est-ce qu'une valeur ? Une déclaration ? Une expression ? Quels sont les liens entre ces notions ? Comment les utilise-t-on pour faire un programme OCaml ? Vous ne devez avoir *aucune* hésitation (ce doit être tout aussi évident que le `;` en C).

29. (NEW) Proposez une blague sur les chameaux.

30. (NEW) Proposez une question 30.

16. Adapté d'un TP de Nicolas Pécheux, merci à lui !
17. C'est un concept à la base de beaucoup d'attaques informatiques. La toute première de l'histoire d'internet était un code autoreproducteur laissé libre de se cloner à l'infini sur un réseau ! C'est aussi une histoire sombre, où l'auteur voulait tester la faisabilité de la chose mais n'avait pas conscience des dégats que son virus allait causer... ni que cela finirait au tribunal.