ENSIBS – 3ème année, spécialité cybersécurité (parcours CyberLog FISE) – Programmation fonctionnelle

TD/TP 2a : programmation fonctionnelle, OCaml

TRÈS IMPORTANT : vérification des programmes

Testez systématiquement tous les exemples OCaml qui ont été donnés. Testez, avec encore plus de soin, les petits programmes OCaml que vous avez vous-même écrit. Vous devez considérer comme faux un programme que vous n'avez pas testé.

Partie 1 : fonctions

Section 1.1 (expressions booléennes)

Exercice 1.1.1

Écrire une fonction « `bissextile` » qui a la signature « `int -> bool` » et qui détermine si une année est bissextile. Votre fonction **ne devra pas utiliser** la structure « `if ... then ... else` ».

Rappel : une année est bissextile si elle est multiple de 4, mais pas de 100 à moins qu'elle soit aussi multiple de 400.

Section 1.2 (type char et string)

Fonction OCaml	Type	Description
`String.length ch`	`string -> int`	Le nombre de caractères de `ch`.
`String.get ch n`	`string -> int -> char`	Le n-ième caractère de `ch`.
`String.sub ch d l`	`string -> int -> int -> string`	La sous-chaîne de `ch` de taille `l`, à partir du `d`-ième caractère.

Exercice 1.2.1 (problème de conjugaison)

L'objectif de cet exercice est d'écrire une fonction de conjugaison des verbes du *premier groupe* au *futur*. Pour cela nous allons réaliser une fonction qui donne la conjugaison à une personne donnée d'un verbe donné.

Ecrire une fonction « `conjugue: bool -> int -> string -> string` » qui prend en argument :

1. un booléen spécifiant le nombre (singulier (true), pluriel (false)),
2. un entier pour le pronom personnel (compris entre 1 et 3) et
3. une chaîne de caractères pour le verbe.

Le résultat de la fonction est la chaîne de caractères formés du bon pronom et du verbe conjugué au futur.

Voici quelques exemples :

```
# conjugue true 2 "chanter";;
(* verbe chanter à la 2eme personne du singulier *)
- : string = "tu chanteras"

# conjugue false 1 "manger";;
(* verbe manger à la 1ere personne du pluriel *)
- : string = "nous mangerons"
```

En cas de paramètres non valides, la fonction devra déclencher une exception comme **par exemple :**

```
# conjugue false 5 "chanter";;
Exception : Failure "Le 2eme parametre doit etre compris entre 1 et 3"

# conjugue true 1 "dormir";;
Exception : Failure "dormir n'est pas un verbe du premier groupe"
```

Pour résoudre ce problème nous devons le décomposer en sous-problèmes plus faciles à traiter. Une décomposition est proposée mais chacun est libre de résoudre le problème à sa manière.

1. Ecrire une fonction « `pronom: bool -> int -> string` » qui prend en argument :
   - un booléen (singulier(false), pluriel(true)) et
   - un entier `i` (1 à 3)

   et retourne un pronom.
   Une exception sera déclenchée si `i` n'est pas valide.
   Exemple :
   ```
   # pronom true 1 ;;
   - : string = "je"
   ```

2. Ecrire une fonction « `premier_groupe : string -> bool` » qui prend en argument :
   - une chaîne de caractères supposée être un verbe

   et vérifie si ce verbe se termine par "er" (utiliser `String.get` et `String.length`).
   Exemple :
   ```
   # premier_groupe "chanter";;
   - : bool = true
   ```

3. Ecrire une fonction « `terminaison : bool -> int -> string` » qui retourne la terminaison des verbes en "er" au futur.
   Exemple :
   ```
   # terminaison false 3;;
   - : string = "a"
   ```

4. La fonction « `conjugue` » consistera à vérifier si le verbe est en "er" dans ce cas le résultat sera la concaténation du pronom, du verbe et de la terminaison, sinon une exception devra être déclenchée.

Partie 2 : système de typage

Chaque expression écrite en OCaml est associée à un type. Une expression mal typée est une expression incorrecte. Par exemple, `let f x = x + true` n'est pas bien typée, car `+` est un opérateur binaire qui attend des expressions de type `int` de chaque côté.

Pour savoir si une expression est bien typée et donner son type, nous regardons sa syntaxe :

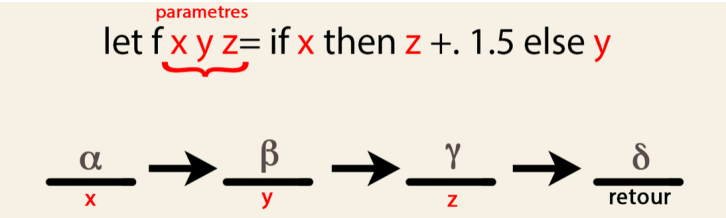
- les expressions simples sont bien typées et leur type est connu, par exemple :
  - `1` est bien typée de type `int` ;
  - "`une phrase`" est bien typée de type `string` ;
  - `true` est bien typée de type `bool` ;
  - '`a`' est bien typée de type `char`.
- Pour une définition de fonction, il faut regarder les paramètres, les expressions à l'intérieur de la fonction, et le type de retour, par exemple :
  - `let f x = x + 1` a le type `int -> int`. La fonction prend un `int` en paramètre et renvoie un `int` ;
  - `let g a = "hello"^a` a le type `string -> string`. L'opérateur `^` concatène des chaînes, donc vous pouvez en déduire que `a` est du type `string` et que la fonction `g` renvoie une chaîne de caractères, donc `string`.

De manière générale, pour typer une fonction, il faut tout d'abord identifier les paramètres et construire un type à trous avec un nombre de trous égal au nombre de paramètres plus un, ces trous étant séparés par des flèches. Le $i^{ème}$ trou correspond au type du $i^{ème}$ paramètre sauf le dernier trou qui correspond au type de retour. Nous déterminons le type des paramètres et du retour à partir du corps de la fonction.

Exemple :

Nous avons une fonction : `let f x y z = if x then z +. 1.5 else y`.

- Nous commençons par donner des noms temporaires aux trous (ici : `α, β, γ, δ`) :



Quand vous avez identifié le nombre de paramètres + retour, vous pouvez remplir les trous :

- **`x`** : `x` est la condition, qui doit renvoyer, soit `true`, soit `false`, donc `x` est du type `bool`. Vous pouvez donc remplacer α par `bool`.
- **`z +. 1.5`** : `z` est du type `float`, car on ajoute `1.5`. Vous pouvez remplacer γ par `float`.
- **`y`** : pour déduire le type de `y`, il faut savoir qu'en OCaml, la structure `if-then-else` demande que le type du `then` soit le même que le type du `else`. Nous savons que `z` est de type `float`, le type du `then` est donc `float`. Donc le type du `else` doit être `float` également.
- La fonction renvoie soit le `then` soit le `else`, dans les deux cas c'est le type `float`.

Donc le type de cette fonction est : `bool -> float -> float -> float`.

Pour chaque application d'une fonction ou d'un opérateur d'une expression les paramètres doivent avoir le type attendu par la fonction ou l'opérateur, le type de l'expression étant alors donné par le type de retour de la fonction ou de l'opérateur.

- `1-'d'` est mal typée car l'opérateur `-` attend `int` pour chacun de ses paramètres alors que `'d'` est un `char` ;
- `1+41` est bien typée de type `int` ;
- `2.+.3.` est bien typée de type `float` ;
- `f true 1.2 3.5` est bien typée de type `float` (type de retour de `f` voir cette fonction dans l'exemple au-dessus). En effet, son premier argument est bien de type `bool` et ses deux arguments suivants sont bien de type `float` ;
- `f true "hello" 3.5` est mal typée car son deuxième argument a le type `string` au lieu de `float` ;
- `1+(2.+.3.)` est mal typée car l'opérateur `+` attendait un `int` comme deuxième argument mais a reçu un `float`.

Exercice 2.1

Faire l'exercice du cours.

Attention, cet exercice doit être réalisé sur papier et sans utiliser l'interpréteur d'OCaml. En effet, il est très important d'abord de comprendre ce que doit faire l'interpréteur avant de voir le résultat.

À la fin de l'exercice, vous pouvez comparer vos résultats avec les résultats de l'interpréteur d'OCaml.

Exercice 2.2

Pour chacun des items suivants, donner une expression du même type. Par exemple, pour le type « `int list` » nous pouvons donner l'expression « `[1;2;3]` ».

1. `(char*char) list`

2. `(int->float) -> float -> float`

3. `(float -> string) list`

4. `'a -> 'b -> 'a`

5. `'a -> 'a -> bool`

Exercice 2.3

Déterminer les types des expressions fonctionnelles suivantes sans, bien sûr, les évaluer sur une machine. Expliquer comment vous avez déduit les types. Comparer vos réponses avec les réponses de l'interpréteur d'OCaml.

1. `let h(f,g) = function x -> f(g x);;`

2. `let i(a,b,c) = function x -> (a b (c * x));;`

3. `let j a b c = if (a>1 || c) then (b c a) else (a,c);;`