Premiers pas en OCaml

Fichiers d'extension `.ml`.
OCaml est un langage de programmation à la fois adapté pour la programmation fonctionnelle, impérative, orientée objet.

Ce langage possède son propre gestionnaire de paquets, opam depuis lequel il est possible d'installer outils et librairies.

Ce langage peut être **compilé** et **interprété**, on le compile avec `ocamlc`

```
ocamlc <file> -o <output>
```

Et on peut l'interpréter dans l'environnement `utop`, en utilisant la directive `#use`.

OCaml est un langage **statiquement typé**, et muni d'un moteur d'inférence de type capable de déterminer le type d'une fonction, valeur, expression si elle n'est pas précisée[1].

⚠ On vérifiera cependant à préciser le typage de toutes nos fonctions.

Enfin, un fichier `.ml` est constitué d'une succession de définitions.

Syntaxe des définitions en OCaml

Définition de valeurs

En général, on définit une valeur par son identifiant

```
let id_var :type = expression
```

Avec `expression` de type `type`.

Définition de fonctions

```
let id_f (arg1 :t1) (arg2 :t2) (argn :tn) :t =
        exp1
```

où :

- `id_f` est l'identifiant
- `arg1` ... `argn` sont les arguments
- `t1` ... `tn` les types des arguments

- `exp1` une expression de type `t` pouvant faire apparaitre `arg1` ... `argn`

Fonction récursive

```
let rec nom_f (arg1 :t1) (arg2 :t2) (argn :tn) :t =
        e
```

où :

- `nom_f` est l'identifiant
- `arg1` ... `argn` sont les arguments
- `t1` ... `tn` les types des arguments
- `e` une expression de type `t` pouvant faire apparaitre `arg1` ... `argn` **ainsi que** `nom_f`

Fonctions mutuellement récursives

```
let rec f1 (arg1 :t1) (arg2 :t2) (argn :tn) :t =
        e1
and f2 (barg1 :ta) (barg2 :tb) (bargm :tm) :t_prime =
        e2
```

où :

- `f1` et `f2` sont les identifiants respectifs des deux fonctions
- `arg1` ... `argn` sont les arguments de `f1`
- `barg1` ... `bargm` sont les arguments de `f2`
- `t1` ... `tn` les types des arguments de `f1`
- `ta` ... `tm` les types des arguments de `f2`
- `e1` une expression de type `t` pouvant faire apparaitre `arg1` ... `argn` **ainsi que** `f1` et `f2`
- `e2` une expression de type `t_prime` pouvant faire apparaitre `barg1` ... `bargm` **ainsi que** `f1` et `f2`

Expressions en OCaml

> Définition
>
> À un instant donné on appelle "environnement courant" l'ensemble des valeurs, fonctions et types définis.

Pour un environnement donné, une expression est un texte formaté selon la syntaxe OCaml auquel on peut donner un type et une valeur.

⚠ Deux expressions ne sont pas identiques parce qu'elles sont du même type et ont la même valeur

Les constantes

Type	Exemple	Remarque
`unit`	`()`	`()` est l'unique valeur
`bool`	`true`, `false`	Uniques valeurs
`int`	`1`, `2`, `-4`, `0`	
`float`	`3.` `4.` `0.`	La notation `.5` n'existe pas.
`char`	`'m'` `'\n'`	
`string`	`"mp1"`	

Les valeurs

Communément appelées à tort "variables" les valeurs définies dans un environnement courant sont des expressions : après avoir executé

```
let a = 2
```

`a` est une expression

Les arguments

Sont évalués dans les corps de fonction comme des valeurs

Les opérations

type 1	type 2		type sortie
`int`	`int`	`*` `+` `/` [2] `mod` `-`	`int`
`'a`	`'a`	`<>` `:` `<` `<=` `>` `>=`	bool
`float`	`float`	`*.` `+.` `/.` `**` `-.`	`float`
`bool`	`bool`	|| `&&`	`bool`
`bool`	---	`not`	`bool`
`string`	`string`	`^`	`string`

Les appels de fonction

```
nomf arg1 arg2 argn
```

Où:

- `nomf` est le nom d'une fonction à n arguments de types `t1`, `t2`... `tn` et de type de sortie `t`
- Pour $i \in [\![1, n]\!]$ `argi` est une expression de type `ti`
  Forme une expression de type `t`.

Les alternatives

```
if cond then e1 else e2
```

Où:

- `cond` est une expression de type `bool`
- `e1` et `e2` sont deux expressions de même type t
  Forme une expression de type `t` et de valeur la valeur de `e1` si la valeur de `cond` est `true`, et la valeur de `e2` sinon.

Les n-uplets

```
e1, e2, en
```

Où, pour $i \in [\![1, n]\!]$, `ei` est une expression de type `ti`.
Cela forme une expression de type `t1 * t2 * ... * tn`

Les définitions locales

```
let id = e in e1
```

Où:

- `e` est une expression de type `t`
- `e1` est une expression de type `t1` qui peut utiliser `id`
  Forme une expression de type `t1` et de valeur la valeur obtenue en évaluant `e1` où `id` a pour valeur celle de l'expression `e`.

Définir des types

Syntaxe de la définition

```
type id = expression_de_type
```

Où:

- `id` est un identifiant qui commence par une minuscule

- `expression_de_type`

Expressions de type déjà vues

Expressions simples

- On retrouve ici les types de base : `unit` `bool` `float` `int` `char` `string`
- Mais aussi les noms des types que l'on a défini au préalable:

```
type truc = int
```

Expressions de type composées

Pour les *n-uplets*

```
t1 * t2 * tn
```

Pour une fonction à n arguments de type respectifs `t1`, `t2` et de type de sortie `tn`

```
t1 -> t2 -> tn
```

Par exemple

```
type 'a triplet = 'a * 'a * 'a
```

Permet de factoriser tous les types de triplets

```
let triplet_entiers = int triplet;;
```

Types paramétrés

> Définition
>
> On peut définir un type dont l'expression dépend d'une variable de type grâce à la syntaxe suivante :

```
type 'var id = e
```

Où:

- `var` est un identifiant local
- `id` est un identifiant

- `e` est une expression qui peut faire apparaitre le type `'var`

On pourra aussi définir un type dont l'expression dépend de plusieurs paramètres :

```
type ('v1, 'v2, 'vn) id = e
```

Dans les deux cas, on parle de types paramétrés, et après de telles définitions, on obtient de nouvelles expressions de type en remplaçant les `'v` par :

- `'un_autre_id`
- une expressions de type

```
type ('a, 'b) triplet_special = 'a * 'b * 'a
type triplets_entiers = (int, int) triplet_special
```

Types avec valeurs

```
type nom_type =
| Cons_1 [of t1]
| Cons_2 [of t2]
| Cons_3 [of t3]
| Cons_n [of tn]
```

Où :

- `nom_type` est un identifiant
- `Cons_i` pour `i` $\in [\![1, n]\!]$
- `ti` sont des expressions de type (ce qui est entre crochets est optionnel)

Types récursifs

```
type 'a pile = | PV | PNV of 'a * pile
```

Types somme et filtrage

> Cartes de Jeu
>
> On comprend d'ores et déjà que pour modéliser des cartes, il nous faudra définir des types correspondants.
>
> ```
> type valeur =
>  | Brele of int
>  | Tete of string
> ```

```
type couleur =
 | Pique
 | Coeur
 | Trefle
 | Carreau
```

Et bien évidemment pour définir une carte, il nous faudra donner un couple de deux valeurs

```
type carte = couleur * valeur
let dame_de_coeur :carte = (Coeur, Tete("Dame"))
let trois_de_pique :carte = (Pique, Brele(3))
```

On commence par écrire une fonction qui vérifie si une carte est rouge

```
let est_rouge (c :carte) : bool =
        (* Teste si c est rouge *)
        let (clr, _) = c in
        clr = Coeur || clr = Carreau
```

On peut évidemment tester cette fonction avec les deux cartes que l'on a défini.

```
let test_est_rouge : unit =
    assert(est_rouge dame_de_coeur);
    assert(not(est_rouge trois_de_pique))
```

Mais on peut définir une autre fonction en utilisant le pattern matching

```
let est_rouge_filtrage (c :carte) : bool =
        (* Teste si c est rouge *)
        let (clr, _) = c in
        match clr with
        | Carreau -> true
        | Coeur -> true
        | _ -> false
```

Qui est une syntaxe équivalente à

```
let est_rouge_filtrage_equivalent (c :carte) : bool =
        (* Teste si c est rouge *)
        let (clr, _) = c in
        match clr with
        | Carreau | Coeur -> true
        | _ -> false
```

Pour se donner une autre idée de comment marche le pattern matching, regardons comment se comporte la syntaxe pour les valeurs définies avec des constructeurs :

```
let est_un_trois (c :carte) : bool =
        (* Teste si c est un trois *)
        let (_, vlr) = c in
        match vlr with
        | Brele (k) -> k = 3
        | Tete (_) -> false
```

En outre, il ne faut pas oublier que le k tel que défini dans le filtrage est une variable muette qui écrase tous les homonymes dans l'expression
ainsi, écrire

```
let nb_points (c :carte) (joker :string) : int =
        (* Teste si c est un trois *)
        let (_, vlr) = c in
        match vlr with
        | Brele (k) -> k
        | Tete "as" -> 14
        | Tete "roi" -> 13
        | Tete "dame" -> 12
        | Tete "valet" -> 11
        | Tete (joker) -> 1000
        | Tete _ -> false
```

ne veut pas dire que le fitre teste la valeur dans le constructeur `Tete` et la compare avec la valeur de `joker`.

Ainsi, pour réalise ce type de filtrage dynamique on peut utiliser les *Motifs Gardés* avec le keyword `when`.

```
let nb_points (c :carte) (joker :string) : int =
        (* Teste si c est un trois *)
        let (_, vlr) = c in
        match vlr with
        | Brele (k) -> k
        | Tete "as" -> 14
        | Tete "roi" -> 13
        | Tete "dame" -> 12
        | Tete "valet" -> 11
        | Tete t when t = joker -> 1000. (* Cela marchera *)
        | Tete _ -> false
```

Attention, la vérification de l'exhaustivité d'un filtrage dans lequel un motif est gardé suppose que l'expression conditionnelle attachée à celui-ci est fausse ; en conséquence

de quoi, ce motif ne sera pas pris en compte. Par exemple, Caml ne peut détecter que le filtrage suivant est exhaustif :

```
let f = function
| x when x >= 0 -> x + 1
| x when x < 0 -> x - 1
```

La syntaxe d'un filtrage par motif est la suivante :

```
match expr0 with
| Cons_1 (var1) -> e1
| Cons_2 (var2) -> e2
| Cons_n (varn) -> en
```

Où :

- `expr0` est une expression de type `nom_type`, un type somme
- `vari` pour `i` $\in [\![1, n]\!]$ est un nom de variable qui est de type `ti`
- `ei` est une expression qui peut utiliser `vari`

⚠ Priorisation

L'ordre dans lequel on essaye de faire correspondre un motif et une valeur a de l'importance :

```
let sinc = function | x -> sin(x) /. x | 0. -> 1.
```

```
Toplevel input :
> | 0. -> 1.
> ^^
Warning : this matching case is unused. sinc : float -> float = <fun>
```

---

1. Le moteur d'inférence de type sera *de toute façon* utilisé pour vérifier la cohérence du type annoncé avec le type de l'expression↩
2. C'est la division euclidienne↩