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

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`.

---

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↩