TP 3 : Introduction à OCaml, Récursivité

3 octobre 2023

1 OCaml

OCaml est une implémentation du langage de programmation CAML. CAML est un langage de programmation conçu et maintenu essentiellement par des chercheurs de l'Inria, développé à partir des années 1980.

OCaml est un langage **fonctionnel**, bien qu'il permette l'utilisation de certains traits de programmation impérative. La programmation fonctionnelle est un paradigme de programmation dans lequel les calculs sont considérés comme des évaluations de fonctions. Cela influe en particulier sur la pratique de programmation, par exemple en privilégiant - ou en contraignant à - l'utilisation de la récursivité plutôt que des méthodes itératives.

OCaml est un langage qui peut aussi bien être compilé (comme C) qu'interprété (comme Python). On enregistre un programme en OCaml dans un fichier .ml. Lorsque l'on a un `fichier .ml`, on peut, en écrivant les lignes suivantes dans un terminal :

- Le compiler vers du bytecode avec :
  `ocamlc fichier.ml -o nomdefichier` (`nomdefichier.exe` sous Windows),
  puis l'exécuter avec `ocamlrun nomdefichier` (.exe)
- Le compiler vers du code natif avec :
  `ocamlopt fichier.ml -o nomdefichier` (`.exe`),
  puis l'exécuter avec `./nomdefichier` (`.exe`)
- Interpréter le contenu du fichier avec `ocaml fichier.ml`

On peut également simplement lancer un toplevel (ce qui s'apparente à une interface en lignes de commandes) avec la commande `ocaml` seule dans le Terminal ; on peut alors écrire les lignes de programme ocaml qui seront directement interprétées dans le Terminal.

Par ailleurs, comme pour C, il existe des IDE en ligne : `https://try.ocamlpro.com/`

2 Premiers Programmes, Déclaration de Variables

Un programme en OCaml est constitué d'une suite de *déclarations*, qui donnent un `nom` de variable à la valeur d'une `expression`. Celles-ci se terminent par un double point-virgule `;;`.

La syntaxe pour donner une valeur `valeur` à une variable `variable` est la suivante :

```
let variable = valeur;;
```

Dans un interpréteur, lorsqu'une déclaration est faite, une ligne affiche la valeur, le type de la valeur et le nom de la variable qui prend cette valeur.

Exercice 1 :

Taper les expressions suivantes :

- `let x = 5;;`
- `let y = x+10;;`
- `x;;`
- `let x = x*x;;`
- `1;;`
- `1.5;;`
- `true;;`
- `'a';;`
- `"bonjour";;`
- `();;`
- `Printf.printf "Hello World!";;`

On observe plusieurs choses, et notamment des différences avec le langage C :

- Il existe un type `bool` prédéfini, pouvant prendre comme valeur `true` et `false`.
- Comme en C, pour calculer le valeur que l'on va attribuer à une variable, on peut faire des opérations sur des valeurs et sur des variables (y compris la variable à laquelle on affecte une nouvelle valeur).

2.1 Portée Lexicale

En C, lorsque l'on donne successivement plusieurs valeurs différentes à une même variable, on *modifiait l'état du programme*, c'est-à-dire le *contenu de la mémoire centrale*. La déclaration d'une variable consiste à associer à un nom de variable un espace en mémoire, et chaque nouvelle affectation de valeur correspondra à une modification de la valeur stockée dans cet espace, qui remplace l'ancienne valeur.

En OCaml, une déclaration définit *toujours* une nouvelle variable : plusieurs `let x = ...` successifs n'entraînent pas la modification ou la disparition des déclarations précédentes : elles sont simplement *masquées* par la dernière déclaration, mais existent toujours en mémoire. Elles ne sont cependant plus accessibles par la mention de x dans une expression.

C'est la *portée lexicale* de x : la portée d'une définition va jusqu'à la déclaration suivante définissant une variable avec le même nom. C'est la valeur de la dernière déclaration de x qui est prise en compte lorsque l'on fait référence à une variable x dans la déclaration d'une nouvelle variable y. Par ailleurs, une nouvelle définition de x ne change pas la valeur de y en fonction de celle-ci ; dans l'exemple précédent, y vaut toujours $5+10 = 15$ après la déclaration `let x = x*x;;`.

2.2 Inférence de Type

On constate, lorsque l'on déclare une variable, qu'OCaml détecte son type sans que celui-ci n'ait eu besoin d'être précisé. En OCaml, toutes les variables doivent avoir un unique type lorsqu'elles sont déclarées : OCaml est un langage *fortement typé*.

La construction de l'expression doit permettre de retrouver le type de façon automatique, et un algorithme permet de le faire en OCaml. Cette procédure s'appelle l'*inférence de type*, et à notamment comme intérêt de permettre de s'assurer qu'un programme est écrit correctement : si l'algorithme d'inférence de type ne parvient pas à trouver le type, alors le type d'une expression, alors celle-ci doit contenir des erreurs.

3 Types de Base et Opérations

On a vu les types de différents éléments dans les lignes de commandes précédentes. Les types de base en OCaml sont les suivants :

- `int` pour les entiers naturels ;
- `float` pour les nombres à virgule flottante ;
- `char` pour les caractères ;
- `string` pour les chaînes de caractères ;
- `bool` pour les booléens, qui valent `true` ou `false` (différent de C, qui utilisait les entiers 0 et 1) ;
- `unit` qui est un type vide (équivalent de `void` en C), qui servira en particulier pour les aspects impératifs du langage.

Exercice 2 :

Taper les expressions suivantes :

- `18 + 7;;`
- `18/7;;`
- `18 mod 7;;`
- `18 * 7;;`
- `18 ** 7;;`
- `18 - 7;;`
- `18.5 - 7;;`
- `18.5 + 7.3;;`
- `18. +. 7.3;;`
- `18. **. 7.;;`

On a testé ici différentes opérations binaires sur des entiers et des flottants. On peut remarquer que :

- Les opérations se font soit entre deux entiers, soit entre deux flottants, mais les types ne se mélangent pas, en raison du typage fort. Le flottant correspondant à un nombre entier s'écrit avec un point à la fin.

— Les opérations sur les flottants se notent avec un point à la fin.

— Il y a une opération de puissance pour les flottants et pas pour les entiers.

On a les opérations suivantes sur les différents types :

— Les opérations sur les entiers : `+`, `-`, `*`, `/`, `mod`

— Les opérations sur les flottants : `+.`, `-.`, `*.`, `/.`, `**`

— Les opérations de conjonction, disjonction, négation sur les booléens : `&&`, `||`, `not`

— L'opération de concaténation sur les chaînes de caractères : `^`

— Les opérateurs de comparaison, soit entre entiers, soit entre flottants, qui renvoient un booléen : `=`, `!=`, `>`, `<`, `>=`, `<=`

4 Commentaires

En OCaml, les commentaires sont notés entre `(* *)` :

```
(* texte en commentaire, ignoré par l'interpréteur *)
```

5 Fonctions

En OCaml, la syntaxe pour désigner une fonction qui a un paramètre `parametre` associe une expression `expression` faisant intervenir le paramètre est la suivante :

```
fun parametre -> expression ;;
```

Pour **appliquer** une fonction `fonction` sur un argument `argument`, on utilise la syntaxe suivante :

```
fonction(argument);;
```

Il existe des fonctions prédéfinies en OCaml. Par exemple, la fonction `int_of_float` associe à un nombre à virgule flottante sa partie entière. Réciproquement, `float_of_int` prend en entrée un entier et lui associe le nombre à virgule flottante ayant la même valeur.

Exercice 3 :

Taper les expressions suivantes :

— `int_of_float(5.3);;`

— `int_of_float;;`

— `fun x -> x+1;;`

— `fun x -> x+.1.;;`

En OCaml, les fonctions sont des valeurs comme les autres, et dans l'interpréteur, lorsqu'on les appelle sans argument, on obtient un affichage similaire à ce qui était obtenu lorsqu'on faisait la même chose avec une variable.

La ligne `int_of_float(5.3);;` fait afficher `- : int = 5` (comme si on avait rentré la ligne `5;;`). En revanche, la ligne `int_of_float;;` affiche `- : float -> int = <fun>`. On remarque deux choses :

— Là où l'on avait la valeur exacte `5` dans le premier cas, on a seulement l'indication qu'il s'agit d'une fonction avec `<fun>` dans le second cas (il est plus difficile d'indiquer ce qu'est exactement la fonction si sa définition est longue).

— Le type de `int_of_float` est noté `float -> int` : il s'agit du type de l'argument en entrée, suivi du type de la valeur renvoyée en sortie. Autrement dit, le type d'une fonction en OCaml correspond à sa *signature*.

Une expression du type `fun x -> y` est une **fonction anonyme** : il s'agit d'une valeur à laquelle on n'a pas encore associé un nom de variable.

Pour déclarer une fonction en lui donnant un nom, on utilise la même syntaxe que celle vue pour les déclarations de variables :

```
let f = fun x -> x+1;;
```

Une autre écriture équivalente à celle ci-dessus, plus concise et plus proche de celles existantes dans d'autres langage de programmation, existe également :

```
let f x = x + 1;;
```

Il n'est pas nécessaire de mettre des parenthèses autour du paramètre de la fonction, mais on peut le faire :

```
let f(x) = x + 1;;
```

Si l'on souhaite indiquer le type des paramètres (bien que ce soit a priori inutile avec l'inférence de type, cela permet une clarté du code et une détection d'erreurs plus précise), on doit utiliser la syntaxe suivante :

```
let f (x:int) = x + 1;;
```

De la même façon, l'application d'une fonction à une valeur ne nécessite pas de parenthèses, bien qu'il soit possible d'en mettre : les écritures `f(1)` et `f 1` sont équivalentes.

On mettra cependant des parenthèses lorsque l'on voudra mettre une expression en argument. En effet, l'application de fonction est prioritaire sur les opérateurs prédéfinis, et une expression sans parenthèse comme `f 1 * 2` correspondra à la valeur $f(1) \times 2$, et non à $f(1 \times 2)$.

Exercice 4 :

1. Ecrire une fonction `carre` qui renvoie le carré d'un entier.
2. Ecrire une fonction `carre_float` qui renvoie le carré d'un flottant.
3. Ecrire une fonction `cube` qui renvoie le cube d'un entier.
4. Ecrire une fonction `cube_float` qui renvoie le cube d'un flottant.

Il est également possible de prendre plusieurs arguments pour une fonction.

Exercice 5 :

Taper les déclarations suivantes :

1. `let add x = fun y -> x + y ;;`
2. `let add x y = x + y ;;`
3. `add 5;;`
4. `let add_bis (x,y) = x + y ;;`

On observe qu'on a plusieurs cas différents sur cet exercice :

— Dans les deux premiers cas, `add` est une fonction qui prend en entrée un entier `x`, et renvoie une fonction qui prend en entrée un entier `y` et renvoie `x+y` : le type est `int -> (int -> int)`. `add` appliqué à un argument renvoie une fonction de type `int -> int`.

— Dans le dernier cas, la fonction `add_bis` prend en entrée un tuple avec deux valeurs `x` et `y`, et renvoie leur somme : c'est une fonction de type `(int * int) -> int`.

6 Définition Locale de Valeur

On a vu en C que l'on pouvait déclarer des variables localement, à l'intérieur du corps des fonctions. Cela permettait notamment de réutiliser plusieurs fois une même valeur sans avoir à la recalculer à chaque fois.

On peut également faire des déclarations locales en OCaml. Si l'on souhaite utiliser une expression `expression1` dans laquelle se trouve une valeur `valeur` définie localement par `expression2`, on utilisera la syntaxe suivante :

```
let valeur = expression2 in expression1;;
```

Et si l'on souhaite déclarer une variable `x` en lui affectant cette valeur, on utilise la syntaxe suivante :

```
let x = let valeur = expression2 in expression1;;
```

Par exemple, les deux expressions suivantes sont équivalentes :

```
let x = 2+3;;
```

```
let x = let y = 2 in y+3;;
```

On peut également écrire la déclaration sur plusieurs lignes, pour plus de clarté :

```
let x =
    let y = 2 in
    y+3;;
```

On voit ici le cas où la syntaxe est employée pour déclarer localement une variable dans la déclaration d'une autre variable. Il est également possible :

— de faire des déclarations locales dans la déclaration d'une fonction ;

— de déclarer localement une fonction dans une déclaration.

Exercice 6 :

1. Définir une fonction `puissance_4` qui prend en entrée un entier x et renvoie la valeur de x^4, en définissant localement à l'intérieur une fonction `carre` qui calcule le carré d'un entier, puis en renvoyant $(x^2)^2$.
2. Définir une fonction `puissance_9` qui prend en entrée un entier x et renvoie la valeur de x^9, en définissant localement à l'intérieur une fonction `cube` qui calcule le cube d'un entier, puis en renvoyant $(x^3)^3$.

7 Expressions Conditionnelles

L'utilisation de conditions existe en OCaml. En C, elles permettent d'exécuter ou non des blocs d'instructions selon qu'une condition soit vérifiée ou non. EN OCaml, elles permettent de définir une fonction/variable par une expression ou une autre selon qu'une condition soit vérifiée ou non.

Supposons que l'on souhaite donner à la variable `x` la valeur correspondant à l'expression `expression1` si la condition `condition1` est vérifiée, celle correspondant à l'expression `expression2` si la condition `condition2` est vérifiée, etc. Et celle correspondant à `expressionN` si aucune condition n'est vérifiée (les conditions sont des booléens).

La syntaxe pour cette déclaration est la suivante :

```
let x =
    if condition1 then expression1
    else if condition2 then expression2
    ...
    else expressionN;;
```

On pourra employer la même syntaxe pour une fonction `f` d'argument `x` :

```
let f x =
    if condition1 then expression1
    else if condition2 then expression2
    ...
    else expressionN;;
```

Exercice 7 :

Définir une fonction qui prend en entrée un entier naturel non-nul n et renvoie le successeur de n dans la suite de Syracuse.

8 Récursivité

Essayons de définir la fonction factorielle d'une façon similaire à C :

```
let fact n =
    if n = 0 then
        1
    else
        n * fact (n-1)
;;
```

Ce code ne marche pas, car `fact` n'est pas considéré comme une fonction récursive si ce n'est pas indiqué dans sa définition. Pour définir une fonction `f` récursive, on utilisera la syntaxe `let rec f x = ... ;;`

La version correcte du programme précédent est donc la suivante :

```
let rec fact n =
    if n = 0 then
        1
    else
```

```
        n * fact (n-1)
;;
```

Exercice 8

Définir une fonction qui prend en entrée un couple d'entiers naturels n et k, avec n non-nul, et renvoie le k-ième successeur de n dans la suite de Syracuse (renvoie n si $k = 0$).

Exercice 9

Définir une fonction qui prend en entrée deux entiers naturels a et n et renvoie a^n, calculé à l'aide de l'exponentiation naïve.

Exercice 10

Définir une fonction qui prend en entrée deux entiers naturels a et n et renvoie a^n, calculé à l'aide de l'exponentiation rapide.