TP 3 : Introduction à OCaml, Récursivité

17 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.

2.3 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 : `=`, `!=`, `>`, `<`, `>=`, `<=`

2.4 Commentaires

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

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

3 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;;
```

4 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;;
```

Exercice 3 :

Définir un entier naturel non-nul x, et définir y dont la valeur est le successeur de x dans la suite de Syracuse.

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 4 :

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 5 :

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 couple avec deux valeurs `x` et `y`, et renvoie leur somme : c'est une fonction de type `(int * int) -> int`.

5.1 Déclaration Locale avec les Fonctions

On a vu 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 $(x^3)^3$.

5.2 Expressions Conditionnelles dans les Fonctions

On a vu la syntaxe pour définir une variable par une expression ou une autre selon qu'une condition soit vérifiée ou non. 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.

Exercice 8 :

1. Définir une fonction qui prend en entrée deux entiers et renvoie le maximum entre les deux.
2. Définir une fonction qui prend en entrée deux entiers et renvoie le minimum entre les deux.

Exercice 9 :

Ecrire une fonction qui prend en entrée un flottant et renvoie sa valeur absolue.

5.3 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.

5.4 Filtrage par Motifs

Exercice 11

Définir une fonction qui prend en entrée trois entiers a, b et c et renvoie le nombre de racines de $ax^2 + bx + c$.

Au lieu d'utiliser des conditions, il est également possible d'utiliser le **filtrage par motifs** (pattern matching en anglais). Le filtrage par motif permet de définir la valeur calculée par la fonction selon les différentes formes que peut prendre la valeur d'une expression.

La syntaxe du filtrage par motifs est la suivante :

```
let fonction expression =
    match expressionbis with
    | cas1 -> valeur1
    | cas2 -> valeur2
```

```
  | ...
  | casN -> valeurN
```

Dans ce cas, la valeur calculée par la fonction sera `valeur1` si `expressionbis` vaut `cas1`, `valeur2` si `expressionbis` vaut `cas2`, etc.

Il est possible de faire un filtrage pour l'ensemble des cas où `expressionbis` prend une valeur `x` qui vérifie une condition `condition sur x` avec la syntaxe suivante :

```
  | x when (condition sur x) -> valeurcondition
```

De la même façon que `else` seul permet de couvrir tous les cas restants dans une instruction conditionnelle avec plusieurs conditions possibles, l'option _ permet de couvrir l'ensemble des cas possibles différents de ceux couverts par les options précédentes.

Exemple : une écriture possible d'une fonction `pgcd` récursive avec le filtrage par motifs en OCaml est la suivante :

```
let rec pgcd a b =
  match b with
  | 0 -> a
  | _ -> pgcd b (a mod b)
;;
```

Exercice 12

Enlever la seconde ligne dans la fonction précédente et observer l'affichage après évaluation.

Si le filtrage ne couvre pas l'ensemble des cas possibles, OCaml mettra un message d'avertissement (Warning) pour **filtrage non-exhaustif**.

Exercice 13

Ecrire une fonction qui prend en entrée deux entiers x et y et renvoie :

1. $x + y$ si $x - y$ est pair
2. $x - y$ sinon.

6 Traits Impératifs

On rappelle que les instructions correspondent aux modifications des éléments d'un ordinateur, comme la mémoire ; contrairement à une expression, une exécution ne renvoie pas de valeur. Le type `unit` permet de de définir des commandes correspondant à des instructions de programmation impérative, comme les instructions d'affichage.

6.1 Affichage et Saisie

De la même façon que l'on peut charger des bibliothèques en C, on peut utiliser des modules en OCaml qui contiennent différentes fonctions pour des tâches données. Lorsque l'on appelle une fonction d'un module, on utilise la syntaxe `Module.fonction` (avec une majuscule au nom du mdule).

L'affichage se fait avec la fonction `printf` dans le module `Printf` :

```
Printf.printf "Hello World!"
```

Comme en C, on peut afficher les valeurs de variables à l'aide de spécificateurs. Ceux-ci sont similaires à ceux utilisés en C :

- `%d` pour les entiers
- `%f` pour les flottants
- `%c` pour les caractères
- `%s` pour les chaînes de caractères

Exemple :

```
let sequence = "World";;
Printf.printf "Hello %s!" sequence;;
```

Exercice 14

Lancer les lignes de codes suivantes, comparer les affichages obtenus :

```
Printf.printf "5 \n" ;;
"5 \n" ;;
let x = "5 \n" ;;
```

Il existe également des fonctions intégrées pour l'affichage de valeurs d'un certain type, comme les fonctions `print_int`, `print_float` et `print_string`.

Exemple :

```
print_int 5 ;;
print_float 5.3 ;;
print_string("cinq") ;;
```

Il est également possible, comme en C, de lire de demander à l'utilisateur de rentrer les valeurs de variables données. On peut utiliser les fonctions `read_int`, `read_float` et `read_string`, avec la syntaxe suivante :

```
let x = read_int() ;;
```

6.2 Séquences

il est possiblement d'évaluer successivement plusieurs instructions en les séparant par un point-virgule, avec une écriture sous la forme `instruction_1 ; instruction_2`.

Exemple : `print_string("cinq") ; print_int(5) ;;`

Les instructions successives peuvent notamment permettre d'afficher des séquences de texte dont le contenu dépend de variable (y compris des variables numériques), par exemple :

```
let affichage_somme x y = print_int(x) ;
  print_string(" + " ) ; print_int(y) ;
  print_string(" = " ) ; print_int(x+y ) ;;
```

6.3 Instructions Conditionnelles

Les conditions, que l'on a vu dans le cadre des déclarations, peuvent aussi être utilisées pour exécuter ou non des blocs d'instructions selon qu'une condition soit vérifiée ou non (on

remarque que l'on peut dans ce cas se dispenser de else). On peut utiliser la syntaxe suivante :

`if condition then instruction_1`

Remarque : dans l'instruction `if condition then instruction_1 ; instruction_2`, la deuxième instruction est à l'extérieur de la condition. Pour pouvoir exécuter une séquence d'instructions selon une condition, on peut :

— La mettre entre parenthèses : `if condition then (instruction_1 ; instruction_2)`

— L'écrire sous la forme d'un bloc d'instruction, balisé par `begin` et `end` :

```
if condition then
    begin
        instruction_1 ;
        instruction_2
    end
```

6.4 Instructions Itératives

Pour les instructions, on a, comme en C, deux types de boucles, les boucles `while` et les boucles `for`, qui permettent de répéter un bloc d'instructions plusieurs fois :

Les boucles `while`, comme en C, permettent de répéter un bloc d'instruction plusieurs fois, jusqu'à ce qu'une condition cesse d'être vérifiée. La syntaxe est la suivante :

```
while condition1 do
    instructions
done;;
```

Les boucles `for` permettent de répéter une instruction pour plusieurs valeurs d'un identificateur, entre une valeur initiale et une valeur finale. Exemple (avec la syntaxe) :

```
for k = 1 to 10 do
    print_int k
done;;
```

Exercice 15

Ecrire une fonction en OCaml qui prend en entrée un entier naturel non-nul n et affiche la phrase "d divise n" pour tous les diviseurs de n.

7 Types Construits

On a vu les types `char`, `string`, `bool`, `int` et `float`.

Il est possible de construire des types plus complexes à partir de ces types initiaux.

7.1 Paires, n-uplets

Soient deux expressions `expression1` et `expression2` de types donnés. On peut déclarer la paire contenant ces deux expressions :

`let p = (expression1,expression2)`

Le type de la paire sera le produit des types des expressions : par exemple, `('a',2)` est de type `char * int`.

On peut également construire des paires dans lesquelles l'un des éléments est une paire :

```
let p = (1,(3,5)) ;;
```

Pour retrouver les valeurs `p1` et `p2` d'un couple `(p1,p2)`, on peut utiliser les fonctions `fst` et `snd` :

```
let p1 = fst p ;;
let p2 = fst p ;;
```

Une autre façon de récupérer les valeurs est en déconstruisant la paire :

```
let (x,y) = p ;;
```

La déconstruction peut être utilisée pour récupérer une unique valeur, en laissant vides les champs dont on ne souhaite pas récupérer les valeurs avec le symbole _ :

```
let (_,(x,_)) = p ;;
```

Le type paire peut se généraliser à un nombre d'élément supérieur à 2 avec le type n-uplet. Pour définir un n-uplet, on met simplement entre parenthèses la succession d'éléments, séparés par des virgules.

```
let nuplet = (3,5,8) ;;
```

Attention : le n-uplet `(3,5,8)` est de type `type1 * type1 * type1`, contrairement à `(3,(5,8))` qui est de type `int * (int * int)`

7.2 Enregistrements

Un problèmes des n-uplets est que l'on peut vouloir stocker un grand ensemble d'informations dedans. Dans ce cas, il peut être utile de définir des types dans lesquels chaque coordonnée a un nom. En OCaml, il est possible de faire cela avec les **enregistrements** ou produits nommés.

La déclaration d'un type de produit nommé se fait de la façon suivante :

```
type typeproduit = { champ1 : type1 ; champ2 ; type2 }
```

Exercice 16

Définir un type complexe dont la partie imaginaire est donnée par le champ `im` et la partie réelle par le champ `re`.

Pour déclarer une variable d'un type construit donné de champs `champ1`, `champ2`..., on utilise la syntaxe suivante :

```
let v = { champ1 = valeur1 ; champ2 = valeur2 }
```

Pour accéder à la valeur d'un champ donné, on utilise la syntaxe `variable.champ`

Soit `z1` une variable d'un type produit défini. Si on veut construire `z2` du même type, égal à `z1` sur plusieurs de ses champs, on peut utiliser la syntaxe suivante pour créer une copie de `z1` en modifiant les valeurs de ses champs :

```
let z2 = z1 with champ = val
```

Exercice 17

Ecrire une fonction qui prend en entrée un nombre complexe et renvoie son conjugué.

Exercice 18

Ecrire une fonction qui prend en entrée deux nombres complexes et renvoie leurs produits.

7.3 Sommes disjointes

On peut souhaiter créer un type représentant un ensemble fini de valeurs spécifiques, différentes les unes des autres. Par exemple, on peut souhaiter définir un type `piece` qui correspondrait à une pièce du jeu d'échecs : Pion, Cavalier, Fou, Tour, Roi ou Dame.

On peut définir un type qui sera représenté par l'*énumération* des valeurs possibles à prendre. Comme c'est une union (ou somme) de valeurs disjointes, on appelle également ce type de définition une `somme disjointe`. La syntaxe pour définir un type avec une énumération est la suivante, par exemple pour les pièces du jeu d'échecs :

```
type piece = Pion | Cavalier | Fou | Tour | Roi | Dame
```

Les valeurs `Pion`, `Cavalier`, etc. qui servent à définir le type `piece` sont appelés ses **constructeurs**.

Attention

Il faut impérativement mettre une **Majuscule** au début de chaque constructeur. Plus précisément, en OCaml, on a la règle suivante :

- Les noms de variables, types et fonctions commencent toujours par des minuscules ;
- Les noms de modules et de constructeurs commencent toujours par des majuscules.

Les types définis par énumération sont particulièrement adaptés aux dissociations de cas pour lesquels on utilise le filtrage par motif. Pour une variable `var` d'un type défini par énumération des constructeurs `Constructeur1`, `Constructeur2`, etc. on peut utiliser la syntaxe suivante :

```
match var with
| Constructeur1 -> expression1
| Constructeur2 -> expression2
[...]
```

Lorsque l'on associe une même expression à deux valeurs de constructeurs possibles, on peut utiliser la syntaxe suivante pour exprimer la disjonction de cas :

```
match var with
| Constructeur1 | Constructeur2 -> expression1
| Constructeur3 -> expression2
[...]
```

Exercice 19

1) Définir un type `chifoumi` qui comprend les constructeurs Pierre, Feuille et Ciseau et un type `resultat` qui comprend les constructeurs Defaite, Nul et Victoire.

2) Ecrire une fonction qui prend en entrée un couple de valeur de type `chifoumi * chifoumi`, correspondant aux coups joué par un joueur 1 et un joueur 2, et renvoie un couple de type `resultat * resultat` correspondant aux résultats respectifs du joueur 1 et du joueur 2.

7.3.1 Sommes disjointes avec Arguments

On peut également définir des constructeurs qui prennent un argument : par exemple, on peut vouloir définir un type `valeur` pour une carte, qui puisse valoir Roi, Dame, Valet, ou un nombre de points qui est un entier.

Dans ce cas, on peut utiliser la syntaxe suivante :

```
type valeur = Roi | Dame | Valet | Point of int
```

Pour construire une carte avec un constructeur prenant un argument, on utilisera la syntaxe ci-dessous :

```
let c = Point 5
```

Cela permet notamment d'avoir plusieurs constructeurs correspondant aux différents types de valeurs que l'on peut chercher à avoir. Par exemple, supposons que l'on ait également un type `couleur` défini ainsi :

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

On pourrait définir un type `carte` comme un type produit d'un élément de type `valeur` et d'un élément de type `coeur`. Cependant, on peut aussi souhaiter qu'une carte soit un Joker. Avec les sommes disjointes, on peut prendre en considération ces deux possibilités :

```
type carte = Joker | Carte of valeur * couleur
```

On remarque que l'on peut avoir un constructeur qui prend plusieurs arguments.

7.3.2 Types récursifs et Listes

On a vu que l'on pouvait définir des types par énumération de constructeurs, et que ces constructeurs pouvaient prendre des arguments de types donnés.

De la même façon que l'on peut définir des fonctions récursivement en appelant les fonctions dans elles-mêmes, on peut **définir des types récursivement** en prenant ces types en arguments de leurs propres constructeurs.

De la même façon que pour les fonctions, on doit définir :

- des valeurs de bases du type, qui ne sont pas des constructeurs prenant un argument du type en cours de définition ; c'est l'équivalent des cas terminaux pour les définitions récursives de fonction
- des valeurs utilisant des constructeurs prenant des arguments du type en train d'être défini.

Un exemple de type pouvant être défini de façon récursive est le type liste : une liste est soit la liste vide, soit la donnée d'un couple premier élément de la liste/suite de la liste privée de ce premier élément. La suite de la liste est elle-même un élément de type liste, on peut donc définir le type récursif suivant, pour une liste d'entiers :

```
type intlist = Vide | Cell of int * intlist ;;
```

Dans une liste, le premier élément s'appelle la tête (head en anglais), et le reste de la liste s'appelle la suite, ou la queue (tail en anglais).

Les types récursifs permettent en particulier de combiner l'utilisation de fonctions récursives avec le filtrage par motif : les cas terminaux sont renvoyés pour les valeurs de base, et les appels récursifs sont effectués pour les constructeurs prenant en argument le type récursif.

Exemple : la fonction `appartient` suivante prend en entrée une liste d'entiers et un entier e, et renvoie Vrai ou Faux selon que l'entier appartienne à la liste ou non. Soit la liste est vide et la valeur Faux et renvoyée, soit la liste n'est pas vide (couple (i,s)) et on renvoie la disjonction $e == i$ OU appartient e s.

```
let rec appartient x l =
    match l with
    | Empty -> False
    | Cell(i,s) -> i == x || appartient x s
;;
```

En OCaml, **il existe un type list prédéfini**, qui est polymorphe : des listes de plusieurs types peuvent être créées. En revanche, tous les éléments à l'intérieur d'une liste sont de même type.

La syntaxe des listes en Python est la suivante : les éléments sont notés entre crochets et séparés par des points-virgules. Pour des éléments `e1`, `e2`, etc. d'un type donné, la liste contenant les éléments se note `[e1;e2;...;en]`. La liste vide se note `[]`.

Pour construire la liste constitue d'une suite s (qui est elle-même une liste) et d'une tête t, on peut noter `t::s`. On peut également enchaîner plusieurs éléments à la suite : `e1::e2::e3::s`.

Le module List fourni également plusieurs fonctions :

- `List.hd l` et `List.tl l` renvoient respectivement la tête et la suite d'une liste `l`
- `List.length l` renvoie la longueur de la liste `l`
- `l1 @ l2` renvoie la concaténation de `l1` et `l2`.

Exercice 20

1) Définir récursivement une fonction `longueur` qui calcule la longueur d'une liste.

2) Définir récursivement une fonction `concat` qui calcule la concaténation de deux listes.

3) Définir récursivement une fonction `max` qui calcule la valeur maximale dans une liste d'entiers.

4) Définir récursivement une fonction `somme` qui calcule la somme des éléments dans une liste d'entiers.

5) Définir récursivement une fonction `produit` qui calcule le produit des éléments dans une liste d'entiers.

6) Définir récursivement une fonction `liste_syracuse` qui prend en entrée un entier naturel non-nul a et un entier n et renvoie la liste à n éléments contenant a et ses successeurs dans la suite de Syracuse.

7) Définir récursivement une fonction `list_concat` qui prend en entrée une liste de listes et renvoie la concaténation de toutes les listes.

8) Définir une fonction `liste_fonction` qui prend en entrée une liste, une fonction et renvoie la liste des $f(x)$ pour tous les éléments x de la liste en entrée.

8 Exercices

Exercice 21

1) On considère le matériel (ensemble des pièces restantes) d'un joueur d'échecs dans une partie. Définir un type produit pour représenter le matériel d'un joueur, où à chaque pièce est associée la quantité dont dispose le joueur (entier).

2) Définir une fonction qui prend en entrée le matériel d'un joueur représenté sous la forme d'une liste de type `piece` (comme défini précédemment) et renvoie le matériel sous la forme d'un produit nommé (comme défini précédemment).

3) Aux échecs, la valuation habituelle des pièces est la suivante : 1 pour le Pion, 3 pour le Cavalier et le Fou, 5 pour la Tour et 9 pour la Dame. Définir une fonction qui prend en entrée le matériel d'un joueur (représenté avec le type défini précédemment) et qui renvoie sa valeur totale.

4) Ecrire une fonction qui prend en entrée les matériels d'un joueur 1 et d'un joueur 2 dans une partie et renvoie 1 si le joueur 1 a l'avantage matériel, -1 si c'est le joueur 2, et 0 si les deux sont égaux.

Exercice 22

Au Poker, les différents types de mains que l'on peut obtenir sont, dans l'ordre, de la plus faible à la meilleure : Carte Haute, Paire, Double Paire, Brelan, Suite, Couleur, Full, Carré, Quinte Flush, Quinte Flush Royale.

1) Créer un type `main` dont les constructeurs correspondent aux différents types de mains possibles.

2) Créer une fonction qui prend en entrée les types de mains de deux joueurs et renvoie 1, 0 ou -1 selon que la main du joueur 1 soit supérieure, égale ou inférieure à celle du joueur 2. *On pourra définir une valeur numérique pour chacune des mains possibles et comparer leurs valeurs.*

Exercice 23

Le tri par insertion fonctionne de la façon suivante : soit une liste de valeur non-triée. On prend le premier élément, on trie la suite de la liste, et on insère l'élément dans la liste triée.

1) Ecrire une fonction `insertion` qui prend en entrée un entier et une liste triée d'entiers et renvoie la liste triée à laquelle on a rajouté l'entier.

2) Ecrire une fonction `tri_insertion` qui prend en entrée une liste et renvoie la liste triée à l'aide de la fonction `insertion` et de la méthode présentée.

Exercice 24

Le tri fusion se fait de la façon suivante :

— On divise une liste en deux sous-listes de tailles égales, à une unité près ;

— On trie chacune des sous-listes ;

— On fusionne les listes triées.

1) Ecrire une fonction `division` qui prend en entrée une liste et la divise en deux listes de tailles égales.

2) Ecrire une fonction `fusion` qui prend en entrée deux listes triées et renvoie la fusion triée des deux listes.

3) Ecrire une fonction `tri_fusion` qui fait le tri d'une liste à l'aide des deux fonctions définies précédemment et de la méthode présentée.