FP1

November 29, 2024

1 Prérequis

Le langage considéré est Microsoft `F#` fournit par la plateforme/outil `dotnet` (.Net) qu'il va falloir installer.

De là, la commande `dotnet fsi` permet de lancer un interprète en ligne de commande qui accepte des *expressions* terminées par `;;` .

Pour éviter de perdre les éléments saisis, il est possible de mettre le code dans un fichier `code.fsx` et d'utiliser la commande `dotnet fsi code.fsx` (et les point-virgules ne sont plus demandés).

Un exemple de code simple est donné par la fonction `printfn`:

[1]:
```
printfn "Welcome to F#"
printfn "Value is %i" 12
```

```
Welcome to F#
Value is 12
```

Notez que le nom de la fonction est toujours le 1er élément d'une expression et que les parenthèses ne sont utiles que pour lever les ambiguités.

2 "Tout est fonction"

Une fonction (anonyme) peut être définie avec le mot-clef `fun` et une fonction peut être *évaluée* en la faisant suivre d'une valeur des paramètres.

[2]:
```
(fun x -> x+1) 2
```

Notez que même les "opérateurs" sont des fonctions et peuvent être utilisés en notation "préfixée":

[3]:
```
1+2 = (+) 1 2
```

Le langage fournit alors des "sucres syntaxiques" pour définir plus simplement des éléments avec le mot-clef `let`:

[4]:
```
let x=2
x+1
```

Les variables peuvent être des fonctions:

[5]:
```
let f = fun x -> x+1
f 2
```

Le code précédent peut s'écrire plus simplement:

[6]:
```
let f x = x+1.0
f 2.0
```

Les fonctions peuvent être passées en paramètre ou en retour d'autres fonctions.

Un exemple bien connu est la dérivée d'une fonction qui est une fonction:

[7]:
```
let derivate f = fun x -> ((f (x+0.01)) - (f x))/0.01
let g = derivate f
g 2
```

3 Pattern `matching`

Un type de données est généralement spécifié par un ensemble fini de constructions.

Par exemple, true/false pour les booléens, 0 et (+1) pour les entiers, etc.

Le code suivant illustre alors comment définr une fonction "par cas" avec l'instruction `match_with`:

[8]:
```
let even_or_odd n =
  match (n%2)=0 with
  | true  -> (string n)+" is even"
  | false -> (string n)+" is odd"

even_or_odd 13
```

[8]:
```
13 is odd
```

Notez que dans le cas des booléens, l'instruction `match_with` est similaire à `if_then_else` qu'il aurait été possible d'utiliser.

4 Récursivité

Une majorité de "types" de données peuvent être définis `réc`ursivement.

Par exemple, un entier peut être vu comme le successeur d'un autre entier (cas général) avec un élément particulier 0 (cas de base), une liste à laquelle on ajoute un élément est encore une liste, etc.

Les fonctions sur un tel type sont alors "récursives" et doivent être définies avec `let_rec`.

Par exemple, la fonction factorielle peut être définie récursivement de la façon suivante:

[9]:
```
let rec factoriel n =
  match n with
  | 0 -> 1
```

```
  | _ -> n * (factoriel (n-1))

factoriel 10
```

Notez que la récursivité permet de réaliser des traitements répétitifs traduits par des boucles `for/while` dans les langages impératifs comme `C/C#/Java/...`

5 Généricité

Le code utilisé dans la fonction `factoriel` peut être généralisé/paramétré pour définir plus rapidement d'autres fonctions comme l'illustre le code suivant:

[10]:
```
let rec gene v f n =
  match n with
  | 0 -> v
  | _ -> f n (gene v f (n-1))

let fact  = gene 1 (*)
let sigma = gene 0 (+)
sigma 10
```

6 Les listes

Les collections d'éléments peuvent être définies à l'aide des listes récursives (là, où les langages impératifs utilisent des tableaux).

Les listes sont définies par la liste vide `[]` et l'opérateur `(::)` qui ajoute une valeur à une autre liste.

Le code suivant propose alors deux exemples d'utilisation de ces éléments:

[11]:
```
let range = gene [] (fun n r -> n::r)
printfn "%A" (range 10)

let rec map f l =
  match l with
  | []    -> []
  | x::xs -> (f x)::(map f xs)
printfn "%A" (map ((+) 1) [14;12;13])
```

```
[10; 9; 8; 7; 6; 5; 4; 3; 2; 1]
[15; 13; 14]
```

7 Exercises

1. Définir une fonction générique `reduce` sur les listes (similaire à la fonction `gene` sur les entiers).
2. Utiliser cette fonction pour définir autrement la fonction `map`.

3. Comment utiliser la fonction pour concaténer/combiner 2 listes (et comment en faire un opérateur `(@)` ?).
4. Donner une expression équivalente à `map f (l1 @ l2)`.
5. Comment utiliser la fonction générique pour obtenir la somme des éléments d'une liste ? La plus grande/petite valeur ?
6. Définir un opérateur `map2 f [x12; ...; xn] [y1; ...; yn] = [f x1 x2; ...; f xn yn]`
7. Comment utiliser l'opérateur précédent pour calculer la somme ou le produit scalaire de 2 vecteurs (un vecteur étant représenté par une liste) ?