Programmation Fonctionnelle

IUT de Lens

DUT S4

Plan

Programmation Fonctionnelle

- Un programme est considéré comme un ensemble de fonctions mathématiques.
- Le calcul est considéré comme comme l'évaluation d'une fonction mathématique.
- Le paradigme fonctionnel trouve son origine dans le λ-calcul.
- Absence d'effet de bord.
- Gestion automatique de la mémoire.

Programmation Fonctionnelle

- Important : la programmation fonctionnelle fait largement appel à la récursivité (les boucles sont une particularité de la programmation impérative).
- Le langage fonctionnel le plus étudié en France : Caml
  - Categorical Abstract Machine Language.
  - Aujourd'hui OCaml successeur de Caml Light : un langage multi-paradigme (fonctionnel, impératif, orionté objet).
  - Un langage fortement typé : toute expression possède un type (même les effets de bord).
  - Un langage compilé avec un mode Intéractif.

Typage

- Analyse la cohérence des programmes soumis avant toute tentative de compilation ou d'exécution.
- Principale analyse de cohérence : typage vérifiant que les opérations utilisées sont déjà définies et que les valeurs appliquées ont un sens.
- Plus d'erreurs détectées (et corrigées) à la compilation ; moins d'erreurs à l'exécution.
- Polymorphisme : accepte des expressions dont les types ne sont pas déterminé (e.g. variables de type) $\longrightarrow$ réutilisabilté du code.

Caml et λ-Calcul

- Un programme est une expression.
- L'exécution d'un programme correspond à l'évaluation d'une expression.
- Une expression est un λ-terme (typé).
- Correspondance entre types et formules logiques (logique intuitionniste).
- L'évalution d'un λ-terme est obtenue par l'application de β-réductions.

Plan

Eléments de base

- Opérateurs de comparaison : $=$, $<>$ (différent), $>$, $<$, $>=$, $<=$.
- Opérateurs arithmétiques sur les entiers : +, -, *, /, mod.
- Opérateurs arithmétiques sur les flottants : +., -., *., /., sqrt, exp, log,...
- Booléens (bool) : true, false
  - Conjonction (et logique) : x && y
  - Disjonction (ou logique) : x || y
  - Négation : not

Eléments de base

Exemple :

```
#1;;
- : int = 1
# 2.3;;
- : float = 2.3
# 1.2 + 3.1 ;;
Error : This expression has type float but ...
# 1.2 +. 3.1 ;;
- : float = 4.3
# exp;;
- : float -> float = <fun>
```

Eléments de base

- Les caractères :

```
# 'a';;
- : char = 'a'
```

- Les chaînes de caractères :

```
# "bonjour" ;;
- : string = "bonjour"
```

- Concaténation : on utilise l'opérateur ^

  Exemple :

```
#"abc" ^ "def" ;;
- : string = "abcdef"
```

- Longueur : `String.length s`

Conversion de type

- Il existe en OCaml des fonctions de conversion qui permettent de passer d'un type à une autre :
  - `float_of_int`
  - `int_of_float`
  - `int_of_string`
  - `string_of_int`
  - `float_of_string`
  - `string_of_float`

- Exemple :

```
float_of_int(4) /. 2.0 ;;
- : float = 2.
```

Déclaration globales

- Les expressions sont déclarées avec le mot-clé : let.
  - En mathématiques : ≪ soit x un entier égal à 2 ≫
  - En Caml : `let x = 1;;`
  - Mode intéractif : `x : int = 2`
- On ne modifie pas la valeur d'un nom, on le redéfinit (pas d'effet de bord).

Déclaration globales

Exemple :

```
#let x =1;;
val x : int = 1
#x=2;;
- = bool = false
#let x = 2;;
val x : int = 2
```

Déclaration locale

- Syntaxe : `let x = e in e';;`
- Sens : `x` ayant la valeur `e` n'est visible que dans `e'`.
- Utilité :
  - Eviter des déclarations globales qui ne sont utilisées que très localement.
  - Eviter l'utilisation de variable qui ne sont utilisées qu'avec peu de valeurs.
- Une variable ne peut pas être réutilisée à l'extérieur de sa portée.
- Il est possible de réutiliser un nom déclaré globalement dans une déclaration locale.

Déclaration locales

Exemple :

```
#let x =1 in x +1;;
- : int = 2
#x;;
Error : Unbound value x
#let y = 2;;
val y : int = 2
#let y =3 in y+1;;
- : int = 4
#y;;
- : int =2
#let z= y in z+y;;
- : int = 4
```

Déclaration locales

Il est possible d'imbriquer des déclarations locales :

```
let ...in let ... in ...;;

#let x =1 in y = x+1 in let z = y+1 in z;;
- : int = 3
```

Fonctions

- Les fonctions forment les constituants élémentaires des programmes en Caml.
- Un programme est constitué d'une collection de définitions de fonctions suivie d'un appel à la fonction qui déclenche le calcul voulu.
- On utilise une syntaxe très proche des notations mathématiques :

```
# let carre x = x*x ;;
carre : int -> int = <fun>
# let carre 4 ;;
- : int = 16
```

Fonctions anonymes

- Il est possible de déclarer une fonction sans lui donner de nom en utilisant le mot-clé : function.

```
# (function x -> x*x) 4 ;;
- : int = 16
```

- Une fonction anonyme est une expression comme une autre qui peut être liée à un identificateur en utilisant **let**

```
# let carre = (function x -> x*x) ;;
carre : int -> int = <fun>
```

Fonctions à plusieurs paramètres

- Une fonction à plusieurs paramètres $x_1, \ldots, x_n$ est construite comme une fonction qui est appliquée à x_1 et fournira une fonction qui appliquée à x_2 et fournira une fonction qui est appliquée à x_3 et ainsi de suite :

  `function x1 -> function x2 -> ··· -> function xn -> e`

- Simplification : il est possible de se passer du mot-clé `function` lorsque la fonction définie possède un nom :

  `let nom x1 x2 ··· xn = e`

Fonctions locales

- Une fonction peut être déclarée localement à l'aide du mot-clé **in**.

```
# let carre x = x*x in carre 4;;
- : int = 16
```

- Une fonction locale ne peut pas être réutilisée à l'extérieur de sa portée : une fonction locale n'est définie qu'à l'intérieur de l'expression qui suit **in**.

Fonctions locales

Exemple :

```
# let carre1 x = x*x ;;
val carre1 : int -> int = <fun>
# let carre2 x = x*x in carre2 3;;
- : int = 9
# carre2;;
Error : Unbound value carre2
```

Imbrication de déclarations de fonctions locales

- Il est possible d'imbriquer plusieurs déclarations locales :
  `# let ··· in ···let··· in ···;;`

- Exemple :

```
# let carre x = x*x in (let somme y z = y+z in
somme (carre 3) 1);;
- : int = 10
```

Expressions conditionnelles

- Construction classique :
  `if ··· then ··· else ··· ;;`

- Les expressions évaluées dans les deux cas **then** et **else** doivent avoir le même type.

- Exemple :

```
# if 3>5 then "3 superieur a 5" else "3
inferieur a 5";;
- : string = "3 inferieur a 5"
# if 3>5 then "3 superieur a 5" else 5;;
This expression has type int but is here used
with type string
```

Exercice

- Ecrire une fonction permettant de vérifier si une variable y et divisible par une variable x :

Exercice

- Ecrire une fonction permettant de vérifier si une variable *y* et divisible par une variable *x* :

```
 #let divisible_par x y = (y mod x) = 0;;
divisible_par : int -> int -> bool = <fun>
#divisible_par 9 10;;
- : bool = false
#divisible_par 3 6;;
- : bool = true
```

Fonctions récurisives

- Une fonction récursive est une fonction qui fait appel à elle-même pour sa construction : $f(x) = \cdots f(\cdots) \cdots$
- Pour que l'évaluation d'une fonction f récursive termine et ne boucle pas à l'infinie, il faut qu'il y ait des cas de base qui ne font pas appel à f.
- En Caml, les définitions récursives sont possibles par l'ajout du mot-clé `rec` :

```
 #let rec factorielle n = if n = 0 then 1 else n
* factorielle (n - 1);;
factorielle : int -> int = <fun>
```

- Le cas de base dans la précédente définition est le cas où la variable `n` vaut `1` (pas d'appel à `factorielle`)

Fonctions récurisives

- Les calculs en algorithmique effectués grâce aux boucles **Pour** et **Tant que** peuvent être réalisés en programmation fonctionnelle grâce à la récursivité.
- Exemple : considérons un algorithme possédant la forme `f(n)`

```
:= Pour i=1 à n faire
                x= x operation g(i);
        fin;
        retourne x;
```

Cette dernière peut être définie comme une fonction récursive de la manière suivante :

```
let rec f n i = if i = n then (g n) else
(g i) operation (f (i+1) n);;
```

Exercice

- Ecrire une fonction permettant de calculer la somme des nombres entiers qui vont x à y :

Exercice

- Ecrire une fonction permettant de calculer la somme des nombres entiers qui vont x à y :

```
 #let rec somme x y = if x = y then x else
if x> y then (somme y x) else x+(somme (x+1)
y);;
somme : int -> int -> int = <fun>
```

Fonctions mutuellement récursives

- Deux fonctions sont mutuellement récursives lorsqu'elles s'appellent l'une l'autre. On parle de récursivité croisée.

- La construction en Caml se fait par l'utilisation des deux mot-clés `rec` et `and` :
  `let rec f1 = e1 and ⋯ and fn = en;;`
- Exemple :

```
 let rec pair k = if k=0 then true
else impair (k-1) and
impair k = if k=0 then false else pair (k-1);;
```

Exercice

- Ecrire des fonctions mutuellement récursives en utilisant la définition suivante de la suite de Fibonacci :
  $u_0 = 1; v_0 = 1; u_n = u_{n-1} + 2 * v_{n-1}; v_n = 3 * u_{n-1} + v_{n-1}$

Exercice

- Ecrire des fonctions mutuellement récursives en utilisant la définition suivante de la suite de Fibonacci :
  $u_0 = 1; v_0 = 1; u_n = u_{n-1} + 2 * v_{n-1}; v_n = 3 * u_{n-1} + v_{n-1}$

```
 #let rec u = function n -> if (n = 0) then 1
else u(n - 1) + 2 * v(n - 1)
and v = function n ->
if (n = 0) then 1 else 3 * u(n - 1) + v(n - 1);;
u : int -> int = <fun>
v : int -> int = <fun>
```

Filtrage

- Le filtrage est une analyse de formes servant à faire des testes.
- Le filtrage est réalisé sur un argument.
- Un filtre doit être du même type que l'argument. Par exemple, on doit utiliser des variables et des nombres entiers comme filtres pour une variable entière.
- Un filtre peut être composé de constantes et de variables.

Filtrage

- En Caml, les filtre sont testés successivement dans l'ordre de présentation.
- Le symbole _ est utilisé comme un filtre universel : il correspond à toute les expressions.
- La construction syntaxique de base en Caml est comme suit :

```
match expression with
| filtre1 -> e1
⋮
| filtren -> en
```

où `e1,...,en` sont du même type.

Filtrage

Exemple :

```
 let f n = match n with
| 0 -> true
|_ -> false;;
```

Cette fonction permet de tester si un nombre est égal à 0 ou non.

Exercices

- Fournir les filtres possibles pour une variable booléenne.

Exercices

- Fournir les filtres possibles pour une variable booléenne.

  `true, false, _`

- Ecrire une fonction permettant de faire la négation d'une variable booléenne.

Exercices

- Fournir les filtres possibles pour une variable booléenne.

```
true, false, _
```

- Ecrire une fonction permettant de faire la négation d'une variable booléenne.

```
let negation b = match b with
|true -> false
|false -> true;;
```

- Ecrire une fonction retournant la disjonction (”ou” logique) de deux variables booléennes.

Exercices

- Fournir les filtres possibles pour une variable booléenne.

```
true, false, _
```

- Ecrire une fonction permettant de faire la négation d'une variable booléenne.

```
let negation b = match b with
|true -> false
|false -> true;;
```

- Ecrire une fonction retournant la disjonction (”ou” logique) de deux variables booléennes.

```
let disjonction b1 b2 = match (b1,b2) with
|(true,_) -> true
|(_,true) -> true
|_ -> false;;
```

Filtrage conditionnel

- En Caml, il est possible d'évaluer une condition booléenne lors du filtrage à l'aide du mot-clé `when` après le filtre :
  `... | filter when condition -> e ...`

- Exemple :

```
let xor b1 b2 = match (b1,b2) with
(b1, b2) when b1 = b2 -> false
| _ -> true ;;
```

Structure de tuple

- La structure de tuple permet de mettre des éléments de types différents dans la même structure : $(e_1, e_2, \ldots, e_n)$
- Construction :

```
# let ville = ("Lens", 62, "Pas-de-Calais");;
val ville : string * int * string = ("Lens", 62,
"Pas-de-Calais")
```

- Déconstruction :

```
# let (nom,code,dpt) = ville ;;
val nom : string = "Lens"
val code : int = 62
val dpt : string = "Pas-de-Calais"
```

Structure de liste

- La liste est un type prédéfini en OCaml :

```
#[1;2;3];;
- : int list = [1;2;3]
#[1.0;2.0;3.0];;
- : float list = [1.0;2.0;3.0]
```

- Dans une liste, tous les éléments doivent être du même type.

- La liste vide :

```
[] ;;
- : 'a list = []
```

Structure de liste

- Ajout d'un élément à la tête d'une liste :

```
#0::[1;2;3];;
- : int list = [0;1;2;3]
```

- Concaténation de deux listes :

```
[1;2] @ [3;4] ;;
- : int list = [1;2;3;4]
```

Filtrage et listes

- En Caml, les filtres généralement utilisés pour une liste sont : `[], [e], [e1;e2], ..., [e1;...;en], t::q, t1::t2::q, t1::···::tn::q, _`

- Le filtre [] signifie que la liste est vide, [e] signifie que la liste contient exactement un élément e (e peut être une variable), [e1 ;e2] signifie que la liste contient exactement deux éléments ...

- Le filtre t : :q signifie que la liste contient au moins un élément dont le premier est t et le reste est la liste q (t et q peuvent être des variables).

- le filtre t1 : :t2 : :q signifie que la liste contient au moins deux éléments dont le premier est t1, le deuxième est t2 et le reste est la liste q.

Exercice

Ecrire une fonction retournant la taille d'une liste.

Exercice

Ecrire une fonction retournant la taille d'une liste.

```
let rec taille l = match l with
|[] -> 0
|_::q-> 1+ (taille q);;
```

Application partielle

- Une fonction possédant plusieurs arguments peut être appelée en instanciant uniquement une partie de ces derniers.

- Exemple :

```
let f x y = x+y ;;
val f : int -> int -> int = <fun>
let f 2;;
- : int -> int = <fun>
```

- Instancier des arguments plus à droite :

```
let g y = f y 2;;
val g : int -> int = <fun>
```

Polymorphisme

- Considérons la définition suivante : `let f x = x;;`
- Quel est le type `x` ? Cette variable peut posséder n'importe quel type !
  `val f : 'a -> 'a = <fun>` (une telle fonction est dite polymorphe)
- `'a` désigne une variable de type qui peut prendre différents types : int, float, bool, etc.

```
# f 1;;
- : int = 1
# f true ;;
- : bool = true
# f 2.5;;
- : float = 2.5
```

Polymorphisme

Exercice : écrire une fonction permettant la composition de deux fonctions.

Polymorphisme

Exercice : écrire une fonction permettant la composition de deux fonctions.

```
# let compose f g x = f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c ->
'b = <fun>
```