Algorithmique fonctionnelle - Présentation, OCaml

©EISTI

21 janvier 2013

1 Fonctions
- Rappels mathématiques
- Rappels informatiques

2 Paradigme fonctionnel

3 OCAML

4 Documentation

Fonctions

Relation binaire

- Le **produit cartésien** de 2 ensembles A et B est l'ensemble $A \times B$ ayant pour éléments les couples (a, b) avec $a \in A$ et $b \in B$.
- Une **relation binaire** définie sur A et B est une partie du produit cartésien $A \times B$ notée aRb, $R \subset A \times B$.
- R est **fonctionnelle** ssi
  $\forall a \in E, \exists \text{ au plus } b \in F \mid aRb$,
  on note $R(a)$, l'unique b tel que aRb

Fonctions

Fonction

- Etant donné 2 ensembles, l'ensemble de départ E et l'ensemble d'arrivée F, une fonction f de E vers F est la relation binaire associant à chaque élément x de l'ensemble de départ un élément $f(x)$ de l'ensemble d'arrivée
- on appelle alors f, une application de E dans F
- E est appelé **domaine**, F, **codomaine**
- $E \rightarrow F$ est la **signature** de la fonction f

On note, par exemple :

$$f : \mathbb{R} \rightarrow \mathbb{R}$$
$$x \mapsto x * x$$

Fonctions

En informatique

Une fonction est une suite d'instructions qui peut prendre ensemble de paramètres en entrée et qui produit (retourne) un résultat à la fonction appelante

Notation

fonction nomDeLaFonction(liste de paramètres) : **type retourné**
variables
  variables locales
debut
  instructions
  retourner(resultat)
fin fonction

Fonctions

Notation

Signature de la fonction :
calculCarre : entier $\rightarrow$ entier
fonction calculeCarre(x : entier) : **entier**
variables
    entier : res
debut
    res $\leftarrow$ x*x
    retourner(res)
fin fonction

Fonctions

Conditions

On notera, en en-tête, les conditions :

- pour les variables en entrée : **préconditions**
- pour l'élément retourné : **postconditions**

Fonctions

Notation

calculCarre : entier $\rightarrow$ entier
precondition : x entier
postcondition : valeur au carré de x
fonction calculeCarre(x : entier) : **entier**
variables
  entier : res
debut
  res $\leftarrow$ x*x
  retourner(res)
fin fonction

Fonctions

Prédicats

Un prédicat est une fonction qui vérifie une propriété. Elle retourne un booléen qui évalue la véracité de cette propriété.

- `estPair(x)`, `estDivisiblePar(x, y)...`

Définition

- La programmation fonctionnelle est un type de programmation basé sur la définition et l'application de fonctions.
- La récursivité est au coeur de ce paradigme.
- Les fonctions s'appliquent sur des valeurs, la fonction étant une valeur comme une autre.
- Les variables "n'existent pas", les paramètres d'entrée d'une fonction sont des **valeurs immutables**.

Réaction

Je veux du Pascal, C, du Java et rien d'autre !

Thése de Church-Turing

De l'origine à la modernité, la boucle est bouclée !

- 1930 : A. Church pose la théorique du λ-calcul, fondement des langages fonctionnels
- 1936 : A. Turing théorise le fonctionnement d'un ordinateur
- Tout algorithme peut être calculé par une machine de Turing
- Le λ-calcul et la machine de Turing sont équivalents
- 2013-2014 : Java 1.8 intègre les λ-expressions

Et alors ?

Avantages et inconvénients

- Avantages
  - simple, intuitif
  - proche d'une description mathématique (récurrence)
  - facile à analyser
  - pas d'effet de bord, tests unitaires simplifiés
  - bien adapté à certaines structures de données
- Inconvenients
  - gourmand en ressources (utilisation d'une pile de stockage des informations)

Langages fonctionnels

Définition

- 1960 : LISP
- 1985 : CAML (INRIA : `http://caml.inria.fr/`)
- 1990 : Caml light, Haskell
- 1996 : Objective Caml : (`http://caml.inria.fr/ocaml/index.fr.html`)

OCaml

Interpréteur

```
# e;;
- : type = val
```

- `ocaml` : lancement
- # est l'invite de commande
- e est l'expression à évaluer, `;;` la marque de fin de ligne
- - : marque la réponse
- type est le type de retour
- val est la valeur évaluée
- `exit(0);;` : sortie

OCaml

Exemple

```
# 7 + 5 ;; (* un commentaire *)
- : int = 12
```

Type

Définition

Un type est défini par :

- un ensemble de valeurs
- un ensemble d'opérations sur ces valeurs
- un nom de type

OCaml est fortement typé, toute donnée a un type. Il utilise *l'inférence de type*, il devine les types. Par contre, il ne fait pas de cast.

Il existe 2 types de type : **type de base** et **type composé**

Entiers

int

- Représente les entiers avec signe 31 bits (environ +/- 1 milliard) avec processeurs 32 bits ou bien 63 bits avec processeurs 64 bits
- Opérations : `+ - * / mod`
- Constantes : `max_int min_int`
- Les opérations sont `binaires`
- Règles mathématiques usuelles pour la priorité des opérateurs

Réels

float

- Distinction des entiers, représente les réels double précisions IEEE
- Opérations : `+.` `-.` `*.` `/.` `**`
- Opérer un entier et un réel : `float_of_int`
  8.2 *. float_of_int(4) ; ;

Booléens

bool

- Valeurs : `true` `false`
- Opérations : `&&` `||` `not`

Caractère et chaînes

char et string

- Valeurs char : 'a' '\n'
- Valeur string : "Bonjour" ^ " les cpi2s" ; ;
- Opération : concaténation ^
- Fonctions : `int_of_char, String.length, ...`

n-uplets

n-uplets

- Ils représentent le produit cartésien de valeurs
- Ils peuvent combiner différents types
- (12, 5, 4, 3) ; ;
  - : int * int * int * int = (12, 5, 4, 3)
- ("Maxime", 0.) ; ;
  - : string * float = ("Maxime", 0.)

Types

Exemples

- 12 ; ;
  - : int = 12
- 12. ; ;
  - : float = 12.
- sqrt(5) ; ;
  Error : This expression has type int but an expression was expected of type float
- 7.2 - 3 ; ;
  Error : This expression has type float but an expression was expected of type int
- ("Maxime", 0.) = ("maxime", 0.) ; ;
  - : bool = false
- ("Maxime", 0.) = (0., "Maxime") ; ;
  Error : This expression has type int * string but an expression was expected of type string * int

Si alors sinon

if then else

- Syntaxe **indissociable** `if test then e1 else e2`
- Cas d'utilisation (if 1=1 then 2 else 3) + 1 ; ; est évalué et donne 2

Définition

let

L'instruction `let` permet de définir des variables ainsi que leur portée

- **let** variable = valeur [**in** expr]
- Cette diapositive a un but introductif
- Il sera communément admis, dans ce cours, que la définition de variables locales ou globales sera proscrite

Définition

let

```
# let x=5;;
val x : int = 5
# x + 2;;
- : int = 7
# let y=2 in y+5;;
- : int = 7
# x;;
- : int = 5
# y;;
Error : Unbound value y
```

Fonctions

Notations

Proche de la théorie mathématique :

En maths :

$$f : \mathbb{N} \to \mathbb{N}$$
$$x \mapsto x * x$$

En OCaml :

```
# let f = function x -> x*x;;
val f : int -> int = <fun>
```

Fonctions

Notations

- `let` nom = `function` var -> expr(var)
- `let` nom = `fun` var -> expr(var)
- `let` nom var = expr(var)

PAS DE **RETURN**, ni de sécification de type !

Fonctions

Exemples

- let moins3 = fun x -> x - 3 ; ;
- let plus x y = x + y ; ;
  équivaut à
  let plus = fun x -> fun y -> (x + y)
  let plus = fun x y -> (x + y)
- let concate s1 s2 = s1 ^ s2 ; ;
- let applik = fun f -> fun x -> (f x) ; ;
- let g f x y = ((f x) -. (f y)) /. (x-.y) ; ;
- let valabs x = if x < 0 then -x else x ; ;
- let et = fun x y -> if x then y else false ; ;

Fonctions

Exemples

- let moins = fun x y -> x - y ; ;
  val moins : int -> int -> int = <fun>
- moins ; ;
  - : int -> int -> int = <fun>
- moins 3 ; ;
  - : int -> int = <fun>
- moins 3 2 ; ;
  équivaut à (moins 3)(2) ; ; équivaut à (moins 3) 2
  - : int = 1
- let moins3 x = moins x 3 ; ;
- let plus = (+) ; ;
  val plus : int -> int -> int = <fun>
- let plus3 = plus 3 ; ;
  val plus3 : int -> int = <fun>

Fonctions

Applications

- moins3 3 ; ;
  - : int = 0
- moins3(3) ; ;
  - : int = 0
- (moins3 3) ; ;
  - : int = 0
- plus 3 2 ; ;
  - : int = 5
- (plus 3 2) ; ;
  - : int = 5
- (plus 3) (2) ; ;
  - : int = 5
- plus (3, 2) ; ;
  Error : This expression has type int * int but an expression was expected of type int

Un peu de lecture...

Documentation

- `http://mirror.ocamlcore.org/ocaml-tutorial.org/fr.html`
- `http://caml.inria.fr/pub/docs/manual-ocaml/`
- `http://caml.inria.fr/pub/docs/manual-ocaml/libref/index_values.html`