Algorithmique fonctionnelle - Récursivité - Pattern-matching

29 janvier 2013

Récursivité

Définition

Une fonction est dite **récursive** si elle s'appelle elle-même dans sa propre définition

Correspondance mathématique

- Principe de récurrence
- Exemple : définition des entiers :
  - 0 est un entier
  - n est un entier, alors $n+1$ est un entier
- Montrer, par récurrence, que

$$\forall\, n \in \mathbb{N},\ \sum_{k=0}^{n} k = \frac{n(n+1)}{2}$$

Récursivité

Représentation

$f(x) =$ Si $c(x)$ Alors
$g(x)$
Sinon
$h(f(s(x)), x)$
FinSi

où

- $c(x)$ est le **critère d'arrêt**, fonction de x, sa valeur est un booléen
- $g(x)$ est le **cas trivial**, en général, retourne une valeur constante lorsque $c(x)$ est vrai
- $s(x)$ est le **successeur** de x, fonction de x
- $h(f(s(x)), x)$ est l'**appel récursif**, fonction de x et de son successeur
- Si $h : y, z \mapsto y$, la récursivité est **terminale**.

Fonctions

Notations

En OCaml, l'instruction `let` permet, notamment, de définir une fonction :

```
let f = fun x -> x*x;;
let f x = x*x;;
```

Fonctions récursives

Notations

En OCaml, l'instruction `let rec` permet de définir une fonction récursive :

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

Fonctions récursives

Traitement de l'erreur

Si aucune précondition n'est définie sur l'entier n, elle est appelable avec n'importe quelle valeur entière.
Utilisation de la commande `failwith` :

```
let rec fact n =
    if n < 0 then
        failwith "valeur invalide"
    else
        if n=0 then 1 else n * (fact n-1);;
```

Fonctions récursives

Problème de pile d'exécution

fact 999999 ; ;
Stack overflow during evaluation (looping recursion ?).

Une seule solution...

... la **terminaison** ! !

Fonctions récursives

Cas terminal

Écrire la version récursive terminale.

Fonctions récursives

Cas terminal

Écrire la version récursive terminale.

```
let rec factRec n res =
    if n=0 then res
    else (factRec (n-1) (n * res)) ; ;

let fact n = factRec n 1 ; ;
```

Fonctions récursives

Récursivité croisée

Deux fonctions peuvent s'inter-appeler dans leur définition.

Exemple

Un nombre est **pair** si son prédécesseur est **impair**.
Un nombre est **impair** si son prédécesseur est **pair**.
0 est pair.

Fonctions récursives

Récursivité croisée

Deux fonctions peuvent s'inter-appeler dans leur définition.

Exemple

Un nombre est **pair** si son prédécesseur est **impair**.
Un nombre est **impair** si son prédécesseur est **pair**.
0 est pair.

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

Fonctions récursives

Récursivité croisée

Deux fonctions peuvent s'inter-appeler dans leur définition.

Exemple

Un nombre est **pair** si son prédécesseur est **impair**.
Un nombre est **impair** si son prédécesseur est **pair**.
0 est pair.

```
let rec pair n = (n = 0) || impair (n-1) and
impair n = (n <> 0) && pair (n-1) ; ;
```

Pattern matching

Filtrage

- Extension de l'expression algorithmique **selon ... cas ...**
- S'emploie avec différents types ou structures de données
- S'écrit `match ... with`, système de **joker**, noté : _
- **Exhaustivité** : si aucun motif n'est validé : erreur

Exemple

```
match expr with
motif1 -> instr1
| motif2 -> instr2
...
| motifn -> instrn
```

Pattern matching

Exemple

```
let rec fact = function n -> match n with
0 -> 1
| n -> n *(fact (n-1)) ; ;
```

Pattern matching

Exemple (avec joker)

```
let rec fact = function n -> match n with
0 -> 1
| _ -> n * (fact (n-1)) ; ;
```

Pattern matching

Exemple

```
let et = fun a b -> match (a,b) with
(true, true) -> true
| (true, false) -> false
| (false, true) -> false
| (false, false) -> false ; ;

#  et (3=3) (7>2)
```

Pattern matching

Exemple simplifié (sans match)

Attention : cette notation modifie le paramètre d'entrée, à éviter...

```
let et = function
(true, true) -> true
| (true, false) -> false
| (false, true) -> false
| (false, false) -> false;;

#  et ((3=3),(7>2))
```

Pattern matching

L'interpréteur est vraiment malin !

Oublions un cas ((false, true) -> false) :

```
let et = fun a b -> match (a,b) with
(true, true) -> true
| (true, false) -> false
| (false, false) -> false ; ;
```

```
Warning 8 : this pattern-matching is not exhaustive.
Here is an example of a value that is not matched :
(false, true)
```

Pattern matching

Exemple encore plus simplifié (avec joker)

```
let et = fun a b -> match (a,b) with
(true, true) -> true
| (_, _) -> false ; ;

#  et (3=3) (7>2)
```

Pattern matching

Exercice

Écrire le prédicat `estVoyelle` qui teste si un caractère en entrée est une voyelle ou non.

Pattern matching

Exercice

Écrire le prédicat `estVoyelle` qui teste si un caractère en entrée est une voyelle ou non.

```
let voyelle = function
'a' -> true
| 'e' -> true
| 'i' -> true
| 'o' -> true
| 'u' -> true
| 'y' -> true
| _ -> false ; ;
```

Pattern matching

Exercice

Écrire le prédicat `estVoyelle` qui teste si un caractère en entrée est une voyelle ou non.

```
let voyelle = function
'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
| _ -> false ; ;
```