Introduction à la programmation fonctionnelle

Notes de cours

Cours 2

25 janvier 2019

Sylvain Conchon

sylvain.conchon@lri.fr

Définitions récursives

Une fonction est récursive si elle fait appel à elle même dans sa propre définition

Par exemple, la fonction factorielle `n!` peut être définie de manière récursive, pour tout entier `n`, par les deux équations suivantes

$$
\begin{aligned}
0! &= 1 \\
n! &= n \times (n-1)!
\end{aligned}
$$

Fonctions récursives en OCaml

La définition d'une fonction récursive est introduite par l'adjonction du mot clé **rec** au mot clé **let**

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

Avantages des définitions récursives

L'écriture à l'aide d'une fonction récursive donne souvent un code plus lisible et plus susceptible d'être correct (car d'invariant plus simple) que son équivalent impératif utilisant une boucle

```
int fact(int n){
  int f = 1;
  int i = n;
  while (i>0){
    f = f * i;
    i--;
  };
  return f;
}
```

L'argument justifiant la correction de cette version est nettement plus complexe que la version récursive

processus d'évaluation

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

```
fact 3
```

processus d'évaluation

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

fact 3

$3 \neq 0 \Rightarrow$ 3 * fact(3-1)

processus d'évaluation

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

```
                 fact 3
3 ≠ 0  ⇒  3 * fact(3-1)
2 ≠ 0  ⇒  3 * 2 * fact(2-1)
```

processus d'évaluation

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

$$\begin{array}{lcl} & & \texttt{fact 3} \\ 3 \neq 0 & \Rightarrow & \texttt{3 * fact(3-1)} \\ 2 \neq 0 & \Rightarrow & \texttt{3 * 2 * fact(2-1)} \\ 1 \neq 0 & \Rightarrow & \texttt{3 * 2 * 1 * fact(1-1)} \end{array}$$

processus d'évaluation

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

$$
\begin{array}{lcl}
 & & \texttt{fact 3} \\
3 \neq 0 & \Rightarrow & \texttt{3 * fact(3-1)} \\
2 \neq 0 & \Rightarrow & \texttt{3 * 2 * fact(2-1)} \\
1 \neq 0 & \Rightarrow & \texttt{3 * 2 * 1 * fact(1-1)} \\
0 = 0 & \Rightarrow & \texttt{3 * 2 * 1 * 1}
\end{array}
$$

processus d'évaluation

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

$$
\begin{array}{rcl}
 & & \texttt{fact 3} \\
3 \neq 0 & \Rightarrow & \texttt{3 * fact(3-1)} \\
2 \neq 0 & \Rightarrow & \texttt{3 * 2 * fact(2-1)} \\
1 \neq 0 & \Rightarrow & \texttt{3 * 2 * 1 * fact(1-1)} \\
0 = 0 & \Rightarrow & \texttt{3 * 2 * 1 * 1} \\
 & \Rightarrow & \texttt{3 * 2 * 1}
\end{array}
$$

processus d'évaluation

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

$$
\begin{array}{rcl}
 & & \texttt{fact 3} \\
3 \neq 0 & \Rightarrow & \texttt{3 * fact(3-1)} \\
2 \neq 0 & \Rightarrow & \texttt{3 * 2 * fact(2-1)} \\
1 \neq 0 & \Rightarrow & \texttt{3 * 2 * 1 * fact(1-1)} \\
0 = 0 & \Rightarrow & \texttt{3 * 2 * 1 * 1} \\
 & \Rightarrow & \texttt{3 * 2 * 1} \\
 & \Rightarrow & \texttt{3 * 2}
\end{array}
$$

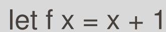
processus d'évaluation

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

		fact 3
$3 \neq 0$	$\Rightarrow$	3 * fact(3-1)
$2 \neq 0$	$\Rightarrow$	3 * 2 * fact(2-1)
$1 \neq 0$	$\Rightarrow$	3 * 2 * 1 * fact(1-1)
$0 = 0$	$\Rightarrow$	3 * 2 * 1 * 1
	$\Rightarrow$	3 * 2 * 1
	$\Rightarrow$	3 * 2
	$\Rightarrow$	6

Pourquoi ce mot clé `rec` ?

La règle de portée du `let`



let f y = if y=0 then f 2 else 4+y

Pourquoi ce mot clé `rec` ?

La règle de portée du `let`

```
let f x = x + 1

let rec f y = if y=0 then f 2 else 4+y
```

Que se passe-t-il si `n` est négatif ?

1. **n !** est définie sur $\mathbb{N}$
2. la fonction `fact` a pour type `int` $\rightarrow$ `int` et ne termine pas si son paramètre est négatif

Que se passe-t-il si `n` est négatif ?

1. **n !** est définie sur $\mathbb{N}$
2. la fonction `fact` a pour type `int` $\rightarrow$ `int` et ne termine pas si son paramètre est négatif

Quelle valeur `fact` peut-elle rendre si `n<0` ?

Que se passe-t-il si `n` est négatif ?

1. **n !** est définie sur $\mathbb{N}$
2. la fonction `fact` a pour type `int` $\rightarrow$ `int` et ne termine pas si son paramètre est négatif

Quelle valeur `fact` peut-elle rendre si `n<0` ?

La seule solution raisonnable est de **stopper** le calcul en cours et d'**expliquer** le problème à l'utilisateur

On utilise pour cela la fonction prédéfinie `failwith` m qui termine un programme en affichant le message m à l'écran

Implantation de `fact` (deuxième version)

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

Implantation de `fact` (deuxième version)

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

Encore mieux, une fonction intermédiaire pour éviter des tests inutiles

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

let fact n =
 if n<0 then failwith "argument negatif";
 factorielle n
;;
```

Implantation de `fact` (troisième version)

On peut cacher la fonction ”incorrecte” comme une fonction locale

```
let fact n =
 let rec factorielle n =
  if n=0 then 1 else n * factorielle (n-1)
 in
 if n<0 then failwith "argument negatif";
 factorielle n
;;
```

Définitions par cas

Analyse par cas d'une valeur avec la construction `match–with`

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

Ceci permet de se rapprocher encore plus de la définition mathématique

La fonction `puissance` (1/3)

$$
\begin{aligned}
x^0 &= 1 \\
x^n &= x * x^{(n-1)}
\end{aligned}
$$

La fonction `puissance` (1/3)

$$
\begin{aligned}
x^0 &= 1 \\
x^n &= x * x^{(n-1)}
\end{aligned}
$$

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

la fonction `puissance` (2/3)

Une version plus rapide qui exploite la parité de `n`

$$\begin{aligned} x^0 &= 1 \\ x^{2n} &= x^n * x^n \\ x^{2n+1} &= x * x^n * x^n \end{aligned}$$

la fonction `puissance` (2/3)

Une version plus rapide qui exploite la parité de `n`

$$
\begin{aligned}
x^0 &= 1 \\
x^{2n} &= x^n * x^n \\
x^{2n+1} &= x * x^n * x^n
\end{aligned}
$$

```
let rec puissance x n =
 match n with
  | 0 -> 1
  | 1 -> x
  | _ -> let e = puissance x (n/2) in
         if n mod 2 = 0 then e*e else x * e * e
;;
```

La fonction `puissance` (3/3)

Encore une fois, pour éviter de boucler sur un argument `n` négatif, on utilise une fonction ”englobante”

```
let puissance x n =
 let rec puissance n =
   match n with
   | 0 -> 1
   | 1 -> x
   | _ -> let e = puissance (n/2) in
          if n mod 2 = 0 then e*e else x * e * e
 in
 if n<0 then failwith "argument négatif";
 puissance n
;;
```

Évaluation de la fonction `puissance`

```
puissance 4 5
```

Évaluation de la fonction `puissance`

```
   puissance 4 5
⇒  let e = puissance 4 2 in 4 * e *e
```

Évaluation de la fonction `puissance`

```
   puissance 4 5
⇒  let e = puissance 4 2 in 4 * e *e
⇒  let e = (let e'= puissance 4 1 in e'*e') in 4 * e * e
```

Évaluation de la fonction `puissance`

```
    puissance 4 5
⇒   let e = puissance 4 2 in 4 * e *e
⇒   let e = (let e’= puissance 4 1 in e’*e’) in 4 * e * e
⇒   let e = (let e’= 4 in e’*e’) in 4 * e * e
```

Évaluation de la fonction `puissance`

```
    puissance 4 5
⇒  let e = puissance 4 2 in 4 * e *e
⇒  let e = (let e'= puissance 4 1 in e'*e') in 4 * e * e
⇒  let e = (let e'= 4 in e'*e') in 4 * e * e
⇒  let e = 4 * 4 in 4 * e * e
```

Évaluation de la fonction `puissance`

```
    puissance 4 5
⇒  let e = puissance 4 2 in 4 * e *e
⇒  let e = (let e'= puissance 4 1 in e'*e') in 4 * e * e
⇒  let e = (let e'= 4 in e'*e') in 4 * e * e
⇒  let e = 4 * 4 in 4 * e * e
⇒  let e = 16 in 4 * e * e
```

Évaluation de la fonction `puissance`

```
    puissance 4 5
⇒   let e = puissance 4 2 in 4 * e *e
⇒   let e = (let e'= puissance 4 1 in e'*e') in 4 * e * e
⇒   let e = (let e'= 4 in e'*e') in 4 * e * e
⇒   let e = 4 * 4 in 4 * e * e
⇒   let e = 16 in 4 * e * e
⇒   4 * 16 * 16
```

Évaluation de la fonction `puissance`

```
    puissance 4 5
⇒  let e = puissance 4 2 in 4 * e *e
⇒  let e = (let e'= puissance 4 1 in e'*e') in 4 * e * e
⇒  let e = (let e'= 4 in e'*e') in 4 * e * e
⇒  let e = 4 * 4 in 4 * e * e
⇒  let e = 16 in 4 * e * e
⇒  4 * 16 * 16
⇒  1024
```

Ensemble de Cantor

Définir une fonction Cantor qui étant donné un entier `n` décide si, dans l'écriture en base 3 (écriture qui utilise les chiffres 0, 1 et 2), `n` s'écrit en utilisant uniquement des 0 et des 1

Ensemble de Cantor

Définir une fonction Cantor qui étant donné un entier `n` décide si, dans l'écriture en base 3 (écriture qui utilise les chiffres 0, 1 et 2), `n` s'écrit en utilisant uniquement des 0 et des 1

```
let rec cantor n =
 match n with
 | 0 -> true
 | _ ->
   if n mod 3 = 2 then false
   else cantor (n / 3)
;;
```

Ensemble de Cantor

Définir une fonction Cantor qui étant donné un entier `n` décide si, dans l'écriture en base 3 (écriture qui utilise les chiffres 0, 1 et 2), `n` s'écrit en utilisant uniquement des 0 et des 1

```
let rec cantor n =
 match n with
  | 0 -> true
  | _ ->
    if n mod 3 = 2 then false
    else cantor (n / 3)
;;
```

Ou plus simplement

```
let rec cantor n =
  (n=0) || (n mod 3 <>2 && cantor (n/3))
;;
```

Récursivité double

Récursivité double : plusieurs appels récursifs peuvent apparaître dans la définition

Récursivité double

Récursivité double : plusieurs appels récursifs peuvent apparaître dans la définition

```
let rec fibonacci n =
 match n with
  | 0 -> 0
  | 1 -> 1
  | _ -> fibonacci (n-1) + fibonacci (n-2)
;;
```

Récursivité mutuelle (1/2)

Récursivité mutuelle : une fonction `f` peut faire référence à une fonction `g` qui elle-même fait référence à `f`

Dans un fichier `pair.ml` :

```
let rec pair n = (n = 0) || impair (n-1) ;;

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

Récursivité mutuelle (1/2)

Récursivité mutuelle : une fonction `f` peut faire référence à une fonction `g` qui elle-même fait référence à `f`

Dans un fichier `pair.ml` :

```
let rec pair n = (n = 0) || impair (n-1) ;;

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

compilation

```
> ocamlc -o pair pair.ml
File "pair.ml", line 1, characters 28-34:
Unbound value impair
```

Récursivité mutuelle (2/2)

Pour définir deux fonctions `f` et `g` mutuellement récursives on utilise la construction let rec f = e1 and g = e2

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

Principes d'une définition récursive

L'analyse par cas dans une définition récursive permet de regrouper

- les cas de base (non récursifs)
- les cas récursifs qui renvoient à la définition en cours

La définition d'une fonction récursive revient toujours à identifier ces différents cas puis à vérifier que :

Principes d'une définition récursive

L'analyse par cas dans une définition récursive permet de regrouper

- les cas de base (non récursifs)
- les cas récursifs qui renvoient à la définition en cours

La définition d'une fonction récursive revient toujours à identifier ces différents cas puis à vérifier que :

1. les cas récursifs finissent par renvoyer aux cas de base (afin de garantir la terminaison de la fonction)

Principes d'une définition récursive

L'analyse par cas dans une définition récursive permet de regrouper

- les cas de base (non récursifs)
- les cas récursifs qui renvoient à la définition en cours

La définition d'une fonction récursive revient toujours à identifier ces différents cas puis à vérifier que :

1. les cas récursifs finissent par renvoyer aux cas de base (afin de garantir la terminaison de la fonction)
2. tous les cas sont couverts dans la définition

Principes d’une définition récursive

L’analyse par cas dans une définition récursive permet de regrouper

- les cas de base (non récursifs)
- les cas récursifs qui renvoient à la définition en cours

La définition d’une fonction récursive revient toujours à identifier ces différents cas puis à vérifier que :

1. les cas récursifs finissent par renvoyer aux cas de base (afin de garantir la terminaison de la fonction)
2. tous les cas sont couverts dans la définition
3. on ne risque pas de boucler sur des cas récursifs, en s’assurant par exemple que chaque appel récursif renvoie à un cas *plus simple*

Exemples de mauvaises définitions récursives

Problème de terminaison : les cas récursifs ne renvoient pas aux cas de base

```
let rec fibonacci n =
 match n with
  | 0 -> 0
  | _ -> fibonacci (n-1) + fibonacci (n-2)
;;
```

Exemples de mauvaises définitions récursives

Problème de terminaison : les cas récursifs ne renvoient pas aux cas de base

```
let rec fibonacci n =
 match n with
  | 0 -> 0
  | _ -> fibonacci (n-1) + fibonacci (n-2)
;;
```

Problème de couverture des cas possibles

```
let mystere x =
  let rec mystere x n =
    match n with
     | 0 -> 1
     | 1 -> x * mystere (x/2) ((x mod 2)+n)
  in
  if x<=0 then 1 else mystere x 1  ;;
```

Récursion infinie

Il arrive cependant parfois de vouloir écrire de ”mauvaises” définitions récursives

Soit la fonction `zero` qui recherche (éventuellement indéfiniment) une valeur pour laquelle une fonction `f` s’annule

```
let zero f =
  let rec recherche i =
   if f i = 0 then i else recherche (i+1)
  in
  recherche 0
;;
```

Récursion terminale

Exemple introductif

Dans un fichier `somme.ml` :

```
let rec somme n =
  match n with
   | 0. -> 0.
   | _ -> n +. somme (n -. 1.)  ;;

print_float (somme 90000.)  ;;
```

Exemple introductif

Dans un fichier `somme.ml` :

```
let rec somme n =
  match n with
   | 0. -> 0.
   | _ -> n +. somme (n -. 1.)  ;;

print_float (somme 90000.)  ;;
```

compilation

```
ocamlc -o somme somme.ml
```

Exemple introductif

Dans un fichier `somme.ml` :

```
let rec somme n =
  match n with
   | 0. -> 0.
   | _ -> n +. somme (n -. 1.)  ;;

print_float (somme 90000.)  ;;
```

compilation

```
ocamlc -o somme somme.ml
```

exécution

```
./somme
Fatal error: exception Stack_overflow
```

Appels en attente

L'exécution du programme précédent devrait correspondre au processus d'évaluation suivant

```
     somme 90000.
⇒    90000. +. somme 89999.
⇒    90000. +. 89999. +. somme 89998.
⇒    ...
⇒    90000. +. 89999. +. 4049865001.
⇒    90000. +. 4049955000.
⇒    4050045000.
```

L'appel à `somme n` est en attente du résultat de l'appel à `somme (n-1)`

Appels en attente

L'exécution du programme précédent devrait correspondre au processus d'évaluation suivant

```
     somme 90000.
⇒   90000. +. somme 89999.
⇒   90000. +. 89999. +. somme 89998.
⇒   ...
⇒   90000. +. 89999. +. 4049865001.
⇒   90000. +. 4049955000.
⇒   4050045000.
```

L'appel à `somme n` est en attente du résultat de l'appel à `somme (n-1)`

Malheureusement, quand le nombre d'appels en attente est trop grand le programme s'arrête !

Pile d'appels

Pour exécuter les appels de fonctions, quelque soit le langage et le compilateur, on utilise une pile d'appel, dans laquelle on stocke (entre autre) les valeurs des arguments et l'adresse de retour de l'appel

La taille de cette pile croît donc en fonction du nombre d'appels en attente et elle "déborde" quand il y a trop d'appels en attente (message *Stack overflow*)

Exemple. sous un Linux standard la taille de la pile est fixée à 8Ko

Appels terminaux

Dans une fonction `f`, un appel à une fonction `g` est terminal si le résultat de cet appel est le résultat de `f`

Exemples :

```
let f x = g x ;;
```

Appels terminaux

Dans une fonction `f`, un appel à une fonction `g` est terminal si le résultat de cet appel est le résultat de `f`

Exemples :

```
let f x = g x ;;

let f x = if ... then  g x  else ...  ;;
```

Appels terminaux

Dans une fonction `f`, un appel à une fonction `g` est terminal si le résultat de cet appel est le résultat de `f`

Exemples :

```
let f x = g x ;;

let f x = if ... then  g x  else ...  ;;

let f x = if ... then ... else g x ;;
```

Appels terminaux

Dans une fonction `f`, un appel à une fonction `g` est terminal si le résultat de cet appel est le résultat de `f`

Exemples :

```
let f x = g x ;;

let f x = if ... then  g x  else ...  ;;

let f x = if ... then ... else g x ;;

let f x = let y = ... in g y ;;
```

Appels terminaux

Dans une fonction `f`, un appel à une fonction `g` est terminal si le résultat de cet appel est le résultat de `f`

Exemples :

```
let f x = g x ;;

let f x = if ... then  g x  else ...  ;;

let f x = if ... then ... else g x ;;

let f x = let y = ... in g y ;;

let f x = match x with ... | p -> g x | ...    ;;
```

Appels terminaux

Dans une fonction `f`, un appel à une fonction `g` est terminal si le résultat de cet appel est le résultat de `f`

Exemples :

```
let f x = g x ;;

let f x = if ... then  g x  else ...  ;;

let f x = if ... then ... else g x ;;

let f x = let y = ... in g y ;;

let f x = match x with ... | p -> g x | ...    ;;
```

Une fonction est récursive terminale si ses appels récursifs sont tous terminaux

Exécution des appels terminaux

Voici une version de `somme` récursive terminale

```
let rec somme_term acc n =
   match n with
    | 0. -> acc
    | _ -> somme_term (n +. acc) (n -. 1.)
;;
let somme n = somme_term 0. n ;;
```

Exécution des appels terminaux

Voici une version de `somme` récursive terminale

```
let rec somme_term acc n =
   match n with
    | 0. -> acc
    | _ -> somme_term (n +. acc) (n -. 1.)
;;
let somme n = somme_term 0. n ;;
```

- Avec cette version plus de débordement de pile
- Le compilateur `ocamlc` traite de manière spéciale les appels terminaux : il remplace dans la pile d'appel la place occupée par `somme_term acc n` par l'appel `somme_term (n +. acc) ( n -. 1.)`

Récursion efficace

Programmer avec des accumulateurs

- Principe analogue à l'ajout de variables auxiliaires en programmation impérative
- Ajout de fonctions auxiliaires avec des paramètres supplémentaires, appelés accumulateurs

Récursion efficace

Programmer avec des accumulateurs

- Principe analogue à l'ajout de variables auxiliaires en programmation impérative
- Ajout de fonctions auxiliaires avec des paramètres supplémentaires, appelés accumulateurs

Autre exemple, la fonction factorielle en version récursive terminale

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