Programmation Fonctionnelle
Cours 04

Michele Pagani

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

pagani@pps.univ-paris-diderot.fr

16 Octobre 2014

Fonctions

Les fonctions sont des valeurs de 1ère classe

Les fonctions peuvent être utilisées sans restriction, par exemple:

- liées à un identificateur (ou variable):

```
# let sum = fun x y -> x+y;;
val sum : int -> int -> int = <fun>
```

- passées en argument à une autre fonction:

```
# List.fold_left sum 0 [1;3;4];;
- : int = 8
```

- retourées comme résultat d'une fonction:

```
# List.fold_left sum 0;;
- : int list -> int = <fun>
```

- faire partie des structures de données:

```
# [sum; (fun x y -> x-y); (fun x y -> x*y)];;
- : (int -> int -> int) list = [<fun>; <fun>; <fun>]
```

Les fonctions sont des valeurs de 1ère classe

Les fonctions peuvent être utilisées sans restriction, par exemple:

- liées à un identificateur (ou variable):

```
# let sum = fun x y -> x+y;;
val sum : int -> int -> int = <fun>
```

- passées en argument à une autre fonction:

```
# List.fold_left sum 0 [1;3;4];;
- : int = 8
```

- retourées comme résultat d'une fonction:

```
# List.fold_left sum 0;;
- : int list -> int = <fun>
```

- faire partie des structures de données:

```
# [sum; (fun x y -> x -y); (fun x y -> x*y)];;
- : (int -> int -> int) list = [<fun>; <fun>; <fun>]
```

Les fonctions sont des valeurs de 1ère classe

Les fonctions peuvent être utilisées sans restriction, par exemple:

- liées à un identificateur (ou variable):

```
# let sum = fun x y -> x+y;;
val sum : int -> int -> int = <fun>
```

- passées en argument à une autre fonction:

```
# List.fold_left sum 0 [1;3;4];;
- : int = 8
```

- retourées comme résultat d'une fonction:

```
# List.fold_left sum 0;;
- : int list -> int = <fun>
```

- faire partie des structures de données:

```
# [sum; (fun x y -> x -y); (fun x y -> x*y)];;
- : (int -> int -> int) list = [<fun>; <fun>; <fun>]
```

Les fonctions sont des valeurs de 1ère classe

Les fonctions peuvent être utilisées sans restriction, par exemple:

- liées à un identificateur (ou variable):

```
# let sum = fun x y -> x+y;;
val sum : int -> int -> int = <fun>
```

- passées en argument à une autre fonction:

```
# List.fold_left sum 0 [1;3;4];;
- : int = 8
```

- retourées comme résultat d'une fonction:

```
# List.fold_left sum 0;;
- : int list -> int = <fun>
```

- faire partie des structures de données:

```
# [sum; (fun x y -> x -y); (fun x y -> x*y)];;
- : (int -> int -> int) list = [<fun>; <fun>; <fun>]
```

Fonctions à un argument

function id –> exp

- C'est une fonction qui prend un argument, dénoté par l'identificateur *id*, et qui renvoie comme résultat la valeur de l'expression *exp*.
- La portée de *id* est *exp*
- Le type de cette valeur est $t_1 -> t_2$, où t_1 est le type de l'argument et t_2 est le type du résultat de la fonction.
- Le type est déterminé utilisant des informations dans l'expression *exp*, et peut être polymorphe. (⇒ voir cours 3)
- en général, on peut avoir un filtrage par motif:

```
function
  | motif_1 -> exp_1
  ...
  | motif_n -> exp_n
```

Fonctions à un argument

function id –> exp

- C'est une fonction qui prend un argument, dénoté par l'identificateur *id*, et qui renvoie comme résultat la valeur de l'expression *exp*.
- La portée de *id* est *exp*
- Le type de cette valeur est $t_1 - > t_2$, où t_1 est le type de l'argument et t_2 est le type du résultat de la fonction.
- Le type est déterminé utilisant des informations dans l'expression *exp*, et peut être polymorphe. (⇒ voir cours 3)
- en général, on peut avoir un filtrage par motif:

```
function
  | motif_1 -> exp_1
  ...
  | motif_n -> exp_n
```

Fonctions à un argument

function id –> exp

- C'est une fonction qui prend un argument, dénoté par l'identificateur *id*, et qui renvoie comme résultat la valeur de l'expression *exp*.
- La portée de *id* est *exp*
- Le type de cette valeur est $t_1 -> t_2$, où t_1 est le type de l'argument et t_2 est le type du résultat de la fonction.
- Le type est déterminé utilisant des informations dans l'expression *exp*, et peut être polymorphe. (⇒ voir cours 3)
- en général, on peut avoir un filtrage par motif:

```
function
  | motif_1 -> exp_1
  ...
  | motif_n -> exp_n
```

Fonctions à un argument

function id –> exp

- C'est une fonction qui prend un argument, dénoté par l'identificateur *id*, et qui renvoie comme résultat la valeur de l'expression *exp*.
- La portée de *id* est *exp*
- Le type de cette valeur est $t_1 - > t_2$, où t_1 est le type de l'argument et t_2 est le type du résultat de la fonction.
- Le type est déterminé utilisant des informations dans l'expression *exp*, et peut être polymorphe. (⇒ voir cours 3)
- en général, on peut avoir un filtrage par motif:

```
function
  | motif_1 -> exp_1
  ...
  | motif_n -> exp_n
```

Exemples

```
# function x -> x+1;;
- : int -> int = <fun>

# function y -> [[y+2; y+3]; [y;y*y]];;
- : int -> int list list = <fun>

# (function x -> x+3) 5;;
- : int = 8

# (function x -> x*x)(function x -> x+1) 3;; (*erreur*)
  ^^^^^^^^^^^^^^^^^^
Error: This function has type int -> int
       It is applied to too many arguments;
       maybe you forgot a ';'.

# (function x -> x*x) ((function x -> x+1)3);; (*OK*)
- : int = 16
```

Déclaration des fonctions

let f x = exp raccourci pour **let f = function x -> exp**

```
# let suc = function x -> x+1;;
val suc : int -> int = <fun>

# suc 5;;
- : suc = 6

# let suc x = x+1 ;;        (*equivalent au precedent*)
val suc : int -> int = <fun>

let rec fib = function
   | 0 -> 0
   | 1 -> 1
   | x -> (fib (x-1))*(fib (x-2));;
val fib : int -> int = <fun>

# let rec fib x = match x with
   | 0 | 1 -> x
   | x -> (fib (x-1))*(fib (x-2));;(*equivalent au precedent*)
val fib : int -> int = <fun>
```

Qu'est-ce qu'il se passe lors d'une application ?

(exp_f)(exp_a)

Le typage nous garantie (si on ignore la polymorphie):

- exp_f est d'un type fonctionnel $t_a \mapsto t_r$
- exp_a est du type t_a

L'évaluation procède par étapes:

1. évaluation de exp_f — **function** x −> exp_c
   (pas d'évaluation dans le corps de **function**)
2. évaluation de exp_a — val_a
3. substitution toute occurrence de x par val_a — exp_c[x=val_a]
4. évaluation de l'expression ainsi obtenue — val_r

Qu'est-ce qu'il se passe lors d'une application ?

(exp_f)(exp_a)

Le typage nous garantie (si on ignore la polymorphie):

- exp_f est d'un type fonctionnel $t_a \mapsto t_r$
- exp_a est du type t_a

L'évaluation procède par étapes:

1. évaluation de exp_f — **function** x −> exp_c
   (pas d'évaluation dans le corps de **function**)
2. évaluation de exp_a — val_a
3. substitution toute occurrence de x par val_a — exp_c[x=val_a]
4. évaluation de l'expression ainsi obtenue — val_r

Qu'est-ce qu'il se passe lors d'une application ?

(exp_f)(exp_a)

Le typage nous garantie (si on ignore la polymorphie):

- exp_f est d'un type fonctionnel $t_a \mapsto t_r$
- exp_a est du type t_a

L'évaluation procède par étapes:

1. évaluation de exp_f — **function** x −> exp_c
   (pas d'évaluation dans le corps de **function**)
2. évaluation de exp_a — val_a
3. substitution toute occurrence de x par val_a — exp_c[x=val_a]
4. évaluation de l'expression ainsi obtenue — val_r

Qu'est-ce qu'il se passe lors d'une application ?

(exp_f)(exp_a)

Le typage nous garantie (si on ignore la polymorphie):

- exp_f est d'un type fonctionnel $t_a \mapsto t_r$
- exp_a est du type t_a

L'évaluation procède par étapes:

1. évaluation de exp_f — **function** x −> exp_c
   (pas d'évaluation dans le corps de **function**)
2. évaluation de exp_a — val_a
3. substitution toute occurrence de x par val_a — exp_c[x=val_a]
4. évaluation de l'expression ainsi obtenue — val_r

Qu'est-ce qu'il se passe lors d'une application ?

(exp_f)(exp_a)

Le typage nous garantie (si on ignore la polymorphie):

- exp_f est d'un type fonctionnel $t_a \mapsto t_r$
- exp_a est du type t_a

L'évaluation procède par étapes:

1. évaluation de exp_f — **function** x −> exp_c
   (pas d'évaluation dans le corps de **function**)
2. évaluation de exp_a — val_a
3. substitution toute occurrence de x par val_a — exp_c[x=val_a]
4. évaluation de l'expression ainsi obtenue — val_r

Qu'est-ce qu'il se passe lors d'une application ?

(**function** x −> 2∗x−x)(1+2)

(pas d'éval au dessous de **function** −> x)

Qu'est-ce qu'il se passe lors d'une application ?

(**function** x −> 2∗x−x)(1+2)

(pas d'éval au dessous de **function** −> x)

⇓ ❷ eval de argument

(**function** x −> 2∗x−x) 3

Qu'est-ce qu'il se passe lors d'une application ?

(**function** x −> 2∗x−x)(1+2)

(pas d'éval au dessous de **function** −> x)

⇓ ❷ eval de argument

(**function** x −> 2∗x−x) 3

⇓ ❸ substitution du parametre

2∗3−3

Qu'est-ce qu'il se passe lors d'une application ?

(function x −> 2∗x−x)(1+2)

(pas d'éval au dessous de **function** −> x)

⇓ ❷ eval de argument

(function x −> 2∗x−x) 3

⇓ ❸ substitution du parametre

2∗3−3

⇓ ❹ eval du résultat

3

Qu'est-ce qu'il se passe lors d'une application ?

snd ("Hello", (**function** x −> 2∗x−x)) (1+2)

⇓ ❶ eval de fonction

(**function** x −> 2∗x−x)(1+2)

(pas d'éval au dessous de **function** −> x)

⇓ ❷ eval de argument

(**function** x −> 2∗x−x) 3

⇓ ❸ substitution du parametre

2∗3−3

⇓ ❹ eval du résultat

3

Pas d'evaluation au dessous de **function** y ->

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

# let g = f 0;;    (*pas d'erreur*)
val g : int -> int = <fun>

# let z = g 5;;  (*Erreur: division par zero*)
Exception: Division_by_zero.
```

Fonctions à plusieurs arguments

fun id_1 ... id_n −> exp
est un raccourci pour
function id_1 −> ... **function** id_n −> exp

- c'est une fonction qui prend une suite de n arguments
- application partielle:
  (**function** id_1 −> ... **function** id_n −> exp) exp1
  est une fonction qui prend une suite de $n-1$ arguments
- filtrage par motif:
  on peut utiliser **match** id_i **with** pour déclarer l'argument id_i dont depend le filtrage.
- déclaration:
  let f id_1 ... id_n = exp est un raccourci pour
  let f = **fun** id_1 ... id_n −> exp

Fonctions à plusieurs arguments

fun id_1 ... id_n −> exp
est un raccourci pour
function id_1 −> ... **function** id_n −> exp

- c'est une fonction qui prend une suite de n arguments
- application partielle:

  (**function** id_1 −> ... **function** id_n −> exp) exp1

  est une fonction qui prend une suite de $n-1$ arguments
- filtrage par motif:
  on peut utiliser **match** id_i **with** pour déclarer l'argument id_i dont depend le filtrage.
- déclaration:
  let f id_1 ... id_n = exp est un raccourci pour
  let f = **fun** id_1 ... id_n −> exp

Fonctions à plusieurs arguments

fun id_1 ... id_n −> exp
est un raccourci pour
function id_1 −> ... **function** id_n −> exp

- c'est une fonction qui prend une suite de n arguments
- application partielle:

  (**function** id_1 −> ... **function** id_n −> exp) exp1

  est une fonction qui prend une suite de $n-1$ arguments
- filtrage par motif:
  on peut utiliser **match** id_i **with** pour déclarer l'argument id_i dont depend le filtrage.
- déclaration:
  let f id_1 ... id_n = exp est un raccourci pour
  let f = **fun** id_1 ... id_n −> exp

Fonctions à plusieurs arguments

fun id_1 ... id_n −> exp
est un raccourci pour
function id_1 −> ... **function** id_n −> exp

- c'est une fonction qui prend une suite de n arguments
- application partielle:

  (**function** id_1 −> ... **function** id_n −> exp) exp1

  est une fonction qui prend une suite de $n-1$ arguments
- filtrage par motif:
  on peut utiliser **match** id_i **with** pour déclarer l'argument id_i dont depend le filtrage.
- déclaration:
  let f id_1 ... id_n = exp est un raccourci pour
  let f = **fun** id_1 ... id_n −> exp

Plusieurs arguments (exemples)

```
# let soit = function x -> function y -> match x with
  | true -> (match y with
               | true -> false
               | false -> true)
  | false -> y;;
val soit : bool -> bool -> bool = <fun>

# let soit = fun x y -> match x with
  | true -> (match y with
               | true -> false
               | false -> true)
  | false -> y;;  (*equivalent au precedent*)
val soit : bool -> bool -> bool = <fun>

# let soit x y = match x with
  | true -> (match y with
               | true -> false
               | false -> true)
  | false -> y;;  (*equivalent au precedent*)
val soit : bool -> bool -> bool = <fun>
```

Plusieurs arguments (exemples)

```
# soit true true;;
- : bool = false

# soit true false;;
- : bool = true

# let neg = soit true;;    (*application partielle*)
val neg : bool -> bool = <fun>

# neg true;;
- : bool = false

# neg false;;
- : bool = true
```

Plusieurs arguments (exemples)

```
# let f x y z = x +2*y+3*z;;
val f : int -> int -> int -> int = <fun>

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

# let h = g 3;;
val h : int -> int = <fun>
```

- Combien ça vaut h 4 ?

```
# h 4;;
- : int = 20
```

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

- Combien ça vaut (f 2)((f 3)4) ?

```
# (f 2)((f 3)4);;
- : int = 24
```

Récurrence terminale

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
   0 -> []
   | n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>

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

# mklist 1000000;;
Stack overflow during evaluation (looping recursion?).
```

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
   0 -> []
   | n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

mklist 3 $\Rightarrow$ 3::(mklist 2) $\Rightarrow$ 3::2::(mklist 1) $\Rightarrow$ 3::2::1(mklist 0)
$\Rightarrow$ 3::2::1::[] $\Rightarrow$ 3::2::[1] $\Rightarrow$ 3::[2;1] $\Rightarrow$ [3;2;1]

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la pile d'exécution (angl. call stack)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Consequence: si trop d'appels imbriquées, on peut épuiser l'espace disponible pour la pile !

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
   0 -> []
   | n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

mklist 3 $\Rightarrow$ 3::(mklist 2) $\Rightarrow$ 3::2::(mklist 1) $\Rightarrow$ 3::2::1(mklist 0)
$\Rightarrow$ 3::2::1::[] $\Rightarrow$ 3::2::[1] $\Rightarrow$ 3::[2;1] $\Rightarrow$ [3;2;1]

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la pile d'exécution (angl. call stack)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Consequence: si trop d'appels imbriquées, on peut épuiser l'espace disponible pour la pile !

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
   0 -> []
   | n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

mklist 3 $\Rightarrow$ 3::(mklist 2) $\Rightarrow$ 3::2::(mklist 1) $\Rightarrow$ 3::2::1(mklist 0)

$\Rightarrow$ 3::2::1::[] $\Rightarrow$ 3::2::[1] $\Rightarrow$ 3::[2;1] $\Rightarrow$ [3;2;1]

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la pile d'exécution (angl. call stack)

Exemple: `DrawSquare` appèle `DrawLine`

Merci, Wikipedia

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
   0 -> []
   | n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

mklist 3 $\Rightarrow$ 3::(mklist 2) $\Rightarrow$ 3::2::(mklist 1) $\Rightarrow$ 3::2::1(mklist 0)

$\Rightarrow$ 3::2::1::[] $\Rightarrow$ 3::2::[1] $\Rightarrow$ 3::[2;1] $\Rightarrow$ [3;2;1]

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la pile d'exécution (angl. call stack)

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
   0 -> []
   | n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

mklist 3 $\Rightarrow$ 3::(mklist 2) $\Rightarrow$ 3::2::(mklist 1) $\Rightarrow$ 3::2::1(mklist 0)
$\Rightarrow$ 3::2::1::[] $\Rightarrow$ 3::2::[1] $\Rightarrow$ 3::[2;1] $\Rightarrow$ [3;2;1]

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la pile d'exécution (angl. call stack)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Consequence: si trop d'appels imbriquées, on peut épuiser l'espace disponible pour la pile !

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
   0 -> []
   | n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

mklist 3 $\Rightarrow$ 3::(mklist 2) $\Rightarrow$ 3::2::(mklist 1) $\Rightarrow$ 3::2::1(mklist 0)
$\Rightarrow$ 3::2::1::[] $\Rightarrow$ 3::2::[1] $\Rightarrow$ 3::[2;1] $\Rightarrow$ [3;2;1]

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la pile d'exécution (angl. call stack)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Consequence: si trop d'appels imbriquées, on peut épuiser l'espace disponible pour la pile !

Récurrence terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
  - appel terminal: une fonction f appèl une fonction g, mais le résultat envoyé par g est envoyé tout de suite par f,
  - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
  - l'espace sur la pile d'exécution peut être libérée avant de lancer g!
- Récurrence terminale (angl.: tail recursion): fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrence terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
  - appel terminal: une fonction f appèl une fonction g, mais le résultat envoyé par g est envoyé tout de suite par f,
  - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
  - l'espace sur la pile d'exécution peut être libérée avant de lancer g!
- Récurrence terminale (angl.: tail recursion): fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrence terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
  - appel terminal: une fonction f appèl une fonction g, mais le résultat envoyé par g est envoyé tout de suite par f,
  - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
  - l'espace sur la pile d'exécution peut être libérée avant de lancer g!
- Récurrence terminale (angl.: tail recursion): fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrence terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
  - appel terminal: une fonction f appèl une fonction g, mais le résultat envoyé par g est envoyé tout de suite par f,
  - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
  - l'espace sur la pile d'exécution peut être libérée avant de lancer g!
- Récurrence terminale (angl.: tail recursion): fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrence terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
  - appel terminal: une fonction f appèl une fonction g, mais le résultat envoyé par g est envoyé tout de suite par f,
  - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
  - l'espace sur la pile d'exécution peut être libérée avant de lancer g!
- Récurrence terminale (angl.: tail recursion): fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrence terminale (exemples)

- Ceci n'est pas une récurrence terminale:

```
# let rec mklist n = match n with
  0 -> []
  | n -> n::(mklist (n-1));;
```

  - on exécute un calcul (n:: ...) avant de renvoyer le résultat de l'appel récursif (mklist (n−1))

- Voici une variante terminale:

```
# let mklist n =
    let rec mkaux n l = match n with
      | 0 -> l
      | n -> mkaux (n-1) (n::l)
    in List.rev (mkaux n []);;
val mklist : int -> int list = <fun>
```

  - aucun calcul après l'appel récursif (mkaux (n−1) (n::l))
  - on utilise une fonction auxiliaire (mkaux) avec un argument accumulateur (l)
  - attention à renverser l'ordre des elements de la liste

Récurrence terminale (exemples)

- Ceci n'est pas une récurrence terminale:

```
# let rec mklist n = match n with
  0 -> []
  | n -> n::(mklist (n-1));;
```

  - on exécute un calcul (n:: ...) avant de renvoyer le résultat de l'appel récursif (mklist (n−1))

- Voici une variante terminale:

```
# let mklist n =
    let rec mkaux n l = match n with
      | 0 -> l
      | n -> mkaux (n-1) (n::l)
   in List.rev (mkaux n []);;
val mklist : int -> int list = <fun>
```

  - aucun calcul après l'appel récursif (mkaux (n−1) (n::l))
  - on utilise une fonction auxiliaire (mkaux) avec un argument accumulateur (l)
  - attention à renverser l'ordre des elements de la liste

Récurrence terminale (exemples)

- Ceci n'est pas une récurrence terminale:

```
# let rec mklist n = match n with
  0 -> []
  | n -> n::(mklist (n-1));;
```

  - on exécute un calcul (n:: ...) avant de renvoyer le résultat de l'appel récursif (mklist (n−1))

- Voici une variante terminale:

```
# let mklist n =
    let rec mkaux n l = match n with
      | 0 -> l
      | n -> mkaux (n-1) (n::l)
   in List.rev (mkaux n []);;
val mklist : int -> int list = <fun>
```

  - aucun calcul après l'appel récursif (mkaux (n−1) (n::l))
  - on utilise une fonction auxiliaire (mkaux) avec un argument accumulateur (l)
  - attention à renverser l'ordre des elements de la liste

Récurrence terminale (exemples)

```
# let mklist n =
    let rec mkaux n l = match n with
      | 0 -> l
      | n -> mkaux (n-1) (n::l)
   in List.rev (mkaux n []);;
val mklist : int -> int list = <fun>
```

$$\begin{aligned}
&\texttt{mklist } 3 \Rightarrow \texttt{List.rev(mkaux 3 [])} \Rightarrow \texttt{List.rev(mkaux 2 (3::[]))} \\
&\quad \Rightarrow \texttt{List.rev(mkaux 2 (3::[]))} \Rightarrow \texttt{List.rev(mkaux 2 [3])} \\
&\Rightarrow \texttt{List.rev(mkaux 1 (2::[3]))} \Rightarrow \texttt{List.rev(mkaux 1 (2::[3]))} \\
&\Rightarrow \texttt{List.rev(mkaux 1 [2;3])} \Rightarrow \texttt{List.rev(mkaux 0 (1::[2;3]))} \\
&\Rightarrow \texttt{List.rev(mkaux 0 (1::[2;3]))} \Rightarrow \texttt{List.rev(mkaux 0 [1;2;3])} \\
&\quad \Rightarrow \texttt{List.rev [1;2;3]} \Rightarrow \cdots \Rightarrow [3;2;1]
\end{aligned}$$

- à chaque appel récursif, la quantité d'informations d'environnement à stocker est constante
- consequence...

Récurrence terminale (exemples)

```
# let mklist n =
    let rec mkaux n l = match n with
      | 0 -> l
      | n -> mkaux (n-1) (n::l)
   in List.rev (mkaux n []);;
val mklist : int -> int list = <fun>
```

mklist 3 $\Rightarrow$ List.rev(mkaux 3 []) $\Rightarrow$ List.rev(mkaux 2 (3::[]))
$\Rightarrow$ List.rev(mkaux 2 (3::[])) $\Rightarrow$ List.rev(mkaux 2 [3])
$\Rightarrow$ List.rev(mkaux 1 (2::[3])) $\Rightarrow$ List.rev(mkaux 1 (2::[3]))
$\Rightarrow$ List.rev(mkaux 1 [2;3]) $\Rightarrow$ List.rev(mkaux 0 (1::[2;3]))
$\Rightarrow$ List.rev(mkaux 0 (1::[2;3])) $\Rightarrow$ List.rev(mkaux 0 [1;2;3])
$\Rightarrow$ List.rev [1;2;3] $\Rightarrow \cdots \Rightarrow$ [3; 2; 1]

- à chaque appel récursif, la quantité d'informations d'environnement à stocker est constante
- consequence...

Récurrence terminale (exemples)

```
# mklist 100000;;
- : int list =
[100000; 99999; 99998; 99997; 99996; 99995; 99994; 99993; 99992; 99991;
 99990; 99989; 99988; 99987; 99986; 99985; 99984; 99983; 99982; 99981; 99980;
 99979; 99978; 99977; 99976; 99975; 99974; 99973; 99972; 99971; 99970; 99969;
 99968; 99967; 99966; 99965; 99964; 99963; 99962; 99961; 99960; 99959; 99958;
 99957; 99956; 99955; 99954; 99953; 99952; 99951; 99950; 99949; 99948; 99947;
 99946; 99945; 99944; 99943; 99942; 99941; 99940; 99939; 99938; 99937; 99936;
 99935; 99934; 99933; 99932; 99931; 99930; 99929; 99928; 99927; 99926; 99925;
 99924; 99923; 99922; 99921; 99920; 99919; 99918; 99917; 99916; 99915; 99914;
 99913; 99912; 99911; 99910; 99909; 99908; 99907; 99906; 99905; 99904; 99903;
 99902; 99901; 99900; 99899; 99898; 99897; 99896; 99895; 99894; 99893; 99892;
 99891; 99890; 99889; 99888; 99887; 99886; 99885; 99884; 99883; 99882; 99881;
 99880; 99879; 99878; 99877; 99876; 99875; 99874; 99873; 99872; 99871; 99870;
 99869; 99868; 99867; 99866; 99865; 99864; 99863; 99862; 99861; 99860; 99859;
 99858; 99857; 99856; 99855; 99854; 99853; 99852; 99851; 99850; 99849; 99848;
 99847; 99846; 99845; 99844; 99843; 99842; 99841; 99840; 99839; 99838; 99837;
 99836; 99835; 99834; 99833; 99832; 99831; 99830; 99829; 99828; 99827; 99826;
 99825; 99824; 99823; 99822; 99821; 99820; 99819; 99818; 99817; 99816; 99815;
 99814; 99813; 99812; 99811; 99810; 99809; 99808; 99807; 99806; 99805; 99804;
 99803; 99802; 99801; 99800; 99799; 99798; 99797; 99796; 99795; 99794; 99793;
 99792; 99791; 99790; 99789; 99788; 99787; 99786; 99785; 99784; 99783; 99782;
 99781; 99780; 99779; 99778; 99777; 99776; 99775; 99774; 99773; 99772; 99771;
 99770; 99769; 99768; 99767; 99766; 99765; 99764; 99763; 99762; 99761; 99760;
 99759; 99758; 99757; 99756; 99755; 99754; 99753; 99752; 99751; 99750; 99749;
 99748; 99747; 99746; 99745; 99744; 99743; 99742; 99741; 99740; 99739; 99738;
 99737; 99736; 99735; 99734; 99733; 99732; 99731; 99730; 99729; 99728; 99727;
 99726; 99725; 99724; 99723; 99722; 99721; 99720; 99719; 99718; 99717; 99716;
 99715; 99714; 99713; 99712; 99711; 99710; 99709; 99708; 99707; 99706; 99705;
 99704; 99703; 99702; ...]
```

Doggy bag

- fonctions
  - valeurs de 1ère class
  - définition par filtrage par motif
- evaluation d'une application
  - substitution
  - évaluation paresseuse
- récurrence terminale
  - pile d'exécution
  - erreur de "stack overflow"
  - appel terminal d'une fonction

Exercices

Transformez les récurrences suivantes en récurrences terminales:

- la factorielle:

```
let rec fact n = match n with
  |0 -> 1
  |n -> n*(fact (n-1));;
val fact : int -> int = <fun>
```

- itération d'une fonction sur une liste:

```
# let rec fold_right f l b = match l with
  | [] -> b
  | t::r -> f t (fold_right f r b) ;;
val fold_right: ('a-> 'b-> 'b)-> 'a list -> 'b-> 'b=<fun>
```

- Fibonacci:

```
# let rec fib n = match n with
  | 0 | 1-> n
  | n -> fib(n-1)+fib(n-2);;
val fib : int -> int = <fun>
```

Évaluer l'e□cacité des deux variants à l'aide de la fonction `Sys.time ()` (voir module `Sys`).