Fonctions et Programmation Fonctionnelle

INF 441 : Programmation Avancée

Xavier Rival

24 avril 2017

Quelques éléments administratifs

Projets :

- **4 sujets** sur la page du cours, texte complet dans Moodle
- **à choisir maintenant**
  deadline pour les choix : vendredi 28 avril 2017

Café-LIX : réunion d'information sur la 3A et les PA d'info

- au **LIX**
- même modèle que la semaine dernière...
- **mercredi 26 avril, de 13h00 à ~~14h00~~13h30**
  ne pas oublier : l'amphi INF 441 démarre à **13h30** :-)
- a priori, il y aura au moins une autre séance

Pourquoi étudier les fonctions ?

La notion de **fonction** en programmation est bien connue...

En OCaml :

- **définition** : `fun x -> corps_de_la_fonction`
- **application** : `fonction argument`

Cela pourrait être la fin du cours... Est-ce aussi simple que cela ?

Tout d'abord les fonctions sont **l'un des principaux mécanismes d'abstraction** en programmation

Ensuite, il s'agit d'une notion **plus subtile** qu'il n'y paraît... À étudier :

- **stratégie d'évaluation** et efficacité
- **typage**
- **expressivité** des programmes fonctionnels

Fonctions et abstraction : paramétrage

Abstraction en programmation :

- paramétrisation
- masquage d'informations

Fonctions et paramétrisation

Par définition, **une fonction** comprend un **fragment de code** qui est **paramétré par ses arguments**...

Essentiellement, si vous avez la tentation de **copier-coller** du code, il faut **s'interroger sur l'opportunité de définir une fonction**.

Que gagne-t'on ?

- un code **plus compact** (moins de lignes à écrire / lire, moins de bugs)
- une meilleure **compréhension** des opérations répétées régulièrement avec à la clé des opportunités d'optimisation...

Fonctions et abstraction : masquage d'information

Abstraction en programmation :

- paramétrisation
- masquage d'informations

Fonctions et localité

Les **variables locales** et **algorithmes** qui apparaissent dans le corps d'une fonction **ne sont pas visibles de l'extérieur**

Autrement dit, la définition d'une fonction permet de **séparer** la conception du **corps** et du **code client**.

Que gagne-t'on ?

- **preuve et vérification** : les détails d'implémentation peuvent être ignorés lorsqu'on prouve le code client...
- **travail de programmation** : un programmeur écrit le corps, un autre le code client...

Outline

Questions liées à l'évaluation d'un appel de fonction

En **mathématiques**, une fonction est tout simplement **un graphe** :

- A ensemble de départ, B ensemble d'arrivée
- graphe fonctionnel $G \subseteq A \times B$

Dans le cas d'un langage de programmation, l'évaluation de `f arg` est **nettement plus compliquée**, en raison :

- des **effets de bords** (affichage, écritures en mémoire, erreurs) : dans quel ordre sont calculés la fonction, son argument, son corps...
- des **ressources** (mémoire, temps) : comment la mémoire est-elle gérée ? où est stocké la valeur d'un argument ? comment représenter des appels imbriqués ?

Pour comprendre tout cela, il faut regarder l'implémentation

Évaluation stricte

Définition : évaluation stricte

On parle **d'évaluation stricte** lorsque les paramètres d'une fonction **sont systématiquement évalués avant le corps de celle-ci**.

- **avantage** : sémantique assez simple
  attention : dans le cas où il y a plusieurs arguments, se pose la question de leur ordre relatif d'évaluation
- **non terminaison** :
  si l'évaluation d'un argument boucle, l'appel boucle
- **paramètre inutilisé** dans le corps de la fonction :
  évalué quand même (le coût peut paraître inutile, mais au moins on sait quels effets de bords attendre)
- **exemples** : Java, OCaml, C, ...

Évaluation non-stricte

Lorsque les arguments ne sont pas forcément évalués avant le corps de la fonction, on parle **d'évaluation non stricte**.

Appel par nom (peu utilisé à présent) :

- pour évaluer `(fun x -> corps)arg`, on **substitue** `arg` à **chaque occurrence** de `x` dans `corps`
- si `arg` ne termine pas, mais n'est pas utilisé, l'évaluation **termine quand même** !
- par contre, si `x` est utilisé plusieurs fois, il sera aussi **réévalué plusieurs fois** !

Évaluation paresseuse :

- pour évaluer `(fun x -> corps)arg`, on **évalue** `arg` **seulement la première fois** que l'on lit `x` en évaluant `corps`
- **minimise le coût** et les cas de **non terminaison** induits par les arguments
- **utilisé en Haskell**, langage fonctionnel pur (sans effets de bord)

Appels de fonctions en OCaml

Considérons le programme : `fonction arg0 arg1`

Alors, les opérations sont effectuées **dans l'ordre suivant** :

1. l'expression `arg1` est évaluée en v_1 ;
2. l'expression `arg0` est évaluée en v_0 ;
3. l'expression `fonction` est évaluée en une fonction `fun x y -> corps` ;
4. `corps` est évalué dans l'environnement `x`: v_0, `y`: v_1.

Si l'évaluation de `fonction` en `fun x y -> corps`, et les évaluations de `arg0`, `arg1` **ne produisent pas d'effet de bord**, alors **l'ordre n'est pas important**...

- sinon, attention à l'ordre d'évaluation
  raison de plus pour utiliser avec parcimonie les effets de bord
- si l'une des sous-expressions ne termine pas, l'appel ne termine pas

Données locales à l'évaluation d'une fonction

```
let f a =
  let x = if a < 0 then -1 else 1 in
  let y = x * a in
  let p = (x, y) in
  p
```

Où sont stockés chacun des noms ?

- l'**argument** `a` est local à la fonction
- de même pour les **noms** `x`, `y` et `p`
- par contre la **paire** `(x, y)` est ensuite **renvoyée comme résultat** et sort donc de la fonction

```
let g b =
  fst (f (b + 1))
```

- pas de nom à stocker, mais **plusieurs valeurs intermédiaires**
- également **plusieurs appels de fonctions à traiter**...

Organisation de la mémoire : pile et tas

Une mémoire découpée en deux régions importantes

- **La pile** contient les données à **durée de vie bornée syntaxiquement** (**arguments**, **noms locaux**), et généralement courte (sauf fonction principale)
- **Le tas** contient les données dont la durée de vie est **dynamique**, généralement impossible à déterminer localement

- la **pile** croît et décroît **linéairement**
  à chaque appel ou retour de fonction, à chaque nouveau nom local...
- en **Java** une allocation sur le tas correspond à un `"new"`, en **OCaml**, lorsqu'on construit une paire, un tableau, ...
- en **Java** et en **OCaml**, la suppression d'un élément sur le tas est à l'initiative du **Garbage Collector (GC, Glaneur de Cellules)**

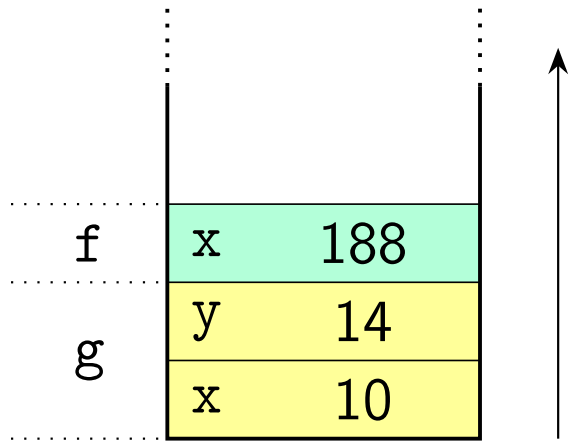
Organisation de la mémoire : pile

Pile :

- les paramètres sont alloués à l'appel, désalloués au retour
- les noms locaux sont alloués lors de leur définition, désalloués lors de la fin de leur portée (scope)

Exemple :

```
let f x =
  if x < 2 then x + 3
  else 3 - x
let g x =
  let y = x + 4 in
  f (y * y)
g 10
```



Note : il existe plusieurs occurrences du nom `x`, la plus récente masque la plus ancienne...

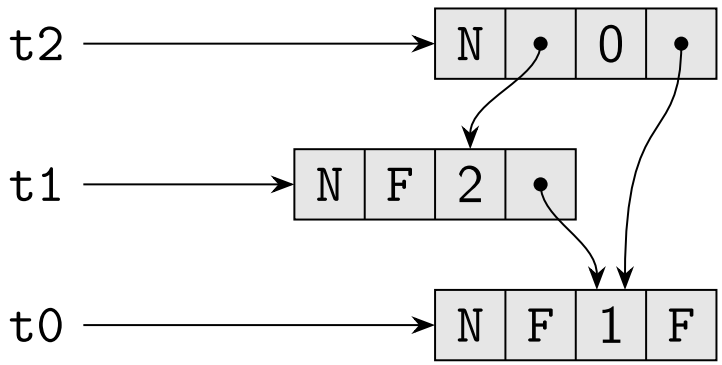
Organisation de la mémoire : allocation sur le tas

Tas :

- géré de manière implicite
- on créé un nouvel élément lorsque on définit des **valeurs construites**
- un élément sur le tas existe **tant qu'il est référencé**
- un élément peut être **partagé** : si plusieurs références

Exemple :

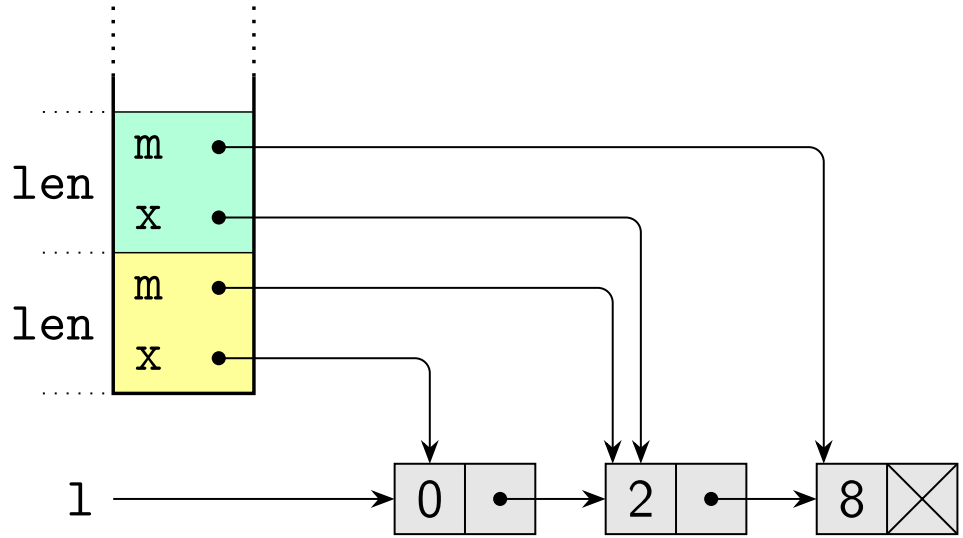
```
type a =
  | F
  | N of a * int * a
let t0 = N (F, 1, F)
let t1 = N (F, 2, t0)
let t2 = N (t1, 0, t0)
```



Organisation de la mémoire

On utilise presque toujours **pile et tas** conjointement :

```
let rec len x =
  match x with
  | [ ] -> 0
  | _ :: m ->
      1 + len m
let l =
  0 :: 2 :: 8 :: [ ]
len l
```



Note : la déclaration d'une fonction récursive nécessite le mot clé `rec`

- sans `rec`, `f` dans la définition de `f` fait référence à un nom `f` défini précédemment (fonction ou pas !)

Organisation de la mémoire : dépassement de pile

Quelques remarques sur **l'espace disponible sur la pile** :

- il est **fixé** au début de l'exécution d'un programme
  (par la machine, l'OS, l'utilisateur...)
- lorsqu'**il ne reste plus d'espace** : **stack overflow**...
- soit sous la forme d'une erreur "propre"
  c'est le cas d'OCaml, qui plante sans corrompre la machine
  idem en Java...
- mais sous d'autres architectures : **corruption de la mémoire**
  i.e., catastrophe garantie...

Outline

Types des fonctions

Type de fonction

Le type des fonctions dont l'argument est de type `t0` et le résultat de type `t1` est noté `t0 -> t1`.

Plus généralement, on note `t0 -> t1 -> ... -> tn -> t` le type des fonctions prenant `n+1` arguments de types `t0`, `t1`, ..., `tn` et retournant une valeur de type `t`.

Inférence de types :

- OCaml est capable de **calculer le type d'une fonction**, sans aucune annotation :

```
let f x y z =
  let n = if x then y + 1 else 3 in
  (n, z n, y) ;;
val f : bool -> int -> (int -> 'a) -> int * 'a * int =
    <fun>
```

- algorithme très intéressant : résout et généralise des contraintes

Appels de fonctions et vérification de types

Typage d'un appel de fonction

Étant donné une expression `f e`, OCaml vérifie que :

1. `f` est une fonction
2. le type de `e` est admissible comme type d'un argument de `f`

Erreurs que l'on souhaite éviter :

- si l'expression `f` **n'est pas une fonction**, alors l'application n'a pas de sens : `true false`
- si l'argument a un type incompatible avec celui du paramètre de la fonction, il est impossible d'évaluer le corps :
  `(fun x -> x < 25) true`

On note que `(fun x -> x) "mot"` a du sens, même si le type de `fun x -> x` est `'a -> 'a` et `"mot"` est de type `string`...

Règles de typage des fonctions

Pour une **définition de fonction**, la règle est très simple :

$$\frac{\ldots, \texttt{x} : \texttt{t0} \vdash \texttt{e} : \texttt{t1}}{\ldots \vdash \texttt{fun x -> e} : \texttt{t0 -> t1}}$$

Pour un **appel, c'est plus complexe** ; on voudrait écrire :

$$\frac{\ldots \vdash \texttt{f} : \texttt{t0 -> t1} \qquad \ldots \vdash \texttt{e} : \texttt{t0}}{\ldots \vdash \texttt{f e} : \texttt{t1}}$$

mais la vraie règle est **plus complexe**, en raison du **polymorphisme** :

- si le type de `f` autorise un argument de type **plus général** que celui de `e`, on peut évaluer l'appel `f e`
  exemple : `(fun x -> x) "mot"` vu précédemment
- sinon, le programme est rejeté... "this expression has type ... but an expression was expected of type ..."

Appels de fonctions polymorphes

Considérons simultanément deux mécanismes d'abstraction :

- **fonctions** : paramétricité vis à vis d'un argument
- **types polymorphes** : paramétricité vis à vis d'un type

La combinaison des deux permet de paramétrer un fragment de code par un argument avec une hypothèse relâchée sur son type...

Voyons un exemple : **transformateur de listes**

```
# let rec map f l =
    match l with
    | [ ] -> [ ]
    | a :: b -> (f a) :: (map f b)
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- `f` est une fonction de type `'a -> 'b` (**doublement polymorphe**)
- `l` est une liste d'éléments de type `'a` (**contrainte 1**)
- le résultat est une liste d'éléments de type `'b` (**contrainte 2**)

Typage des fonctions : une surprise...

Considérons à nouveau les règles de typage, et **effaçons les programmes** et remplacçons -> par l'implication mathématique $\Longrightarrow$

Définition de fonction :

$$\frac{\mathtt{H}, \mathtt{x} : \mathtt{t0} \vdash \mathtt{e} : \mathtt{t1}}{\mathtt{H} \vdash \mathtt{fun\ x\ ->\ e} : \mathtt{t0\ ->\ t1}} \longrightarrow \frac{\mathtt{H}, \mathtt{t0} \vdash \mathtt{t1}}{\mathtt{H} \vdash \mathtt{t0} \Longrightarrow \mathtt{t1}}$$

Cela ressemble à une règle logique :
pour prouver $\mathtt{t0} \Longrightarrow \mathtt{t1}$, on pose l'hypothèse t0, et on prouve t1...

Appel de fonction :

$$\frac{\mathtt{H} \vdash \mathtt{f} : \mathtt{t0\ ->\ t1} \qquad \mathtt{H} \vdash \mathtt{e} : \mathtt{t0}}{\mathtt{H} \vdash \mathtt{f\ e} : \mathtt{t1}} \longrightarrow \frac{\mathtt{H} \vdash \mathtt{t0} \Longrightarrow \mathtt{t1} \qquad \mathtt{H} \vdash \mathtt{t0}}{\mathtt{H} \vdash \mathtt{t1}}$$

Encore une fois, cela ressemble à une règle logique :
si on a prouvé $\mathtt{t0} \Longrightarrow \mathtt{t1}$ et t0, on déduit t1 (Modus Ponens)

Typage et preuve

Il existe un parallèle entre **typage** et **preuve** :

Correspondance de Curry Howard : programmer = prouver

programme	preuve
type	théorème

Ce lien entre logique et programmation est très solide, et sert de fondation à plusieurs **systèmes de preuves** dont **Coq** :

- permet de **prouver des théorèmes mathématiques** et de **vérifier des programmes**
- preuve largement automatisée du **théorème des quatre couleurs**
- preuve du **théorème de Feith-Thompson** (structure des groupes de cardinal impair)
- **compilateur CompCert** formellement vérifié

Outline

Notion de clôture

On peut procéder à une **application partielle** de fonction à plusieurs arguments, en définissant une **fonction qui renvoie une fonction** et en l'appliquant à **un argument**...

Définition : clôture

On appelle **clôture** une expression formée d'une **fonction** qui est **appliquée** à un ou plusieurs arguments, mais dont on ne peut pas encore évaluer le corps.

Une clôture peut être **appliquée** ensuite, **comme toute fonction** :

```
# let add_vec (x0, y0) (x1, y1) = (x0 + x1, y0 + y1)
  let add_spec = add_vec (12, -3)
  let res = add_spec (-7, 8) ;;
val add_vec : int * int -> int * int -> int * int = <fun>
val add_spec : int * int -> int * int = <fun>
val res : int * int = (5, 5)
```

Clôtures : utilisation

Principe général : lorsqu'on peut réutiliser plusieurs fois une application partielle, l'utilisation d'une clôture peut contribuer à la clarté du code

Exemple : fonction d'encodage paramétrée par le **protocole** et le **message**

```
let encodage protocol message =
  (* ... *)

let my_enc = encodage my_proto

let enc_0 = my_enc msg_0
let enc_1 = my_enc msg_1
let enc_2 = my_enc msg_2
```

Dans la suite, on verra d'autres applications pour les clôtures

Clôtures : représentation

Considérons le cas particulier d'une fonction à deux arguments :

```
let f arg_0 arg_1 =
  corps

let g = f v_0
```

Vue intuitive de la **représentation interne** de g : un type **paire**

- premier argument : **corps de la fonction** `f`
- second argument : **valeur du premier argument** `v0`

Lorsqu'on applique une clôture, on construit une nouvelle clôture étendue, ou bien on évalue le corps (lorsque tous les arguments sont présents).

Application :

- **simuler les clôtures dans un langage qui en est dépourvu** (e.g., Java)

Curryfication

Curryfication

Plutôt que d'écrire `f = fun (x, y)-> ...`,
on peut définir `f = fun x -> (fun y -> ...)`

- **avantage de la forme curryfiée** :
  on peut procéder à des applications **partielles** de fonctions
- **exemples** :

```
List.mem: 'a -> 'a list -> bool
Arg.parse: (bytes * Arg.spec * bytes) list -> Arg.
    anon_fun -> bytes -> unit
```

Choix de **l'ordre des arguments** :

- **astuce 1** : placer en premier l'argument qui sera connu le plus tôt afin de permettre la construction et le partage de **clôtures**
- **astuce 2** : utiliser un ordre standard
  e.g., dernier argument = structure (liste, arbre...)

Outline

Principe

Nous avons vu que les fonctions en OCaml sont **des expressions comme les autres**, et par conséquent, elles **peuvent être passées en paramètres** :

Définition : fonction d'ordre supérieur

Une **fonction d'ordre supérieur** est une fonction dont **au moins un des paramètres est une fonction**.

Un mécanisme d'abstraction puissant :

- une **fonction** =
  une **action**
- une **fonction d'ordre supérieure** =
  une **action** qui est **paramétrée par une action**

Application : action sur une valeur optionnelle

Action optionnelle, sur une **valeur optionnelle**

```
# type 'a option = None | Some of 'a
  let opt_act f o =
    match o with
    | None -> None
    | Some x -> Some (f x) ;;
val opt_act : ('a -> 'b) -> 'a option -> 'b option
  = <fun>
```

- si l'argument décrit l'absence de valeur :
  on renvoie la valeur vide `None`
- sinon :
  on applique la fonction passée en argument

Application : itérateur

On peut effectuer une action sur plusieurs éléments d'une liste :

```
# let rec list_iter f l =
    match l with
    | [ ] -> ( )
    | a :: b -> f a ; list_iter f b ;;
val list_iter : ('a -> 'b) -> 'a list -> unit
  = <fun>
```

- la fonction `f` est appliquée à chaque élément de la liste
- ordre d'application : à partir du début de la liste

Note : les itérateurs seront étudiés en détail **dans un autre cours**

Application : tri

Exercice : écrire une fonction de **tri** sur des **listes polymorphes** `'a list`

Difficulté :

- on doit utiliser une **relation d'ordre** pour comparer les éléments
- mais celle-ci **dépend du type des éléments de la liste**

Solution : une **fonction d'ordre supérieur** prenant l'ordre en paramètre

```
# type ordre = Inf | Eq | Sup
  let tri comparaison liste =
    (* ... *)
val tri : ('a -> 'a -> ordre) -> 'a list -> 'a list
  = <fun>
```

Fonctions d'ordre supérieur et clôtures

On peut combiner :

- **clôtures** : applications partielles de fonctions
- **ordre supérieur** : fonction prenant une fonction en paramètre

Exemple : une requête sur plusieurs bases

- `List.filter: ('a -> bool)-> 'a list -> 'a list` (librairie) prend une liste et extrait les éléments satisfaisant une condition
- `condition: requete` décrit une formule logique
- `build_cond: requete -> t -> bool`
- listes sur `l0`, ..., `ln`

```
let extrait = List.filter (build_cond condition) in
let res0 = extrait l0
and res1 = extrait l1
and ...
and resn = extrait ln
```

Outline

Valeurs persistantes et valeurs mutables

Rappel de l'amphi précédent :

Définition : type / valeur immutable (ou persistant)

Une **valeur immutable** est une valeur qu'on peut lire mais pas modifier.

Structures **persistantes** :

- **paires** / uplets
- **enregistrements**
- types **somme**

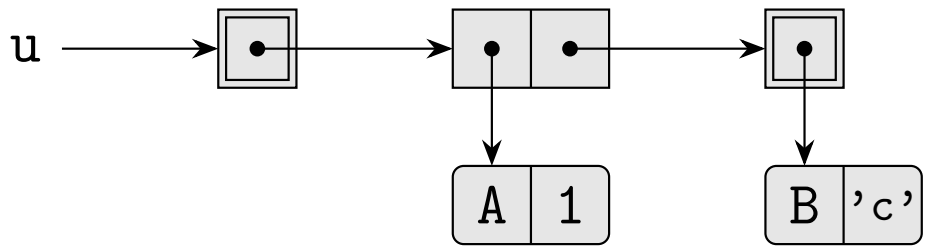
Structures **mutables** :

- **références**
- **tableaux**
- champ **mutables** d'enregistrements

Exemple :

```
type c = A of int | B of char
type r = c * c ref
let u = ref (A 1, ref (B 'c'))
```

Mémoire (mutables : double cadre) :



Fonctions et valeurs mutables

Pourquoi s'inquiéter des structures mutables / immutables dans un cours sur les fonctions ?

```
type trec = { mutable u: int ; ... }
let f0 x = let v = ... in x.u <- v ; ...
let f1 x = ... f0 x ...
  ...
let f10 x = ... f9 x ...
let r = { u = 0 ; ... }
(* ici, r.u vaut 0 *)
f10 r
(* que vaut r.u ? *)
```

Un mécanisme **puissant** mais qui peut se révéler **dangereux** :

- puissant : `f0` peut "communiquer" une partie du résultat via `x.u`
- dangereux : on peut ainsi **détruire des propriétés globales** depuis `f0` (`f0` peut être à des milliers de lignes de `f10 r`)

Types mutables : il est difficile de garantir la préservation de l'état

Fonctions et valeurs persistantes

Fonctions à arguments persistants (ou immutables) :

- limitent la **portée des effets de bords** : seulement I / O et erreurs
- donc les noms dans l'environnement sont **préservés en profondeur**
- `f = fun arg -> res`
  on peut inclure une partie de l'état dans `arg` / `res`

fonctions à arguments mutables :

- peuvent modifier indirectement l'état courant via les champs mutables
- donc **n'autorisent pas le même niveau d'abstraction** :
  pour connaître le contenu de la mémoire après appel,
  il faut comprendre **tout** le programme

Programmation en style fonctionnel pur

- Pas de fonctions avec des arguments (même partiellement) mutables
- Pas d'accès à des références globales depuis le corps d'une fonction

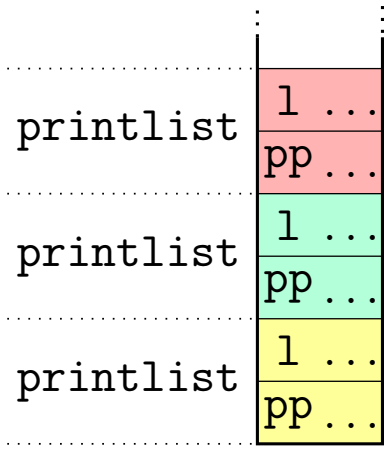
Note : Haskell, langage fonctionnel sans effets de bords...

Outline

Déclarations locales

Considérons cette fonction d'affichage des éléments d'une liste :

```
let rec printlist pp l =
  match l with
  | [ ] -> ( )
  | a :: b ->
      pp a;
      printlist pp b
```



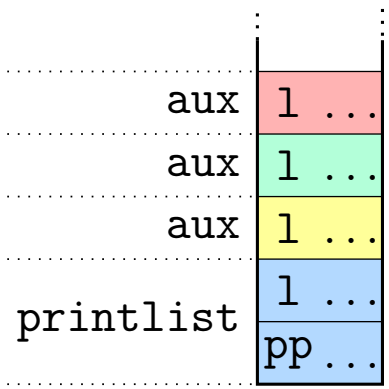
L'argument pp est constant et propagé à chaque appel :

- définition **plus lourde** :
  un argument supplémentaire, qui gagnerait à être global
- **consommation de pile** plus importante :
  le code compilé crée une nouvelle référence à pp à chaque appel...

Fonctions locales

Solution pour un code plus clair : **fonction auxiliaire**

```
let printlist pp l =
  let rec aux l =
    match l with
    | [ ] -> ( )
    | a :: b -> pp a; aux b in
  aux l
```



Avantages :

- **clarté** :
  fonction récursive plus compacte, argument constant explicite
- **efficacité** :
  moins de consommation de pile, moins de temps à passer des paramètres

Fonctions récursives terminales "tail recursive"

Comment économiser la place utilisée en pile ?

- espace en pile précieux (stack overflow...)
- par défaut : quelques centaines de ko / quelques Mo (modifiable)

Voyons une catégorie particulière de fonctions :

Fonction récursive terminale (ou "tail-recursive")

Une fonction **récursive terminale** est telle que tout appel récursif est **la dernière opération sur un chemin d'exécution** du corps de la fonction.

Récursive terminale :

```
let rec mem x l =
  match l with
  | [ ] -> false
  | a :: l ->
      (x = a) || (mem x l)
```

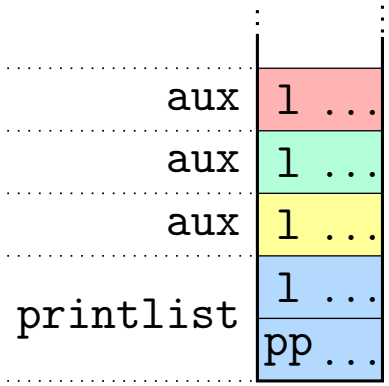
Non récursive terminale :

```
let rec len l =
  match l with
  | [ ] -> 0
  | _ :: l ->
      1 + len l
```

Avantage de la récursion terminale

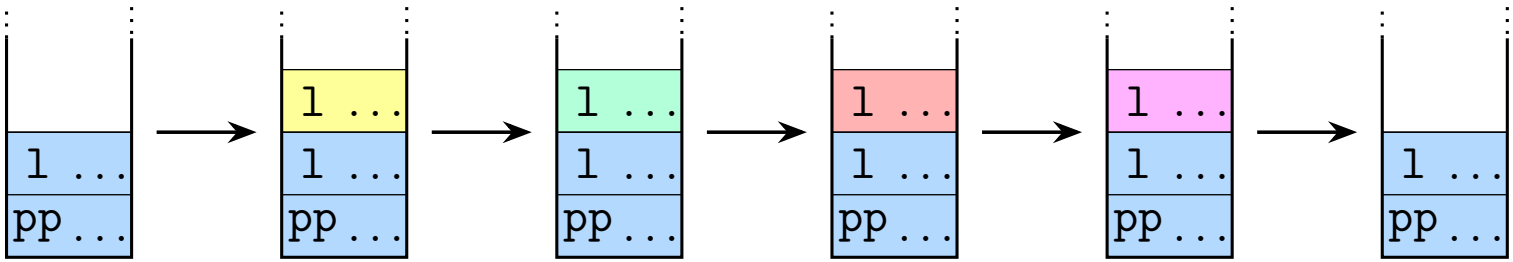
Nous avons vu la fonction suivante :

```
let printlist pp l =
  let rec aux l =
    match l with
    | [ ] -> ( )
    | a :: b -> pp a; aux b in
  aux l
```



- lorsque la fonction aux retourne d'un appel récursif à elle même, l'exécution du code appelant est terminée...
- donc tous les appels à aux peuvent **retourner directement vers** `printlist`

C'est le principe d'une **optimisation** de certains **compilateurs** :



Transformation en style récursif terminal

Considérons à nouveau la fonction longueur :

```
let rec len l =
  match l with
  | [ ] -> 0
  | _ :: l -> 1 + len l
```

Elle n'est pas récursive terminale, car après le retour d'un appel récursif qui renvoie `v`, il faut encore calculer `1 + v`...

Astuce : changer la signature et utiliser un **accumulateur**

```
let len =
  let rec aux acc l =
    match l with
    | [ ] -> acc
    | _ :: l -> aux (1 + acc) l in
  aux 0
```

Outline

Principaux éléments à retenir

Un mécanisme d'abstraction très puissant :

- **masquage des détails d'implémentation** et **généricité**
- **ordre supérieur** : fonction appliquées à des fonctions
- **clôtures** : fonctions partiellement appliquées
  peuvent être passées en paramètre plus tard

Quelques techniques pour **des programmes plus clairs** et **efficaces** :

- privilégier les **types immutables** :
  meilleure abstraction, ordre d'évaluation sans importance, plus grande clarté
- **définitions locales** (fonctions, autres valeurs) :
  clarté, efficacité (moins d'arguments) et économie de pile
- **récursivité terminale** : économie de pile, meilleure localité

Fonctions et langages de programmation

Une brève comparaison de quelques langages :

	Java	OCaml	Haskell
Évaluation de paramètres	par valeur	par valeur	paresseuse
Fonctions de première classe (manipulables comme valeurs)	simulable	en natif	en natif
Fonctions d'ordre supérieur	simulable	en natif	en natif
Clôtures	simulable	en natif	en natif
Optimisation réc. terminale	JVMstd: non	oui	oui

- Java peut simuler de nombreuses constructions à l'aide d'objets
- exercice : ajouter votre langage préféré (C, C++, JavaScript)...