Plan des 4 prochains cours

- Environnements
- Fonctionnalité
- Application de fonction
- Récursivité
- Traits impératifs
  - Mémoire, références
  - Exceptions
- Correction de programmes

Renaud Rioboo[a]

[a]Renaud.Rioboo@lip6.fr

Évaluation

Espace d'**expressions**	Espace de **valeurs**
1+2	3
`if true then 1 else 2`	1
`let y = 1 in y+1`	2
`function x -> x + 1`	$\ll x \leadsto x+1\ ,\ E \gg$
`let y = 1 in` `  function x -> x + y`	$\ll x \leadsto x+y\ ,\ E_1 \gg$

Une **expression** Caml-Light prend une **valeur** dans un **environnement**.
C'est l'**évaluation**.

Un **identificateur** est un nom[a].

[a]que l'on utilise souvent dans une fonction ou pour pour éviter des re-calculs

Un **environnement** permet d'associer une valeur à un identificateur.

Une **liaison** est l'association d'un identificateur à une valeur.

Dans un environnement il peut y avoir **plusieurs** liaisons pour un même identificateur, c'est le **masquage**.

Si E est un environnement, $\mathcal{E}$ est une expression et V une valeur, on notera $\mathcal{E} \longrightarrow_E V$ si $\mathcal{E}$ s'évalue en V dans l'environnement E.

Ainsi si E est l'environnement courant, on a :

– `true` $\longrightarrow_E$ `true`,
– $1 \longrightarrow_E 1$,
– (`if true then` 1 `else` 2) $\longrightarrow_E 1$

L'environnement **global** est l'environnement dans lequel le programme se trouve.

Portée dynamique

– En portée **dynamique** la valeur d'un identificateur se trouve dans l'**environnement global** (**interprétation**, langages de shell)

`let x = 1`	$E \;= \lfloor\, (\mathrm{x}{=}1) \triangleleft E \,\rfloor$
`let f =`	$E \;= \lfloor\, (\mathtt{f}{=}F) \triangleleft E \,\rfloor$
`function y -> x + y`	
`f 1`	2
`let x = 2`	$E \;= \lfloor\, (\mathrm{x}{=}2) \triangleleft E \,\rfloor$
`f 1`	3

La portée dynamique n'est pas compatible avec le typage statique.

Portée statique

– En portée **statique** (ou lexicale) la valeur d'un identificateur se trouve dans l'**environnement de définition** (**compilation**)

`let x = 1`	$E_1 \ = \lfloor\ (\mathtt{x}{=}1) \triangleleft E_0\ \rfloor$
`let f =`	$E_2 \ = \lfloor\ (\mathtt{f}{=}F) \triangleleft E_1\ \rfloor$
` function y -> x + y`	
`f 1`	2
``let x = `a` ``	$E_3 \ = \lfloor\ (\mathtt{x}{=}\text{`}\mathtt{a}\text{`}) \triangleleft E_2\ \rfloor$
`f 1`	2

Interaction avec le système de fichiers

$$E_g = E_u \oplus E_m \oplus E_b$$

L'environnement global est la juxtaposition de

- E_u, session utilisateur en cours,
- E_m, les modules ouverts
- E_b, la base.

Une compilation (`camlc -c`) produit :

- un fichier `.zo` de code et
- un fichier `.zi` d'interface obtenu à partir d'un fichier de description d'interface `.mli`.

La directive `#open` *mod* permet de rendre visible les **identificateurs** décrits dans *mod*`.zi` sans avoir à les nommer par *mod__ident*.

L'interprète possède

– une fonction `load( : string -> unit)` qui permet de charger un module source (`.ml`) en mémoire.

– une fonction `load_object( : string -> unit)` qui permet de charger un module source (`.zo`) en mémoire.

En mode compilé il faut inclure le fichier `.zo` dans la ligne de commande

```
camlc -o bar foo.zo bar.ml
```

pour obtenir les **valeurs** définies dans le module `foo` **et** rendre visibles les **identificateurs** de `foo` en incluant la directive

```
#open "foo"
```

dans le fichier `bar.ml`

Terminologie

MATHS	INFORMATIQUE
variable	paramètre formel
paramètre	paramètre libre
argument	paramètre effectif
fonction	fonction partiellement définie
application	fonction
calculer f(exp)	appliquer f à exp
f(exp)	f exp

On **doit** distinguer l'expression $f(exp)$ de la valeur qu'elle prend dans un environnement. On appelle **application** une expression de la forme $\mathcal{E}_1 \;\; \mathcal{E}_2$.

Les fonctions

Les fonctions sont des valeurs Caml-Light que l'on nomme des **fermetures**.
En Caml-Light elles sont caractérisées par :

- un **nom** de variable, c'est le **paramètre formel** (`nom`),
- une **expression** définissant la valeur de la fonction en tout point que l'on nomme le **corps** de la fonction (`corps`),
- un **environnement de définition** de la fonction (`Env`) :

On les note : $\ll$`nom` $\rightsquigarrow$ `corps` , `Env` $\gg$

On notera souvent

- les **expressions** en lettres calligraphiques ($\mathcal{E}$) ou sous forme de pseudo code (`expr`)
- les **valeurs** ou les **environnements** en notation mathématique (V ou E)

Appel par Nom

```
let deux = (print_string "1+1" ; 1 + 1) in
  deux + deux ;;
```

1. liaison de `deux` à

   `(print_string "1+1"; 1 + 1)`

   puis évaluation de `deux + deux`

2. substitution de `deux` à sa “valeur” soit :

   ```
   (print_string "1+1" ; 1 + 1) +
   (print_string "1+1" ; 1 + 1)
   ```

3. affichage de `"1+1"` calcul de `1 + 1 = 2`
4. affichage de `"1+1"` calcul de `1 + 1 = 2`
5. calcul de `2 + 2 = 4`

Les identificateurs sont remplacés par des expressions qui sont réévaluées.

Appel par Valeur

```
let deux = (print_string "1+1" ; 1 + 1) in
  deux + deux ;;
```

1. calcul de la valeur de
   `(print_string "1+1"; 1 + 1)`
2. affichage de `"1+1"` retour de 2
3. liaison de `deux` à sa valeur 2
4. calcul de `deux + deux` soit 2+2
5. retour de 4

Les identificateurs sont remplacés par des valeurs et ne sont pas réévalués.

La liaison

– Définitive : on modifie l'environnement global

```
let x = 1;;
```

1. évaluation de 1.
2. enrichissement de l'environnement global de la liaison de `x` à 1

   $E \;=\lfloor$ (x=1) $\triangleleft E \rfloor$

– Temporaire : on créée un nouvel environnement pour évaluer une expression (portée lexicale) :

```
let x = 1 in x+1;;
```

1. évaluation de 1.
2. création d'un nouvel environnement contenant la liaison de `x` à 1

   $E_1 \;=\lfloor$ (x=1) $\triangleleft E \rfloor$

3. évaluation de `x+1` dans E_1

L'environnement global n'est pas modifié !

Les fonctions comme valeurs

On peut étendre les opérations booléennes aux fonctions d'une variable booléenne :

```
type fonctions_booleennes = Fbool of bool -> bool ;;

let vrai = Fbool (function x -> true)
and faux = Fbool (function x -> false);;

let ou = function f1 -> function f2 ->
  match (f1,f2) with
   ((Fbool f1), (Fbool f2)) ->
      Fbool (function x -> (f1 x) || (f2 x)) ;;
```

Les fonctions sont des valeurs spéciales

```
let applique = function f1 ->
  match f1 with
    Fbool f1 -> function x -> f1 x ;;

let est_vrai = ou vrai vrai ;;

applique est_vrai true = applique vrai true ;;
applique est_vrai false = applique vrai false;;

est_vrai = vrai;;
```

Liaison Caml-Light

La liaison permet de **déstructurer** des valeurs et d'introduire des **identificateurs** :

```
type point = {abs : int ; ord : int} ;;
let ({abs = x ; ord = y} as un_point) =
  {abs = 10 ; ord = 20 } ;;
```

Introduit globalement trois nouveaux noms :

– `x : int = 10`,

– `y : int = 20` et

– `un_point : point = {abs = 10; ord = 20}`

il n'y a pas de calcul dans les parties gauches d'un let, ce sont des **motifs lexicaux**. Une telle liaison peut être **mal typée** ou **échouer** :

```
let (a,b) = (1,2,3) in a+b ;;
let (a,1) = (1,2) in a+1 ;;
```

```
let projection_verticale = function a_point ->
  match a_point with
    {abs = x ; ord = y} ->
      if (y = 0) then a_point
      else {abs = x ; ord = 0};;
```

Le motif “_” (souligné) permet de déstructurer sans nommer.

```
let abscisse = function un_point ->
  match un_point
    with {abs = x ; ord = _} -> x ;;
```

Application

Appliquer une fonction f à une valeur x c'est calculer la valeur de $f(x)$.

Il faut évaluer l'expression constituant le **corps** de la fonction dans l'environnement de définition de la fonction augmenté de la liaison où le paramètre formel est lié à la valeur d'appel (le paramètre effectif).

Soient $\mathcal{F}_e$ et $\mathcal{X}_e$ deux **expressions** Caml-Light, la valeur de l'**application** $\mathcal{F}_e(\mathcal{X}_e)$ dans l'environnement E est la valeur de la simplification de $F(x_v)$ où F est la valeur de $\mathcal{F}_e$ dans E et x_v est la valeur de $\mathcal{X}_e$ dans E.

Si $F = \ll \mathtt{x} \rightsquigarrow \mathcal{E}_{\mathtt{x}}\ ,\ E_d \gg$, alors $F(x_v)$ a la même valeur que celle de $\mathcal{E}_{\mathtt{x}}$ dans l'environnement $\lfloor\ (\mathtt{x}{=}x_v) \triangleleft E_d\ \rfloor$

On peut donc écrire une **règle** de simplification d'une application :

$\mathcal{F}_e \longrightarrow_E (\ll \mathtt{x} \rightsquigarrow \mathcal{E}_{\mathtt{x}} \, , E_d \gg)$ $\mathcal{X}_e \longrightarrow_E x_v$ $\mathcal{E}_{\mathtt{x}} \longrightarrow_{\lfloor\, (\mathtt{x}=x) \triangleleft E_d \,\rfloor} V$	$\mathcal{F}_e(\mathcal{X}_e) \longrightarrow_E V$

pour obtenir la valeur d'une application il faut :

– évaluer l'expression en position fonctionnelle,
– évaluer l'expression en position d'argument,
– augmenter l'environnement de définition de la fermeture de la liaison entre le paramètre formel et le paramètre effectif pour évaluer le corps de la fermeture.

Fonctions à plusieurs arguments

Soit E l'environnement courant, a-t-on deux choix ?

- `let foo = function (x,y) ->` $\mathcal{E}(\mathtt{x},\mathtt{y})$
  L'expression n'a qu'**un seul** argument qui est le couple (x,y) et `foo` a pour type $\sigma_x {*} \sigma_y \to \sigma_{\mathcal{E}}$
  Sa valeur est une fermeture $F_1 = \ll \mathtt{x,y} \leadsto \mathcal{E}(\mathtt{x,y})\ ,\ E \gg$
- `let foo = function x -> function y ->` $\mathcal{E}(\mathtt{x},\mathtt{y})$
  L'expression est une fonction **curryfiée** et `foo` a pour type $\sigma_x \to \sigma_y \to \sigma_{\mathcal{E}}$.
  Sa valeur est une fermeture $F_2 = \ll \mathtt{x} \leadsto \mathcal{E}_{\mathtt{x}}\ ,\ E \gg$
  où $\mathcal{E}_x$ est l'expression `function y ->` $\mathcal{E}(\mathtt{x},\mathtt{y})$.

Application partielle

Soit $E_{\texttt{foo}}$ l'environnement de définition de F_2.

On peut appliquer F_2 à une valeur x_0 de type σ_x.

C'est une **application partielle**, le résultat en est une autre fermeture :

$F_{x_0} = \ll \texttt{y} \rightsquigarrow \mathcal{E}(\texttt{x},\texttt{y})\ ,\ \lfloor\ (\text{x}{=}x_0) \triangleleft E_{\texttt{foo}}\ \rfloor \gg$

qui a pour type $\sigma_y \to \sigma_E$

– F_1 et F_2 n'ont **pas le même type**,
– On évalue F_2 en liant x et y **en même temps**,
– F_1 et F_2 ne sont **pas équivalentes**.

Fonctions

La construction `fun` $x_1 \ldots x_n$ `->` E

est équivalente à `function` x_1 `->` ...`function` x_n `->` E

L'écriture `let` F $x_1 \ldots x_n$ = E

est équivalente à `let` F `=` `fun` $x_1 \ldots x_n$ `->` E

La forme où `fun` ou `function` est explicite permet de préciser son **environnement de définition** :

```
let incremente_compteur = let compteur = ref(0) in
    function n ->
      begin
      compteur := !compteur + n ;
      !compteur
      end ;;
```

Le Filtrage

$$\texttt{match}\ \mathcal{V}\ \texttt{with}\ (\mid \mathcal{F}\ \texttt{->}\ \mathcal{E})$$

avec la notation

$$\mid \mathcal{F}\texttt{->}\mathcal{E} \quad = \quad \begin{cases} \mid \mathcal{F}_1\texttt{->}\mathcal{E}_1 \\ \quad \dots \\ \mid \mathcal{F}_n\texttt{->}\mathcal{E}_n \end{cases}$$

- Les $\mathcal{F}_i$ sont des **motifs lexicaux** qui introduisent de nouveau noms. Ils permettent de produire un environnement **d'exécution** de la clause.
- Les $\mathcal{E}_i$ sont des expressions qui constituent le **corps** de la clause, elles sont évaluées dans l'environnement de définition de la fonction augmenté des liaisons produites pendant le filtrage.

- Chaque **clause de filtrage** $\mid \mathcal{F}_i \rightarrow \mathcal{E}_i$ va :
  - permettre de **sélectionner** un sous-ensemble de valeurs du type σ_v que doit avoir $\mathcal{V}$.
  - permettre de produire un environnement dans lequel va s'évaluer l'expression $\mathcal{E}_i$ qui définit le **corps de la clause** comme dans :
    `let` $\mathcal{F}_i = \mathcal{V}$ `in` $\mathcal{E}_i$ exécuté dans un environnement E tel que $(\mathcal{V} \longrightarrow_E V)$.
    On notera E_i les liaisons produites pendant ce filtrage $E_i \oplus E$ l'environnement obtenu après avoir augmenté E de ces liaisons.

À l'appel, le paramètre effectif V qui est l'évaluation de l'expression $\mathcal{V}$ dans l'environnement courant est "superposé" avec les filtres $\mathcal{F}_i$ **dans l'ordre** du texte.

On peut donc énoncer une règle pour l'évaluation d'un filtre :

$\mathcal{V} \longrightarrow_E V$, $\exists i\ (\mid \mathcal{F}_i \to \mathcal{E}_i) \in (\mid \mathcal{F} \to \mathcal{E})$ et i est le premier indice qui produise des liaisons E_i de `let` $\mathcal{F}_i = \mathcal{V}$, $\mathcal{E}_i \longrightarrow_{(E_i \oplus E)} R$	[`match` $\mathcal{V}$ `with` $(\mid \mathcal{F} \to \mathcal{E})] \longrightarrow_E R$

La construction `match` $\mathcal{V}$ `with` $\mid \mathcal{F} \to \mathcal{E}$
est équivalente à (`function` $\mid \mathcal{F} \to \mathcal{E}) \, \mathcal{V}$.

On peut ajouter des **conditions** a une clause de filtrage $\mathcal{F}_i$ qui a alors la forme :

$$| \ \mathcal{F}_i \ \texttt{when} \ \mathcal{C}_i \ \texttt{->} \ \mathcal{E}_i$$

Où $\mathcal{C}_i$ est une expression qui doit être de type `bool`. Elle est évaluée dans l'environnement d'exécution du corps de la clause :

Si V est le paramètre effectif, et si E est l'environnement courant, $\mathcal{E}_i$ et $\mathcal{C}_i$ s'évaluent tous deux dans $E_i \oplus E$ qui est E augmenté des liaisons E_i produites pendant le filtrage.

cette construction est à éviter

La Récursivité

Dans le **corps** d'un `let` les identificateurs (*noms de fonctions*) sont cherchés dans l'environnement courant.

La construction

`let rec`

introduit un identificateur récursif.

Soit E_c l'environnement courant, la déclaration

$$\texttt{let rec foo =}\ \mathcal{E}$$

crée une valeur $V_{\texttt{foo}}$ qui est le résultat de l'évaluation de l'expression $\mathcal{E}$ dans l'environnement

$$E_{\texttt{foo}} = \lfloor\ (\texttt{foo=}V_{\texttt{foo}}) \triangleleft E_c\ \rfloor \text{ où } \mathcal{E} \longrightarrow_{E_{\texttt{foo}}} V_{\texttt{foo}}$$

et $E_{\texttt{foo}}$ devient l'environnement courant.

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

définit un environnement $E_{\texttt{fib}}$, une fermeture
$F_{\texttt{fib}} = \ll \texttt{n} \rightsquigarrow \mathcal{E}_{\texttt{fib}} \, , E_{\texttt{fib}} \gg$

l'évaluation d'une fonction récursive se fait comme une évaluation normale.

À l'appel d'une fonction récursive, il faut **empiler** les environnements :

- Soit $E_{\texttt{fib}}$ l'environnement de définition de `fib` et E l'environnement courant.
- Évaluation de `fib(3)` dans E :
  $F_{\texttt{fib}}(3)$ puisque `fib` $\longrightarrow_{\mathrm{E}} F_{\texttt{fib}}$ et `3` $\longrightarrow_{E}$ 3. On évalue
  $\mathcal{E}_3 =$ `fib(n-1) + fib(n-2)` dans $E_3 = \lfloor$ (n=3) $\triangleleft E_{\texttt{fib}} \rfloor$,
  évaluation de `fib(2)` dans E_3 en suspens et :

- évaluation de `fib(1)` dans E_3 soit l'évaluation de $F_{\texttt{fib}}(1)$ qui vaut 1.
  Donc `fib(1)` $\longrightarrow_{E_3} 1$
- reprise de l'évaluation de `fib(2)` dans E_3 :
  évaluation de $F_{\texttt{fib}}(2)$ puisque `fib` $\longrightarrow_{E_3} F_{\texttt{fib}}$ et `2` $\longrightarrow_{E_3} 2$
  On doit évaluer $\mathcal{E}_2 =$ `fib(n-1) + fib(n-2)`
  dans $E_2 = \lfloor$ (n=2) $\triangleleft E_{\texttt{fib}} \rfloor$:
  - évaluation de `fib(0)` dans E_2 soit 0
  - évaluation de `fib(1)` dans E_2 soit 1
- Donc $\mathcal{E}_2 \longrightarrow_{E_2} 1$

- et enfin $\mathcal{E}_3 \longrightarrow_{E_3} 2$

les types Somme

La déclaration

```
type foo = C | C1 of t1 | C2 of t2
```

introduit des

- **constructeurs de valeurs** du type `foo` :
  - une constante `C : foo`
  - deux **fonctions**
    - `C1 : t1 -> foo`
    - `C2 : t2 -> foo`

    qui **allouent** de nouvelles valeurs du type `foo`
- **motifs lexicaux** de filtrage des valeurs de type `foo` :
  - un motif `C` qui filtre cette valeur,
  - deux motifs `C1 _` et `C2 _` qui filtrent les autres.

    qui n'allouent pas mais **introduisent des noms**

les Types Sommes

Sont **récursifs** :

```
type somme_formelle =
  | Variable
  | Entier of int
  | Somme of somme_formelle * somme_formelle ;;
```

```
let rec eval = function (expr, valeur) ->
   | (Variable , _ ) -> valeur
   | (Entier n , _ ) -> n
   | (Somme (e1,e2) , _ ) ->
        eval(e1,valeur) + eval(e2,valeur) ;;
```

Ensuite :

`eval(Somme(Variable,Entier 2) , 4)` vaut 6.

Ordre supérieur

Sommer tous les éléments d'une liste ?

```
let rec somme = function l -> match l with
  | [] -> 0
  | h::r -> h + (somme r) ;;
```

Ici `0 :int` et `(prefix +) :int -> int -> int`.

```
let rec somme_abs =
  function zero -> function plus -> function l ->
    match l with
      | [] -> zero
      | h::r ->
        (plus h (somme_abs zero plus r)) ;;
```

On généralise `zero :` α et `plus :` β`->`α`->`α.

```
let somme = function l -> somme_abs 0 (prefix +) l;;
```

Pour tout environnement E si `F` est un identificateur (monomorphe) tel que

`F` $\longrightarrow_E F$ et si

`(function x -> F(x))` $\longrightarrow_E G$ on a :

$G \quad \equiv \quad F$ dans le sens où

$\forall x \; G(x) = F(x)$ et

`function x -> F(x)` $\Leftrightarrow$ `F`

```
let somme = somme_abs 0 (prefix +) ;;
let mult = somme_abs 1 (prefix *) ;;
```

Récursion terminale

L'**enveloppe** d'un appel récursif est l'ensemble des calculs qui doivent d'effectuer **après** cet appel.

Lors d'un appel récursif, les différents environnements doivent être conservés car d'autres calculs doivent s'exécuter **après** :

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

Soit E l'environnement courant, si `foo` $\longrightarrow_E F_{\texttt{foo}}$,
$F_{\texttt{foo}} = \ll \texttt{n} \rightsquigarrow \mathcal{E}_{\texttt{foo}} \text{ , } E_{\texttt{foo}} \gg$

Pour évaluer `foo(1)` dans E, on doit évaluer $F_{\texttt{foo}}(1)$ et donc évaluer (`foo(n-1) + n`) dans $E_1 = \lfloor \text{ (n=1)} \triangleleft E_{\texttt{foo}} \rfloor$. Si `foo(n-1)` est évalué en premier il faut restaurer E_1 pour évaluer `n`.

Un appel récursif est **terminal** s'il n'y a pas d'autres calculs à effectuer ensuite. Le résultat de la fonction est alors lui même un appel récursif.

```
let rec foo_bis =
  function n -> function res -> match n with
    | 0 -> res
    | _ -> foo_bis (n-1) (res + n);;
```

$$\forall x \; F_{\texttt{foo}}(x) = (F_{\texttt{foo\_bis}} \; x \; 2)$$

Si (`n` $\longrightarrow_{E_{n+1}} n$) et (`res` $\longrightarrow_{E_{n+1}} r$)

il est inutile de stocker E_{n+1} pour déterminer la valeur de

`foo_bis (n-1) (res + n)` dans E_{n+1}

puisque (`(n-1)` $\longrightarrow_{E_{n+1}} n-1$) et

(`(res + n)` $\longrightarrow_{E_{n+1}} r+n$) peuvent être calculés **avant**

Récursion terminale

Lorsqu'un appel récursif est **enveloppé** par une **opération associative** on peut **rajouter un paramètre** pour obtenir un appel terminal.

```
let rec foo = function arg ->
  if (arg = arg_0)
  then val_0
  else op(foo(g(arg)), f(arg));;
```

où `f` et `g` ne contiennent pas `foo`, devient lorsque `op` est associative :

```
let rec foo = function (arg,accu)
  if (arg = arg_0)
  then op(accu,val_0)
  else foo(g(arg) , op(accu, f(arg)));;
```

Dichotomie

$$\text{Calcul de } a^n = \begin{cases} (a^{n/2})^2 & \text{si } n \text{ est pair} \\ a(a^{n/2})^2 & \text{si } n \text{ est impair} \end{cases}$$

```
let rec pow = function a -> function n ->
  if ((n mod 2) = 0)
  then
    if (n = 0)
      then 1
      else
        let res = pow a (n / 2) in res * res
  else
    if (n = 1)
    then a
    else
      let res = pow a (n / 2) in a * res * res ;;
```

On peut généraliser 0, 1 et * :

```
let rec dich = fun zero un op a n ->
  if ((n mod 2) = 0) then
    if (n = 0)
    then zero
    else
      let res = dich zero un op a (n/2) in
        op res res
  else
    if (n = 1)
    then un
    else
      let res = dich zero un op a (n/2) in
        op a (op res res) ;;
```

Évaluation des arguments

Pour des raisons d'**efficacité** Caml-Light évalue les arguments d'une application du dernier au premier :

```
let foo = function x -> function y ->!x +!y;;

let essai = ref 0 in
  foo (essai := 1; essai) (essai := 2; essai);;
```

$\mathcal{E}_1$ $\mathcal{E}_2$

– 4 si l'ordre d'évaluation est `foo` puis $\mathcal{E}_1$ puis $\mathcal{E}_2$ ou bien
– 2 l'ordre d'évaluation est si $\mathcal{E}_2$ puis $\mathcal{E}_1$ puis `foo`

Ainsi lorsqu'une application est totale il n'y a pas besoin de créer les fermetures intermédiaires :

$$F\ x_1\ x_2\ \ldots\ x_n = F_1\ x_2\ \ldots\ x_n = F_{n-1}\ x_n$$

Les exceptions

Caml-Light offre un mécanisme d'erreurs simple grâce au type `exn`.

```
exception Foo ;;
exception Bar of int;;
```

Les exceptions sont des valeurs :

```
let to_foo = function an_exn -> match an_exn with
  | Bar _ -> Foo
  | an_exn -> an_exn ;;
```

marquées exceptionnelles (**déclencher** ou **lever** une exception) par :

$$\texttt{raise}\,\mathcal{I}$$

On peut **récupérer** une exception :

$$\texttt{try}\,\mathcal{E}\,\texttt{with}\ \mid \mathcal{F} \rightarrow \mathcal{E}_{\mathcal{I}}$$

```
let v = 1 in
  try (let v = 2 in raise Foo)
  with
   | Foo -> v
;;
```

Soit E l'environnement courant.

– Si l'évaluation de $\mathcal{E}$ dans E ne déclenche pas d'exception c'est une valeur V (de type σ_v) qui est le résultat.
– Sinon, soit $\mathcal{I}$ l'exception déclenchée, le résultat de l'évaluation est alors le même que celui de

$$\texttt{match}\,\mathcal{I}\,\texttt{with}\ \mid \mathcal{F} \to \mathcal{E}_{\mathcal{I}}$$

dans l'environnement E.

En particulier les motifs des clauses $\mathcal{F}_i$ doivent être des exceptions et les corps des clauses $\mathcal{E}_{\mathcal{I}i}$ doivent avoir des types compatibles avec σ_v.

Le filtrage n’a pas besoin d’être exhaustif,

on peut récupérer **n’importe quelle** exception.

c’est à éviter

Une exception marquée n’est plus une valeur :

```
let foo = function x ->
  try (x+1) with
    | Foo -> -1
;;
foo (raise Foo);;
```

Traits impératifs

Les **références** Caml-Light correspondent aux **variables** des langages impératifs. Ce sont des “boites” dans lesquelles on peut “mettre” des valeurs de type fixé.

Caml-Light	C
`let r = ref 1`	`int r = 1`
On distingue la référence de son contenu	
`r :=` $\boxed{\texttt{!r}}$ `+ 1`	`r =` $\boxed{\texttt{r}}$ `+ 1`

On a alors un modèle dynamique de **mémoire** analogue aux environnements mais dont les “clés” sont des références.

L'affectation

$$\texttt{(prefix :=)} : \alpha\ \texttt{ref} \texttt{ -> } \alpha \texttt{ -> unit}$$

modifie la valeur associée à la clé, la création

$$\texttt{ref} : \alpha \texttt{->} \alpha\ \texttt{ref}$$

– rajoute une nouvelle référence dans l'environnement,
– crée une entrée associée à cette valeur dans la mémoire.

Soit M l'**état mémoire** d'un programme et E l'environnement courant ,

`let r = ref v` introduit

– un nouvel environnement $E' = \lfloor\ (\texttt{r}{=}v_r) \lhd \mathrm{E}\ \rfloor$
  où v_r est une nouvelle valeur telle que $\texttt{r} \longrightarrow_{E'} v_r$
– un nouvel état mémoire $M' = \lceil\ (v_r \leftarrow v) \lhd \mathrm{M}\ \rceil$.

on dit parfois que v_r est un **pointeur** sur v ou l'**adresse** de v.

Si $\texttt{r} \longrightarrow_E v_r$

- on consulte le contenu de v_r par $!\,\texttt{r}$ et on note $M \triangleright v_r$
- l'instruction `r := v'` ne modifie pas l'environnement courant E mais affecte la mémoire courante M : $\lceil\, (M\triangleright v_r)\leftarrow v' \,\rceil$ où $\texttt{v'} \longrightarrow_E v'$

Il n'y a pas de masquage dans la mémoire :

$$\boxed{\begin{array}{c} \lceil\, (M\triangleright v_r)\leftarrow v' \,\rceil \\ = \\ \lceil\, (\lceil\, (v_r\leftarrow v)\triangleleft M' \,\rceil\triangleright v_r)\leftarrow v' \,\rceil \\ = \\ \lceil\, (v_r\leftarrow v')\triangleleft M' \,\rceil \end{array}}$$

L'**état** d'un programme est alors donné par le couple (E, M).

Les **vecteurs** (`vect`), **enregistrements** (`record`) dont les champs sont déclarés **mutables** (`mutable`) suivent le même modèle.

```
let foo =
  let c = ref 0 in
    function () ->
      begin c := !c + 1 ; !c end ;;
```

incrémente un compteur local à la fonction à chaque appel.

```
let foo = function () ->
  let c = ref 0 in
  begin c := !c + 1 ; !c end ;;
```

Crée un nouveau compteur à chaque appel.

Que fait ? `function c -> begin c :=!c + 1; c end`

La séquence, les boucles

l'expression	`begin`
	$\mathcal{E}_1$ `;`
	...
	$\mathcal{E}_{n-1}$ `;`
	$\mathcal{E}_n$
	`end`
		équivaut à	`let void =` $\mathcal{E}_1$ `in`
			`let void = ... in`
			`let void =` $\mathcal{E}_{n-1}$ `in`
			$\mathcal{E}_n$

La boucle

`for i =` $\mathcal{I}_m$ `to` $\mathcal{I}_M$ `do` $\mathcal{E}$ `done`

introduit un nouvel identificateur `i : int`.

Dans un environnement E, si $\mathcal{I}_m \longrightarrow_E i_m$ et $\mathcal{I}_M \longrightarrow_E i_M$, on évalue successivement l'expression $\mathcal{E}$ dans les environnements $E_i = \lfloor$ (`i`=i) $\triangleleft E \rfloor$ pour $i \in i_m \ldots i_M$ et on retourne la valeur `()`.

```
let i = "i" ;;
for i = 1 to 9
do
  begin
    print_int i ;
    print_newline() ;
    i
  end
done ;;
i;;
```

l’identificateur `i` n’est ainsi pas lié en sortie de boucle !

Impératif/Fonctionnel

```
let fact = function n ->
  let res = ref 1 in
    begin
    for i = 1 to n do
      res := !res * i
    done ;
    !res
    end;;
```

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

Consommation de pile !

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

les paramètres `n` et `r` sont analogues à `i` et `!res`

Impératif/Fonctionnel

```
let fib = function n ->
  let f_0 = ref(0)
  and f_1 = ref(1)
  and temp = ref(0) in
    begin
    for i = 1 to n do
      temp := !f_0;
      f_0 := !f_1;
      f_1 := !temp + !f_1
    done;
    !f_0
    end ;;
```

```
let fib =
  let rec f_r = function
    | 0 -> (0, 1)
    | n ->
      let (f_0,f_1) =
          f_r(n - 1)
      in (f_1, f_0 + f_1)
  in fun n -
> fst(f_r n) ;;
```

Comment rendre terminal l'appel `f_r(n - 1)`

Correction de programmes

– Terminaison de fonctions récursives

```
let rec ack = function n -> function m ->
  if (n = 0)
  then
    m + 1
  else
    if (m = 0)
    then
      ack (n - 1) 1
    else
      ack (n - 1) (ack n (m - 1)) ;;
```

– Démonstration de propriétés

```
let min_list = function l -> match l with
  | [] -> failwith "Liste vide"
  | h::t -> it_list min h t ;;
```

Équivalent à

```
let min_list =
  let rec aux = function m -> function
    | [] -> m
    | h::t ->
      if (h < m)
      then aux h t
      else aux m t
  in
    function l -> match l with
      | [] -> failwith "Liste vide"
      | h::t -> aux h t ;;
```

Terminaison

Mathématique	Caml-Light
Ensemble inductif	type récursif
fonction récursive	fonction récursive

```
type arbre_bin =
  | Feuille of int
  | Noeud of (arbre_bin * arbre_bin) ;;

let rec it_arbre = fun op -> function arbre ->
  match arbre with
    | Feuille n -> n
    | Noeud (fg,fd) ->
      op (it_arbre op fg) (it_arbre op fd) ;;
```

On **n'explicite pas** les **environnements**, les **expressions**, les **valeurs** !

On **identifie** les **fermetures** et leurs **noms**, les **paramètres** et leurs **valeurs**

Par induction on a deux cas :

– l'arbre est une feuille et
  $\texttt{it\_arbre}(F, \texttt{Feuille}\ n)$ vaut n
– on suppose que A_1 et A_2 sont des `arbre_bin` tels que :
  – $\texttt{it\_arbre}(F, A_1) = a_1$
  – $\texttt{it\_arbre}(F, A_2) = a_2$

  La valeur de $\texttt{it\_arbre}(F, \texttt{Noeud}\ (A_1, A_2))$ est alors $F(a_1, a_2)$.

Correction

```
let rec append = function l ->
  match l with
    | [] -> function l -> l
    | h::t -> function l -> h::(append t l) ;;
```

A-t-on l'associativité :

$$\texttt{append}\ l_1\ (\texttt{append}\ l_2\ l_3) = \texttt{append}\ (\texttt{append}\ l_1\ l_2)\ l_3$$

- si l_1 est vide, $\texttt{append}\ l_1\ (\texttt{append}\ l_2\ l_3) = \texttt{append}\ l_2\ l_3$ et $\texttt{append}\ (\texttt{append}\ l_1\ l_2)\ l_3 = \texttt{append}\ l_2\ l_3$
- sinon $l_1 = h_1{::}l'_1$ et

$$\begin{array}{rcl} \texttt{append}\ l_1\ (\texttt{append}\ l_2\ l_3) & = & h_1 :: (\texttt{append}\ l'_1\ (\texttt{append}\ l_2\ l_3)) \\ \texttt{append}\ (\texttt{append}\ l_1\ l_2)\ l_3 & = & \texttt{append}\ (h_1 :: (\texttt{append}\ l'_1\ l_2))\ l_3 \\ & = & h_1 :: (\texttt{append}\ (\texttt{append}\ l'_1\ l_2)\ l_3) \end{array}$$

On conclut par induction en utilisant le résultat sur l'_1, l_2 et l_3.

Exemple : le pgcd

```
let rec gcd = function n -> function m ->
  if (m = 0)
  then n
  else gcd m (n mod m) ;;
```

Termine pour m et n **positifs** ?

- supposons $m < n$.
  - Si $m = 0$ le résultat est correct.
  - Sinon on a $0 \leq (n \bmod m) < m$ et $n \bmod m < m$
    et le couple (n, m) décroît strictement pour l'ordre produit.
- si $m \geq n$ alors $m < (n \bmod m)$ [a] et on utilise le point précédent.

[a] sauf si $m = n = 0$ mais le résultat reste correct

Tout diviseur commun à n et m divise le pgcd :

Si d divise n et m alors $n = dn'$ et $m = dm'$ et on a :

$$\begin{aligned} n \texttt{ mod } m &= (dn') \texttt{ mod } (dm') \\ &= d\,(n' \texttt{ mod } m') \end{aligned}$$

en supposant le **bon fonctionnement** de `mod`

et d divise ($n \texttt{ mod } m$).