

# Machines Virtuelles – MV6

## Séance 4

Peter Habermehl  
(Merci à Michele Pagani et ...)

Université Paris Cité  
UFR Informatique  
IRIF

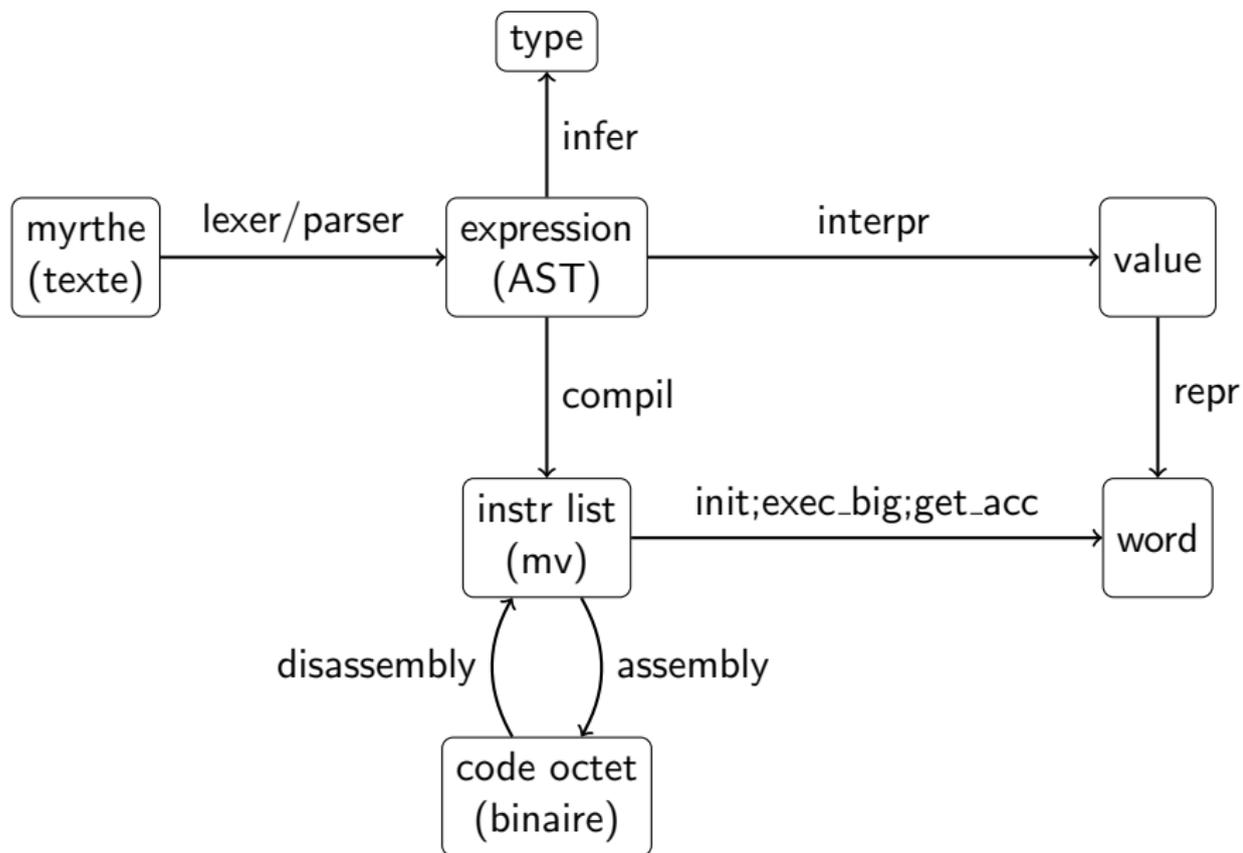
`Peter.Habermehl@irif.fr`

13 Février 2025

## Solution TP3: compil avec variables

```
1 let compil e =
2   let succ = List.map (fun (k,v) -> (k,v+1)) in
3   let rec aux e env =
4     match e with
5     | Const v ->
6       [Consti (repr v)]
7     | Unop (Succ, e) ->
8       (aux e env) @ [Push; Consti 1; Addi]
9     | Binop (Plus, e1, e2) ->
10      (aux e1 env) @ [Push] @ (aux e2 (succ env)) @ [Addi]
11     | Var x -> [Acc (List.assoc x env)]
12     | Let (x,e1,e2) ->
13       let l1 = aux e1 env in
14       let l2 = aux e2 ((x,0)::(succ env)) in
15       l1 @ [Push] @ l2 @ [Popi 1]
16     | If (e1, e2, e3) ->
17       let l1 = aux e1 env in
18       let l2 = aux e2 env in
19       let l3 = aux e3 env in
20       l1 @ [Branchif (List.length l3+2)]
21       @ l2 @ [Branch (List.length l2+1)] @ l3
22     | _ -> failwith "Students, this is your job!"
23   in
24   aux e []
```

# Résumé



## de Myrthe à OCaml

```
unop ::= not | ++  
binop ::= + | * | && | || | = | <=  
expression ::= value  
           | unop expression  
           | expression binop expression  
           | if expression then expression else expression  
           | var_id  
           | let var_id = expression in expression  
           | ( expression )
```

Question :

qu'est-ce qu'il manque pour avoir un **vrai** langage fonctionnel ?

## de Myrthe à OCaml

```
unop ::= not | ++  
binop ::= + | * | && | || | = | <=  
expression ::= value  
          | unop expression  
          | expression binop expression  
          | if expression then expression else expression  
          | var_id  
          | let var_id = expression in expression  
          | ( expression )
```

### Question :

qu'est-ce qu'il manque pour avoir un **vrai** langage fonctionnel ?

- données structurées (types algébriques, tableaux, etc...)
- définition des fonctions, applications

## de Myrthe à OCaml

```
unop ::= not | ++  
binop ::= + | * | && | || | = | <=  
expression ::= value  
           | unop expression  
           | expression binop expression  
           | if expression then expression else expression  
           | var_id  
           | let var_id = expression in expression  
           | ( expression )
```

### Question :

qu'est-ce qu'il manque pour avoir un **vrai** langage fonctionnel ?

- données structurées (types algébriques, tableaux, etc...)
- définition des fonctions, applications

Pour traiter ce type de données il faut une MV avec un tas.

# ocamlrun

## la machine virtuelle de OCaml

## Rappel

Il existe (au moins) deux compilateurs pour le langage OCaml

`ocamlopt` compile vers du code natif

(IA-32, IA-64, AMD64, POWERPC, SPARC, MIPS,  
ARM, ...)

`ocamlc` compile vers du code-octet (`.cmo`)

OCAMLRUN

(autres compilateurs: `ocamljava`, `js_of_ocaml`, ...)

La machine virtuelle `ocamlrun`:

- machine à a-pile + tas
- représentation mémoire homogène des données
- les fonctions sont des données
- son code-octet `.cmo`

## Le format de code-octet .cmo

Compilons le fichier test.ml:

```
1 let f x = x*10
2 let g y = (f (f y))
```

avec

```
ocamlc test.ml
```

## Le format de code-octet .cmo

On obtient le fichier test.cmo.

Avec la commande `xxd test.cmo`:

```
00000000: 4361 6d6c 3139 3939 4f30 3238 0000 0084  Caml19990028....
00000010: 5400 0000 0d00 0000 0000 0000 1a00 0000  T.....
00000020: 2100 0000 1a00 0000 2500 0000 0200 0000  !.....%.
00000030: 6700 0000 0a00 0000 0b00 0000 7000 0000  g.....p...
00000040: 2800 0000 0100 0000 2b00 0000 0000 0000  (.
00000050: f8ff ffff 0a00 0000 2b00 0000 0100 0000  .....+.
00000060: eeff ffff 0a00 0000 0c00 0000 4000 0000  .....@...
00000070: 0000 0000 1300 0000 0200 0000 3900 0000  .....9...
00000080: 0000 0000 8495 a6be 0000 007a 0000 0014  .....z....
00000090: 0000 004d 0000 0042 0800 0028 0024 5465  ...M...B...($Te
000000a0: 7374 5000 74a0 a092 9204 0500 7040 a0a0  stP.t.....p@..
000000b0: 0407 9030 cd22 6fa5 1046 9349 dc9c 0e7b  ...0."o..F.I...{
000000c0: 6f41 0898 a0a0 2653 7464 6c69 6290 30c2  oA...&Stdlib.0.
000000d0: 1c5d 2641 6461 b543 3218 72a5 51ea 0da0  .]&Ada.C2.r.Q...
000000e0: a038 4361 6d6c 696e 7465 726e 616c 466f  .8CamlinternalFo
000000f0: 726d 6174 4261 7369 6373 9030 3a3c a183  rmatBasics.0:<..
00000100: 8627 f776 2f49 679c e027 8ad1 4040 4040  .'v/Ig..'..@@@
00000110: 4040                                     @@
```

## Le format de code-octet .cmo

- Il existe une spécification du format .cmo:

`http://cadmium.x9c.fr/distrib/caml-formats.pdf`

- Il existe aussi des outils pour désassembler leur contenu :

`ocamldumpobj` affiche la description textuelle des instructions

`ocamlobjinfo` affiche des informations supplémentaires

- On peut même demander au compilateur d'afficher le code produit:

```
$ ocamlc -c -dinstr test.ml
```

```
$ ocaml -dinstr
```

## Le format de code-octet .cmo

```
$ ocamlc -dinstr test.ml
```

```
branch L3
```

```
L1: acc 0
```

```
push
```

```
envacc 1
```

```
apply 1
```

```
push
```

```
envacc 1
```

```
appterm 1, 2
```

```
L2: const 10
```

```
push
```

```
acc 1
```

```
multint
```

```
return 1
```

```
L3: closure L2, 0
```

```
push
```

```
acc 0
```

```
closure L1, 1
```

```
push
```

```
acc 0
```

```
push
```

```
acc 2
```

```
makeblock 2, 0
```

```
pop 2
```

```
setglobal Test!
```

## Ocamlrun: état

Son **état** est constitué de:

- un programme (liste de bytecode)
- une pile **S**
- un tas **H** (heap) où placer les données allouées
- et sept registres:
  - ① **pc**: *program counter* (adresse de la prochaine instruction)
  - ② **accu**: *accumulator*
  - ③ **sp**: *stack pointer*
  - ④ **env**: *environment pointer* (pour la fermeture courante des fonctions)
  - ⑤ **extraArgs**: (nombre d'arguments en plus d'une application)
  - ⑥ **global**: (pour des données globales)
  - ⑦ **trapSp**: (pour le gestionnaire d'exceptions)

# Ocamlrn: jeu d'instructions

Il y a presque 150 instructions,

- la plus part sont des variants similaires
- la description complète se trouve sur:

`http://cadmium.x9c.fr/distrib/caml-instructions.pdf`

On va les regrouper en 3 parties

- ① le fragment commun à Myrthe  
(gestion de la pile, arithmétique, branchement);
- ② la représentation des données structurées;
- ③ la manipulation des fonctions.

# Ocamlrun: jeu d'instructions

Il y a presque 150 instructions,

- la plus part sont des variants similaires
- la description complète se trouve sur:

`http://cadmium.x9c.fr/distrib/caml-instructions.pdf`

On va les regrouper en 3 parties

- ① le fragment commun à Myrthe  
(gestion de la pile, arithmétique, branchement);
- ② la représentation des données structurées;
- ③ la manipulation des fonctions.

## Ocamlrn: représentation des valeurs immédiates

Toute valeur OCaml est représentée par un mot machine (32/64 bits). Son premier bit détermine son utilisation:

- si c'est 1, il représente une valeur immédiate (codée sur 31/63 bits)
- si c'est 0, il représente un pointeur vers un bloc (données structurées, fonctions, cf. plus loin)

Représentation des valeurs immédiates par des mots 31/63 bits:

`int` entier signé

`char` entier correspondant au code ASCII

`bool` `false` est codé par 0, `true` par 1

`unit` `()` est codé par 0

## Ocamlrn: gestion de la pile

- `push` empile `accu`
- `pop n` dépile `n` éléments du sommet de `S` (c'est tout)
- `acc n` copie dans `accu` la valeur du `n`-ème élément de `S`

Variantes:

`acc0, ..., acc7` instructions spécialisées pour  $n \leq 7$

`pushacc n` même effet que `push`; `acc n`

`pushacc0...` cf. `acc0 ... acc7`

`assign n` déplace `accu` dans la `n`-ème case de `S`  
(il vaut `()` après)

## Ocamlrun: arithmétique

`constint n` met `n` dans `accu`

Avec des variantes:

`const0 ...const3` `constint` spécialisés pour  $n \leq 3$

`pushconstint n` même effet que `push`; `constint n`

`pushconstint0 ...3` cf. `pushconstint n`

`offsetint n` incrémente `accu` de `n`

`negint` `accu` reçoit l'opposé de sa valeur

`addint` `accu` reçoit `accu +` sommet de `S` (qui est dépilé)

`subint`, `mulint`, `divint`, `modint`, `andint`, `orint`, `(u)ltint...`

sont des opérateurs binaires qui fonctionnent de la même manière que `addint`.

`ltint`, `leint`, `gtint`, `geint...` sont des comparaisons.

## Ocamlrn: branchements

Le saut se fait par décalage (offset), mais utilisation de labels

`branch n` saute  $n$  instructions en avant  
(ou en arrière si  $n < 0$ )

`branchif n` saute  $n$  instructions si  $\text{accu} = 1$

`branchifnot n` saute  $n$  instructions si  $\text{accu} = 0$

`beq m n` saute  $n$  instructions si  $\text{accu} = m$

`bneq`, `b(u)ltint`, `bleint`, `bgtint`, `b(u)geint` sont des sauts conditionnels binaires qui fonctionnent de la même façon que `beq`

## Quiz: devine l'expression !

Quelle expression génère les code-octets suivants:

```
const 3
offsetint 4
offsetint -2
return 1
```

Solution:

$3+4-2$

---

```
const 3
push
const 2
eqint
branchifnot L1
const 4
return 1
L1: const 0
return 1
```

if 3 = 2 then 4 else 0

## Quiz: devine l'expression !

Quelle expression génère les code-octets suivants:

Solution:

```
const 3
offsetint 4
offsetint -2
return 1
```

$3+4-2$

---

```
const 3
push
const 2
eqint
branchifnot L1
const 4
return 1
L1: const 0
return 1
```

**if 3 = 2 then 4 else 0**

## Quiz: devine l'expression !

Quelle expression génère les code-octets suivants:

```
const 3
push
const 2
push
acc 0
push
acc 2
addint
return 3
```

Solution:

```
let x = 3 in let y = 2 in x + y
```

## Quiz: devine l'expression !

Quelle expression génère les code-octets suivants:

Solution:

```
const 3
push
const 2
push
acc 0
push
acc 2
addint
return 3
```

**let** x = 3 **in** **let** y = 2 **in** x + y

# Ocamlrn: jeu d'instructions

Il y a presque 150 instructions,

- la plus part sont des variants similaires
- la description complète se trouve sur:

`http://cadmium.x9c.fr/distrib/caml-instructions.pdf`

On va les regrouper en 3 parties

- ① le fragment commun à Myrthe  
(gestion de la pile, arithmétique, branchement);
- ② la représentation des données structurées;
- ③ la manipulation des fonctions.

# Ocamlrn: jeu d'instructions

Il y a presque 150 instructions,

- la plus part sont des variants similaires
- la description complète se trouve sur:

`http://cadmium.x9c.fr/distrib/caml-instructions.pdf`

On va les regrouper en 3 parties

- ① le fragment commun à Myrthe  
(gestion de la pile, arithmétique, branchement);
- ② la représentation des données structurées;
- ③ la manipulation des fonctions.

## Données structurées

En OCaml, on a plus que les types `int`, `bool` et `unit` : on a aussi

- les  $n$ -uplets `(x,y,z,...)`
- les enregistrements `{nom = "Alice"; age = 9}`
- les tableaux `[|1;2;3;4;...|]`
- les chaînes de caractères `"abc"`
- les fonctions
- les types algébriques: listes, mais aussi tout type défini par l'utilisateur:

```
type t = A | B | M of t*t | N of t
```

Les valeurs de tous ces types sont représentées en mémoire sur le `tas` (en angl. `heap`).

## Qu'est-ce que le tas?

À un nouveau processus est assigné par l'OS

- un espace mémoire constant pour stocker son code
- un espace pour ses données constantes (string etc.)
- un espace de pile extensible (mais contigu)
- à la demande, de l'espace utilisateur (`malloc`, `free`)

La zone mémoire pointée par le retour de `malloc` fait partie du tas. Le processus est chargé de la libérer après utilisation.

On mime cela dans la MV.

## Données structurées = bloc + pointeur

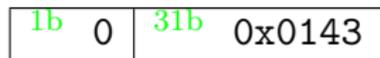
Par exemple, la donnée ('a', 'b') est représenté par:

Un **bloc** de trois cases contiguës dans le tas:



adresse dans le tas: 0x0143

et un **pointeur** (adresse) vers la premier case de ce bloc :



## Données structurées = bloc + pointeur

Par exemple, la donnée ('a', 'b') est représenté par:

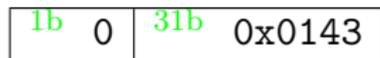
Un **bloc** de trois cases contiguës dans le tas:



adresse dans le tas: 0x0143

- **tag** : type de donnée,  
ex. string  $\mapsto$  252, float  $\mapsto$  253, fonction  $\mapsto$  247, *n*-uplet,  
enregistrements et tableau  $\mapsto$  0, types somme  $\mapsto$  0-245
- **color** : pour le ramasse miettes
- **size** : nombre cases bloc en plus de l'entête (valeurs à l'intérieur de la donnée structurée)

et un **pointeur** (adresse) vers la premier case de ce bloc :



## Données structurées = bloc + pointeur

Par exemple, la donnée ('a', 'b') est représenté par:

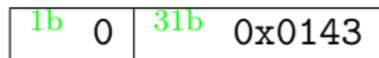
Un **bloc** de trois cases contiguës dans le tas:



adresse dans le tas: 0x0143

- **tag** : type de donnée,  
ex. string  $\mapsto$  252, float  $\mapsto$  253, fonction  $\mapsto$  247,  $n$ -uplet,  
enregistrements et tableau  $\mapsto$  0, types somme  $\mapsto$  0-245
- **color** : pour le ramasse miettes
- **size** : nombre cases bloc en plus de l'entête (valeurs à l'intérieur de la donnée structurée)

et un **pointeur** (adresse) vers la premier case de ce bloc :



## Données structurées = bloc + pointeur

Par exemple, la donnée ('a', 'b') est représenté par:

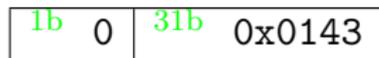
Un **bloc** de trois cases contiguës dans le tas:



adresse dans le tas: 0x0143

- **tag** : type de donnée,  
ex. string  $\mapsto$  252, float  $\mapsto$  253, fonction  $\mapsto$  247,  $n$ -uplet,  
enregistrements et tableau  $\mapsto$  0, types somme  $\mapsto$  0-245
- **color** : pour le ramasse miettes
- **size** : nombre cases bloc en plus de l'entête (valeurs à l'intérieur de la donnée structurée)

et un **pointeur** (adresse) vers la premier case de ce bloc :



## Données structurées = bloc + pointeur (II)

La donnée (3, ('a', 'b'), false) est représentée par:

Un pointeur:

0	0x0123
---	--------

vers le bloc:

0	(col.)	3	1	3	0	0x02b3	1	0
---	--------	---	---	---	---	--------	---	---

adresse: 0x0123

qui contient un autre pointeur vers le bloc:

0	(col.)	2	1	97	1	98
---	--------	---	---	----	---	----

adresse: 0x02b3

## Données structurées = bloc + pointeur (III)

Avec `ocaml -dinstr` :

```
[tag: data1 data2 data3 ....]
```

Donc

- ('a', 'b')

```
[0: 'a' 'b']
```

- (3, (25, 67))

```
[0: 3 [0: 25 67]]
```

- ('a', 10, (3, 'b'), 0)

```
[0: 'a' 10 [0: 3 'b'] 0]
```

## Types sommes: un tag par constructeur

type t = A | B of int | C | D | E of t  
          0   0          1   2   1

A, C et D: valeurs immédiates;      B et E: pointeurs.

La valeur D de type t est codée par le mot:

1b 1	31b 2
------	-------

(une valeur immédiate)

## Types sommes: un tag par constructeur

type t = A | B of int | C | D | E of t  
          0   0          1   2   1

A, C et D: valeurs immédiates;      B et E: pointeurs.

La valeur A de type t est codée par le mot:

1b	1	31b	0
----	---	-----	---

(une valeur immédiate)

## Types sommes: un tag par constructeur

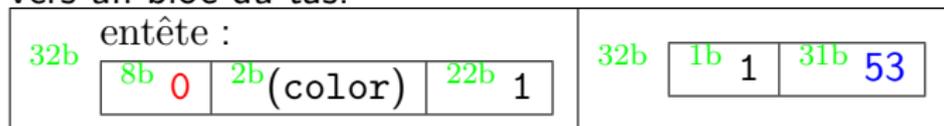
type t = A | B of int | C | D | E of t  
          0   0          1   2   1

A, C et D: valeurs immédiates;      B et E: pointeurs.

La valeur B 53 de type t est codée par un pointeur:



vers un bloc du tas:



## Types sommes: un tag par constructeur

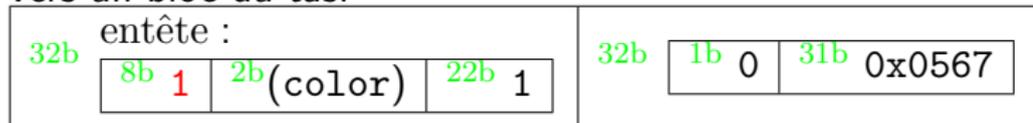
type t = A | B of int | C | D | E of t  
          0    0            1    2    1

A, C et D: valeurs immédiates;      B et E: pointeurs.

La valeur E (B 53) de type t est codée par un pointeur:

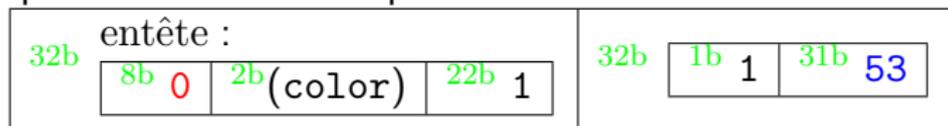


vers un bloc du tas:



adresse 0x143

qui contient un autre pointeur vers un autre bloc du tas:



adresse 0x0567

## Types sommes: un tag par constructeur

```
type t = A | B of int | C | D | E of t
         0  0           1  2   1
```

A, C et D: valeurs immédiates;      B et E: pointeurs.

avec `ocamlc -dinstr`:

```
— D                2
— A                0
— B 53             [0: 53]
— E (B 53)        [1: [0: 53]]
— E D             [1: 2]
```

## Ocamlrn: manipulation des blocs

`makeblock n k` alloue un bloc de taille `n` et de tag `k` dans le tas et y copie les `n` éléments de `accu` et du sommet de la pile `S`; met dans `accu` un pointeur vers ce bloc.

`getfield n` `accu` doit pointer vers un bloc; la `n`-ième donnée de ce bloc est mis dans `accu`

`setfield n` Le sommet de `S` est mis comme `n`-ième donnée du bloc qui est pointé par `accu`

+ des variantes

## Exemple

**let**  $x = 2$  **in**  $(x, x+1)$   
donne

```
const 2
push
acc 0
offsetint 1
push
acc 1
makeblock 2, 0
return 2
```

## Exemple

**let** x = (3,4,5) **in** **let** (y1,y2,y3) = x **in** y2  
donne

```
const [0: 3 4 5]  
push  
acc 0  
getfield 1  
push  
acc 0  
return 3
```

## Exemple

**match** [1;2;3] **with** []  $\rightarrow$  0 | x::\_  $\rightarrow$  x  
donne

```
const [0: 1 [0: 2 [0: 3 0]]]  
push  
acc 0  
branchifnot L1  
acc 0  
getfield 0  
push  
acc 0  
return 3  
L1: const 0  
return 2
```

# Ocamlrn: jeu d'instructions

Il y a presque 150 instructions,

- la plus part sont des variants similaires
- la description complète se trouve sur:

`http://cadmium.x9c.fr/distrib/caml-instructions.pdf`

On va les regrouper en 3 parties

- ① le fragment commun à Myrthe  
(gestion de la pile, arithmétique, branchement);
- ② la représentation des données structurées;
- ③ la manipulation des fonctions.