Machines Virtuelles – MV6

Séance 2

Benoît Valiron
Université Paris Diderot

2015

Retour sur la 1ère séance

On a défini une MV dont un état consiste en

- une pile S
- un registre A (l'accumulateur)
- un tableau d'instructions C
- un pointeur PC vers l'instruction courante

Jeu d'instructions constituant C :

push, pop, consti n, addi, andi, eqi,

À chaque fois, on incrémente PC de 1 unité.

Retour sur la 1ère séance

Les instructions sont codés en code-octet :

On utilise pas les instructions comme langage « processeur » pour la machine virtuelle.

Dans la machine virtuelle :
- les booléens sont codés comme `false` $= 0$ et `true` $= 1$.
- les données sont non-typés dans la machine (`int` et `bool`)

Au cours de la compilation :
Le code est compilé en instructions puis assemblé en code-octet

La machine virtuelle :
Lit un fichier contenant le code octet, le désassemble en instructions puis exécute les instructions.

Le langage Myrte

Syntaxe

On considère le langage des expressions formées par :

– les constantes true, false, 0, 1, 2, ...

– les opérations binaires $+$, $=$, $\wedge$.

– les parenthèses

et sa sémantique habituelle.

Exemple

– $2 + 2$ vaut 4

– $(1 = 0 + 1) \wedge (1 = 1) \wedge$ true vaut true

– true $+ 1$ ne vaut rien (erreur)

Syntaxe

Question : Étant donnée la chaine de caractères (sans espaces)

```
1 2 + 3 + 4 = ( 3 2 + 3 )
```

comment l'interpréter en une expression ?

Deux étapes :

– Lexing : passer d'une chaine de caractères à une liste de symboles.

– Parsing : donner du sens à ces symboles.

Lexing

```
1 2 + 3 + 4 = ( 3 2 + 3 )
```

Étape 1 : Découpage en sous-chaines.

```
"1"  "2+"  "3"  "+"  "4=("  "32+"  "3)"

"1"  "2" "+"  "3"  "+"  "4"  "=" "("  "3" "2"  "+"  "3"  ")"

"12" "+"  "3"  "+"  "4"  "=" "("  "32"  "+"  "3"  ")"
```

Lexing

```
1 2 + 3 + 4 = ( 3 2 + 3 )
```

Étape 2 : Interprétation de chaque sous-chaine.

```
type token =
  INT of int | PLUS | AND | EQ | LPAREN | RPAREN | EOL
```

Propositions :

- `INT(1), INT(2), PLUS, INT(3), PLUS, INT(4), EQ, LPAREN, INT(3), INT(2), PLUS, INT(3), RPAREN, EOL`
- `INT(12), PLUS, INT(3), PLUS, INT(4), EQ, LPAREN, INT(32), PLUS, INT(3), RPAREN, EOL`

Parsing

```
12 + 3 + 4 = ( 32 + 3 )
```

Quelle « expression » cherche-t-on ici ?

```
type expr =
| ConsInt of int
| ConsBool of bool
| Add of expr * expr
| And of expr * expr
| Eq of expr * expr
```

Parsing

```
12 + 3 + 4 = ( 32 + 3 )
```

Quelle « expression » cherche-t-on ici ?

```
type value =
| Int of int
| Bool of bool

type binop = Add | Eq | And

type expr =
| Const of value
| Binop of binop * expr * expr
```

Plusieurs choix possibles

Parsing

```
(12 + 3) + (4  = ( 32 + 3 ))
```

Quelle « expression » cherche-t-on ici ?

```
Binop(Add,
      Binop(Add,
            Const(Int 12),
            Const(Int 3)),
      Binop(Eq,
            Const(Int 4)
            Binop(Add,
                  Const(Int 32),
                  Const(Int 3))))
```

Règles de précédance (Poids des opérateurs).

Parsing

```
((12 + 3) + 4) = ( 32 + 3 )
```

Quelle « expression » cherche-t-on ici ?

```
Binop(Eq,
      Binop(Add,
            Binop(Add,
                  Const(Int 12),
                  Const(Int 3)),
            Const(Int 4)),
      Binop(Add,
            Const(Int 32),
            Const(Int 3)))
```

Parsing

```
(12 + (3 + 4)) = ( 32 + 3 )
```

Quelle « expression » cherche-t-on ici ?

```
Binop(Eq,
      Binop(Add,
            Const(Int 12),
            Binop(Add,
                  Const(Int 3),
                  Const(Int 4)))
      Binop(Add,
            Const(Int 32),
            Const(Int 3)))
```

Règles d'associations (à droite ou à gauche).

Parsing

```
12 +  3 + 4   = ( 32 + 3 )
```

Il y a donc des choix à faire pour parser une expression

- Choix de la représentation des expression
- Choix dans la lecture : précédance, association.

Ocaml possède un outil de génération de lexer/parser : `ocamllex` et `ocamlyacc`.

(Mais nous ne verrons pas comment l'utiliser)

Arbre de syntaxe

Structure d'une expression = arbre de syntaxe.

– Constructeurs de type = noeud de l'arbre

– Valeur = feuilles

```
type value =
| Int of int
| Bool of bool
```

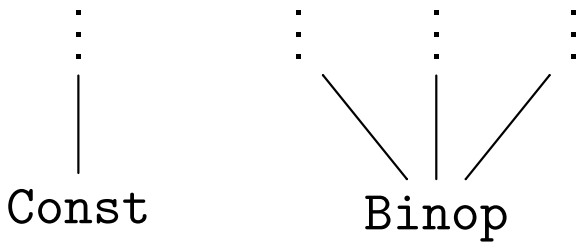
$\longmapsto$


```
type binop =
   Add | Eq | And
```

$\longmapsto$ Des feuilles de l'arbre

```
type expr =
| Const of value
| Binop of
     binop * expr * expr
```

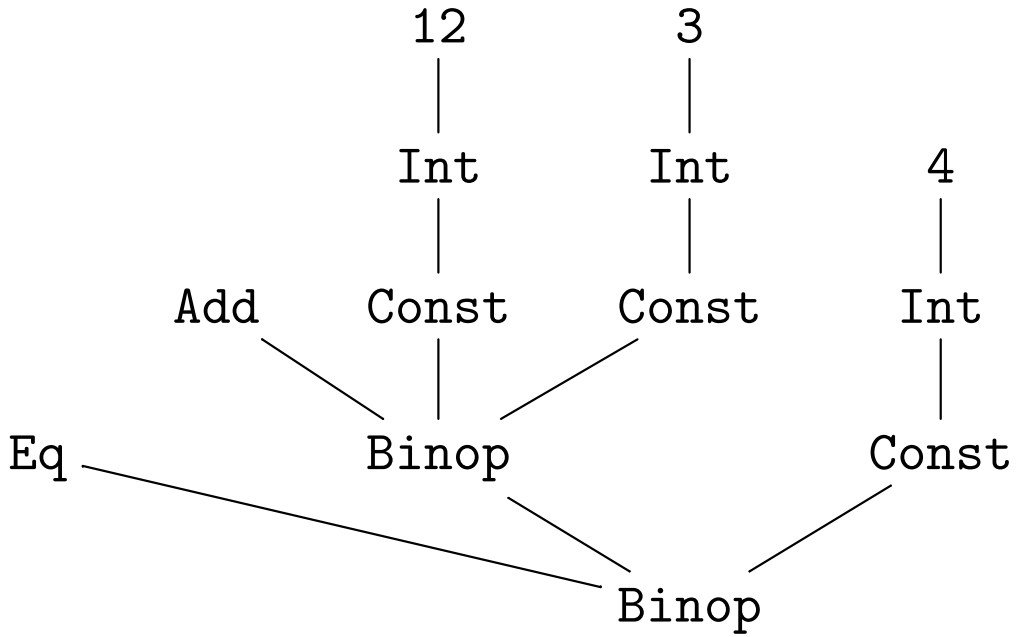
$\longmapsto$



Arbre de syntaxe

Exemple : `12 + 3 = 4`

Avec la syntaxe utilisée plus haut :



Arbre de syntaxe

Exemple :

```
type n = Z | S of n
```

Quel arbre pour `S(S(S Z))` ?

Arbre de syntaxe

Exemple :

```
type l = Nil | Cons of int * l
```

Quel arbre pour `Cons(1,Cons(2,Cons(3,Nil)))` ?

Arbre de syntaxe

Exemple :

```
type t = Nil | Node of t * int * t
```

Quel arbre pour
`Node(Node(Nil,1,Nil),2,Node(Node(Nil,3,Nil),4,Nil))` ?

Arbre de syntaxe

Exemple : ajouter à `expr` la construction

```
if(...)then(...)else(...)
```

```
type expr =
| Const of value
| Binop of binop * expr * expr
```

Arbre de syntaxe

Exemple : ajouter à `expr` la construction

```
if(...)then(...)else(...)
```

```
type expr =
| Const of value
| Binop of binop * expr * expr

| If of expr * expr * expr
```

Arbre de syntaxe

Exemple : ajouter à `expr` la construction

```
let x = ... in ...
```

```
type expr =
| Const of value
| Binop of binop * expr * expr
| If of expr * expr * expr
```

Arbre de syntaxe

Exemple : ajouter à `expr` la construction

```
let x = ... in ...
```

```
type expr =
| Const of value
| Binop of binop * expr * expr
| If of expr * expr * expr

| Let of string * expr * expr
```

Arbre de syntaxe

Exemple : ajouter à `expr` la construction

```
let x = ... in ...
```

```
type expr =
| Const of value
| Binop of binop * expr * expr
| If of expr * expr * expr

| Let of var    * expr * expr

type var = string
```

Arbre de syntaxe

Exemple : ajouter à `expr` la construction

```
let f x1 x2 ... xn = ... in ...
```

```
type expr =
| Const of value
| Binop of binop * expr * expr
| If of expr * expr * expr
| Let of var * expr * expr
```

Arbre de syntaxe

Exemple : ajouter à `expr` la construction

```
let f x1 x2 ... xn = ... in ...
```

```
type expr =
| Const of value
| Binop of binop * expr * expr
| If of expr * expr * expr
| Let of var * expr * expr

| Letf of    ????      * expr * expr
```

Arbre de syntaxe

Exemple : ajouter à `expr` la construction

```
let f x1 x2 ... xn = ... in ...
```

```
type expr =
| Const of value
| Binop of binop * expr * expr
| If of expr * expr * expr
| Let of var * expr * expr

| Letf of var * (var list) * expr * expr
```

La représentation en arbre (concret) est moins claire, mais cela reste le langage de description d'un arbre de syntaxe abstrait.

Différence code-octet / instructions

Les instructions de la MV forment un langage.

Objectifs dans le choix du code-octet :

- Représentation de toutes les instructions
- nécessitant le moins de lexing/parsing possible
- en particulier pour les valeurs : “12” versus 12

Les choix pour Myrte :

- Taille fixe d'encodage pour chaque instruction (un ≪ mot ≫).
- Même structure pour chaque mot : une paire ⟨argument, op-code⟩.

On aurait pu (ou du) faire des choix différents. Par exemple, si une instruction avait un nombre arbitraire d'argument.

Le langage Myrte

Mini-langage, avec pour arbre de syntaxe

```
type value =
| Int of int
| Bool of bool

type binop = Add | Eq | And

type expr =
| Const of value
| Binop of binop * expr * expr
```

Le langage Myrte

Un interpréteur peut être écrit comme suit

```
let rec interp : expr -> value = function
| Const v              -> v
| Binop (b, e1, e2) ->
      match b, interp e1, interp e2 with
        | Add, Int i,  Int j  -> Int  (i + j)
        | Eq,  Int i,  Int j  -> Bool (i = j)
        | And, Bool i, Bool j -> Bool (i && j)
        | _ -> failwith "ill-formed expression"
```

Le langage Myrte

Un interpréteur peut être écrit comme suit

```
let rec interp : expr -> value = function
| Const v              -> v
| Binop (b, e1, e2) ->
      match b, interp e1, interp e2 with
        | Add, Int i,  Int j  -> Int  (i + j)
        | Eq,  Int i,  Int j  -> Bool (i = j)
        | And, Bool i, Bool j -> Bool (i && j)
        | _ -> failwith "ill-formed expression"
```

```
Binop(Eq,Binop(Add,Const(Int 2),Const(Int 3)),Const(Int 5))
```

Le langage Myrte

Un interpréteur peut être écrit comme suit

```
let rec interp : expr -> value = function
| Const v              -> v
| Binop (b, e1, e2) ->
      match b, interp e1, interp e2 with
        | Add, Int i,  Int j  -> Int  (i + j)
        | Eq,  Int i,  Int j  -> Bool (i = j)
        | And, Bool i, Bool j -> Bool (i && j)
        | _ -> failwith "ill-formed expression"
```

```
Binop(Add,Binop(Eq,Const(Int 2),Const(Int 3)),Const(Int 5))
```

Typage de Myrte

Problème

L'évaluation de certaines expressions peut échouer (p.e. `2 + true`)

Question

Peut-on analyser une expression pour détecter les cas d'échec statiquement, c'est à dire sans évaluer sa valeur ?

Solution : Le typage.

Well-typed programs can't go wrong – Milner (1978)

Typage de Myrte

Un typage est une annotation statique de code qui donne des garanties sur l'exécution :

```
check : expr -> bool
```

répond `true` si l'expression est typable, `false` sinon.

Pour Myrte, en l'état actuel, le système de type est simple :

```
type tp = TInt | TBool
```

On veut que

Si `check e = true` alors `eval e` n'échoue pas.

Typage de Myrte

```
type tp = TInt | TBool

let rec infer : expr -> tp = function
| Const (Int _)  -> TInt
| Const (Bool _) -> TBool
| Binop (b, e1, e2) ->
   match b, infer e1, infer e2 with
     | Add, TInt,  TInt  -> TInt
     | Eq,  TInt,  TInt  -> TBool
     | And, TBool, TBool -> TBool
     | _ -> failwith "expression mal typee"

let check e =
      try ignore (infer e); true
      with Failure _ -> false
```

Typage de Myrte

Quelques remarques

- Dans le cas de Myrte, la fonction `check` « est comme » l'évaluateur : elle « évalue » aussi la fonction
- Avec un langage plus complexe (tests, fonctions, récursion), le typage prend tout son sens (p.e. ocaml !)
- Le type n'est pas forcément une garantie tout-risque :
  - C (n'assure rien...)
  - java (null pointer exception ?)
- Le typage peut-être dynamique (python, lisp)

Compilation en instructions

- Sur un exemple comme celui-ci, un interprète fait l'affaire
- Pour un langage plus riche, il devient beaucoup trop lent
- Il faut « linéariser » l'expression en code, et au besoin l'optimiser

Compilation en instructions

On doit d'abord fixer des conventions d'encodage :

Étape 1 : Encodage des valeurs en états

- Quand la machine a fini de traiter l'encodage d'une expression `e`
  - le résultat est lu dans A
  - `pc` indique l'instruction juste après l'encodage de `e`
  - la pile est restaurée
- Les entiers sont encodés par... eux-même
- Les booléens true et false sont codés resp. par 1 et 0

Note : `false` and 0 sont tous deux codés pareil.

$\rightarrow$ La MV est non-typée !

Compilation en instructions

On doit d'abord fixer des conventions d'encodage :

Étape 1 : Encodage des valeurs en états

```
let repr : value -> int = function
     | Bool true -> 1
     | Bool false -> 0
     | Int i -> i
```

Compilation en instructions

Étape 2 : Compilation des expressions en instructions

Écrire une fonction

```
val compil : expr -> instr list
```

telle que pour toute expression e,

```
eval (compil e) = repr (interp e)
```

avec

```
let eval c =
      let s = machine (init c) in s.acc
```

Compilation en instructions

Étape 2 : Compilation des expressions en instructions

On utilise l'**invariant** qui dit que quand la machine a fini de traiter l'encodage d'une expression,

- **le résultat est lu dans A**
- **le PC pointe juste après l'encodage**
- **la pile est restaurée**

```
let rec compil : expr -> instr list = function
  | Const v            -> [Consti (repr v)]
  | Binop (o, e1, e2) -> compil e1   @
                         [Push]      @
                         compil e2   @
                         [op o; Pop]

let op = function And -> Andi | Add -> Addi | Eq -> Eqi
```

Compilation en instructions

Exemple avec

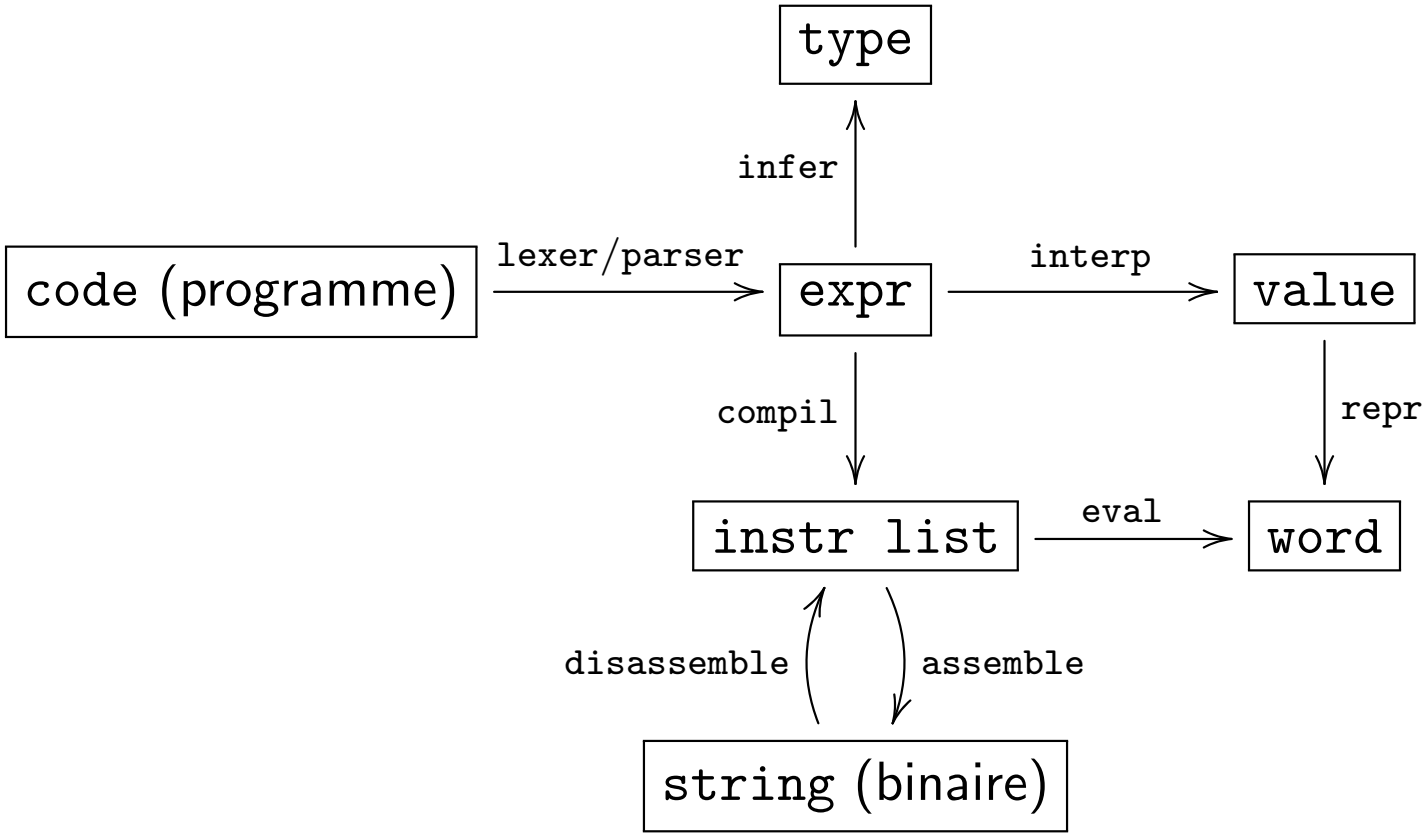
```
(1 + 2) + 3 = 7
```

Compilation en instructions

Remarques

- La compilation d'une expression place le résultat dans A
- L'exécution de son code restaure la pile telle qu'il l'a trouvé (mais écrase A)
- À chaque expression correspond (au moins) une série d'instructions
- Un programme qui correspond à une expression bien typée n'échoue pas
- Un série d'instructions ne correspondant pas à un programme peut échouer

Résumé


type
infer
code (programme)
lexer/parser
expr
interp
value
compil
repr
instr list
eval
word
disassemble
assemble
string (binaire)