Machines Virtuelles – MV6
CM 3
(Séance 3)

Peter Habermehl (merci aux auteurs)

Université Paris Cité
UFR Informatique
Institut de Recherche en Informatique Fondamentale

Peter.Habermehl@irif.fr

15 février 2024

Rappel des derniers cours/TP

On a défini une MV dont un état consiste en:

- une pile S
- un registre A (l'accumulateur)
- un tableau d'instructions P
- un registre PC, pointeur vers l'instruction suivante

Jeu d'instructions constituant P:
`Push`, `Pop`, `Consti n`, `Addi`, `Multi`, `Ori`, `Eqi`, `Acc n`, `Popi n`, `Branch n`, `Branchif n`, `Assign n`, `Leqi`

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.

Rappel des derniers cours/TP

Une instruction est codé sur trois octets:

Instruction 1			Instruction 2			Instruction 3			...
opcode	signe	valeur	opcode	signe	valeur	opcode	signe	valeur	...

Les op-codes sont définis comme suit:

`Push` $\mapsto$ 00000000	`Addi` $\mapsto$ 00000001	`Eqi` $\mapsto$ 00000010
`Ori` $\mapsto$ 00000011	`Consti` $\mapsto$ 00000100	`Pop` $\mapsto$ 00000101
`Acc` $\mapsto$ 00000110	`Popi` $\mapsto$ 00000111	`Branch` $\mapsto$ 00001000
`Branchif` $\mapsto$ 00001001	`Assign` $\mapsto$ 00001010	`Multi` $\mapsto$ 00001011
`Leqi` $\mapsto$ 00001100		

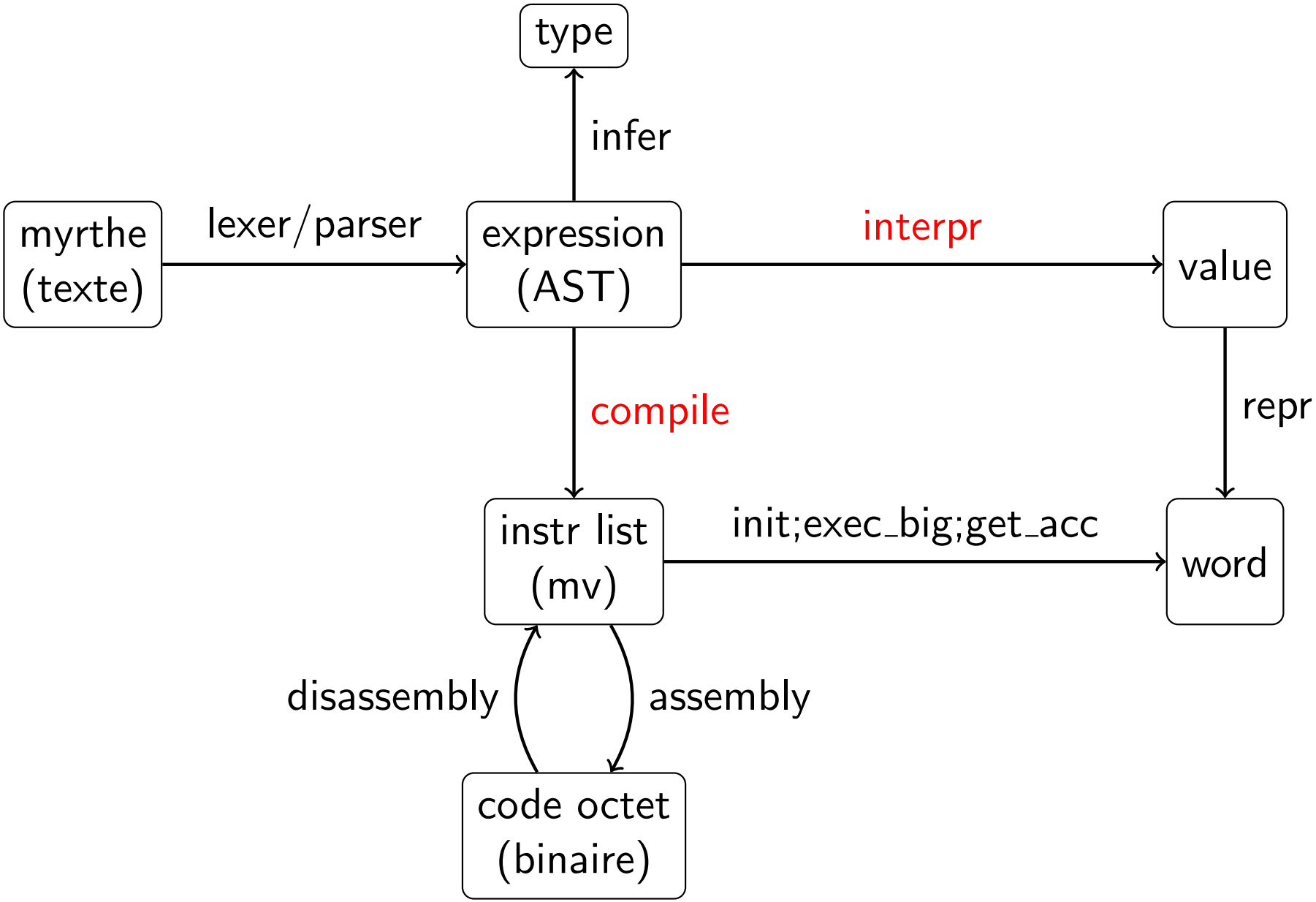
Les deux octets “signe” et “valeur” ne sont utilisés que pour les instructions de la MV avec arguments:

- le signe est 0 si l'argument est positif ou nul, 1 sinon.
- La valeur est l'argument, en valeur absolue.

Séance 3

Aller d'un langage (très) simple de haut niveau (Myrthe) vers la machine virtuelle

Traitement de Myrthe, résumé



Le langage Myrthe

La grammaire de Myrthe

value ≡ `0` | `[1-9][0-9]*` | `-[1-9][0-9]*` | `true` | `false`

var_id ≡ `[a-z][A-Z a-z 0-9 _]*`

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* `)`

Attention: cette grammaire est ambiguë. Il faut fixer des règles de priorité.

Exemples

```
(3+4)*15

3+4*15

(3+-4 = 15) || (4 <= 9)

let x = x+1 in if 4 <= x then x else ++x

let x=3 in let x = 4+x in if 4 <= x then x else ++x

let x=3 in let x = 4+x in (if 4 <= x then x else ++x )*2
```

Implémentation en OCaml

Abstract Syntax Tree

```
if 3+4<=2 then 4+2 else 0
```

- comment representer un programme de Myrthe ?
  - ce n'est pas une liste d'instructions !
  - plusieurs sous-expressions en parallel

- programmation structurée:

$$\begin{aligned} \text{un programme} &\mapsto \text{un arbre} \\ \text{constructeurs du langage} &\mapsto \text{noeuds internes} \\ \text{constantes} &\mapsto \text{feuilles} \end{aligned}$$

Abstract Syntax Tree

```
if 3+4<=2 then 4+2 else 0
```

- comment representer un programme de Myrthe ?
  - ce n'est pas une liste d'instructions !
  - plusieurs sous-expressions en parallel
- programmation structurée:

$$\begin{aligned} \text{un programme} &\mapsto \text{un arbre} \\ \text{constructeurs du langage} &\mapsto \text{noeuds internes} \\ \text{constantes} &\mapsto \text{feuilles} \end{aligned}$$

Abstract Syntax Tree

```
if 3+4<=2 then 4+2 else 0
```

- comment representer un programme de Myrthe ?
  - ce n'est pas une liste d'instructions !
  - plusieurs sous-expressions en parallel
- programmation structurée:

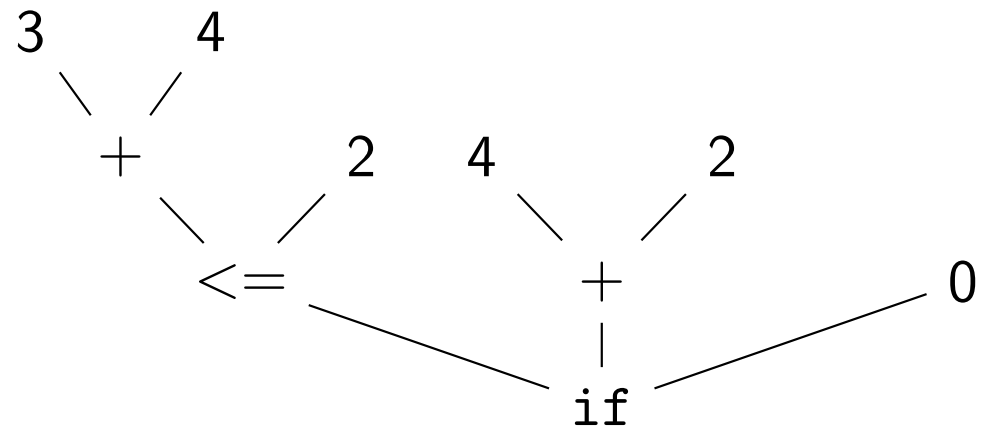
un programme	$\mapsto$	un arbre
constructeurs du langage	$\mapsto$	noeuds internes
constantes	$\mapsto$	feuilles

Abstract Syntax Tree

```
if 3+4<=2 then 4+2 else 0
```

- comment representer un programme de Myrthe ?
  - ce n'est pas une liste d'instructions !
  - plusieurs sous-expressions en parallel
- programmation structurée:

un programme	$\mapsto$	un arbre
constructeurs du langage	$\mapsto$	noeuds internes
constantes	$\mapsto$	feuilles



Exercice

Dessinez les AST des expressions suivantes:

```
(3+4)*15
```

```
3+4*15
```

```
(3+-4 = 15) || (4 <= 9)
```

```
let x = x+1 in if 4 <= x then x else ++x
```

```
let x=3 in let x = 4+x in if 4 <= x then x else ++x
```

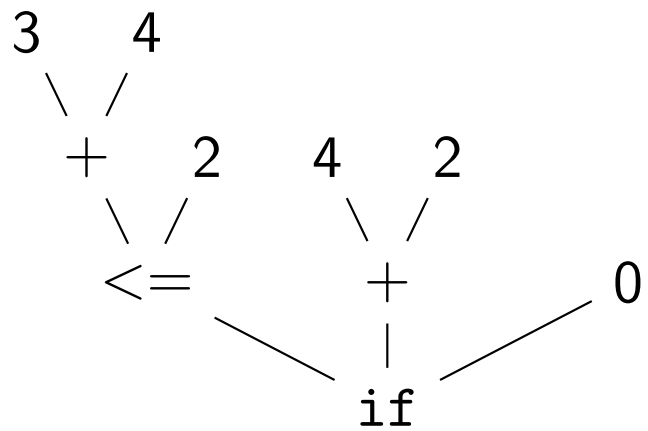
```
let x=3 in let x = 4+x in (if 4 <= x then x else ++x)*2
```

AST en OCaml

```
1  type value = Int of int | Bool of bool
2
3  and var_id = string
4
5  and unop = Not | Succ
6
7  and binop = Plus | Times | Et | Or | Eq | Leq | Minus
8  and expression =
9     | Const of value
10    | Unop of unop*expression
11    | Binop of binop*expression*expression
12    | If of expression*expression*expression
13    | Var of var_id
14    | Let of var_id*expression*expression
```

Example

```
if 3+4<=2 then 4+2 else 0
```




```
1   If (
2       Binop (
3           Leq,
4           Binop (
5               Plus,
6               Const (Int 3),
7               Const (Int 4)
8             ),
9           Const (Int 2)
10        ),
11      Binop (
12          Plus,
13          Const (Int 4),
14          Const (Int 2)
15        ),
16      Const (Int 0)
17    )
```

Exercice

Ecrivez les valeurs de type `expression` des expressions suivantes:

```
(3+4)*15
```

```
3+4*15
```

```
(3+-4 = 15) || (4 <= 9)
```

```
let x = x+1 in if 4 <= x then x else ++x
```

```
let x=3 in let x = 4+x in if 4 <= x then x else ++x
```

```
let x=3 in let x = 4+x in (if 4 <= x then x else ++x)*2
```

Lexing, parsing

Voir cours Grammaires et Analyse syntaxique !

Question: étant donnée une chaine de caractères

```
12 + 3 +4= (32 +5)
```

comment la transformer dans une valeur `expression` ?

Deux étapes:

- Lexing : passer d'une chaine de caractères à une liste de symboles.
- Parsing : ranger ces symboles dans un AST.

Lexing, parsing

Voir cours Grammaires et Analyse syntaxique !

Question: étant donnée une chaine de caractères

```
12 + 3 +4= (32 +5)
```

comment la transformer dans une valeur `expression` ?

Deux étapes:

- Lexing : passer d'une chaine de caractères à une liste de symboles.
- Parsing : ranger ces symboles dans un AST.

Lexing

```
12 + 3 +4= (32 +5)
```

Étape 1 : Découpage en sous-chaines.

```
"1" "2+" "3" "+" "4=(" "32+" "3)"
"1" "2" "+" "3" "+" "4" "=" "(" "3" "2" "+" "3" ")"
"12" "+" "3" "+" "4" "=" "(" "32" "+" "3)"
```

Étape 2: Transformer chaque sous-chaîne en lexème (*token*).

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

Propositions :
premier découpage n'est pas possible

```
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
```

ocamllex

- il y a des programmes générant automatiquement des lexeurs, étant donnée une specification sous forme d'expressions rationnelles.
- Il y a des générateurs de lexeurs en OCaml
  - ocamllex utilise le module Lexing de OCaml
  - voir: le manuel de OCaml
  - Nous ne verrons pas comment l'utiliser ici

    voir cours GRAMMAIRES ET ANALYSE SYNTAXIQUE

ocamllex

- il y a des programmes générant automatiquement des lexeurs, étant donnée une specification sous forme d'expressions rationnelles.
- Il y a des générateurs de lexeurs en OCaml
  - ocamllex utilise le module Lexing de OCaml
  - voir: le manuel de OCaml
  - Nous ne verrons pas comment l'utiliser ici

    voir cours GRAMMAIRES ET ANALYSE SYNTAXIQUE

Parsing

```
INT 12,PLUS,INT 3,PLUS,INT 4,EQ,LPAREN,INT 32,PLUS,INT 3,RPAREN,EOL
```

- Comment traduire flux de lexemes dans valeur `expression` ?
- il y a plusieurs choix possibles:

- règles d'association (à droite ou à gauche)
- règles de précédance (Poids des opérateurs).

Parsing

```
INT 12,PLUS,INT 3,PLUS,INT 4,EQ,LPAREN,INT 32,PLUS,INT 3,RPAREN,EOL
```

- Comment traduire flux de lexemes dans valeur `expression` ?
- il y a plusieurs choix possibles:

```
1   Binop (
2       Eq,
3       Binop (
4           Plus,
5           Const (Int 12),
6           Binop (
7               Plus,
8               Const (Int 3),
9               Const (Int 4)
10            )),
11      Binop (
12          Plus,
13          Const (Int 32),
14          Const (Int 3)
15        ))
```

- règles d'association (à droite ou à gauche)
- règles de précédance (Poids des opérateurs).

Parsing

```
INT 12,PLUS,INT 3,PLUS,INT 4,EQ,LPAREN,INT 32,PLUS,INT 3,RPAREN,EOL
```

- Comment traduire flux de lexemes dans valeur `expression` ?
- il y a plusieurs choix possibles:

```
1   Binop (
2       Eq,
3       Binop (
4           Plus,
5           Binop (
6               Plus,
7               Const (Int 12),
8               Const (Int 3)
9             ),
10          Const (Int 4)),
11      Binop (
12          Plus,
13          Const (Int 32),
14          Const (Int 3)
15        ))
```

- règles d'association (à droite ou à gauche)
- règles de précédance (Poids des opérateurs).

Parsing

```
INT 12,PLUS,INT 3,PLUS,INT 4,EQ,LPAREN,INT 32,PLUS,INT 3,RPAREN,EOL
```

- Comment traduire flux de lexemes dans valeur `expression` ?
- il y a plusieurs choix possibles:

```
1   Binop (
2       Plus,
3       Binop (
4           Plus,
5           Const (Int 12),
6           Const (Int 3)
7         ),
8       Binop (
9           Eq,
10          Const (Int 4),
11          Binop (
12              Plus,
13              Const (Int 32),
14              Const (Int 3)
15            ))
16    )
```

- règles d'association (à droite ou à gauche)
- règles de précédance (Poids des opérateurs).

ocamlyacc/Menhir

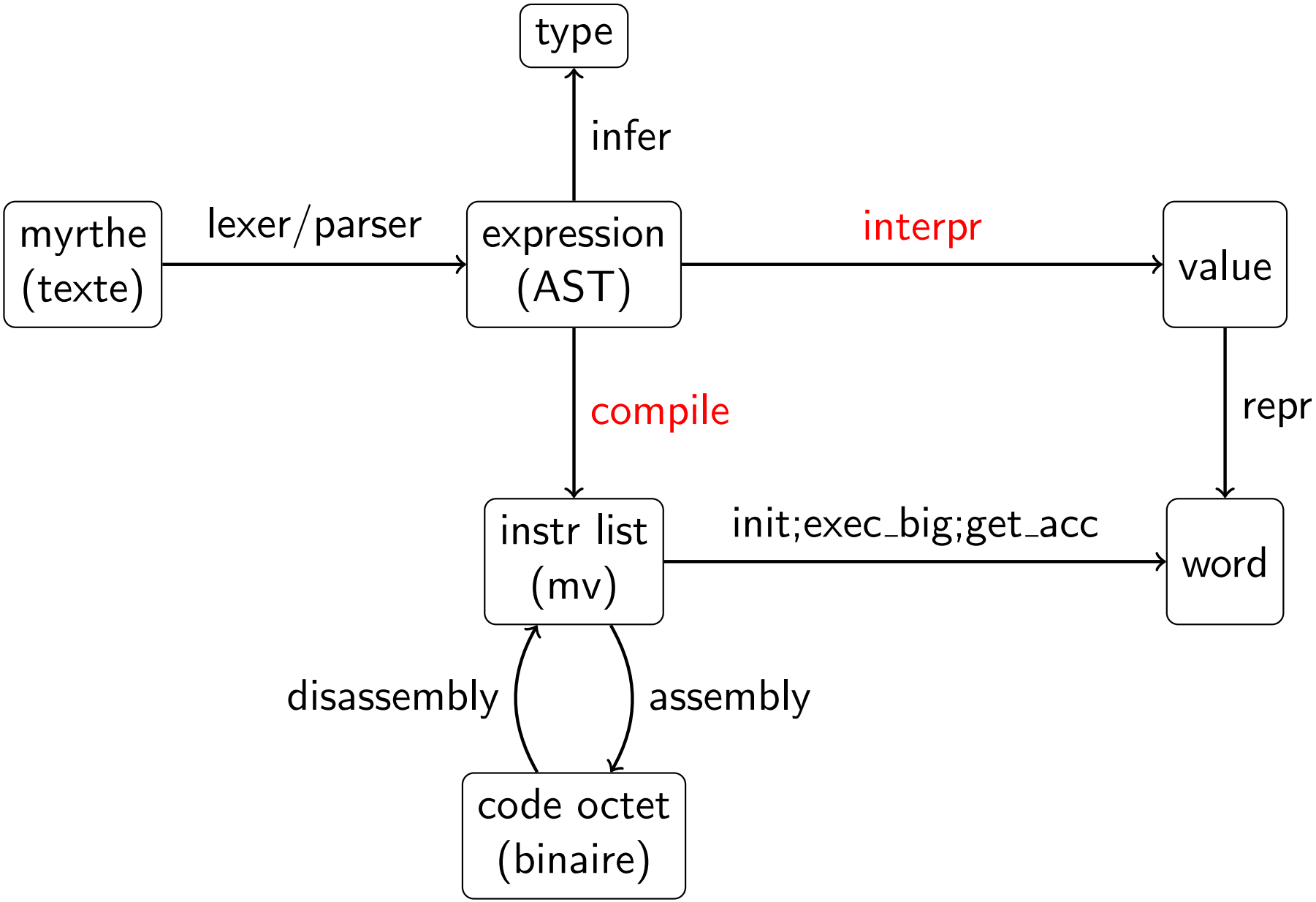
- Étant donnée une specification sous forme de grammaire hors-contexte (*context-free*), les analyseur syntaxiques (parseurs, *parsers*) peuvent aussi être générés automatiquement.
- en OCaml
  - ocamlyacc/Menhir utilisent le module Lexing de OCaml
  - voir: le manuel de OCaml
  - Nous ne verrons pas comment l'utiliser ici
    voir cours GRAMMAIRES ET ANALYSE SYNTAXIQUE
  - Nous utilisons directement l'AST.

ocamlyacc/Menhir

- Étant donnée une specification sous forme de grammaire hors-contexte (*context-free*), les analyseur syntaxiques (parseurs, *parsers*) peuvent aussi être générés automatiquement.
- en OCaml
  - ocamlyacc/Menhir utilisent le module Lexing de OCaml
  - voir: le manuel de OCaml
  - Nous ne verrons pas comment l'utiliser ici
    voir cours GRAMMAIRES ET ANALYSE SYNTAXIQUE
  - Nous utilisons directement l'AST.

Interprétation de Myrthe

Traitement de Myrthe, résumé



Interpréteur Myrthe

- On peut implémenter d'abord les opérateurs unaires et binaires,
  par exemple:

```
1  let udecode (o : unop) (v: value) : value =
2    match o,v with
3    | Not, Bool x -> Bool (not x)
4    | Succ, Int x -> Int (x+1)
5    | _, _ -> failwith "Erreur de typage"
```

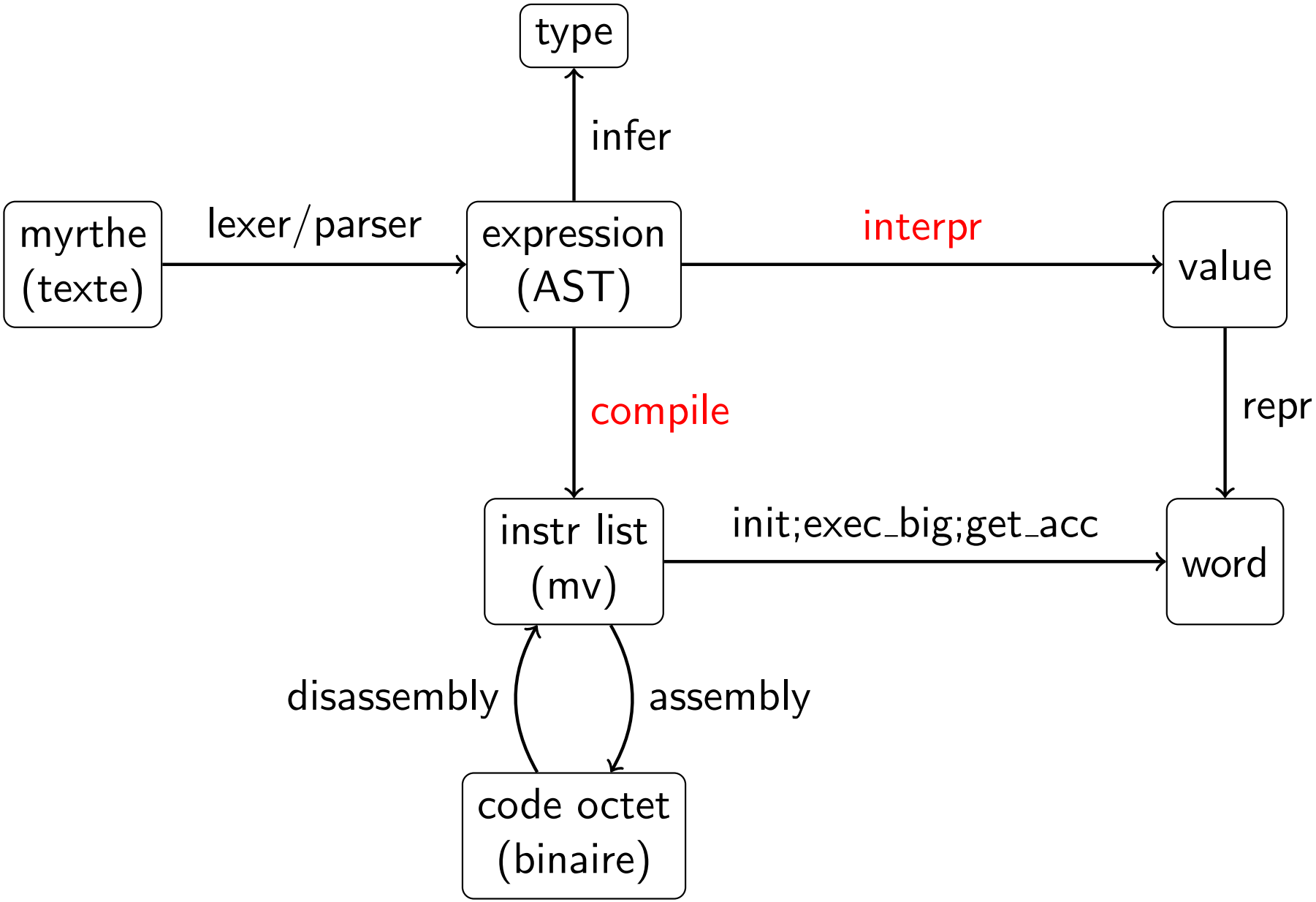
- remarquez l'erreur de typage,
  - détecté au cours de l'évaluation (*run-time*)
  - on verra comment le detecter avant d'évaluer le programme (*compile-time*)
- Pour évaluer une expression Myrthe on a besoin d'un environnement qui associe un valeur à des variables

Interpréteur Myrthe

```
let interpr (e : expression) : value =
  let rec aux e (env : (var_id*value) list) = match e with
    | Const v -> v
    | Unop (o,e1) ->
      let v = aux e1 env
      in udecode o v
    | Binop (o,e1,e2) ->
      let v1 = aux e1 env and v2 = aux e2 env
      in bdecode o v1 v2
    | If (e1,e2,e3) ->
      begin
        match (aux e1 env) with
          Bool b -> if b then aux e2 env else aux e3 env
        | Int _ -> failwith "Erreur de typage"
      end
    | Var x ->
      begin
        try (List.assoc x env) with (* on récupère la valeur de x
          Not_found -> failwith "Variable non definie"
      end
    | Let (x,e1,e2) ->
      let v = aux e1 env in (* évaluation de e1 *)
      let env = (x,v)::env in (* ajoute x à env avec valeur de e1
      aux e2 env
  in aux e []  (* environnement vide au départ *)
```

Compilation de Myrthe dans le code octet de la MV

Traitement de Myrthe, résumé



Compilation

- Pour un langage comme celui-ci, un interprète fait l'affaire
- Pour un langage plus riche, il devient trop lent
- linéariser l'expression en code, et au besoin l'optimiser
- Ici, nous compilons dans le code de la machine virtuelle présentée avant

Compilation

Étape 1. Fixer des conventions d'encodage:

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

Étape 2. Compilation des expressions en listes d'instructions :

```
val compile : expression -> instr list
```

telle que pour toute expression e:

```
get_acc (
    exec_big (
        init
            (Array.of_list (compile e))
            [ ]
    )
) = repr (interpr e)
```

Compilation

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

- le résultat est dans **A**
- le **PC** pointe juste après l'encodage
- la **pile** est restaurée (la même qu'avant)

```
let rec compile = function
  | Const v ->
    [Consti (repr v)]
  | Unop (Succ, e) ->
    (compile e) @ [Push; Consti 1; Addi]
  | Binop (Plus, e1, e2) ->
    (compile e1) @ [Push] @ (compile e2) @ [Addi]
  | Binop (Eq, e1, e2) ->
    (compile e1) @ [Push] @ (compile e2) @ [Eqi]
  | If (e1, e2, e3) ->
    let l1 = compile e1 in
    let l2 = compile e2 in
    let l3 = compile e3 in
    l1 @ [Branchif (List.length l3+2)]
    @ l3 @ [Branch (List.length l2+1)]
    @ l2
```

Example

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

Compilation

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
- Une série d'instructions ne correspondant pas à un programme peut échouer

Typage de Myrthe

Typage

Problème

Qu'est-ce qui se passe si on compile cette expression ?

```
2+true
```

- le comportement du code compilé simule l'interpréteur sur les expressions correctes,
  - mais sur des expressions incorrectes on peut avoir un comportement incontrôlable.
- Comment détecter ces pathologies avant d'exécuter le code (c.à-d. statiquement) ?
- Solution: le typage.

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

Typage

Problème

Qu'est-ce qui se passe si on compile cette expression ?

```
2+true
```

- le comportement du code compilé simule l'interpréteur sur les expressions correctes,
  - mais sur des expressions incorrectes on peut avoir un comportement incontrôlable.
- Comment détecter ces pathologies avant d'exécuter le code (c.à-d. statiquement) ?
- Solution: le typage.

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

Typage

Problème

Qu'est-ce qui se passe si on compile cette expression ?

```
2+true
```

- le comportement du code compilé simule l'interpréteur sur les expressions correctes,
  - mais sur des expressions incorrectes on peut avoir un comportement incontrôlable.
- Comment détecter ces pathologies avant d'exécuter le code (c.à-d. statiquement) ?
- Solution: le typage.

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

Un exemple de typeur

```
1  type type_myrthe = Integer | Boolean
2
3
4  let rec infer (e: expression) : type_myrthe =
5    match e with
6      Const (Int _) -> Integer
7    | Const (Bool _) -> Boolean
8    | Unop (Not,e) ->
9       begin
10        match infer e with
11          Integer -> failwith "Type error"
12        | Boolean -> Boolean
13      end
14   | Unop (Succ,e) ->
15      begin
16        match infer e with
17          Integer -> Integer
18        | Boolean -> failwith "Type error"
19      end
20   | _ -> failwith "Students, this is your job !"
```