UE LI101

Université Pierre et Marie Curie
Programmation récursive
Cours 12

A.Brygoo, C.Queinnec, M.Soria

PLAN DU COURS 12

- Langages
- Évaluation
- Conclusions

LANGAGES

- Système d'écriture (syntaxe/sémantique)
  - mots-clés
  - fonctions (bibliothèques)
- Complet au sens de Turing
  - régularité
  - puissance
  - complétude
  - confort
  - précision
  - efficacité
- Niche écologique

La perfection n'est pas quand on ne peut plus rien ajouter mais quand on ne peut plus rien retirer !

SCHEME

- mots-clés : `define`, `if`, `let`, `quote`, variable, application
- mots-clés dérivés : `let*`, `and`, `or`, `cond`, `begin`
- fonctions : 86 dans carte de référence
- composabilité
- récursion, structures de données récursives

Toutefois la moitié de Scheme reste encore à étudier

- concepts : fonctions anonymes, affectation et données modifiables, tour des nombres, continuations, macros
- styles : programmation par objets, par flots, paresseuse, dirigée par les données, par passage de continuation, etc.

ÉVALUATION

```
(define (f ...) ...)
(verifier f
   ... == ... )
(define (g ...) ...)
(verifier g
   ... == ... )
```

***Bouton* Run**

;;; DrScheme: string -> Valeur
;;; (DrScheme chaîne) calcule la valeur du programme écrit dans chaîne.

CONDITIONNEMENT

Regroupement en une seule S-expression :

```
(let ()
  (define (f ...) ...)
  (define (g ...) ...)
  (verifier f
     ... == ... )
  (verifier g
     ... == ... ) )
```

***Bouton* Run**

;;; DrScheme: S-expression -> Valeur
;;; (DrScheme sexp) calcule la valeur du programme représenté par sexp.

AUTO-ÉVALUATION

```
(DrScheme '(+ 2 3))   →5

(DrScheme '(let ()
              (define (valeur p) ...)
              (valeur '(+ 2 3)) ))       →5

(DrScheme '(let ()
              (define (valeur p) ...)
              (valeur '(let ()
                          (define (valeur p) ...)
                          (valeur '(+ 2 3)) )) ))   →5
```

`valeur` ⊂ `DrScheme`

LA FONCTION `eval`

- la base de tout interprète ou compilateur
- la base de tout metteur au point
- la possibilité d'engendrer des programmes
- une autre façon de comprendre Scheme

Les détails sont dans le code distribué ou dans le livre[a] !

[a] `http://www.infop6.jussieu.fr/cederoms/li101/`

UPMC LI101 2004-2005 – ©2004 A.Brygoo, C.Queinnec, M.Soria

S-EXPRESSION, ARBRES ET GRAMMAIRE

- Le programme est une S-expression
- Une S-expression correspond à un arbre
- qui obéit à une grammaire

```
;;;;    programme := expression
;;;;   expression := variable
;;;;              | constante | (QUOTE donnée)       ; citation
;;;;              | (COND clause*)                   ; conditionnelle
;;;;              | (IF condition conséquence [alternant]); alternative
;;;;              | (BEGIN expression*)              ; séquence
;;;;              | (LET (liaison*) corps)           ; bloc
;;;;              | (fonction argument*)             ; application
;;;;       clause := (condition expression*)
```

UPMC LI101 2004-2005 – ©2004 A.Brygoo, C.Queinnec, M.Soria

```
;;              | (ELSE expression*)
;;   constante := nombre | chaîne | booléen | caractère
;;      donnée := constante | symbole | (donnée*)
;;      liaison := (variable expression)
;;        corps := definition* expression expression*
;;   définition := (DEFINE (nom-fonction variable*) corps)
```

UPMC LI101 2004-2005 – ©2004 A.Brygoo, C.Queinnec, M.Soria

BARRIÈRE D'ABSTRACTION SYNTAXIQUE

```
;;; variable?: Expression -> bool
;;; citation?: Expression -> bool
;;; alternative?: Expression -> bool
...
;;; alternative-condition: Alternative -> Expression
;;; alternative-consequence: Alternative -> Expression
;;; alternative-alternant: Alternative -> Expression

;;; definition?: Corps -> bool
;;; definition-nom-fonction: Definition -> Variable
;;; definition-variables: Definition -> LISTE[Variable]
;;; definition-corps: Definition -> Corps
...
```

UPMC LI101 2004-2005 – ©2004 A.Brygoo, C.Queinnec, M.Soria

Couches

livre-eval
evaluation

variable-val sequence-val ...

fonction
primitive programmeur

environnement

environnement initial

bloc d'activation

Moteur d'évaluation

```
(define (livre-eval p)
  (evaluation p (env-initial) ) )

(define (evaluation exp env)
  (cond
   ((variable? exp)       (variable-val exp env))
   ((citation? exp)       (citation-val exp))
   ((alternative? exp)    (alternative-eval
                           (alternative-condition exp)
                           (alternative-consequence exp)
                           (alternative-alternant exp) env))
   ((conditionnelle? exp) (conditionnelle-eval
                           (conditionnelle-clauses exp) env))
   ((sequence? exp)       (sequence-eval (sequence-exps exp) env))
```



```
((bloc? exp)           (bloc-eval (bloc-liaisons exp)
                                  (bloc-corps exp) env))
((application? exp)    (application-eval
                        (application-fonction exp)
                        (application-arguments exp) env))
(else (livre-erreur 'evaluation "pas un programme" exp))) )
```


Couches

livre-eval
evaluation

variable-val sequence-val ...

fonction
primitive programmeur

environnement

environnement initial

bloc d'activation

Alternative

```
efine (alternative-eval condition consequence alternant env)
(if (evaluation condition env)
    (evaluation consequence env)
    (evaluation alternant env)) )
```

UPMC LI101 2004-2005 – ©2004 A.Brygoo, C.Queinnec, M.Soria

Conditionnelle

```
(define (conditionnelle-eval clauses env)
  (evaluation (cond-expansion clauses) env) )

;;; (cond (predicat consequence) clause ...) ==
;;;     (if predicat consequence (cond clause ...))
;;; (cond) == (begin)
```

UPMC LI101 2004-2005 – ©2004 A.Brygoo, C.Queinnec, M.Soria

Application

```
efine (application-eval exp-fn arguments env)
;; eval-env : Expression -> Valeur
;; (eval-env exp) rend la valeur de «exp» dans l'environnement «env»
(define (eval-env exp)
  (evaluation exp env))
;; expression de (application-eval exp-fn arguments env) :
(let ((f (evaluation exp-fn env)))
  (if (invocable? f)
      (invocation f (map eval-env arguments))
      (livre-erreur 'application-eval
                    "pas une fonction" f ) ) ) )
```

UPMC LI101 2004-2005 – ©2004 A.Brygoo, C.Queinnec, M.Soria

Citation

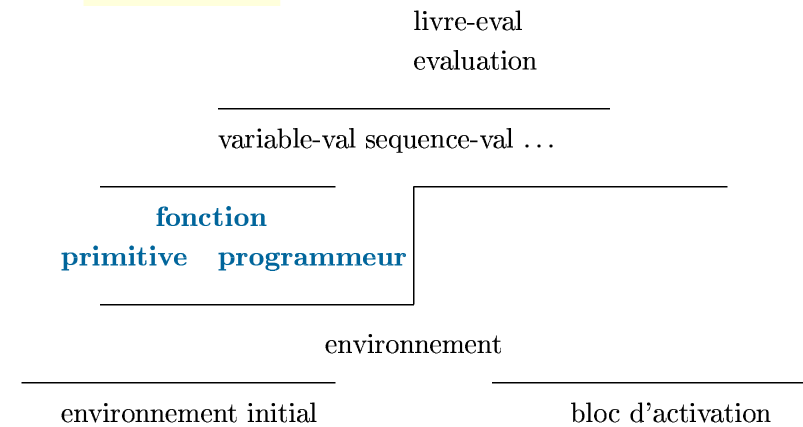
```
(define (citation-val cit)
  (if (pair? cit)
      (cadr cit)
      cit ) )
```

UPMC LI101 2004-2005 – ©2004 A.Brygoo, C.Queinnec, M.Soria

VARIABLE

```
efine (variable-val var env)
(if (env-non-vide? env)
    (let ((bloc (env-1er-bloc env)))
      (let ((variables (blocActivation-variables bloc)))
        (if (member var variables)
            (blocActivation-val bloc var)
            (variable-val var (env-reste env)))))
    (valeur-erreur 'variable-val "variable inconnue" var)))
```


COUCHES



FONCTIONS

- fonctions primitives
- fonctions de l'utilisateur

Que faire sur une fonction ?

- la créer,
- la reconnaître,
- l'invoquer

FONCTIONS

```
(define (invocable? val)
  (if (primitive? val)
      #t
      (fonction? val) ) )

(define (invocation f vals)
  (if (primitive? f)
      (primitive-invocation f vals)
      (fonction-invocation f vals) ) )
```

PRIMITIVES

```
efine (primitive? val)
(if (pair? val)
    (equal? (car val) '*primitive*)
    #f) )

; primitive-creation: N-UPLET[(Valeur... -> Valeur) (num * num ->
ol) num] -> Primitive
efine (primitive-creation f-c-n)
(cons '*primitive* f-c-n) )
```

29 primitives dont

```
ist '*primitive* car = 1)
ist '*primitive* list >= 0)
```



```
(define (primitive-invocation primitive vals)
  (let ((n       (length vals))
        (f       (cadr primitive))
        (compare (caddr primitive))
        (arite   (cadddr primitive)))
    (if (compare n arite)
        (cond
         ((= n 0) (f))
         ((= n 1) (f (car vals)))
         ((= n 2) (f (car vals) (cadr vals)))
         ((= n 3) (f (car vals) (cadr vals) (caddr vals)))
         ((= n 4) (f (car vals) (cadr vals)
                     (caddr vals) (cadddr vals) ))
         (else
          (valeur-erreur 'primitive-invocation
```



```
         "limite implantation (arité quelconque < 5)" vals)))
      (valeur-erreur 'primitive-invocation
                     "arité incorrecte" vals) ) ) )
```

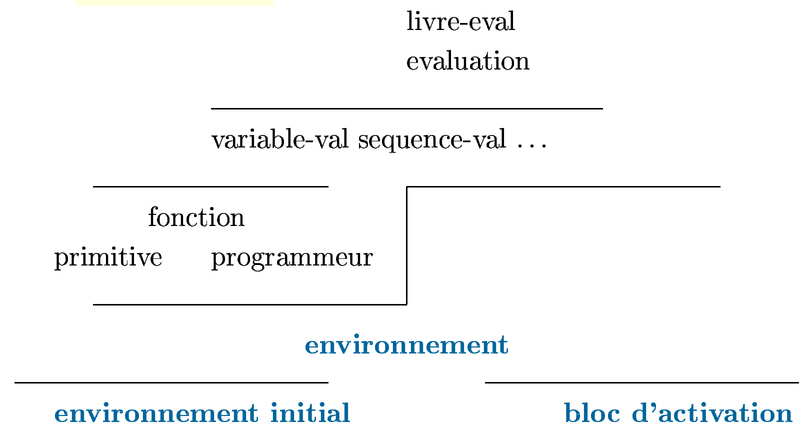

FONCTIONS DE L'UTILISATEUR

```
(define (fonction? val)
  (if (pair? val)
      (equal? (car val) '*fonction*)
      #f ) )

(define (fonction-invocation f vals)
  (let ((variables (cadr f))
        (corps     (caddr f))
        (env       (cadddr f)) )
    (corps-eval corps (env-extension env variables vals)) ) )
```

```
; fonction-creation: Definition * Environnement -> Fonction
efine (fonction-creation definition env)
(list '*fonction*
      (definition-variables definition)
      (definition-corps definition)
      env ) )
```

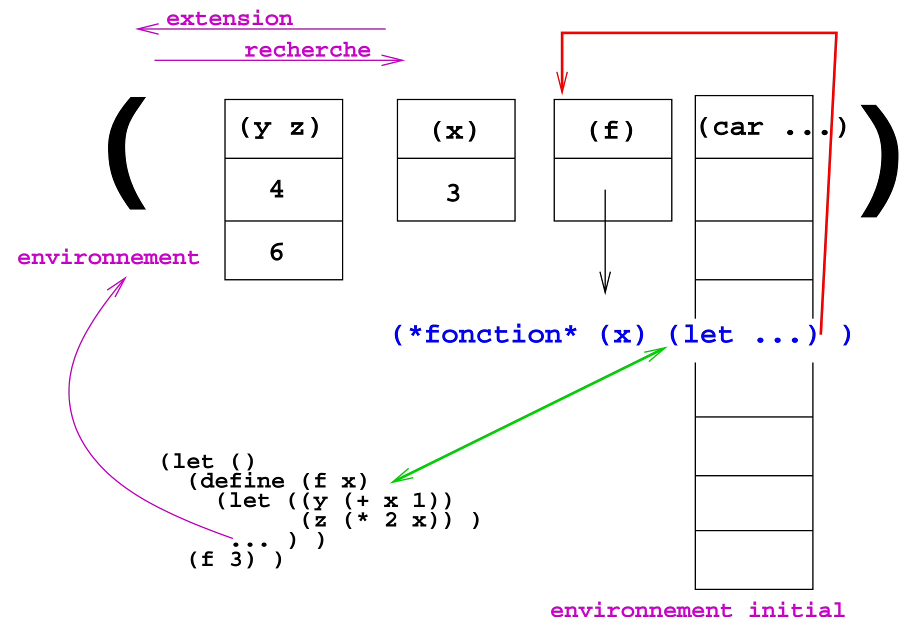

Couches



Environnement

L'environnement associe variables et valeurs.

- l'environnement initial (global)
- chercher dans l'environnement
- étendre l'environnement



Conclusions sur l'évaluateur

À un détail près (l'usage d'un vecteur et de `vector-set!` pour l'implantation des fonctions récursives)

tout le reste n'est que parcours de listes ou d'arbres généraux (programmes, environnement)

Scheme est le seul langage normalisé dont l'interprète a une taille étudiable.

Points abordés

- L'informatique est une science (mais aussi une technique)
- Bases des langages de programmation
- Principes de programmation (récursion, test, barrière d'abstraction)
- Structures de données (liste, arbre)
- Base d'algorithmique (dichotomie, parcours arborescent)
- Structure d'un évaluateur

Futur

La suite avec l'UE LI102 « Programmation impérative »

- style impératif
- usage de larges bibliothèques
- graphe de dépendance

Bonne année !