Affectation et effets de bord (6)

- Scheme offre aussi des fonctions de **mutation** pour modifier le contenu des données structurées
  - paires: `(set-car! p d)` et `(set-cdr! p d)`
  - chaînes: `(string-set! s i c)`
  - vecteurs: `(vector-set! v i d)`
- Exemple: renverser une liste destructivement

```
(define reverse!
  (lambda (lst)

    (define rev!         ; définition de fonction ''locale''
      (lambda (lst res)
        (if (null? lst)
            res
            (let ((temp (cdr lst)))
              (begin
                (set-cdr! lst res)
                (rev! temp lst))))))

    (rev! lst '())))

(define x (list 3 4 5))
(define y (cons 1 (cons 2 x)))

(reverse! y)  =>  (5 4 3 2 1)
x  =>  (3 2 1)
y  =>  (1)
```


Définition de macros (1)

- Les macros en Lisp et Scheme tirent avantage de la similitude des données et expressions (la syntaxe des expressions est un sous-ensemble de la syntaxe des données)
- Ainsi l'**expression** `(and x y)` se représente avec **la liste** `(and x y)`
- Une macro sera vue comme une **fonction de transformation** qui prend en paramètre la **représentation** des paramètres actuels de l'appel de macro, et qui retourne comme résultat la **représentation** de l'expression qui est l'expansion de la macro
- Exemple: définition d'une macro "`and`"

  Il faut transformer `(and` X Y`)` en `(if` X Y `#f)`

```
(define-macro and (lambda (x y) (list 'if x y #f)))
```

- Cette fonction de transformation est appelée **à la compilation** sur l'ASA qui est justement représenté sous forme de liste

Définition de macros (2)

- Ainsi le programme

```
(define-macro and
  (lambda (x y)
    (list 'if x y #f)))

(define f
  (lambda (x)
    (if (and (<= 0 x) (<= x 9)) (g x) 'erreur)))
```

sera expansé en

```
(define f
  (lambda (x)
    (if (if (<= 0 x) (<= x 9) #f) (g x) 'erreur)))
```

- Puisque les formes spéciales prédéfinies et les macros utilisent la **même syntaxe** préfixe parenthésée, il n'y a pas de distinction d'utilisation entre elles et l'usager peut ainsi intégrer naturellement ses propres **extensions au langage de base**
- On parle de **langage à domaine spécifique** ("domain specific language")

Définition de macros (3)

- Par exemple, pour écrire des "boucles" dans le style de Pascal on aimerait une macro "`for`" avec la syntaxe

  `(for` ⟨*variable*⟩ `:=` ⟨*expr*$_1$⟩ `to` ⟨*expr*$_2$⟩ `do` ⟨*expr*$_3$⟩`)`

  permettant de faire: `(for i := 1 to 10 do (write i))`
- Cette macro doit s'expandre à

```
(let ((debut ⟨expr1⟩) (fin ⟨expr2⟩))
  (define iteration
    (lambda (⟨variable⟩)
      (if (<= ⟨variable⟩ fin)
          (begin ⟨expr3⟩ (iteration (+ ⟨variable⟩ 1))))))
  (iteration debut))
```

- Définition basée sur "`list`"

```
(define-macro for
  (lambda (variable ignore1 expr1 ignore2 expr2 ignore3 expr3)
    (list 'let
          (list (list 'debut expr1) (list 'fin expr2))
          (list 'define
                'iteration
                (list 'lambda
                      (list variable)
                      (list 'if
                            (list '<= variable 'fin)
                            (list 'begin
                                  expr3
                                  (list 'iteration
                                        (list '+ variable 1))))))
          '(iteration debut))))
```


Définition de macros (4)

- L'utilisation de "`list`" pour cette macro donne une définition difficile à lire; il est mieux de construire des listes "quasi-constantes" à l'aide de la forme spéciale "`quasiquote`":

  ```
  `(a b ,X c (,Y d)) = (list 'a 'b X 'c (list Y 'd))
  ```

  La donnée construite correspond au patron avec des valeurs calculées insérées aux endroits précédés d'une virgule

- Définition basée sur "`quasiquote`"

```
(define-macro for
  (lambda (variable ignore1 expr1 ignore2 expr2 ignore3 expr3)
    `(let ((debut ,expr1) (fin ,expr2))
       (define iteration
         (lambda (,variable)
           (if (<= ,variable fin)
               (begin ,expr3 (iteration (+ ,variable 1))))))
       (iteration debut))))
```

- Ces définitions engendrent malheureusement un conflit de nom dans certains cas (par exemple, si la variable se nomme "`fin`" ou le corps fait référence à une variable "`fin`")

Définition de macros (5)

- Solution 1: utilisation de fonctions d'O.S.

```
(define-macro for
  (lambda (variable ignore1 expr1 ignore2 expr2 ignore3 expr3)
    `((let ()
        (define iteration
          (lambda (i fin corps)
            (if (<= i fin)
                (begin (corps i) (iteration (+ i 1) fin corps)))))
        iteration)
      ,expr1
      ,expr2
      (lambda (,variable) ,expr3))))
```

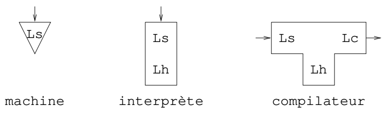
- Solution 2: utilisation de "`gensym`" qui crée des symboles qui sont garantis d'être différent de tout autre symbole

```
(define-macro for
  (lambda (variable ignore1 expr1 ignore2 expr2 ignore3 expr3)
    (let ((debut (gensym)) (fin (gensym)) (iteration (gensym)))
      `(let ((,debut ,expr1) (,fin ,expr2))
         (define ,iteration
           (lambda (,variable)
             (if (<= ,variable ,fin)
                 (begin ,expr3 (,iteration (+ ,variable 1))))))
         (,iteration ,debut)))))
```

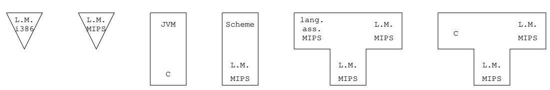
- Solution 3: utilisation de "`syntax-rules`" (propre à R5RS) un mécanisme de définition de macros "hygiéniques" définis par des règles de transformation à base de patrons

Interprètes et compilateurs (1)

- Une **machine** est capable d'exécuter un programme exprimé dans son langage source **Ls**
- Un **interprète** est un programme exprimé en langage hôte **Lh** capable d'exécuter un programme exprimé dans le langage source **Ls**
- Un **compilateur** est un programme exprimé en langage hôte **Lh** capable de traduire un programme en langage source **Ls** en un programme en langage cible **Lc**
- La notation graphique suivante permet de spécifier les langages impliqués dans chaque cas:

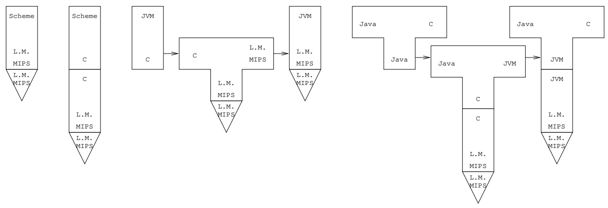


- Exemples



Interprètes et compilateurs (2)

- Un interprète ou compilateur **exécutable** peut être obtenu en combinant des machine(s), interprète(s) et compilateur(s)

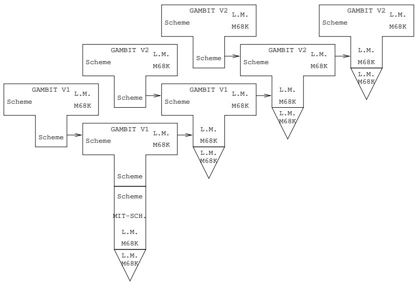


- Problème du **"bootstrap"** d'un compilateur: comment créer un nouveau compilateur pour un langage **Ls**?
- Le compilateur peut être écrit dans n'importe quel langage hôte **Lh**
- Évidemment c'est mieux si **Lh** est un langage de haut niveau (par exemple Scheme ou C) car si **Lh** est le langage machine/assembleur cela prendra beaucoup d'efforts de développement (mais c'est ce qui a été fait pour FORTRAN!)

Interprètes et compilateurs (3)

- S'il existe déjà un interprète ou compilateur pour **Ls** il est pratique d'écrire un **compilateur autogène** c'est-à-dire où **Ls = Lh**

- Exemple du compilateur Gambit:



- Avantages

  1. **autosuffisance** (pas besoin d'un autre compilateur ou interprète pour la maintenance et extensions)

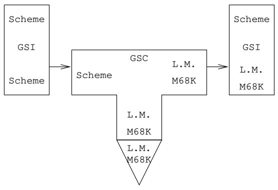
  2. amélioration de la qualité du code généré a un impact sur la **performance du compilateur** (p.e. réduction du temps de compilation)

Interprètes et compilateurs (4)

- Le cas analogue pour les interprètes, c'est-à-dire **Ls = Lh**, se nomme **interprète méta-circulaire**

- La compilation d'un interprète méta-circulaire permet d'obtenir un interprète exécutable

- Exemple de l'interprète Gambit:



- De plus, un interprète méta-circulaire peut servir à décrire et éclaircir la sémantique d'un langage de programmation et le fonctionnement des interprètes (méta-circulaires ou non)

Interprète méta-circulaire (1)

- L'implantation d'un interprète méta-circulaire pour **Scheme** est simplifiée par le fait que la syntaxe du langage source est un **sous-ensemble de la syntaxe des données**

- Ainsi la **liste** `(set! x (cdr x))` peut servir de **représentation pour l'expression** `(set! x (cdr x))`

- De plus, il est possible de représenter les données manipulées par le programme interprété par elles-mêmes (par exemple si le programme interprété exécute `(cons 1 2)` l'interprète construira la paire `(1 . 2)` directement)

- Nous allons écrire un interprète pour un **sous-ensemble de Scheme** (petit nombre de fonctions et seulement les formes spéciales `quote`, `set!`, `lambda`, `begin`, `cond`, `define` et `define-macro`)

- Problèmes principaux

  1. représentation de l'environnement d'évaluation

  2. représentation des fonctions

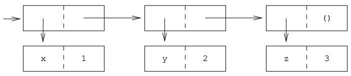
Interprète méta-circulaire (2)

- L'environnement d'évaluation peut être représenté avec une **liste d'association** (qui associe la valeur d'une variable avec le symbole qui est le nom de la variable)

- Par exemple si l'environnement contient uniquement les variables `x`, `y` et `z` et que ces variables ont respectivement les valeurs `1`, `2` et `3`:

  `((x . 1) (y . 2) (z . 3))`

  qui a la forme suivante en mémoire:



- L'accès aux variables se fait avec `assoc`:

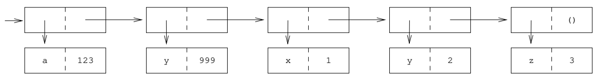
```
(define env-read
  (lambda (env var)
    (cdr (assoc var env))))

(define env-set!  ; cette implantation supporte également ‘‘define’’
  (lambda (env var val)
    (cond ((eq? (car (car env)) var) (set-cdr! (car env) val))
          ((null? (cdr env))         (set-cdr! env (list (cons var val))))
          (#t                        (env-set! (cdr env) var val)))))
```

Interprète méta-circulaire (3)

- Le traitement correct des formes spéciales `lambda` et `define` demande d'ajouter des nouvelles variables à l'environnement d'évaluation

- Pour `lambda` on peut simplement ajouter des nouvelles associations **à l'avant** de l'environnement

  Ainsi l'environnement construit pendant l'évaluation de `((lambda (a y) (list a y z)) 123 999)` est




```
(define env-extend
 (lambda (env parms vals)
  (cond ((null? parms) env)
        (#t           (cons (cons (car parms) (car vals))
                            (env-extend env (cdr parms) (cdr vals)))))))
```

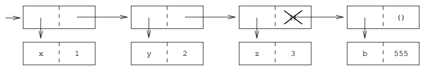
- Note: l'ancienne association pour `y` est cachée (aux yeux de `env-read` et `env-set!`) ce qui implante correctement la **portée lexicale**

Interprète méta-circulaire (4)

- Pour `define`, si la variable existe déjà on change sa valeur comme pour un `set!`

- Sinon on doit ajouter une nouvelle association **à l'arrière** de l'environnement **destructivement**

  Ainsi l'environnement sera modifié comme suit pendant l'évaluation de `(define b 555)`



- L'environnement doit être étendu **destructivement** pour implanter correctement les références "vers l'avant":

```
(define f (lambda (a) (list a b)))
(define b 555)
```

- Le même mécanisme est utilisé pour implanter `define-macro`

Interprète méta-circulaire (5)

- Il y a 2 types de fonctions: les **primitives** (les fonctions prédéfinies) et les **fermetures** (créées par l'évaluation de `lambda`-expressions)

- Pour simplifier, les fonctions sont représentées par des paires

  – si le `car` est le symbole `prim`, la paire représente la fonction primitive dont le nom est dans le champ `cdr`, par exemple `(prim . write)`

  – sinon, la paire représente une fermeture, le champ `car` donne les paramètres et le corps de la `lambda`-expression qui a engendré cette fermeture et le champ `cdr` est l'environnement dans lequel cette `lambda`-expression fut évaluée

- Par exemple, si `(lambda (a y) (list a y z))` est évaluée dans l'environnement `((x . 1) (y . 2) (z . 3))` le résultat est

```
(((a y) (list a y z)) . ((x . 1) (y . 2) (z . 3)))
```

- Cette fermeture sera scrutée par l'interprète lors de son appel

Interprète méta-circulaire (6)

- La fonction principale de l'interprète est la fonction `eval`:

  `(eval` ⟨*repr_d'une_expression*⟩ ⟨*environnement*⟩`) =>` ⟨*valeur_de_l'expression*⟩

  Par exemple: `(eval '(cons 1 2) global-env) => (1 . 2)`

- La boucle d'interaction "read-eval-print" de l'interprète s'implante simplement avec:

```
(define repl
  (lambda ()
    (begin
      (write '>)
      (write (eval (read) global-env))
      (newline)
      (repl))))
```

Interprète méta-circulaire (7)

```
(define global-env
  (list '(read     . (prim . read    )) '(newline  . (prim . newline))
        '(write    . (prim . write   )) '(symbol?  . (prim . symbol?))
        '(null?    . (prim . null?   )) '(pair?    . (prim . pair?  ))
        '(car      . (prim . car     )) '(cdr      . (prim . cdr    ))
        '(cadr     . (prim . cadr    )) '(cons     . (prim . cons   ))
        '(set-cdr! . (prim . set-cdr!)) '(assoc    . (prim . assoc  ))
        '(eq?      . (prim . eq?     )) '(list     . (prim . list   ))))

(define env-read
  (lambda (env var)
    (cdr (assoc var env))))

(define env-set!
  (lambda (env var val)
    (cond ((eq? (car (car env)) var) (set-cdr! (car env) val))
          ((null? (cdr env))         (set-cdr! env (list (cons var val))))
          (#t                        (env-set! (cdr env) var val)))))

(define env-extend
  (lambda (env parms vals)
    (cond ((null? parms) env)
          (#t           (cons (cons (car parms) (car vals))
                              (env-extend env (cdr parms) (cdr vals)))))))

(define eval
  (lambda (expr env)
    (cond ((symbol? expr) (env-read env expr))
          ((pair? expr)   (eval-composite (car expr) (cdr expr) env))
          (#t             expr))))

(define eval-composite
  (lambda (x lst env)
    (cond ((assoc x macros)  (eval (apply (env-read macros x) lst) env))
          ((eq? x 'defmacro) (env-set! macros (car lst) (eval (cadr lst) env)))
          ((eq? x 'set!)     (env-set! env (car lst) (eval (cadr lst) env)))
          ((eq? x 'quote)    (car lst))
          ((eq? x 'lambda)   (eval-lambda lst env))
          ((eq? x 'begin)    (eval-begin #f lst env))
          ((eq? x 'cond)     (eval-cond lst env))
          (#t                (eval-call x lst env)))))

(define eval-lambda
  (lambda (parms-and-body def-env)
    (cons parms-and-body def-env)))
```



```
(define eval-begin
  (lambda (val lst env)
    (cond ((null? lst) val)
          (#t          (eval-begin (eval (car lst) env) (cdr lst) env)))))

(define eval-cond
  (lambda (lst env)
    (cond ((eval (car (car lst)) env) (eval (cadr (car lst)) env))
          ((pair? (cdr lst))          (eval-cond (cdr lst) env)))))

(define eval-call
  (lambda (fn-expr arg-exprs env)
    (apply (eval fn-expr env) (eval-list arg-exprs env))))

(define eval-list
  (lambda (lst env)
    (cond ((null? lst) '())
          (#t          (cons (eval (car lst) env)
                             (eval-list (cdr lst) env))))))

(define apply
  (lambda (fn lst)
    (cond ((eq? (car fn) 'prim)
           (cond ((eq? (cdr fn) 'read)     (read))
                 ((eq? (cdr fn) 'newline)  (newline))
                 ((eq? (cdr fn) 'write)    (write    (car lst)))
                 ((eq? (cdr fn) 'symbol?)  (symbol?  (car lst)))
                 ((eq? (cdr fn) 'null?)    (null?    (car lst)))
                 ((eq? (cdr fn) 'pair?)    (pair?    (car lst)))
                 ((eq? (cdr fn) 'car)      (car      (car lst)))
                 ((eq? (cdr fn) 'cdr)      (cdr      (car lst)))
                 ((eq? (cdr fn) 'cadr)     (cadr     (car lst)))
                 ((eq? (cdr fn) 'cons)     (cons     (car lst) (cadr lst)))
                 ((eq? (cdr fn) 'set-cdr!) (set-cdr! (car lst) (cadr lst)))
                 ((eq? (cdr fn) 'assoc)    (assoc    (car lst) (cadr lst)))
                 ((eq? (cdr fn) 'eq?)      (eq?      (car lst) (cadr lst)))
                 ((eq? (cdr fn) 'list)     lst)))
          (#t
           (eval (cadr (car fn))
                 (env-extend (cdr fn) (car (car fn)) lst))))))

(define macros
  (list (cons 'define
              (eval-lambda '((var expr) (list 'set! var expr)) global-env))))

(define repl
  (lambda ()
    (begin (write '>) (write (eval (read) global-env)) (newline) (repl))))
```


Interprète méta-circulaire (8)

Exemple 1: trace de l'évaluation de `list`

```
| > (eval 'list '((read prim . read) ...))
| > (env-read '((read prim . read) ...) 'list)
| (prim . list)
```

Exemple 2: trace de l'évaluation de
`(set! read (list 1 2))`

```
| > (eval '(set! read (list 1 2)) '((read prim . read) ...))
| > (eval-composite 'set! '(read (list 1 2)) '((read ...)))
| | > (eval '(list 1 2) '((read prim . read) ...))
| | > (eval-composite 'list '(1 2) '((read prim . read) ...))
| | > (eval-call 'list '(1 2) '((read prim . read) ...))
| | | > (eval 'list '((read prim . read) ...))
| | | > (env-read '((read prim . read) ...) 'list)
| | | (prim . list)
| | | > (eval-list '(1 2) '((read prim . read) ...))
| | | | > (eval 1 '((read prim . read) ...))
| | | | 1
| | | | > (eval-list '(2) '((read prim . read) ...))
| | | | | > (eval 2 '((read prim . read) ...))
| | | | | 2
| | | | | > (eval-list '() '((read prim . read) ...))
| | | | | ()
| | | | (2)
| | | (1 2)
| | > (apply '(prim . list) '(1 2))
| | (1 2)
| > (env-set! '((read prim . read) ...) 'read '(1 2))
| #<void>
```


Interprète méta-circulaire (9)

Exemple 3: trace de l'évaluation de
`((lambda (write) (write read)) car)`

```
| > (eval '((lambda (write) (write read)) car) '((read 1 2) ...))
| > (eval-composite '(lambda (write) (write read)) '(car) '(...))
| > (eval-call '(lambda (write) (write read)) '(car) '((read 1 2) ...))
| | > (eval '(lambda (write) (write read)) '((read 1 2) ...))
| | > (eval-composite 'lambda '((write) (write read)) '((read 1 2) ...))
| | > (eval-lambda '((write) (write read)) '((read 1 2) ...))
| | (((write) (write read)) . ((read 1 2) ...))
| | > (eval-list '(car) '((read 1 2) ...))
| | | > (eval 'car '((read 1 2) ...))
| | | > (env-read '((read 1 2) ...) 'car)
| | | (prim . car)
| | | > (eval-list '() '((read 1 2) ...))
| | | ()
| | ((prim . car))
| > (apply '(((write) (write read)) . ((read 1 2) ...)) '((prim . car)))
| | > (env-extend '((read 1 2) ...) '(write) '((prim . car)))
| | | > (env-extend '((read 1 2) ...) '() '())
| | | ((read 1 2) ...)
| | ((write prim . car) (read 1 2) ...)
| > (eval '(write read) '((write prim . car) (read 1 2) ...))
| > (eval-composite 'write '(read) '((write prim . car) (read 1 2) ...))
| > (eval-call 'write '(read) '((write prim . car) (read 1 2) ...))
| | > (eval 'write '((write prim . car) (read 1 2) ...))
| | > (env-read '((write prim . car) (read 1 2) ...) 'write)
| | (prim . car)
| | > (eval-list '(read) '((write prim . car) (read 1 2) ...))
| | | > (eval 'read '((write prim . car) (read 1 2) ...))
| | | > (env-read '((write prim . car) (read 1 2) ...) 'read)
| | | (1 2)
| | | > (eval-list '() '((write prim . car) (read 1 2) ...))
| | | ()
| | ((1 2))
| > (apply '(prim . car) '((1 2)))
| 1
```