Interprétation

Un mini-Scheme
programmé en…
Scheme !

But : la sémantique

- Le problème de la **syntaxe** sera "réglé" par Lex/Yacc (cours 13).
- Le langage Scheme (à la suite de LISP) utilise des **listes** pour représenter à la fois les données et les programmes, ce qui est for-mi-da-ble car on peut les penser comme des **arbres**.
- On veut programmer un **interprète** d'un langage proche de Scheme.
- En se focalisant sur la **sémantique** :
  - des formes spéciales →
  - de l'application fonctionnelle (f a b …)
  - avec un accent particulier sur le concept de **fermeture**.

```
if
let
letrec
lambda
begin
set!
define
```

Rappel : une fermeture est …

La grammaire de MIPS (Mips Is Pseudo-Scheme)

- On interprète en Scheme des expressions parenthésées [donc des **arbres**]. La fonction (read) est gratuite…

```
<expr> ::=
       <constante>
     | IDENT
     | <forme-spéciale>
     | <application>
<forme-spéciale> ::=
       ($lambda (IDENT ...) <expr> <expr> ...)
     | ($if <expr> <expr> <expr>)
     | ($let ((IDENT <expr>) ...) <expr> <expr> ...)
     | ($letrec ((IDENT <expr>) ...) <expr> <expr> ...)
     | ($set! IDENT <expr>)
     | ($begin <expr> <expr> ...)
     | ($define IDENT <expr>)
<application> ::= (<expr> <expr> ...)
<constante> ::= NUMBER | #t | #f
```


- Exemple de session au toplevel :

```
> (%mips)
========= Entering MIPS toplevel =========
? ($define x 1)
--> #<void>
? ($+ ($if ($> x 0) ($+ x 1) ($- x 1))
      ($let ((x ($+ x 1)) (y x))
        ($+ x y)))
-->
? ($define foo
    ($lambda (a b) ($+ x ($* a b))))
--> #<void>
? ($set! x ($+ x 9))
--> #<void>
? ($let ((x 1000))
    (foo x 5))
-->
? ($define fac
    ($letrec ((f ($lambda (x) ($if ($= x 0) 1 ($* x (f ($- x 1)))))))
      f))
--> #<void>
? (fac x)
--> 3628800
```

Evaluer mais où ? Dans un environnement

- **Evaluer** une expression n'a de sens que dans un **environnement** de variables :

```
(let ((x 2))
  (+ x a))
```

sachant que x vaut temporairement 2, évaluer x + a

Dans quel contexte ?

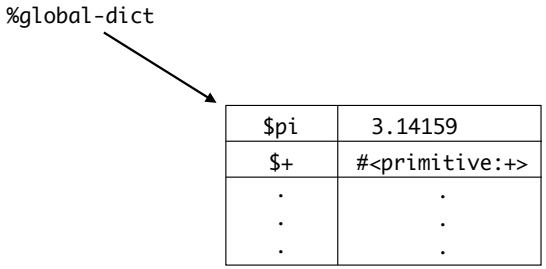
Que vaut la variable a ?

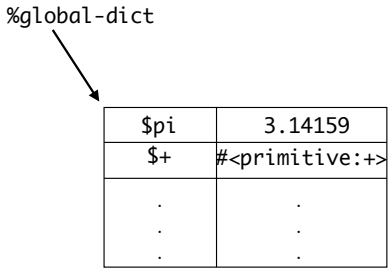
Que vaut la variable + ?

```
(let ((+ *) (a 3))
  (let ((x 2))
    (+ x a)))
```


Dictionnaires et Environnements

- Un **dictionnaire** sera un ensemble de couples ‹variable, valeur› organisé comme une table [accès séquentiel, en arbre, par hash-code]. Toutes ses variables sont distinctes.
- Par exemple le **dictionnaire global** du toplevel :





« **dictionnaire** » = « cadre » chez Abelson & Sussman (SICP)

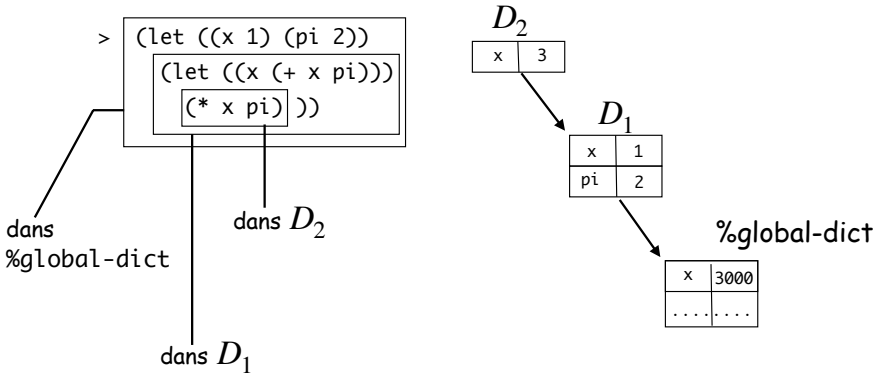
[en anglais « frame »]

Le couple ‹$pi, 3.14159› est une « association » ou « liaison »

- Un **environnement** sera une suite ordonnée [liste] de dictionnaires aboutissant au dictionnaire global [accès séquentiel] :

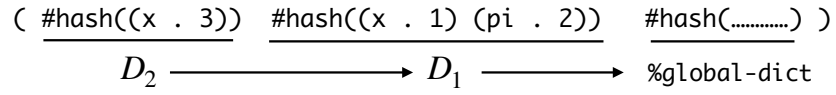
$$D_n \rightarrow D_{n-1} \rightarrow \cdots \rightarrow D_1 \rightarrow D_0 \equiv \texttt{\%global-dict}$$

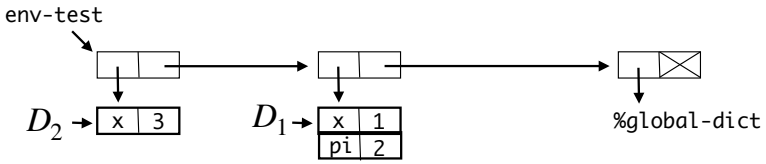
- Par exemple **l'environnement étendu** en provenance d'un let :



Implémentation des environnements

Un **environnement** sera une **liste** de dictionnaires.
Un **dictionnaire** sera implémenté par une **table de hash-code**. Soit l'environnement `env-test` ci-dessous :





Gestion des environnements

- Il faut connaître l'API minimale des tables de hash-code :

```
(make-hash)                 ; construit une table de hash mutable, vide
(hash-set! H k val)         ; modifie la valeur de la clé k de la table H
(hash-has-key? H k)         ; k est-elle une clé de la table H ?
(hash-ref H k)              ; la valeur de la clé k dans la table H
```

NB. Les tables de hash-code sont **extrèmement efficaces** !

- La **recherche de la liaison** courante d'un symbole var dans un environnement env se fait en recherchant le premier dictionnaire de env contenant la clé var :

```
(define (%lookup var env)          ; valeur de var dans env ou erreur
  (define dict (%dict var env))    ; premier dictionnaire de env contenant var
  (if dict
      (hash-ref dict var)
      (error "Variable MISS inconnue : " var)))
```


- Un **nouveau dictionnaire** est construit à partir d'une liste Lvar de variables et d'une liste Lval de valeurs.

```
(%new-dictionnary Lvar Lval)
```

```
> (%new-dictionnary '(x y z) '(1 2 3))
#hash((x . 1) (y . 2) (z . 3))
```

- **L'extension d'un environnement** par ajout d'un nouveau dictionnaire en tête est une fonction [un constructeur] analogue à cons :

```
(%extend-env Lvar Lval env)
```

Retourne un nouvel environnement obtenu en ajoutant en tête de env *un nouveau dictionnaire liant les variables de* Lvar *aux valeurs de* Lval.

- L'**extension physique d'un dictionnaire** existant (les dictionnaires sont mutables contrairement aux environnements) :

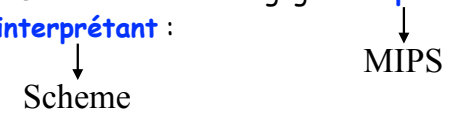
```
(define (%extend-dict! var val dict)   ; ajout de var/val au dictionnaire dict
  (hash-set! dict var val))
```

- L'**environnement initial** contiendra les **primitives** : nous profitons de celles du langage interprétant (Racket) :

```
(define (%init-global-env)
  ; la primitive binaire $+ aura pour valeur dans la hash-table
  ; %global-dict : (prim 2 #<primitive:+>)
  (let ((Lprims `(($+      prim  2   ,+)
                  ($-      prim  2   ,-)
                  ($*      prim  2   ,*)
                  ($/      prim  2   ,/)
                  ($=      prim  2   ,=)
                  . . . . . . .
                  ($zero?  prim  1   ,zero?))))
    (set! %global-env
          (%extend-env (map car Lprims)
                       (map cdr Lprims)
                       %global-env))         ; qui est vide au départ !
    (set! %global-dict (car %global-env))))  ; le dictionnaire initial
```

Valeur d'une constante

- La valeur d'une constante est cette constante.
- Les booléens du langage **interprété** sont ceux du langage **interprétant** :




```
(define (%constant? expr)
  (or (number? expr) (boolean? expr)))
```

- La fonction (%eval expr env) est guidée par la grammaire des expressions :

```
(define (%eval expr env)
  (cond ((%constant? expr) expr)
        ......
```


Valeur d'un symbole

- ***dans un environnement !***... On cherche la liaison la plus récente du symbole dans env. Les liaisons sont donc gérées en pile.
- La recherche de la liaison d'une variable var dans l'environnement env se fait par :

```
($lookup var env)
```
cf page 10

```
(define (%eval expr env)
  (cond ((%constant? expr) expr)
        ((symbol? expr) (%lookup expr env))
        ......
```


Les « formes spéciales »

- Une **forme spéciale** débute par un **mot-clé** [**keyword**]. Leur traitement est spécial, il ne suit pas l'ordre applicatif, ce ne sont PAS des fonctions !

```
(define (%special? expr)
  (and (pair? expr)
       (member (car expr)
               '($if $let $letrec $lambda $begin $set! $define))))
```

- Certains arguments seront évalués, d'autres pas...
- Pour ceux qui le sont, l'ordre d'évaluation peut être ou non spécifié.

```
(define (%eval expr env)
  (cond ((%constant? expr) expr)
        ((symbol? expr) (%lookup expr env))
        ((%special? expr) (%eval-special expr env))
        ......
```


L'application fonctionnelle

```
<expr> ::=  <constante>
          | IDENT
          | <forme-spéciale>
          | <application>
<application> ::= (<expr> <expr> ...)
```

- Dans (f a b c ...) la fonction f est donc calculée tout comme ses arguments a, b, c... On **applique** ensuite la valeur F de f aux valeurs A, B, C... des arguments : c'est l'ORDRE APPLICATIF d'évaluation (grosso modo **l'appel par valeur**) :

```
(define (%eval expr env)
  (cond ((%constant? expr) expr)
        ((symbol? expr) (%lookup expr env))
        ((%special? expr) (%eval-special expr env))
        (else (%apply (%eval (car expr) env)
                      (%evlis (cdr expr) env))))
```

(%apply F Lvals) ???

• Ah, ah ! Et pourquoi pas (%apply F Lvals env) ? L'environnement env serait-il devenu inutile ?...

• Réponse : OUI, car de deux choses l'une :

- ou bien F est une **primitive** comme + qui se contente d'additionner deux constantes numériques.
- ou bien F est une **fermeture**, qui contient en son sein [un pointeur vers] **l'environnement de création**.

```
> (define foo (λ (x) (+ x a)))
> (define a 10)
> (let ((a 1000))
    (foo 1))
???
```

la fermeture :
$< x \mapsto x + a,$ %global-env>

C'est la **liaison statique** [ou **lexicale**]...
La portion de texte où a été créée une lambda est d'une importance capitale !

Représentation interne d'une fermeture

• Ce qui précède suggère de représenter une **fermeture** comme un triplet contenant :
- la liste des paramètres
- le corps [une expression]
- [un pointeur vers] l'environnement de création.

Le tout agrémenté d'une petite marque de fabrique `*closure*`

```
foo == (*closure* (x) ($+ x a) %global-env)
```

```
(define (%closure? proc)
  (and (pair? proc) (eq? (car proc) '*closure*)))
```

• Une fermeture est une valeur perçue comme un tout, dont les composantes sont accessibles à l'interprète mais pas au programmeur :

```
> foo
#<procedure:foo>
```


Qu'est-ce que la liaison dynamique ?

• En Lisp classique [ex: Emacs], la valeur d'une lambda n'est que le **texte** de la lambda. Pas de pointeur sur l'environnement de création !

• Donc on passe à apply l'environnement courant [dynamique] :

(%apply F Lvals env)

• On dit alors que l'interprète fonctionne en **liaison dynamique** :

```
> (define foo
    (let ((a 1000))
      (lambda (x) (+ x a))))
> (let ((a 1))
    (foo a))
2
```

Ce n'est plus une fermeture !

N.B. En liaison statique, on aurait obtenu 1001.

Représentation d'une primitive

• La valeur de la primitive binaire $+ sera représentée par une liste contenant l'arité et une valeur procédurale Scheme, considéré comme langage de bas niveau.

Avec une marque de fabrique `*prim*` *:*

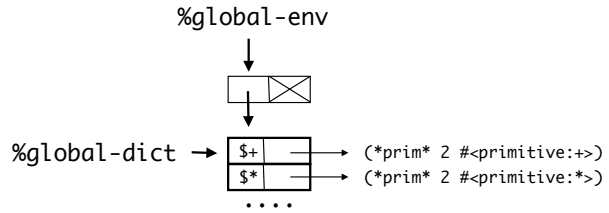
```
(*prim* 2 #<primitive:+>)
```

• On utilise donc les primitives du langage interprétant [ici Scheme]...

```
(define (%primitive? proc)
  (and (pair? proc) (eq? (car proc) '*prim*)))
```

• Ces primitives seront installées dans le dictionnaire global %global-dict avant de lancer la boucle toplevel...

• Donc au lancement de la boucle toplevel, les primitives seules sont présentes dans l'environnement global :



• Rappel : env = (<D_n> ... <D_1> %global-dict)

L'application fonctionnelle [suite]

• Distinguons l'application d'une **primitive** de celle d'une **fermeture** :

```
(define (%apply proc Largs)
  (cond ((%primitive? proc) (%apply-internal proc Largs))
        ((%closure? proc) ....))
        (else (error "Bad ! Un-apply-able object !" proc))))
```

*Si la procédure à appliquer est une **fermeture** [valeur d'une lambda-expression], le résultat de l'application n'est autre que la valeur du corps de la fermeture dans l'environnement obtenu en étendant celui de la fermeture par les liaisons des paramètres aux valeurs des arguments...*

• En ce qui concerne une primitive, on se laisse guider par l'arité :

```
(define (%apply-internal proc Largs)
  (case (cadr proc)
    ((1) . . . .)
    ((2) . . . .)
    (else (error "Bad arity" proc)))))
```


Les traitements spéciaux

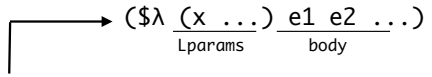
• A chaque **mot-clé** [keyword] est associé un traitement ad-hoc :

```
(define (%eval-special expr env)
  (case (car expr)
    (($lambda) (%eval-lambda expr env))
    (($if)     (%eval-if expr env))
    (($let)    (%eval-let expr env))
    (($letrec) (%eval-letrec expr env))
    (($set!)   (%eval-set! expr env))
    (($begin)  (%eval-begin (cdr expr) env))
    (($define) (%eval-define expr env))
    (else     (error "Not yet implemented" (car expr)))))
```

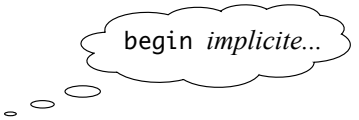
• Les formes spéciales manquantes comme and, or, cond, ... peuvent être rajoutées :
- soit explicitement en augmentant la liste ci-dessus.
- soit par un système de **macros** qui transformerait par ex. le source ($and a b) en ($if a b #f). Mézalor, quel sera le noyau essentiel ?...

Forme spéciale : $lambda

• La valeur d'une λ-expression créée dans un environnement env est une **fermeture** qui capture [un pointeur vers] env :




```
(define (%eval-lambda expr env)
  (let ((Lparams (cadr expr)) (body (cddr expr)))
    (list . . . .)))
```




```
(*closure* (x ...) ($begin e1 e2 ...) env)
```

Forme spéciale : $let

• La forme spéciale `let` [*inutile en théorie*] sert à introduire des variables locales.

On évalue son corps dans l'environnement étendu obtenu à partir de env *en liant temporairement les variables locales à leurs valeurs dans* env.

```
                                  → ($let ((var val) ...) e1 e2 ...)
(define (%eval-let expr env)
  (let ((Lvars (map car (cadr expr)))
        (Lvals (map cadr (cadr expr)))
        (body (cddr expr)))
    ....))
```


Forme spéciale : $letrec

• La forme spéciale `letrec` sert à obtenir des variables locales fonctionnelles mutuellement récursives.

On créé un dictionnaire temporaire liant les symboles `var...` *à des valeurs indéfinies, puis on évalue les expressions* `func...` *dans l'environnement étendu E et les valeurs obtenues remplacent les valeurs indéfinies précédentes. Enfin le corps* `e1 e2...` *est évalué dans l'environnement E. Chaque variable* `var...` *a pour portée l'expression* `(letrec...)` *tout entière [R5RS].*

```
                                  → ($letrec ((var func) ...) e1 e2 ...)
(define (%eval-letrec expr env)
  (let ((Lvars (map car (cadr expr))) (Lfuncs ...))
    (let ((new-env (%extend-env ...)))
      .....)))
```


Forme spéciale : $define

• Nous prenons des libertés par-rapport à la norme Scheme :

```
($define x ($begin ($define y 3) ($+ y 1)))   ← good
($let ((x 1)) ($define y (+ x 3)) (+ x y)))   ← bad
```

La forme `define` n'est acceptée **que dans l'environnement global** !
Pas de `define` interne : utilisez `letrec` !

```
                                  → ($define var e)
(define (%eval-define expr env)
  (let ((var (cadr expr)) (e (caddr expr)))
    (if (not (eq? env %global-env))
        (error  "Bad define" expr)
        ....))))
```

eq?

Forme spéciale : $begin

• La forme de **séquencement** pour la programmation impérative. La valeur de `(begin e1 e2 …)` est obtenue en évaluant **en séquence** les expressions e1, e2 … dans l'environnement lexical courant et le résultat est la valeur de la dernière expression.

```
                                  → (e1 ... en)
(define (%eval-begin expr env)
  .....)
```

• Le `begin` est quasiment **implicite** partout en Scheme sauf dans la conditionnelle `if`…

Forme spéciale : $set!

set-bang

- La forme de **mutation des variables** pour la programmation impérative. L'évaluation de (`$set! var expr`) dans un env. E est obtenu en cherchant [avec `%dict`] la « première » liaison du symbole `var` dans E et en y remplaçant sa valeur par celle de `expr` dans E. C'est une erreur d'essayer d'affecter un symbole non lié dans E.

- Le « ! » est là pour dire « Danger ! Ici on modifie ! ». Il se lit *Bang…*

- La valeur de retour de `$set!` est non spécifiée [par ex. `#<void>`].

```
                              ┌──────────→ ($set! var e)
(define (%eval-set! expr env)
  (let ((var (cadr expr)) (e (caddr expr)))
    .....))
```


La boucle toplevel

- alias « REP loop » [*Read-Eval-Print* loop] :

```
(define (%mips)
  (printf "========= Entering MIPS toplevel =========\n")
  (%init-global-env)
  (printf "Installing software...\n")
  (%install-software)
  ;(%dump-env "$GLOBAL-ENV" $global-env)     ; histoire de voir…
  (%rep-loop))

(define (%rep-loop)
  (let ((expr (%read)))
    (if (eq? expr 'quit)     ; une "commande" de sortie…
        (printf "Program first, think later !")     ; gasp
        (let ((val (%eval expr $global-env)))
          (%print val)
          (%rep-loop)))))
```

N.B. Il n'y a pas de fonction `(quit)` *mais une commande* `quit` *au toplevel de MIPS !*

Le lecteur et le scribe

- On profite ici des entrées-sorties de Scheme, mais on pourrait les redéfinir, notamment le scribe qui pourrait tester le type de `val` [par exemple si `val` est une fermeture !] :

```
(define (%read)            ; le lecteur
  (printf "? ")
  (read))

(define (%print val)       ; et le scribe
  (printf "--> ~a~n" val))
```

Biblio :

- ***Premiers Cours de Programmation avec Scheme***, chap. 16
- ***Structure and Interpretation of Computer Programs***, 2nd ed, chap. 4
- https://paracamplus.com/Books/Cours/LiSP/4/extrait.pdf