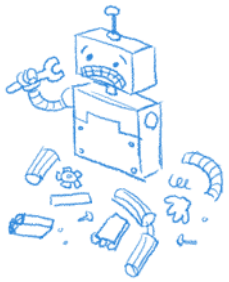
*OPTION PF2 - Automne 2016*
*Jean-Paul Roy*
*L2-Info&Maths, Faculté des Sciences*

http://deptinfo.unice.fr/~roy

*Cours n°11B*
*(enfin presque...)*

# Implantation d'un micro-Scheme en Python



d'après Peter Norvig
Directeur de la Recherche, Google

## Programmation d'un interprète μ-Scheme

- On veut programmer un **interprète** d'un petit langage proche de Scheme nommé μ-Scheme.
- Un **interprète de code B** est un programme écrit dans un langage A capable de lire du texte (string) contenant du code rédigé en langage B et d'exécuter ce code au sein de A.
- Prenons A = *Python* par exemple et B = *μ-Scheme*.
- Etant donnée une string Python contenant du μ-Scheme, comme :

```
'(* (+ 2014 10) 2)'
```

on veut l'évaluer en Python pour obtenir sa valeur, ici 4048.

- Une expression μ-Scheme pourra avoir un effet sans valeur de retour :

```
'(define foo (lambda (x) (* x 2)))'
```

Nous devrons donc gérer une mémoire Scheme en Python...



- Exemple de session au toplevel Python 3 :

```
>>> micro_scheme()                    # lancement de l'interprète
Welcome to micro-scheme (PF2)
? 2014                                # avec son propre toplevel !
= 2014
? (define x 2014)                     # aucun résultat
? (+ (* x 10) 2)
= 20142
? (lambda (x) (* x 2))
= #<closure>
? ((lambda (x) (* x 2)) 3)
= 6
? (define fac (lambda (n) (if (= n 0) 1 (* n (fac (- n 1))))))
? (fac 20)
= 2432902008176640000
? .quit                               # une commande au toplevel
>>> x                                 # retour au toplevel Python
NameError: name 'x' is not defined
```



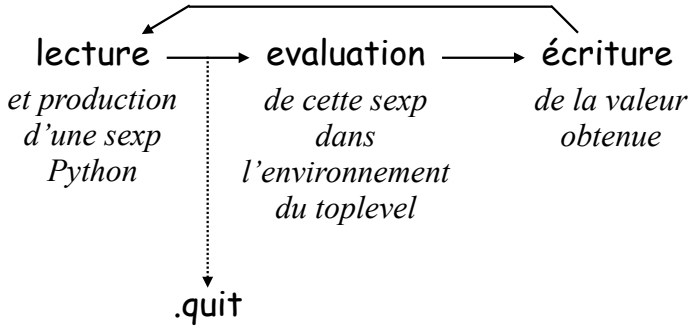
## Grammaire du langage μ-Scheme

```
<expr> ::=
     NUMBER | VAR | <forme-spéciale> | <application>

<forme-spéciale> ::=
     | (if <expr> <expr> <expr>)
     | (lambda (VAR ...) <expr> <expr> ...)
     | (begin <expr> <expr> ...)
     | (set! VAR <expr>)
     | (define VAR <expr>)

<application> ::=
       (<expr> <expr> ...)
```

- L'interprète sera là pour donner un **sens** à une expression, ce sera la partie **sémantique**. Le lecteur s'occupe de la partie **syntaxique**.

# Le toplevel et sa boucle read-eval-print

- Le toplevel est une boucle infinie (ou presque) :




- La commande **.quit** n'est pas une sexp mais permet d'agir sur le fonctionnement de l'interprète (quitter, tracer, etc).

- Donc deux parties essentielles : le **lecteur** et l'**évaluateur**.



- La programmation du **toplevel** est simple, elle anticipe sur les fonctions **read** (le lecteur) et **micro_eval** (l'évaluateur).

```
def micro_scheme(reader=False):
    print('Welcome to micro-scheme (PF2)')
    while True:             # boucle REP: Read-Eval-Print loop
        s = input('? ')     # l'utilisateur entre du µ-Scheme
        sexp = read(s)                                  # lecteur
        if reader: print('sexp --> ',sexp)
        val = micro_eval(sexp, global_env)              # évaluateur
        if val != None: print('=',to_string(val)) # scribe
```

```
>>> micro_scheme(reader = True)
Welcome to micro-scheme (PF2)
? (define x (* 3 4))
sexp -->  ['define', 'x', ['*', 3, 4]]
? (+ x 1)
sexp -->  ['+', 'x', 1]
= 13
```



# Première partie : le lecteur de sexp

- Il doit être capable de lire une string contenant du code µ-Scheme puis la traduire en une sexp Python (qui sera ensuite passé à l'évaluateur) : nombre, symbole, liste.

- Une telle string va contenir des parenthèses, des espaces, des nombres, et des symboles. Parmi ces symboles, des noms de variables, des noms de fonctions (primitives arithmétiques ou fermetures écrites en Scheme), mais aussi des mots-clés de formes spéciales comme `lambda`, `if`, `define`…

| ***string*** | ***sexp*** |
|---|---|
| `'2014'` | `2014` |
| `'(* (+ x 2) y)'` | `['*', ['+', 'x', 2], 'y']` |



- On commence par décoller les parenthèses pour les isoler, de manière à obtenir des unités lexicales bien séparées (**tokens**) :

```
def read(s):
    s = s.replace('(',' ( ')
    s = s.replace(')',' ) ')
    s = s.split()
    return s
```

*version provisoire*

```
>>> read('(+ (* hauteur 0.25) h)')
 ['(', '+', '(', '*', 'hauteur', '0.25', ')', 'h', ')']
```

- Il s'agit maintenant de transformer cette liste plate en une sexp :

```
['+', ['*', 'hauteur', 0.25], 'h']
```

- C'est un peu plus délicat (les ***parsers*** du cours d'Analyse de L3), nous allons procéder à une analyse récursive de la liste.

```
def read(s):                          # version définitive
    s = s.replace('(',' ( ').replace(')',' ) ').split()
    return read_from(s)
```

*je continue par le parser récursif*

- Etudiez bien cette très belle récurrence :

```
def read_from(Ltokens):    # Read an expression from a sequence of tokens
    if len(Ltokens) == 0:
        raise SyntaxError('Unexpected EOF while reading')
    token = Ltokens.pop(0)          # on dépile le 1er élément
    if token == '(':                # je vais lire une liste
        L = []
        while Ltokens[0] != ')':
            L.append(read_from(Ltokens))
        Ltokens.pop(0)              # on dépile la ')'
        return L
    elif token == ')': raise SyntaxError('unexpected )')
    else : return atom(token)       # nombre ou symbole
```



- En dernière ligne, **atom** transforme une string contenant un symbole ou un nombre (entier, flottant) en string ou nombre.

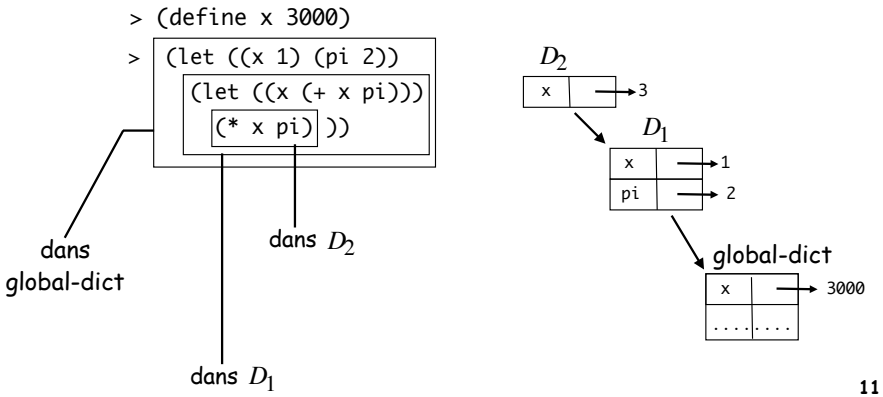
```
def atom(token):
    try: return int(token)
    except ValueError :
        try: return float(token)
        except ValueError:
            return str(token)
```

```
>>> atom('foo')          # symbole
'foo'
>>> atom('-56')          # entier
-56
>>> atom('56.3')         # flottant
56.3
```



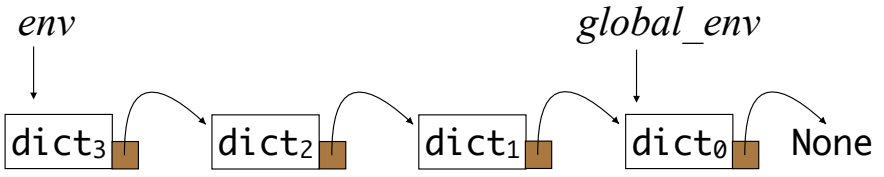
# Seconde partie : gestion des environnements

- Vous savez qu'un calcul a lieu dans un **environnement**, qui est une suite de dictionnaires. Un tel **dictionnaire** en Python sera de la forme `{var:val,...}`. On notera `global_env` l'environnement réduit au dictionnaire du toplevel.






- Un nouvel environnement **étend** toujours un autre environnement.
- Pour calculer l'expression `(let ((x 1) (y 2)) (+ x y z))`, invoquée dans un environnement E, l'interprète va commencer par créer un dictionnaire local D = `((x 1) (y 2))` qui va étendre E, d'où un environnement étendu E' = D → E, et l'expression `(+ x y z)` sera évaluée dans E'. *Cf PCPS § 16.2...*
- L'environnement global étend la vacuité None.



• La classe Env sera une **sous-classe** de la classe primitive dict. Un environnement est ainsi vu comme un dictionnaire mais muni en plus d'un pointeur vers l'environnement qu'il étend.

```
class Env(dict):          # Env comme sous-classe de dict
    "Environment: a dict of {'var':val} pairs, with an outer Env"
    def __init__(self, params=(), args=(), outer=None):
        self.update(zip(params,args))     # zip est un itérateur
        self.outer = outer

    def find(self, var):
        "Find the first (innermost) Env where var appears"
        <à compléter en TP>
```

```
>>> E = Env(('x','y'),(10,20),Env(('x','z'),(2,3),None))
>>> E
{'y': 20, 'x': 10}
```

```
>>> E.find('x')
{'y': 20, 'x': 10}
>>> E.find('z')
{'z': 3, 'x': 2}
```



• Construction de l'environnement global :

```
def add_globals(env):
    "Add some Scheme standard procedures to an env."
    import operator as op              # local import
    env.update(
      {'+':op.add, '-':op.sub, '*':op.mul,
       '/':op.truediv, 'quotient':op.floordiv,
       'not':op.not_, '>':op.gt, '>=':op.ge,
       '=':op.eq})
    return env

global_env = add_globals(Env())
```

```
>>> *(2,3)
Syntax Error
>>> import operator as op
>>> op.mul(2,3)
6
```

```
>>> op.eq(3,1+2)
True
>>> op.pow(2,3)
8
```



# Troisième partie : l'évaluateur

• Il s'agit de définir l'évaluateur `micro_eval(expr,env)` chargé de calculer la valeur (ou l'effet) d'un expression dans un environnement de dictionnaires. Grâce à lui, je pourrai même faire des calculs µ-Scheme directement en Python :

```
>>> micro_eval(read('(+ 3 (* 2 4))'),global_env)
11
```

• Laissons-nous guider par la grammaire de µ-Scheme :

## 1. L'expression est un **symbole**

```
def micro_eval(expr,env):
    if type(expr) == str:
        <à compléter en TP>
    elif ...
```



## 2. L'expression n'est pas une liste

• Ni un symbole, ni une liste, il s'agit donc d'une constante.

```
def micro_eval(expr,env):
    ...
    elif type(expr) != list:
        <à compléter en TP>
    elif ...
```

## 3. L'expression est une **conditionnelle** `(if p q r)`

```
def micro_eval(expr,env):
    ...
    elif expr[0] == 'if':
        [_, p, q, r] = expr
        <à compléter en TP>
```

## 4. L'expression est une **affectation** (set! var e)

• On cherche dans l'environnement courant env le premier dictionnaire qui contient la valeur de var, et on met cette valeur à jour [mutation !] avec la valeur de e dans env. On utilise à cet effet les méthodes find de la classe Env et update de la classe dict.

```
def micro_eval(expr,env):
    ...
    elif expr[0] == 'set!':
        <à compléter en TP>
```

```
>>> import operator as op
>>> global_env.find('+').update({'+':op.mul})
>>> micro_eval(read('(+ 2 3)'),global_env)
6
```




## 5. L'expression est une **définition** (define var e)

• On introduit de force une nouvelle variable dans le premier dictionnaire de l'environnement env. Contrairement à Racket, on peut définir plusieurs fois var dans le même dictionnaire !

```
def micro_eval(expr,env):
    ...
    elif expr[0] == 'define':
        <à compléter en TP>
```

```
? (define foo (lambda (x) (begin (define y (+ x 1)) (+ x y))))
? (foo 5)
= 11
```

• La variable y est locale à la fonction foo en Scheme !

• La lecture doit se faire sur une même ligne : désagréable…



## 6. L'expression est une **séquence** (begin e1 e2 …)

• On évalue en séquence les expressions e1, e2… dans l'environnement env, et on retourne la dernière valeur obtenue.

```
def micro_eval(expr,env):
    ...
    elif expr[0] == 'begin':
        <à compléter en TP>
```

```
? (define x 1)
? (begin (define y (+ x 1)) (set! x (- x 1)) (+ x y))
= 2
? x
= 0
? y
= 2
```



## 7. L'expression est une **abstraction** (lambda (var …) expr)

• Le résultat est une **fermeture**. Il faut rendre la valeur d'une lambda Python dans l'environnement obtenu en étendant l'environnement env par le dictionnaire des liaisons var:$val_{env}$

```
def micro_eval(expr,env):
    ...
    elif expr[0] == 'lambda':          # fermeture !
        [_, Lvars, e] = expr
        return lambda *L: micro_eval(e,Env(Lvars,L,env))
```

• La notation Python lambda *L: e correspond en Scheme à la notation (lambda L e) où L représente la liste des paramètres dans une fonction d'arité variable.

```
>>> (lambda *L: len(L))(10,20,30,40)
4
```

## 7. L'expression est une **application** (expr$_1$ expr$_2$ …)

• **L'application d'une fonction à des arguments**. En Scheme la fonction en tête peut résulter d'un calcul, donc toutes les expressions sont évaluées dans env, puis on applique la valeur de tête (qui doit être une fonction) aux valeurs des arguments.

```
def micro_eval(expr,env):
    ...
    else:          # une application
        Lvals = [micro_eval(e,env) for e in expr]
        proc = Lvals.pop(0)
        return proc(*Lvals)
```

```
? (define x 3)
? ((if (> x 0) + *) (* 10 10) (* 20 20))
= 500
```



## Quatrième partie : le scribe

• Le **scribe** est chargé d'**écrire** les résultats dans la boucle REP. En effet, l'évaluateur passe la main à Python, donc retourne une sexp en Python, qu'il faut afficher sous la forme Scheme. La fonction `to_string(sexp)` se chargera de cette transformation.

• Oui, mais nous ne faisons que de l'arithmétique ! A quoi bon ?

• Exact, mais en TP vous allez implémenter les **listes** avec cons, car, cdr, etc. Et vous ne voudrez pas afficher une liste Python !

```
def to_string(sexp):
    <à compléter en TP>
```

```
? (cons (+ 1 2) (quote (bateaux en mer)))
= (3 bateaux en mer)   # et non [3, 'bateaux', 'en', 'mer']
```



## Faisons un point sur la situation

• Nous avons ouvert une fenêtre sur la programmation d'un interprète de langage A immergé dans un langage B.

• On peut ainsi se tailler un petit langage sur mesure (***DSL***) tout en restant au sein d'un langage hôte. Un programme en B peut donc faire appel à un programme en A (avec A implémenté en B).

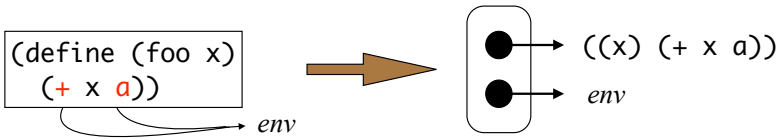
```
>>> def scheme(s):
        return micro_eval(read(s),global_env)
>>> 2 * scheme('(+ 3 4)')
14
```

• Attention cependant : on profite du langage hôte, mais on peut aussi être pénalisé par ses faiblesses…

***DSL*** == *Domain-Specific Language*



• L'interprète présenté **profite du langage hôte**, par exemple de la présence de `lambda` en Python (même si elle est restreinte). En grandeur réelle, on ne devrait pas, en construisant les fermetures à la main, sous la forme de couples *(texte,env)*, ce qui obligerait à programmer `apply`. Voir PCPS chap. 16 :




• L'interprète présenté **hérite des lacunes du langage hôte**, par exemple de la mauvaise gestion de la récurrence par Python avec lequel `(fac 1000)` fera claquer la pile de récursion.

• Ces problèmes sont résolubles. Voir par exemple (plus difficile) **Peter Norvig**, auteur de l'interprète présenté ici : `lispy1.html`.

*Directeur de la Recherche, Google !*

`http://norvig.com/lispy2.html`

Google

# Google's Peter Norvig: 'I have the best job in the world'

Google's director of research talks artificial intelligence, personal computing, mapping, and what the internet giant is planning next

norvig.com

research.google.com

**Director of Research** Google