RACKET - SCHEME

Mounir T. El Araki

mounir.elarakitantaoui@uic.ac.ma

Intelligence Artificielle

- Introduction à l'intelligence artificielle
- Historique de l'IA
- Problèmes de recherche
  - Recherche dans les graphes
  - CSP (Problèmes de Satisfaction de Contraintes)
- Les algorithmes génétiques
- Machine Learning
  - Nearest Neighbor
  - Les arbres de décisions
  - Les réseaux de neurones
- Représentation de la connaissance (Knowledge) et Inférence
  - Logique propositionnelle et 1ière degré
  - Système basé sur les règles
  - Langage naturel
- **LISP (LISt Processing) (Dr RACKET)**

Les expressions préfixées

- Notation infixe :
  - 2 + 3x/4-5 → (2+ ((3x)/4))-5 priorités des opérateurs
- Notation préfixe :
  - f(x, g(x))
- Notation postfixe : n!
- Les 3 notations : f(x+1, n!)

- Lisp, Scheme, DrRacket → Notation préfixe;
  - 2 + 3 * x; (+ 2 (* 3 x)) → Arbre
  - Données comme des arbres → Langages naturels, bases de connaissances, plan d'action, expressions algébriques, documents XML, compilateurs, etc.
- + * x log y + z 1 → (+ (* x (log y)) (+ z 1))

Les expressions préfixées

Maths	Lisp/Scheme
f(x,y,z)	(f x y z)
f(x+1, y)	(f (+ x 1) y)
x + f(y)	(+ x (f y))
p et q	(and p q)
si x alor x+1 sinon y	(if (> x 0) (+ x 1) y)

Le TopLevel

- Bienvenue dans DrRacket, version 5.3.6 [3m].
- Langage: Etudiant niveau avancé; memory limit: 128 MB.
- > (+ 2 3 4)
- 9
- > (+ (* 2 3)
- (* 3 4)
- (* 4 5)) ; ceci est un commentaire (ignoré)
- 38
- > (+2 3 4)
- function call: expected a function after the open parenthesis, but received 2
- La boucle TopLevel est donc un processus qui consiste à :

1. Lire une expression SCHEME grammaticalement correcte (construction d'un objet interne A
2. Evaluer l'objet A construit pour produire un objet B
3. Afficher une représentation externe de l'objet B sous la forme d'une suite de caractère

Retourner au point 1 …

Le TopLevel (Dictionnaire globale)

- > pi
- #i3.141592653589793
- > +
- +
- > foo
- foo: this variable is not defined
- > (define degre (/ pi 180))
- > (* 30 degre)
- #i0.5235987755982988
- > log
- log
- > ln
- ln: this variable is not defined

- Define est plus une définition de constante

Les nombres (les entiers)

- 0 , ensembles, Z, Q, R, Z
- Integer?, rational?, real?, complex?, number?,

- (modulo p q), (quotient p q) (gcd n_1 n_2 …) (lcm n_1 n_2 …) (random n), (zero? n), (even? n) (odd? n)
  - > (quotient 9 2)
  - 4
  - > (/ 9 2)
  - 4.5
- On peut définir des naturels positifs
  - (define (naturel? x)

    (and (integer? x) (>= x 0)

)
- Comment peut définir que deux nombres sont premier?
  - (define (premier? x y)

    (……)

)

Les rationnels

- ½ et 2/4 sont pareils
- Une seule représentation irréductible (numérateur et dénominateur sont premiers entre eux) et dénom. >0
  - > (define r (- 2/4 15/9))
  - > r
  - -1.16 (nombre rationnel)
  - > (numerator r)
  - -7
  - > (denominator r)
  - 6
- Pi est un rationnel inexact (#i3.141592653589793)
- > (exact? 1.5)
- true
- > (exact? #i1.5)
- false
-

Les réels

- (abs x), (floor x), (round x), (min x_1 x_2 …), (max x_1 x_2 …), (random), (<= x_1 x_2 …), (< x_1 x_2 …), (>= x_1 x_2 …), (> x_1 x_2 …), (= x_1 x_2 …)
- 2 types de réels 'exacts' et 'inexacts'
  - (conversion) exact-> inexact, inexact → exact
  - > (exact? 1.5)
  - true
  - > (exact? #i1.5)
  - false
  - > (exact->inexact 1/3) ;perte de précision!
  - #i0.3333333333333333
  - > (inexact->exact #i0.3333333333333333)
  - 6004799503160661/18014398509481984
  - > (floor 3.25) ; partie entière de 3.25
  - 3
  - > (min 5 #i6.7) ; 5 est converit en inexact
  - #i5.0
  - > (random)
  - #i0.36976432937918713
  - > (+ 1 #i2.3e-14)
  - #i1.000000000000023
  - > (+ 1 #i2.3e-17)
  - #i1.0
  - > (< 2 3 4 5 6)
  - true
  - > (> 98 43 3 1)
  - true
  - > (= 1 1.0 #i1.0)
  - true

Booléens et les expressions conditionnelles

- (if p q r)
- 'IF' n'est pas une fonction → forme spéciale (comme 'define', 'and', 'or', 'cond' et d'autres)
- (cond (t_1 e_1)

  ……

  (t_{n-1} e_{n-1})

  (else e_n)
- (and p q r …)
  - (if (not p) #f (if (not q) #f r))
- (or …..)

- > (boolean? false)
- true
- > (boolean? 1)
- false
- > (if (not (integer? (sqrt 2))) (* 2 3) (/ 1 0))
- 6
- > (and (integer? (sqrt 2)) (=0 (/ 1 0)))
- false
- > (or (= 2 3) (= 3 3) (= 0 (/ 1 0)))
- true
- >

L'évaluation d'une expression arithmétique

- Supposons une expression (f a b …)
- L'élément de tête de l'expression n'est pas toujours une fonction
  - > (procedure? log)
  - true
  - > (procedure? and)
  - and: expected an open parenthesis before and, but found none
- Si f est une fonction les éléments sont évalués de gauche à droite (dans le cas de Racket) mais non spécifié en général.
- La fonction f peut elle-même être calculée
  - (if (> x 0) (+ y 1) (- y 1)
  - ((if (> x 0) + -) y 1) ;; équivalente

L'évaluation d'une expression arithmétique

- Supposons une expression (f a b …) dont nous cherchons une valeur V au TopLevel.
  - Si f est le mot-clé d'une forme spéciale, on procède à un traitement spéciale!
  - Sinon on **évalue** tous les éléments de la forme parenthèsées et on obtient resp. les valeurs F A B ….
    - Si F n'est pas une procédure, erreur et revenir au TopLevel
    - Sinon on **applique** la procédure F aux valeurs trouvées A B …et obtenir la valeur V

- Dans le cas où on applique la fonction F sur tout les éléments évalués → Appel par valeur (la valeur est transmise à la fonction)
  - L'ordre d'évaluation est de l'intérieur vers l'extérieur
  - Exemple f(x,y) = x
  - f(1+1, 10^{20}) = f(2, 100000000000000000000) = 2 (appel par valeur)
  - f(1+1,10^{20}) = 1+1= 2 (Evaluation paresseuse → retarder l'évaluation des arguments)

Les fonctions

- (define (aire r)
- (* pi r r))

- > (define G 9.81)

- > (define (periode L)
- (* 2 pi (sqrt (/ L G))))

- > (aire 2)
- #i12.566370614359172

- > (periode 0.3)
- #i1.0987679728847353

- > (printf "l'aire d'un cercle ~a est ~a\n" 2 (aire 2))
- l'aire d'un cercle 2 est 12.566370614359172
- Printf n'est pas une fonction Racket

- > (check-expect (+ 2 3) 5)
- Le test est réussi !
- >
- (check-within (periode 0.3) 1.1 0.1)
- Les deux tests ont réussi !

- > (define (doubler x)
- (if (number? x)
- (* 2 x)
- (error "on attendait un nombre et non" x)))
- >
- (doubler 4)

Les fonctions anonymes

- Fonctions lambda
  - On peut construire des fonctions qui retournent des fonctions
  - Lambda, paramètres , corps

    - (lambda (x) (*2 x)) → fonction f(x)=2*x
    - (lambda (x y) (sqrt (+ x y)) → fonction f(x,y)=sqrt(x+y)
    - (lambda () 5)) → fonction f=5

    - > ((lambda (x) (* x x)) 3)
    - 9
    - > ((lambda (x y) (+ (sqrt x) y)) 5 (* 3 2))
    - #i8.23606797749979
    - > +
    - +
    - >

  - + n'est pas l'addition c'est le nom que porte l'addition.
  - Les fonctions anonymes sont gérées par un 'Garbage Collector' (GC)

Les variables globales

- > (define x 5)

- > (define (foo y)
  (* 2 x y)) ; x est globale

- > (foo 3)
- 30

- > (define (foo1 x y)
  (* 2 x y)) ; x est masquée

- > (foo1 3 4) ; x=3 dans foo1 est non 5
- 24
- >

Les variable locales

```
> (define (f x)
    (+ (* x x) (sqrt (+ (* x x) 1))))
> (f 2)
#i6.23606797749979

> (define (f1 x)
    (local [(define u (* x x))]
      (+ u (sqrt (+ u 1)))))
> (f1 2)
#i6.23606797749979
```

```
> (define (f3 x y)
    (local [(define x^2 (sqr x))
            (define y^2 (sqr y))
            ]
      (/ (+ x^2 y^2) (+ 1 (sqrt (+ x^2 (sqr y^2)))))))
> (f3 2 2)
#i1.4619519810524544
```

Les structures

- 'bonjour
  - 'bonjour
- > '(+ 2 3)
  - (list '+ 2 3)
- > (number->string 123)
  - "123"
- > (format "~a*~a=~a" 3 5 (* 3 5))
  - "3*5=15 »
- > (define-struct point (x y))
- > (define A (make-point 3 10))
- > A
  - (make-point 3 10)
- > (define A1 (make-posn 3 10))
- > A1
  - (make-posn 3 10)
- > (define B (make-point -8 1))
- > B
  - (make-point -8 1)
- > (point-x A)
  - 3
- > (point-y B)
  - 1
- > (point? A)
  - true
- > (point? 5)
  - false

Exemple: modélisation d'un nombre rationnel

```
> (define-struct rat (n d))

> (define (rationnel p q) ; retourne le rationnel p/q simplifié
    (cond ((= q 0) (error 'rationnel "Dénominateur nul!"))
          ((< q 0) (rationnel (- p) (- q))) ; on monte le signe
          (else (local [(define g (gcd p q))] ; calcul du pgcd
                  (make-rat (quotient p g) (quotient q g))
                  )
                )
          )
    )

> (rationnel 4 8)
(make-rat 1 2)
>
```

Programmation par récurrence

- n! = (n-1)! * n

```
> (define (fac n)  ; n entier naturel, retourne n!
    (if (= n 0)
        1
        (* (fac (- n 1)) n)))

>(define f50 (fac 50))

> f50
30414093201713378043612608166064768844377641568960512000000000000

> (string-length (number->string (fac 50)))
65
```

- (nbchiffres) ~ (+ 1 (nbchiffres (quotient n 10))) ; pour avoir le nombre de chiffres (hypothèse de récurence différentes de (n-1)

```
> (define (nbchiffres n)
    (if (< n 10)
        1
        (+ 1 (nbchiffres (quotient n 10)))))

> (nbchiffres 12323923)
8
```

Programmation par récurrence

```
> (define (fac n)
    (cond ((< n 0) (error 'fac "On attendait un entier positif:" n))
          ((= n 0) 1)
          (else (* (fac (- n 1)) n))))
> (fac 4)
24
> (fac -6)
fac: On attendait un entier positif:-6
> (define (fact n)
    (local [(define (aux n)
              (if (= n 0)
                  1
                  (* (aux (- n 1)) n)))]
      (if (>= n 0)
          (aux n)
          (error 'fact "On attendait un entier positif :" n))))
> (fact -6)
fact: On attendait un entier positif :-6
```

Les listes (ou listes chainées)

- > empty

empty

- > (and (empty? '()) (empty? empty) (list? empty))

true

- > (define L (cons 1 (cons 2 (cons 3 empty))))
- > L

(list 1 2 3)

- > (list? L)

true

- > (empty? L)

false

- > (cons 0 L)

(list 0 1 2 3)

- > (cons 'et '(les 2 ou 3 bateaux)) ; les symboles dans les listes

(list 'et 'les 2 'ou 3 'bateaux)

- > (define L '(les 2 ou 3 bateaux))
- > (first L)

'les

- > (second L)

2

- > (third L) ;;jusqu'à eighth

'ou

- > (equal? 'x (first (cons 'x L)))

true

- > (equal? L (rest (cons 'x L)))

True

- > (equal? L (cons (first L)(rest L)))

true

- > (equal? '(Do Ré Mi Fa) (cons 'Do '(Ré Mi Fa)))

true

- >

Primitives sur les listes

```
> (length empty)
0
(length '())
0
> (length '(()))
1
> (length '(les 2 ou 3 bateaux))
5
> (length '(les (2 ou 3) bateaux))
3
> (define (longueurListe L)
   (if (empty? L)
            0
        (+ 1 (longueurListe (rest
L)))))
> (longueurListe '(2 4 5 2 4 3))
6
>
```

```
> (member 'petit '(le petit poisson))
true
> (member 'petit '(le (petit poisson)
rouge))
false
> (define (EstCeMembre x L)
  (cond ((empty? L) false)
        ((equal? (first L) x) true)
        (else (EstCeMembre x (rest L)))))
> (EstCeMembre 3 '(les 3 petits
poissons))
True
>
 (list 'Do 'Ré 'Mi 'Fa) ; équivalent à (cons
   'Do (cons 'Ré (cons ..
(list 'Do 'Ré 'Mi 'Fa)
```

Primitives sur les listes
Construction/ Concaténation/ Reverse

- > (build-list 10 (lambda (i) (sqr (+ 1 i))))

(list 1 4 9 16 25 36 49 64 81 100)

- > (append '(le petit poisson) '(rouge est))

(list 'le 'petit 'poisson 'rouge 'est)

- > (define (MonAppend L1 L2)
  (if (empty? L1)
  L2
  (cons (first L1) (MonAppend (rest L1) L2))))

- > (MonAppend '(le petit poisson) '(rouge est))

(list 'le 'petit 'poisson 'rouge 'est)

- > (reverse '(le petit (poisson est) rouge))

(list 'rouge (list 'poisson 'est) 'petit 'le)

- > (define (MonReverse L)
  (if (empty? L)
  L
  (append (MonReverse (rest L)) (list (first L)))))

- > (MonReverse '(4 5 2 402 2))

(list 2 402 2 5 4)

Décomposition d'une Liste avec Match

(match expressions cas_1 cas_2 …)

```
> (define (traiter L)
    (match L
      ((list x y) (* x y))
      ((list x y z) (+ x y z))
      (_ 0)))
> (define L '(1 3 5))
> (traiter L)
9
> (traiter (rest L))
5
```

```
> (define (MonFirst L)
    (match L
      ((list x y ...) x)
      (_ (error "Invalide"))))
> (MonFirst '(1 4 5))
1
```

```
> (MonFirst 4)
```
Invalide

```
> (define (MonRest L)
    (match L
      ((list x y ...) y)
      (_ (error "Invalide"))))
> (MonRest '(4 3 4))
```
(list 3 4)

```
>
```

Recherche dans une liste

Accès a l'élément k d'une liste

```
> (list-ref '(une liste est un objet séquentiel) 4)
'objet

> (define (MonListeRef L k)
    (cond ((empty? L) (error 'MonListeRef "Pas assez d'éléments"))
          ((= k 0) (first L))
          (else (MonListeRef (rest L) (- k 1)))))

> (MonListeRef '(une liste est un objet séquentiel) 4)
'objet

> (MonListeRef '() 9)
MonListeRef: Pas assez d'éléments

> (MonListeRef '() 0)
MonListeRef: Pas assez d'éléments
```

Recherche d'un élément avec une condition (rechercher pred L echec)

```
> (define (rechercher pred L echec)
    (cond ((empty? L) echec)
          ((pred (first L)) (first L))
          (else (rechercher pred (rest L) echec))))

> (rechercher number? '(les deux ou trois ou quatre) #f)
false

> (rechercher number? '(les deux ou 3 ou 4) #f)
3

> (rechercher (lambda (n) (> n 10)) '(3  4 10 11 39 8 ) #f)
11

> (rechercher (lambda (x) (equal? x #f)) '(le boolean #t vaut vrai) '*Echec*)
'*Echec*
```

Recherche en profondeur dans une liste

```
> (define (symboles x)
    (cond ((empty? x) empty)
          ((symbol? x) (list x))
          ((number? x) empty)
          (else (append (symboles (first x)) (symboles (rest x))))))
```

```
> (symboles '(a ((b (c 3)) 4) d))
'(a b c d)
```

Les fonctions map/apply

- > (map (lambda(x y) (+ x y)) '(1 2 3 4 5) '(1 2 3 4 5))

'(2 4 6 8 10)

```
> (define (map f L)
      (if (empty? L)
          L
          (cons (f (first L)) (map f (rest L)))))
```

> (map sqr '(1 2 3 4 5))

'(1 4 9 16 25)

> (+ 1 2 3 4)

10

> (+ '(1 2 3 4))

+: contract violation

  expected: number?

  given: '(1 2 3 4)

> (apply + '(1 2 3 4))

10

> (apply max '(6 2 3 47 28 82 2 3))

82

Recherche (dans un graph) (Filter)

```
(define (filter select? L)
    (if (empty? L) empty
        (let ([elt (first L)]
              [le-rest (rest L)])
          (if (select? elt)
              (cons elt (filter select? le-rest))
              (filter select? le-rest)))))
```

```
(filter (lambda (x) (> x 0)) '(-4 5 6 9 8 -4 -2 9))
'(5 6 9 8 9)

(filter (lambda (x) (equal? (car x) 'A)) '((A B) (A C) (B D)(E F)(A M)))
'((A B) (A C))

(define (Succ-Graph Succ Graph)
   (filter (lambda (x) (equal? (car x) Succ)) Graph))
```

Recherche (dans un graph) (Map)

- (map (lambda (x) (first (rest x))) '((A B) (A C) (A M)))

'(B C M)

- (define (Successeurs Etat Graph)
  (map (lambda (x) (first (rest x)))
  (Succ-Graph Etat Graph)))

(Successeurs 'A '((A G) (A F) (G D) (D G) (A M) (M K)))

'(G F M)

Recherche en Largeur dans un graphe

```
(define (Rech-Larg Node Goal Successeurs Develop Graph)
  (cond ((empty? Node) empty)
        ((equal? (first Node) Goal)
          (printf "Succès ~a Objectif Atteint \n" (first Node)))
       ((member (first Node) Develop)
         (Rech-Larg (rest Node) Goal Successeurs Develop Graph))
       (else
         (printf "la liste développée est : ~a \n" Develop)
         (Rech-Larg
          (append (rest Node) (apply Successeurs (list (first Node) Graph)))
           Goal
           Successeurs
           (cons (first Node) Develop)
           Graph)
       )))
```

Recherche en profondeur dans un graphe

```
(define (Rech-Prof Node Goal Successeurs Develop Graph)
   (cond ((empty? Node) empty)
         ((equal? (first Node) Goal)
            (printf "Succès ~a Objectif Atteint \n" (first Node)))
         ((member (first Node) Develop)
           (Rech-Prof (rest Node) Goal Successeurs Develop Graph))
         (else
          (printf "la liste développée est : ~a \n" Develop)
          (Rech-Prof
          (append (apply Successeurs (list (first Node) Graph)) (rest Node))
          Goal
          Successeurs
          (cons (first Node) Develop)
          Graph)
        )))
```

Recherche dans un graphe (Appel)

- (define Graph1 '((S A) (S B) (A D) (A C) (B D) (B G) (D C) (D G)))
- (define Graph2 '((D G) (D C) (B G) (B D) (A C) (A D) (S B) (S A)))
- (Rech-Larg '(S) 'G Successeurs '() Graph1)
- (Rech-Larg '(S) 'G Successeurs '() Graph2)
- (Rech-Prof '(S) 'G Successeurs '() Graph1)
- (Rech-Prof '(S) 'G Successeurs '() Graph2)

Recherche dans un graphe (avec stratégie)

```
(define (Append-Rech Liste ListeSucc strategie)
    (cond ((equal? strategie 'largeur)
           (append Liste ListeSucc))
          ((equal? strategie 'profondeur)
           (append ListeSucc Liste))))
```

```
> (Append-Rech '(une liste et) '(et une autre) 'largeur)
'(une liste et et une autre)
> (Append-Rech '(et une liste) '(une autre liste)
   'profondeur)
'(une autre liste et une liste)
```

Recherche avec stratégie

```
(define (Rechercher Node Goal Successeurs Develop Graph strategie)
   (cond ((empty? Node) empty)
         ((equal? (first Node) Goal)
            (printf "Succès ~a Objectif Atteint \n" (first Node)))
         ((member (first Node) Develop)
            (Rechercher (rest Node) Goal Successeurs Develop Graph strategie))
         (else
            (printf "la liste développée est : ~a \n" Develop)
            (Rechercher
             (Append-Rech (rest Node)(apply Successeurs (list (first Node) Graph)) strategie)
             Goal
             Successeurs
             (cons (first Node) Develop)
             Graph
             strategie)
          )))
```

```
(Rechercher '(S) 'G Successeurs '() Graph1 'largeur)
```