

Le langage Racket (Lisp)

Un langage de programmation *fonctionnelle*

Historique

- ▶ Langage conçu par John McCarthy entre 1956 – 1959 au MIT pour des applications liées à l'intelligence artificielle
 - donc l'un des plus vieux langages toujours utilisés
 - 1960: McCarthy publie un article sur Lisp
- ▶ LISP = LISt Processor
- ▶ Issu de la théorie du λ -calcul
 - permet aux fonctions d'être les valeurs d'une expression
- ▶ Plusieurs dialectes:
 - Lisp 1.5 (1960)
 - Scheme (1975)
 - Common Lisp (1985)
 - PLT Scheme (Racket) (1995)

Naissance de Lisp

- ▶ Avec quelques opérateurs simples, une notation riche pour les fonctions et une structure de données simple:
 - On a un langage de programmation complet et expressif
- ▶ Langage riche: fonctionnel, symbolique.
- ▶ Syntaxe et sémantique simples et uniformes

9 concepts clé

1. Conditions (if–then–else)
2. Fonctions en tant que type de données
3. Récursivité
4. Variables en tant que pointeurs
5. Ramasse-miette
6. Le programme est une expression (non une suite d'énoncés)
7. Les symboles ou atomes
8. L'utilisation des listes et des arbres
9. Langage complet disponible en tout temps (read–eval–print)

Programmation fonctionnelle pure

- ▶ Un programme correspond à l'appel d'une fonction
- ▶ Une fonction est une composition de fonctions
- ▶ Les fonctions sont non-causales (ne dépendent que des paramètres transmis)
- ▶ Pas de variables, pas d'affectations
- ▶ Pas de boucles, pas d'énoncé de contrôle (outre la fonction if–then–else)

Programmation fonctionnelle

▶ Quelques concessions:

- Permettre la définition locale de certaines valeurs
- Permettre les affectations (donc les variables à portée lexicale)
- Permettre l'exécution en séquence (afin de pouvoir morceler le programme).

Applications de calcul symbolique

- ▶ Toute application non numérique, en particulier:
 - Intelligence artificielle
 - systèmes experts, interfaces en langages naturel, etc.
 - Raisonnement automatique (preuves de théorèmes, preuves de programmes,...)
 - Calcul formel
 - Jeux

Programmation fonctionnelle et Schème

- ▶ Dialecte de LISP concu au MIT en 1975, principalement pour l'éducation
- ▶ Initialement petit, est maintenant un langage complet.
- ▶ Standardisé par ANSI/IEEE, le langage continue à évoluer
- ▶ Généralement interprété, il peut aussi être compilé afin d'être efficacement exécuté.

Programmation fonctionnelle et Racket

- ▶ Développé par M. Felleisen en 1995
 - Initialement appelé PLT
 - But: créer un environnement de programmation orienté vers la pédagogie
 - e.g. intégration d'éléments graphiques simples
- ▶ 2010 PLT devient Racket
 - et son outil de programmation devient Dr Racket
- ▶ C'est un langage multi-paradigme
 - Mais appartenant à la famille Lisp

Notions de base

- ▶ La liste est la structure de données fondamentale
- ▶ Atome: un nombre, une chaîne de caractères ou un symbole.
 - Tous les types de données sont égaux
- ▶ Expression: un atome ou une liste
- ▶ Liste: une série d'expression entre parenthèses
 - Incluant la liste vide () nil, à la fois liste et atome
- ▶ Une fonction est un objet de première classe (first-class data) qui peut être créée, assignée à des variables, passée comme paramètre ou retournée comme valeur.

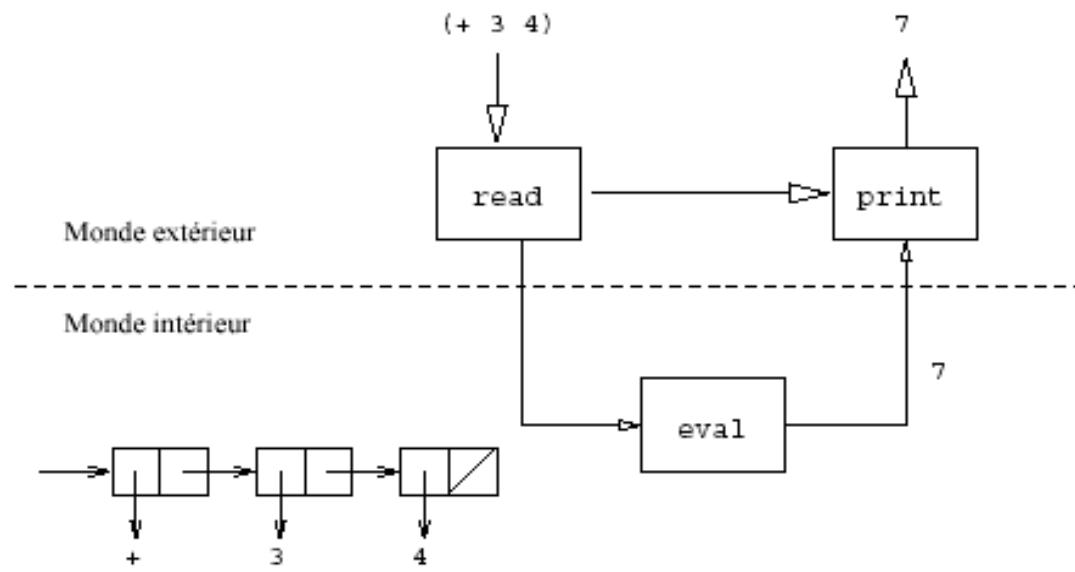
Règles d'évaluation

- ▶ Les constantes s'évaluent pour ce qu'elles sont.
- ▶ Les identificateurs s'évalue à la valeur qui leur est couramment attribuée.
- ▶ Les listes s'évalue en évaluant d'abord la première expression qui la compose;
 - la valeur de cette expression doit être une fonction
 - Les arguments de cette fonction sont les valeurs obtenues par l'évaluation des expressions contenues dans le reste de la liste

Une Session Lisp

- ▶ Dans sa forme la plus simple, Lisp utilise le modèle de programmation interactive READ–EVAL–PRINT

```
> (+ 3 4)  
7  
> (quit)
```



Évaluation des expressions

- ▶ La notation préfixée est utilisée dans l'écriture d'une expression
 - $3+4*5$ devient $(+ 3 (* 4 5))$
- ▶ Pour évaluer une expression, toutes les sous-expressions doivent être évaluées d'abord.
 - L'évaluation suit donc l'ordre normal de réduction
 - $(+ 3 (* 4 5))$
 - $(+ 3 20)$
 - 23

Représentation exacte et inexacte des nombres

- ▶ Les nombres réels peuvent être représentés de façon exacte ou inexacte
- ▶ La représentation exacte utilise une représentation par nombre rationnels, ainsi:

> (/ 50 6)
8 1/3 ; i.e. 8 et 1/3

- ▶ La plupart des opérations mathématiques retournent des nombres exactes
 - Sauf celles susceptibles de retourner des nombres irrationnels
 - e.g. sqrt

Représentation exacte et inexacte des nombres

- ▶ Un nombre écrit avec un point est un nombre inexact
 - La représentation IEEE754 est alors adoptée

```
> (/ 50.0 6)  
8.333333333333334
```

- ▶ La conversion de inexacte à exacte s'écrit:

```
> (inexact->exact (/ 50.0 6))  
8 187649984473771/562949953421312
```

- ▶ La fonction **truncate** retournera l'entier le plus près

```
> (truncate (/ 10 3))  
3  
> (truncate 3.7)  
3.0
```

Formes syntaxiques spéciales

- ▶ Certaines fonctions n'obéissent pas à la règle d'évaluation normale
 - ces fonctions sont dites de formes syntaxiques spéciales.
- ▶ L'évaluation de leurs arguments est plutôt différée jusqu'à ce qu'il soit requis d'en connaître la valeur.
- ▶ Les principales formes spéciales sont:
 1. L'alternative
 2. Le branchement conditionnel
 3. La création de portée locale
 4. La citation

1. L'alternative

```
(if (= x 0) infini (/ 1 x))
```

- ▶ L' expression qui suit le if est d' abord évaluée, si sa valeur est vraie (#t) alors le second argument est évalué et sa valeur est retournée sans évaluer le troisième argument
- ▶ sinon c' est le troisième argument qui est évalué et retourné.

2. Le branchement conditionnel

```
(cond ((<x xmin) xmin)
      ((>x xmax) xmax) (#t x))
```

- ▶ La fonction cond est suivie d' une série de listes composée de deux expressions. Si la première des deux expressions d' une de ces listes s'évalue à #t alors la valeur de la seconde expression est retournée
- ▶ sinon il faut passer a la liste suivante.
- ▶ Si aucune des listes s'évalue à T alors la valeur *nil* est retournée.

Exemple

```
(define (cout age)
  (cond ((or (<= age 3) (>= age 65)) 0)
        ((<= 4 age 6) 0.5)
        ((<= 7 age 12) 1.0)
        ((<= 13 age 15) 1.5)
        ((<= 16 age 18) 1.8)
        (else 2.0)))
```

3. La création de portée locale

```
(let ((pi 3) (d 4)) (* pi d))
```

12

- ▶ Le premier argument de cette fonction est une liste de liens créés entre un identificateur et une valeur
- ▶ Ces liens ne sont valides que pour l'évaluation de l'expression qui suit (il peut même y en avoir plusieurs afin de permettre l'exécution d'une séquence).

4. La citation

```
(quote (1 2 3))  
      (1 2 3)
```

- ▶ La fonction quote permet d'éviter que la liste en argument soit évaluée.
- ▶ Cette liste est plutôt retournée telle quelle.
- ▶ L'utilisation de cette fonction est nécessaire lorsque la première expression d'une liste ne s'évalue pas à une fonction.
 - La fonction quote s'écrit plus simplement:
`' (1 2 3)`
 - (write 'pi) affiche le symbole pi
 - (write pi) affiche 3.141592
 - (* 2.0 pi) retourne 6.283184
 - (* 2.0 'pi) *paramètre invalide*

Un exemple

```
(let ((a '(1 2 3)) (b '(3 4 5)))
  (traite a b))
```

équivaut à

```
(traite '(1 2 3) '(3 4 5))
```

Une fonction pour construire des listes

```
(list `a `b `c)  
(a b c)  
(list `(a b c))  

```

Définition d'une fonction

- ▶ Une définition associe l'expression d'une fonction à un nom:

```
(define (carre x) (* x x))
```

ou, de façon équivalente:

```
(define carre (lambda (x) (* x x)))  
(carre 2)
```

4

- ▶ L'expression (lambda(var1, var2, ...) exp1
exp2 ...) retourne une fonction où les variables sont des paramètres qui seront appliqués aux expressions.

```
((lambda (x) (* x x)) 3)
```

9

Définition d'une fonction

```
(define (fact n)
  ( if (> n 0)
    ( * n (fact (- n 1)))
    1
  )
)

(fact 40)
815915283247897734345611269596115894272000000000
```

Définition d'une fonction

```
(define (F-a-C temperature)
  ; conversion de oF a oC
  (/ (- temperature 32) 1.8))
```

```
(F-a-C 95)  
35
```

```
(define congelation 32)  
56  
(F-a-C congelation)  
0
```

Définition d'une fonction avec lambda

```
(define fct (lambda (f x) (f x x)))  
  
(fct + 13)  
26  
(fct * 4)  
16  
  
(let ((x `a))  
    (let ((f (lambda (y) (list x y))))  
        ; le x est celui défini dans le let englobant  
        (f `b)))  
(a b)
```

Lambda et Let

```
(let ((x 2) (y 3)) (+ x y))
```

est équivalent à:

```
((lambda (x y) (+ x y)) 2 3)
```

De façon générale:

```
((let (var val) ...) expr...)    <=>  
          ((lambda (var ...) expr...) val...))
```

GCD

```
(define gcd
  (lambda (a b)
    (if (= a b) a
        if (> a b)
        (gcd (- a b) b)
        (gcd a (- b a))))))
```

Fonctions Primitives

▶ Prédicats ?:

- des fonctions qui retournent #t ou #f.
 - (symbol? x)
#t si x est un symbole,
 - (number? x)
#t si x est un nombre,
 - (eq? x y)
#t si x et y sont des symboles égaux
 - (equal? x y)
si x et y sont des objets identiques (pas nécessairement atomiques)
 - (null? x)
si x est () - la liste vide
 - (pair? x)
si x est soit une liste ou soit une pair
 - (procedure? x)
si x est une fonction
 - (list? x)
si x est une liste

Tests d'égalité: eq?

- ▶ eq? compare si il s'agit du même objet
(compare les les adresses)
 - Ne pas utiliser pour comparer des nombres

```
(define chaine "bonjour")
(eq? chaine chaine)
#t
(eq? "bonjour" "bonjour")
#t
```

Tests d'égalité: eqv?

- ▶ eqv? Compare les valeurs (et types)
 - Ne pas utiliser sur des listes, des chaines de caractères et des fonctions

```
(eqv? 1 1)
#t
(eqv? 2 (+ 1 1))
#t
(eqv? 1 1.0)
#f
```

Tests d'égalité: equal?

- ▶ equal? compare les représentations

```
(equal? '(a 1 2) '(a 1 2))  
#t  
(equal? "bonjour" "bonjour")  
#t  
(equal? (list 1 2) '(1 2))  
#t  
(equal? 'a 'a)  
#t  
(equal? 2 2)  
#t
```

Structures du contrôle

Les structures de contrôle en Scheme sont simples. Il n'existe pas de boucles. Il y a l'application de fonctions, l'expression conditionnelle, et la séquence (une concession aux programmeurs habitués aux langages impératifs):

```
> (begin (print 'okay) (print ' (great) ))  
okay  
(great)
```

La valeur renvoyée par (begin ...) est la valeur du dernier terme.