Programmation fonctionnelle : Fiche

Sommaire

---

1. Introduction

1.1. Définition

> Paradigme de programmation de type déclaratif qui considère le calcul en tant qu'évaluation de fonctions mathématiques.

---

2. *Racket*

> Langage de programmation se basant sur ***LISP*** (*LISt Processing*, créé en 1958).

- Ce langage ce base sur l'utilisation de listes.
    - Soit une liste vide `()`
    - Soit quelque chose de la forme `(Tete|Queue)`. `Queue` étant une liste :

```
(1,2,3) = (1|(2,3)) = (1|(2|(3))) = (1|(2|(3|())))
```

- Toute fonction est récursive.
- Pas de boucles `for` ou `while`...

2.1. Exemples :

```
#lang racket

(define somme (lambda(x y))
    (+ x y))

(somme 5 7)           ; retourne 12
```

```
#lang racket

(define m-in-km 1000)

(define km->m (lambda(km))
    (* km m-in-km))

(km->m 23)            ; retourne 23000
```

```
#lang racket

#|
  Programme permettant de calculer n!
  Appel récursif
|#

(define facto (lambda(n)
        (if(zero? n)        ; ou (if(= n 0)
            1
```

```
            (* (facto(- n 1)) n))))

(facto 5)             ; retourne 120
```

2.2. Constantes prédéfinies

- Les chaînes de caractères :
  - `"Bonjour"`
  - `"1"`
  - `"hello world"`
- Les symboles :
  - `zero?`
  - `a`
  - `coucou`
- Les nombres :
  - `123`
  - `3.14158`
  - `6.5e-8`
- Les booléens
  - `#t` → vrai
  - `#f` → faux

2.3. Fonctions

- `define` permet d'associer un identifiant à une expression :

```
(define pi 3.14)          ; pi -> 3.14
(define a (+ 3 4))        ; a -> 7
```

- `'` permet d'empêcher l'évaluation de l'expression passée en argument

```
(define b '(+ 3 4))       ; a -> +34
```

- `lambda` permet d'avoir des paramètres

```
(define c (lambda(n) (+ n 1))   ; c -> n + 1
```

2.4. Opérateurs arithmétiques

- addition : `(+ 1 2 3 4)` → `10`
- soustraction : `(- 2 3)` → `-1`
- multiplication : `(* 2 3)` → `6`
- division : `(/ 6 3)` → `2`
- racine carrée : `(sqrt 4)` → `2`

2.5. Opérations sur les listes

- accès à la tête : `(car '(1 2 3))` → `1`
- accès à la queue : `(cdr '(1 2 3))` → `(2 3)`
    - **Attention :** la queue est une liste, pour accéder au dernier élément :

```
(car(cdr(cdr '(1 2 3))))  ; -> 3
(cadr(cdr '(1 2 3)))      ; -> 3

(car(car()))              ; = caar()
(car(cdr()))              ; = cadr()
```

2.6. Construction d'une liste

- Ajout de `E` en tête de la liste `L` : `cons(E L)`

```
(cons 1 '(2 3))           ; -> '(1 2 3)
(cons '() '())            ; -> (())
(cons 1 (cons('() '())))  ; -> (1())
```

2.6.1. Exercice

```
(cons 1 (cons 2 (cons(3 '())))                          ; -> (1 2 3)
(cons 1 (cons 2 (cons (cons 3 '()) '())))               ; -> (1(2(3)))
(cons (cons (cons '() '()) '()) '())                    ; -> ((((()))))
(cons (cons (cons 1 '()) (cons 2 '())) (cons 3 '()))    ; -> (((1) 2) 3)
```

2.7. Prédicats prédéfinis

```
(null? l)       ; #t si l = ()
                ; #f sinon
```

```
(list? x)       ; #t si x est une liste
                ; #f sinon

(boolean? x)    ; #t si x est un booléen
                ; #f sinon

(procedure? x)  ; #t si x est une procédure
                ; #f sinon

(number? x)     ; #t si x est un nombre
                ; #f sinon

(integer? x)    ; #t si x est un entier
                ; #f sinon
```

2.7.1. Alternative : "si ... alors ... sinon"

```
(if <expression booléenne>
  <expression alors>
  <expression sinon>)

; Exemple :

(if (integer? n)
  (+ 3 n)
  "pas un entier")
```

- Une seule ligne pour `alors` et `sinon` car sinon on serait dans de la programmation procédurale et non fonctionnelle

2.8. Exercices

2.8.1. Somme d'une liste d'entiers

```
#lang racket

(define somme-liste
  (lambda(L)
    (if(null? L)
    0
    (+ (car L) (somme-liste(cdr L))))))
```

```
(somme-liste '(1 2 3 4))        ; retourne 10
```

2.8.2. Somme jusqu'à n

```
#lang racket

(define somme
  (lambda(n)
    (if(= n 1)
    1
    (+ n (somme(- n 1))))))

(somme 5)
```

2.8.3. Copie d'une liste

```
#lang racket

(define copie
  (lambda(L)
    (if(null? L)
    '()
    (cons (car L) (copie(cdr L))))))

(copie '(1 2 3))
```

2.8.4. Egalité de deux listes

```
#lang racket

(define comp
  (lambda(L1 L2)
    (if(and (null? L1) (null? L2))
        #t
        (if(or (null? L1) (null? L2))
            #f
            (if(= (car L1) (car L2))
                (comp (cdr L1) (cdr L2))
                #f)))))
```

```
(comp '(1 2 3) '(1 2 3))      ; #t
(comp '(1 2 3) '(1 5 3))      ; #f
(comp '(1 2 3 4) '(1 2 3))    ; #f
(comp '() '())                ; #t
```

2.9. La conditionnelle `cond`

```
(cond [(<expression booléenne 1> <expression alors>)]
      [(<expression booléenne 2> <expression alors>)]
      [(<expression booléenne 3> <expression alors>)]
      ...
      [#t <expression sinon>])
```

2.10. Test d'égalité

```
; Valide pour les nombres
(= <expression 1> <expression 2>)

; #t si les expressions ont la même valeur
(eq? <expression 1> <expression 2>)

; #t si les expressions ont la même structure
(equal? <expression 1> <expression 2>)
```

2.11. Définition de variables locales

```
; L'évaluation des expressions se fait en parallèle
(let ((<variable 1> <expression 1>)
      (<variable 2> <expression 2>)
      ...
      (<variable n> <expression n>))
  <expression>)

; L'évaluation des expressions se fait en série
(let* ((<variable 1> <expression 1>)
       (<variable 2> <expression 2>)
       ...
```

```
        (<variable n> <expression n>))
  <expression>)
```

2.11.1 Exemple

```
; En parallèle
(let  ((a 20)         ; a = 20
       (b (* a 10))   ; b = 10
       (c (+ a b)))   ; c = 3
  (+ a b c))          ; -> 60

; En série
(let* ((a 20)         ; a = 20
       (b (* a 10))   ; b = 200
       (c (+ a b)))   ; c = 220
  (+ a b c))          ; -> 440
```

2.12 Algorithme de tri : Quicksort

```
#lang racket

(define concatenation
  (lambda (L1 L2)
    (if (null? L1)
      L2
      (cons (car L1) (concatenation (cdr L1) L2)))))

(concatenation '(1 2) '(2 8 9))   ; retourne (1 2 2 8 9)

(define scinder
  (lambda (L P op)
    (if(null? L)
      '()
      (if (op (car L) P)
        (cons (car L) (scinder (cdr L) P op))
        (scinder (cdr L) P op)))))

(scinder '(4 1 3 7 1) 2 <)        ; retourne (1 1)
(scinder '(4 1 3 7 1) 2 >)        ; retourne (4 3 7)


(define scinder2
```

```
  (lambda (L P C)
    (if (null? L)
    C
    (let* ((LI (car C))
           (LS (car (cdr C))))
          (if ( < (car L) P)
            (scinder2 (cdr L) P (list (cons (car L) LI) LS))
            (scinder2 (cdr L) P (list LI (cons (car L) LS)))))))))

;list crée une liste avec les éléments données en paramètre


(define scinder
  (lambda (L P)
    (scinder2 L P '(()()) )))

(scinder '(1 4 3) 2)              ; retourne ((1) (3 4))

(define quicksort
  (lambda (L)
    (if (null? L)
      '()
      (let* ((P (car L))
             (C (scinder (cdr  L) P))
             (A (car C))
             (B (cdr C))
             (AT (quicksort A))
             (BT (quicksort B)))
            (concatenation AT (cons P BT))))))

(quicksort '(1 4 2 3 5))          ; retourne (1 2 3 4 5)
```

3. Les arbres binaires ordonnés

- Un arbre c'est :
    - Soit l'arbre vide **(Liste vide)**
    - Soit un noeud qui a au plus un fils gauche et un fils droite
        - Le fils gauche est un arbre
        - Le fils droite est un arbre

⇒ On peut représenter un arbre binaire ordonné par une liste de listes `(V G D)` (ex : `(10 (5 (2 () ()) (7 () ())) (15 () (20 () ())))`)

3.1. Représentation d'un arbre binaire ordonné

```
(define estVide?
  (lambda (ND)
    (null? ND)))

(define egaux?
  (lambda (ND1 ND2)
    (if (estVide? ND1)
        (estVide? ND2)
        (if (estVide? ND2)
            #f
            (and (= (valeur ND1) (valeur ND2))
                 (egaux? (gauche ND1)(gauche ND2))
                 (egaux? (droite ND1)(droite ND2)))))))

(define valeur
  (lambda (ND)
    (car ND)))

(define gauche
  (lambda (ND)
    (cadr ND)))

(define droite
  (lambda (ND)
    (caddr ND)))
```

3.2. Parcours d'un arbre binaire ordonné

- **Parcours infixé**
    - On parcourt le sous-arbre gauche
    - On visite la racine
    - On parcourt le sous-arbre droit

```
(define infixe
  (lambda (ND)
    (if (estVide? ND)
        '()
        (concatenation (infixe (gauche ND))
                       (cons (valeur ND) (infixe (droite ND)))))))
```

- **Parcours préfixé**
    - On visite la racine
    - On parcourt le sous-arbre gauche
    - On parcourt le sous-arbre droit

```
(define prefixe
  (lambda (ND)
    (if (estVide? ND)
      '()
      (concatenation (cons (valeur ND) (prefixe (gauche ND)))
                     (prefixe (droite ND))))))
```

- **Parcours postfixé**
    - On parcourt le sous-arbre gauche
    - On parcourt le sous-arbre droite
    - On visite la racine

```
(define postfixe
  (lambda (ND)
    (if (estVide? ND)
      '()
      (concatenation (concatenation (postfixe (gauche ND))
                                    (postfixe (droite ND))
                     (cons (valeur ND)
                           '()))))))
```

3.3. Construction d'un noeud

```
;Construit un arbre
(define construction
  (lambda (V G D)
    (list V G D)))

(define copie
  (lambda (ND)
    (if (estVide? ND)
        '()
        (construction (valeur ND)
                      (copie (gauche ND))
                      (copie (droite ND))))))
```

```
;Ajoute un noeud
(define ajout
  (lambda(ND V)
    (if (estVide? ND)
      (construction V '() '())
      (if (> V (valeur ND))
          (construction (valeur ND)
                        (copie (gauche ND))
                        (ajout (droite ND) V))
          (construction (valeur ND)
                        (ajout (gauche ND) V)
                        (copie (droite ND)))))))
```

3.4. Exercice

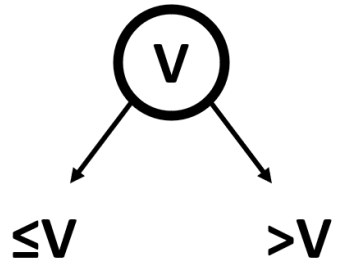
> Suppression d'une valeur dans un ABO

```
(define greffeDroite
  (lambda(greffe A)
    (if (estVide? A)
      greffe
      (creationNoeud (valeur A)
                     (gauche A)
                     (if (estVide? (droite A))
                       greffe
                       (greffeDroite greffe (droite A)))))))

(define detruire
  (lambda(N V)
    (if (estVide? N)
      '()
      (if (= (valeur N) V)
        (greffeDroite (droite N) (copie (gauche N)))
        (if (> V (valeur N))
          (creationNoeud (valeur N)
                         (gauche N)
                         (detruire (droite N) V))
          (creationNoeud (valeur N)
                         (detruire (gauche N) V)
                         (droite N)))))))
```

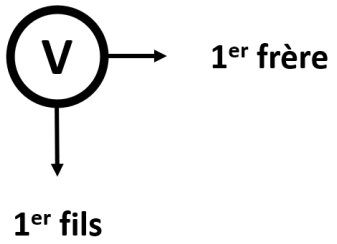
3.5. Autres arbres binaires

Jusqu'à présent, un arbre était ceci :



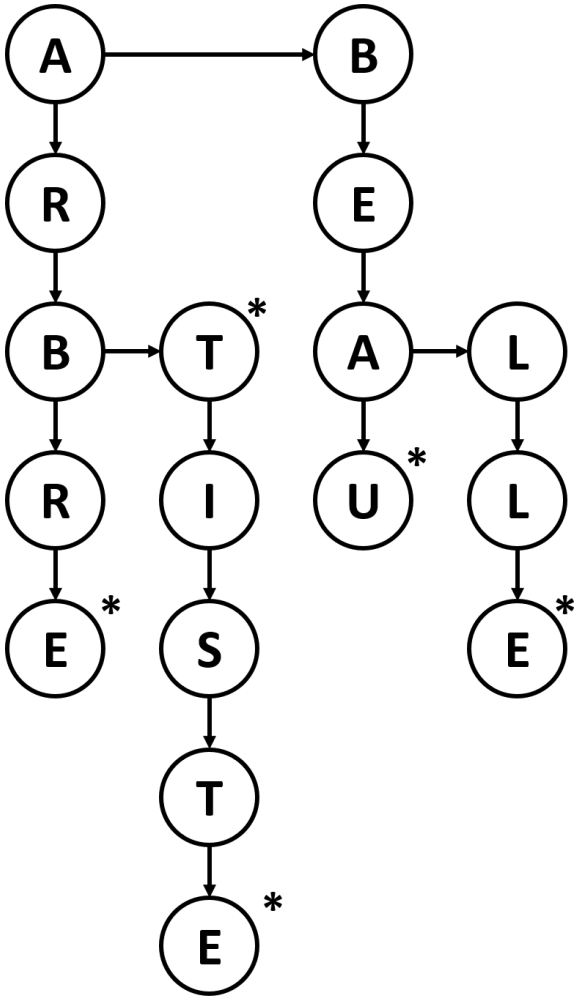
> *N.B. :* $V = valeur$

Mais désormais, nous allons étudié des arbres sous cette forme :



Par exemple, nous pouvons dresser un dictionnaire :

- Mots : ARBRE, ART, ARTISTE, BEAU, BELLE
- Arbre :



Nous représenterons ces arbres ainsi : `V FILS FRERE FIN` (Avec `FIN` un booléen, qui vaut vrai si le mot est terminé (représenté par ∗ sur le dessin))

3.6. Exercice

> Fonction qui vérifie si un mot est présent dans un dictionnaire

```
(define valeur
  (lambda (D)
    (car D)))

(define fils
  (lambda (D)
    (cadr D)))

(define frere
  (lambda (D)
    (caddr D)))

(define fin?
  (lambda (D)
    (cadddr D)))

(define present?
  (lambda (M D)
    (if (null? D)
      #f
      (if (equal? (car M) (valeur D))
        (if (null? M)
          (fin? D)
          (present? (cdr M) (fils D)))
        (present? M (frere D))))))
```

> Fonction qui ajoute un mot dans un dictionnaire

```
(define creationNoeud
  (lambda (V F Fr F?)
    (list V F Fr F?)))

(define copie
  (lambda (D)
    (if (null? D)
      '()
      (creationNoeud (valeur D)
                     (copie (fils D))
                     (copie (frere D))
                     (fin? D)))))

(define ajout
  (lambda(M D)
    (if (null? D)
      (if (null? M)
        D
        (if (null? (cdr M))
          (creationNoeud (car M) '() '() #t)
          (creationNoeud (car M) (ajout (cdr M) D) '() #f)))
      (if (eq? (car M) (valeur D))
        (creationNoeud (valeur D)
                       (ajout (cdr M) (fils D))
                       (frere D)
                       (fin? D))
        (creationNoeud (valeur D)
                       (fils D)
                       (ajout M (frere D))
                       (fin? D))))))
```

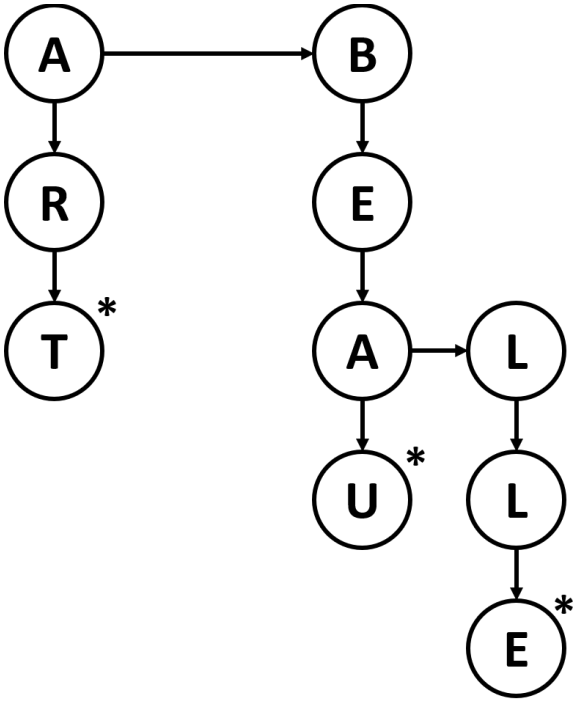
TEST

- Entrée :

```
(define D (ajout '(B E L L E) (ajout '(B E A U) (ajout '(A R T) '()))))

D
```

- Sortie (représentation graphique) :


A
B
R
E
T
*
A
L
U
*
L
E
*

> Fonction qui supprime un mot dans un dictionnaire

```
(define suppression
  (lambda (M D)
    (if (or (null? D) (null? M))
      D
      (if (eq? (car M) (valeur D))
        (if (null? (cdr M))
          (if (fin? D)
            (creationNoeud (valeur D)
                           (fils D)
                           (frere D)
                           #f)
            D)
          (creationNoeud (valeur D)
                         (suppression (cdr M) (fils D))
                         (frere D)
                         (fin? D)))
        (creationNoeud (valeur D)
                       (fils D)
                       (suppression M (frere D))
                       (fin? D))))))
```

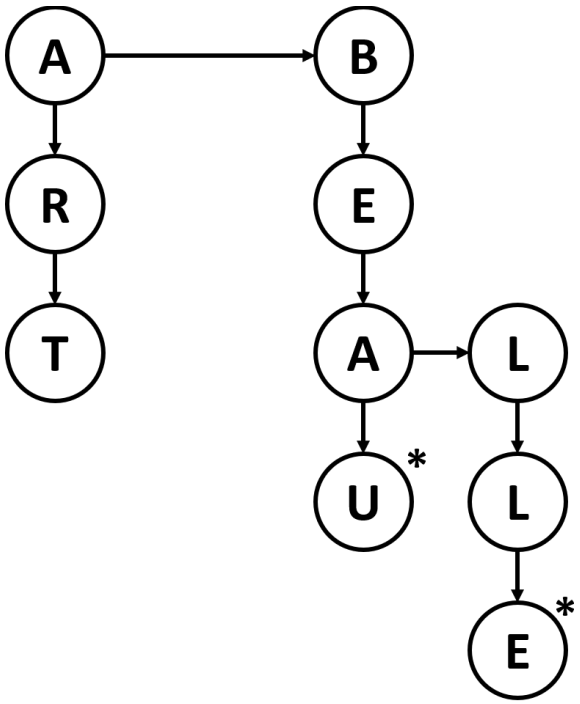
TEST

- Entrée :

```
(define D1 (suppression '(A R T) D))

D1
```

- Sortie (représentation graphique) :



> Fonction qui compte le nombre de mots dans un dictionnaire

```
(define compteur
  (lambda (D)
    (if (null? D)
      0
      (if (fin? D)
        (+ (compteur (frere D)) (compteur (fils D)) 1)
        (+ (compteur (frere D)) (compteur (fils D)))))))
```

TEST

- Entrées :

```
(compteur D)
(compteur D1)
```

- Sorties :

```
3
2
```

> Version alternative

```
(define compteurV2
  (lambda (D)
    (if (null? D)
      0
      (+ (if (fin? D) 1 0)
        (compteurV2 (fils D))
        (compteurV2 (frere D))))))
```

TEST

- Entrées :

```
(compteurV2 D)
(compteurV2 D1)
```

- Sorties :

```
3
2
```

> Fonction qui affiche tous les mots d'un dictionnaire

```
(define renverse-handler
  (lambda (L P)
    (if (null? L)
      P
      (renverse-handler (cdr L) (cons (car L) P)))))

(define renverse
  (lambda (L)
    (renverse-handler L '())))

(define liste-mots-handler
  (lambda (D M)
    (if (null? D)
        '()
        (if (fin? D)
          (cons
            (renverse (cons (valeur D) M))
            (append
              (liste-mots-handler (fils D) (cons (valeur D) M))
              (liste-mots-handler (frere D) M)))
          (append
            (liste-mots-handler (fils D) (cons (valeur D) M))
            (liste-mots-handler (frere D) M))))))

(define liste-mots
  (lambda (D)
    (liste-mots-handler D '())))
```

TEST

- Entrée :

```
(liste-mots D)
```

- Sortie :

```
'((A R T) (B E A U) (B E L L E))
```

> Fonction qui contruit tous les sous-ensembles d'une liste

```
(define sous-listes
  (lambda (L)
    (if (null? L)
      '(())
      (append
        (map
          (lambda (x) (cons (car L) x))
          (sous-listes (cdr L)))
        (sous-listes (cdr L))))))
```

TEST

- Entrée :

```
(sous-listes '(1 2 3))
```

- Sortie :

```
'((1 2 3) (1 2) (1 3) (1) (2 3) (2) (3) ())
```