

Le langage Racket

- Un langage de programmation fonctionnelle
- 10/11/2021

Plan du cours

- 1. Correction contrôle 2
- 2. . Arbres (suite)
 - (Quelques rappels)
 - 2.5. Arbres de recherche
- 3. Exercices

```

#lang racket
(require racket/trace)
"hello, world"

; contrôle 2

; Exo 1:
; premier et deuxième éléments ont été échangés |
(define (echange l)
(define (f l k i)
  (if (> i 0)
      (f (rest l) (cons (first l) k) (- i 1))
      (append k l)))
  (f l '() 2)
)
(trace echange)
(echange '(2 10 4 9 6 7 9))

```

; où le premier et le troisième éléments ont été échangés

```

(define (swap l)
(define (f l k i)
  (if (> i 0)
      (f (rest l) (cons (first l) k) (- i 1))
      (append k l)))
  (f l '() 3)
)
(trace swap)
(swap '(2 10 4 9 6 7 9))

```

```

; Exo 2:
; Une fonction qui prend une liste de nombres en argument et qui renvoie son maximum.
(define (max l)
(define (f l M)
  (if (empty? l) M
      (let ([i (first l)])
        (if (>= i M)
            (f (rest l) i)
            (f (rest l) M)
            )
        )
      )
  (f l (first l))
)

(trace max)
(max '(-5 -3 -100 -8 -2 -14 -10))

```

;Une fonction qui prend en argument une liste de nombre et qui en retourne la moyenne
(define (moyenne l)
(define (f l k i)
 (if (empty? l) (/ k i)
 (f (rest l) (+ k (first l)) (+ i 1))
))
 (f l 0 0)
)
;(trace moyenne)
;(moyenne '(2 4 9 0 10))

- -

```
(define (member l k)
  (if (empty? l) l
      (if (= k (first l))
          l
          (member (rest l) k)
      )))
```

;Exo 5

```
(define (permute l R)
  (if (= (length l) 1) (append R (list (last l)))
      (if (empty? l) R
          (let ([i (first (rest l))])
            (permute (rest (rest l)) (append R (list i) (list (first l)))))))
  )
  )
)
)
```

1. Arbres binaires (Rappel)

- ① Considérer les nœuds non-terminaux comme des listes de taille variable et les feuilles comme des éléments.

En utilisant la représentation ①, dessiner les arbres suivants :

arbre1 : '(1 (2 3 (4 5 6))),

arbre2 : '(1 (2 3 (4 5 6)) (7 (8 9 10))),

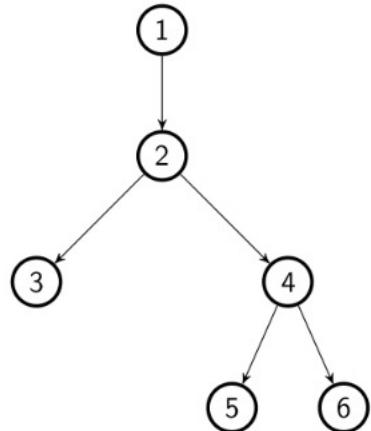


Fig. 17 – arbre1.

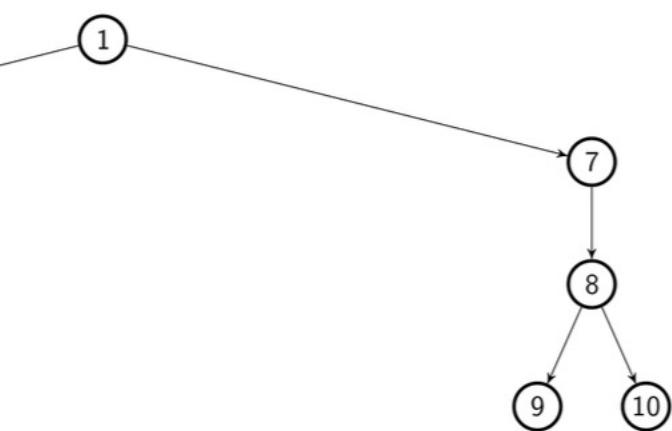
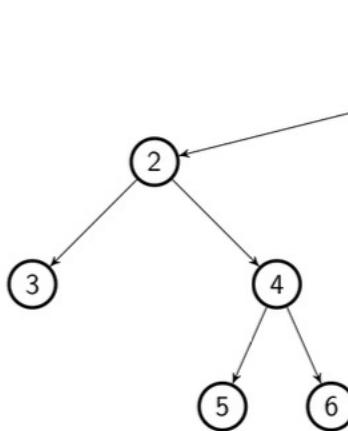


Fig. 18 – arbre2.

1. Arbres binaires (Rappel)

Afin de simplifier/clarifier l'utilisation des arbres, il est possible de définir des fonctions de manipulation des arbres dans une interface (nommée **bt1**) :

```
1 (define tree list)
2 (define (leaf e) e)
3 (define (leaf? e) (not(list? e)))
4 (define (not-leaf? e) (list? e))
5 (define (root T) (first T))
6 (define (L-subtree T) (first(rest T)))
7 (define (R-subtree T) (first(rest(rest T))))
8 (define (has-L-subtree? T) (>(length T) 1))
9 (define (has-R-subtree? T) (>(length T) 2))
```

Ici, comme présenté dans ①, les nœuds intermédiaires sont des listes de taille variable et les feuilles sont des éléments.

1. Arbres de recherche

- Les nœuds sont classés selon la valeur des étiquettes.
- Pour un nœud n de valeur v , les nœuds de valeurs $< v$ seront classés dans les sous arbres gauche de n .
- Les nœuds de valeurs $> v$ seront classés dans les sous arbres droits de n .

Exemple: Dessiner les arbres suivants:

L1: '(6 3 7 8 5 2)

L2: '(7 6 8 3 2 5)

1. Arbres de recherche

Exemple:

L1: '(6 3 7 8 5 2)

L2: '(7 6 8 3 2 5)

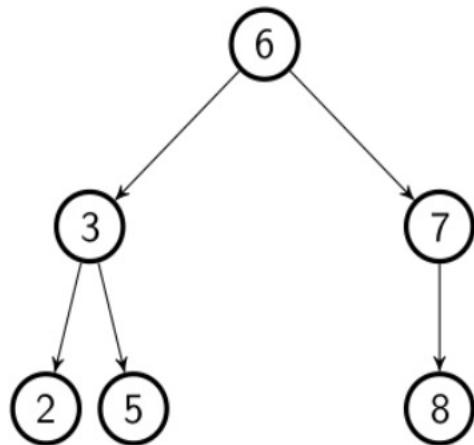


Fig. 4 – arbre de L1.

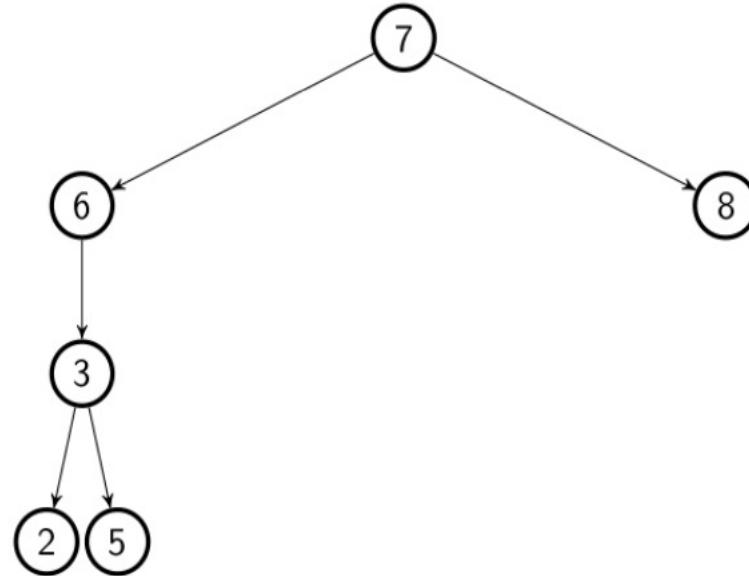


Fig. 5 – arbre de L2.

1. Arbres de recherche

En utilisant la représentation 6 , il est possible de définir des fonctions de manipulation des arbres binaires de recherche dans une interface (nommée bt6)

1. Arbres de recherche

Rappel:

La représentation 6:

- ⑥ Considérer les nœuds non-terminaux comme des listes de taille 3 (en les complétant si nécessaire avec des listes vides) et les feuilles comme des listes de taille 3.

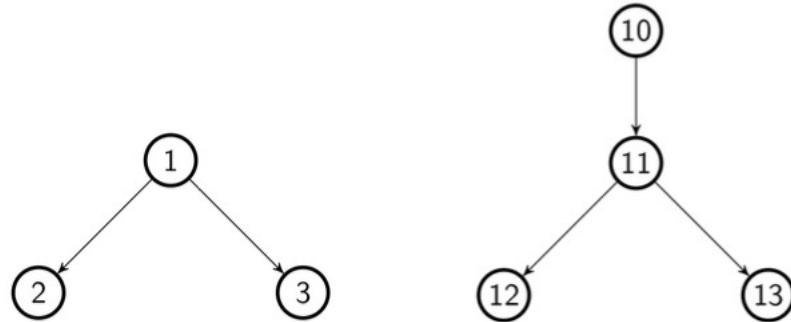


Fig. 2 – Arbres à 3 et 4 nœuds.

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

2 ' (10 (11 (12 () ()) (13 () ())) ())

```
1 (define (tree val L R) (list val L R))
2 (define (leaf val) (list val empty empty))
3 (define (root T) (first T))
4 (define (L-subtree T) (first(rest T)))
5 (define (R-subtree T) (first(rest(rest T))))
6 (define (has-L-subtree? T) (not (empty? (L-subtree T))))
7 (define (has-R-subtree? T) (not (empty? (R-subtree T))))
8 (define (add T val)
9   (if (empty? T) (leaf val)
10     (let ([r (root T)])
11       (if (< val r)
12         (if (has-L-subtree? T)
13           (tree r (add (L-subtree T) val) (R-subtree T))
14           (tree r (leaf val) (R-subtree T)))
15         (if (has-R-subtree? T)
16           (tree r (L-subtree T) (add (R-subtree T) val))
17           (tree r (L-subtree T) (leaf val)))
18         ))))
19 (define (addl T L)
20   (if (empty? L) T
21     (addl (add T (first L)) (rest L))
22   ))
```

1. Arbres de recherche

```
1 (add(add(add(add(add empty 6) 3) 7) 8) 5) 2)
2 (addl empty '(6 3 7 8 5 2))
3 (add(add(add(add(add empty 7) 6) 8) 3) 2) 5)
4 (addl empty '(7 6 8 3 2 5))
```

```
'(6 (3 (2 () ()) (5 () ())) (7 () (8 () ())))
'(6 (3 (2 () ()) (5 () ())) (7 () (8 () ())))
'(7 (6 (3 (2 () ()) (5 () ())) ()) (8 () ()))
'(7 (6 (3 (2 () ()) (5 () ())) ()) (8 () ()))
```

3. Exercices

Question 16

Conformément à la représentation ⑥, dessiner les arbres suivants :

arbre8 : (addl empty '(7 1 13 17 9))

arbre9 : (addl empty '(5 8 7 6 13 3 9 1 15))

arbre10 : (addl empty '(5 4 7 6 2 11 12 9 3 10 1 8 13))

3. Exercices

(Solution Exo 16)

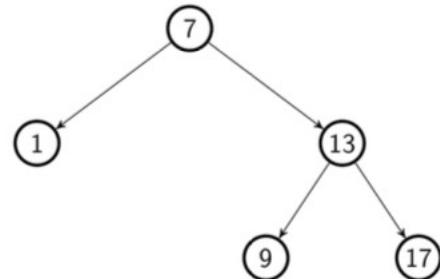


Fig. 22 – **arbre8**

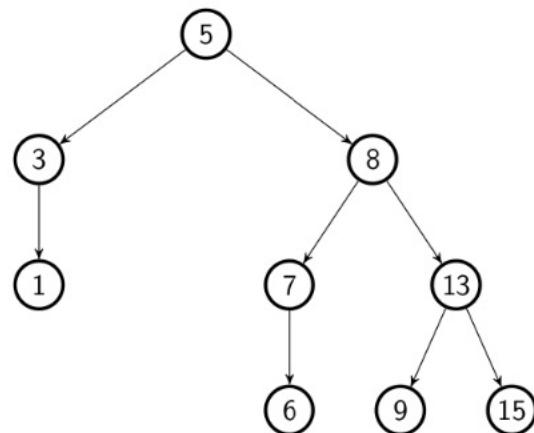


Fig. 23 – **arbre9**

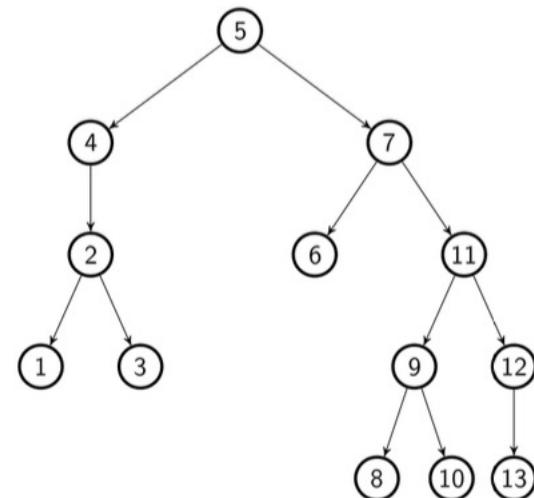


Fig. 24 – **arbre10**

3. Exercices

Question 2

En utilisant l'interface **bt1**, définir une fonction **nb-nodes** permettant de connaître le nombre de noeuds d'un arbre.

Appeler (**nb-nodes arbre1**) retourne 6.

Appeler (**nb-nodes arbre2**) retourne 10.

(*fonctions utiles : leaf?, has-R-subtree?, L-subtree, R-subtree*)

arbre1 : '(1 (2 3 (4 5 6))),

arbre2 : '(1 (2 3 (4 5 6)) (7 (8 9 10))),

3. Exercices

```
1 (define tree list)
2 (define (leaf e) e)
3 (define (leaf? e) (not(list? e)))
4 (define (not-leaf? e) (list? e))
5 (define (root T) (first T))
6 (define (L-subtree T) (first(rest T)))
7 (define (R-subtree T) (first(rest(rest T))))
8 (define (has-L-subtree? T) (>(length T) 1))
9 (define (has-R-subtree? T) (>(length T) 2))
```

- ① Considérer les noeuds non-terminaux comme des listes de taille variable et les feuilles comme des éléments.

3. Exercices

Solution 2

```
1 (define (nb-nodes T)
2   (if (leaf? T) 1
3     (if (has-R-subtree? T)
4       (+ 1 (nb-nodes (L-subtree T)) (nb-nodes (R-subtree T)))
5         (+ 1 (nb-nodes (L-subtree T))))
6       )))
7 (nb-nodes '(1 (2 3 (4 5 6))))
8 (nb-nodes '(1 (2 3 (4 5 6)) (7 (8 9 10))))
```

```
6  
10
```

3. Exercices

Question 3

En utilisant l'interface `bt1`, définir une fonction `h` permettant de connaître la hauteur d'un arbre.

Appeler (`h arbre1`) retourne 3.

Appeler (`h arbre3`) retourne 2.

(*fonctions utiles : leaf?, max, has-R-subtree, L-subtree, R-subtree*)

3. Exercices

Solution 3

```
1 (define (h T)
2   (if (leaf? T) 0
3     (if (has-R-subtree? T)
4       (+ 1 (max (h (L-subtree T))
5                  (h (R-subtree T))))
6       (+ 1 (h (L-subtree T))))
7     )))
8 (h '(1 (2 3 (4 5 6))))
9 (h '(1 (2 3)))
```

```
3
2
```

3. Exercices

Question 4

En utilisant l'interface **bt1**, définir une fonction **nb-leaves** permettant de connaître le nombre de feuilles d'un arbre.

Appeler (**nb-leaves arbre1**) retourne 3.

Appeler (**nb-leaves arbre2**) retourne 5.

(*fonctions utiles : leaf?, has-R-subtree, L-subtree, R-subtree*)

3. Exercices

Solution 4

```
1 (define (nb-leaves T)
2   (if (leaf? T) 1
3     (if (has-R-subtree? T)
4       (+ (nb-leaves (L-subtree T))
5           (nb-leaves (R-subtree T)))
6       (nb-leaves (L-subtree T))
7     )))
8 (nb-leaves '(1 (2 3 (4 5 6))))
9 (nb-leaves '(1 (2 3 (4 5 6)) (7 (8 9 10))))
```

```
3
5
```

3. Exercices

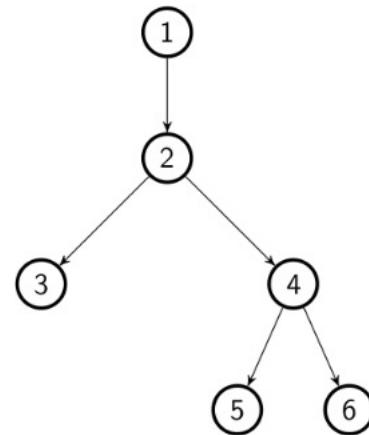


Fig. 17 – arbre1.

Question 5

En utilisant l'interface `bt1`, définir une fonction `nbn-at-d` qui retourne la largeur d'un arbre à la profondeur d .

Appeler `(nbn-at-d arbre1 2)` retourne 2.

Appeler `(nbn-at-d arbre1 3)` retourne 2.

(fonctions utiles : leaf?, has-R-subtree?, L-subtree, R-subtree)

Question 8

En utilisant l'interface `bt1`, définir un prédictat `isst-root?` qui retourne vrai si la racine de l'arbre est supérieure à tous les éléments du sous-arbre gauche et inférieure à tous les éléments du sous-arbre droit.

Appeler `(isst-root? '(2 1 4))` retourne `#t`.

Appeler `(isst-root? '(1 2 3))` retourne `#f`.

Appeler `(isst-root? '(4 (2 1 3) (6 5)))` retourne `#t`.

(fonctions utiles : `leaf?`, `root`, `chk?`, `flatten`, `has-R-subtree?`, `L-subtree`, `R-subtree`)

```
1 (define (chk? v ope L)
2   (if (empty? L) #t
3       (if (ope v (first L)) (chk? v ope (rest L)) #f)
4       )))
5 (chk? 0 < '(1 2 3 4 5))
6 (chk? 6 >= '(1 3 7 5 2))
```

```
#t
#f
```

3. Exercices

Question 10

Pour compléter l'interface `bt6`, définir le prédictat `leaf?`.

Appeler `(leaf? (leaf 1))` retourne `#t`.

Appeler `(leaf? (tree 1 (leaf 2) (leaf 3)))` retourne `#f`.

(fonctions utiles : `empty?`, `L-subtree`, `R-subtree`)

En utilisant la représentation ⑥, il est possible de définir des fonctions de manipulation des arbres binaires de recherche dans une interface (nommée **bt6**) :

```
1 (define (tree val L R) (list val L R))
2 (define (leaf val) (list val empty empty))
3 (define (root T) (first T))
4 (define (L-subtree T) (first(rest T)))
5 (define (R-subtree T) (first(rest(rest T))))
6 (define (has-L-subtree? T) (not (empty? (L-subtree T))))
7 (define (has-R-subtree? T) (not (empty? (R-subtree T))))
8 (define (add T val)
9   (if (empty? T) (leaf val)
10    (let ([r (root T)])
11      (if (< val r)
12        (if (has-L-subtree? T)
13            (tree r (add (L-subtree T) val) (R-subtree T))
14            (tree r (leaf val) (R-subtree T)))
15        (if (has-R-subtree? T)
16            (tree r (L-subtree T) (add (R-subtree T) val))
17            (tree r (L-subtree T) (leaf val)))
18        )))
19 (define (addl T L)
20   (if (empty? L) T
21     (addl (add T (first L)) (rest L))
22   ))
```

3. Exercices

Solution 10

```
1 (define (leaf? T)
2   (and (empty? (L-subtree T))
3        (empty? (R-subtree T)))
4   ))
5 (leaf? (leaf 1))
6 (leaf? (tree 1 (leaf 2) (leaf 3)))
```

```
#t
#f
```

3. Exercices

Question 11 |

Pour compléter l'interface `bt6`, définir le prédictat `has-LR-subtree?` qui retourne vrai si la racine courante possède un sous-arbre gauche et un sous-arbre droit.

Appeler `(has-LR-subtree? (leaf 1))` retourne #f.

Appeler `(has-LR-subtree? (tree 1 (leaf 2) (leaf 3)))` retourne #t.

(fonctions utiles : `has-L-subtree?`, `has-R-subtree?`)

3. Exercices

Solution 11

```
1 (define (has-LR-subtree? T)
2   (and (has-L-subtree? T)
3         (has-R-subtree? T)
4         ))
5 (has-LR-subtree? (leaf 1))
6 (has-LR-subtree? (tree 1 (leaf 2) (leaf 3)))
```

```
#f
```

```
#t
```

3. Exercices

Question 18

En utilisant l'interface `bt6`, écrire un prédictat `isin?` qui vérifie si une valeur `v` est dans un arbre.

Appeler `(isin? (addl empty '(6 3 7 8 5 2)) 2)` retourne `#t`.

Appeler `(isin? (addl empty '(6 3 7 8 5 2)) 1)` retourne `#f`.

(fonctions utiles : `root`, `leaf?`, `has-L-subtree?`, `has-R-subtree?`, `L-subtree`, `R-subtree`)