

Le langage Racket

- Un langage de programmation fonctionnelle
- 27/11/2021

Plan du cours

- 1. Exercices sur les listes
- 2 . Arbres
 - 2.1. Contexte
 - 2.2. Arbres binaires
 - 2.3. Parcours en profondeur
 - 2.4. Parcours en largeur

1. Exercices

Question 7

Définir un prédictat `isflat?` permettant de savoir si une liste est plate.

Appeler `(isflat? '(1 2 3 4 5))` retourne vrai.

Appeler `(isflat? '(1 (2) 3 4 5))` retourne faux.

(fonctions utiles : empty?, list?, first, rest)

1. Exercices

Première solution :

```
1 (define (isflat? L)
2   (if (empty? L) #t
3       (if (list? (first L)) #f
4           (isflat? (rest L)))
5       )))
6 (isflat? '(1 2 3 4 5))
7 (isflat? '(1 (2) 3 4 5))
```

```
#t
```

```
#f
```

1. Exercices

Deuxième solution :

```
1 (define (isflat? L) (not(ormap list? L)))
2 (isflat? '(1 2 3 4 5))
3 (isflat? '(1 (2) 3 4 5))
```

```
#t
#f
```

1. Exercices

Question 9

Définir une fonction **sublist** retournant une demi-liste d'une liste ; la liste est passé en argument ; un deuxième argument définit si la liste retournée est la première demi-liste (celle des valeurs d'indices pairs) ou la deuxième demi-liste (celle des valeurs d'indices impairs).

Appeler (**sublist** '(1 2 3 4 5 6) 0) retourne la liste (1 3 5).

Appeler (**sublist** '(1 2 3 4 5 6) 1) retourne la liste (2 4 6).

Appeler (**sublist** '(1 2 3 4 5) 1) retourne la liste (2 4).

(*fonctions utiles : empty?, cons, first, rest, reverse*)

1. Exercices

```
1 (define (sublist L i)
2   (define (f L R i)
3     (if (empty? L) (reverse R)
4         (if (= i 0)
5             (f (rest L) (cons (first L) R) 1)
6             (f (rest L) R 0)
7             )))
8   (f L '() i))
9 (sublist '(1 2 3 4 5) 0)
10 (sublist '(1 2 3 4 5) 1)
```

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

1. Exercices

Question 10

Définir une fonction **rev-dup** inversant une liste en dupliquant ses éléments ; ne pas utiliser **reverse**.

Appeler (**rev-dup** '(1 2 3 4)) retourne la liste (4 4 3 3 2 2 1 1).

(*fonctions utiles : empty?, append, first, rest, list*)

1. Exercices

Solution 10

```
1 (define (rev-dup L)
2   (if (empty? L) L
3     (let ([i (first L)])
4       (append (rev-dup (rest L)) (list i i)))
5     )))
6 (rev-dup '(1 2 3 4 5))
```

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

1. Exercices

Question 12

Définir une fonction `remove` permettant de supprimer un élément d'une liste.

Appeler `(remove '(1 2 3 4 5) 6)` retourne la liste `(1 2 3 4 5)`.

Appeler `(remove '(1 2 3 4 5) 3)` retourne la liste `(1 2 4 5)`.

(fonctions utiles : empty?, cons, first, rest)

1. Exercices

Solution 12

```
1 (define (remove L e)
2   (if (empty? L) empty
3       (let ([i (first L)])
4         (if (= e i)
5             (remove (rest L) e)
6             (cons i (remove (rest L) e)))
7         )))
8   )
9 (define L (list 1 2 3 4 5))
10 (remove L 6)
11 (remove L 3)
```

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

1. Exercices

Question 15

Définir une fonction `last` retournant le dernier élément d'une liste.

Appeler `(last '(1 2 3 4 5))` retourne 5.

Appeler `(last '(5 4 3 2 1))` retourne 1.

(fonctions utiles : length, rest, first)

1. Exercices

Solution 15

```
1 (define (last L)
2   (if (= (length L) 1) (first L)
3       (last (rest L))
4   )))
5 (last '(1 2 3 4 5))
6 (last '(5 4 3 2 1))
```

```
5
1
```

1. Exercices

Question 16

Définir une fonction `firssts` retournant les éléments d'une liste sans le dernier.

Appeler `(firssts '(1 2 3 4 5))` retourne `'(1 2 3 4)`.

Appeler `(firssts '(5 4 3 2 1))` retourne `'(5 4 3 2)`.

(fonctions utiles : length, rest, first)

1. Exercices

Solution 16

```
1 (define (firsts L)
2   (if (= (length L) 1) empty
3       (cons (first L) (firsts (rest L))))
4   )
5 (firsts '(1 2 3 4 5))
6 (firsts '(5 4 3 2 1))
```

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

1. Exercices

Question 19

Définir une fonction **set-at** permettant de redéfinir une valeur dans une liste ; quand la liste considérée n'a pas de valeur à cet indice, la fonction **set-at** retourne une liste non-modifiée ; réaliser cette fonction sans utiliser **list-set**.

Appeler **(set-at '(1 2 3 4 5) 0 22)** retourne la liste **(22 2 3 4 5)**.

Appeler **(set-at '(1 2 3 4 5) -1 22)** retourne la liste **(1 2 3 4 5)**.

Appeler **(set-at '(1 2 3 4 5) 22 22)** retourne la liste **(1 2 3 4 5)**.

(fonctions utiles : empty?, length, first, rest)

1. Exercices

Solution 19

```
1 (define (set-at L pos val)
2   (define (f L pos val)
3     (if (= pos 0) (cons val (rest L))
4         (cons (first L) (f (rest L) (- pos 1) val)))
5   )
6   (if (or (empty? L)
7           (< pos 0) (>= pos (length L)))
8       L
9       (f L pos val)
10  ))
11 (set-at (list 1 2 3 4) 0 22)
12 (set-at '() 0 22)
13 (set-at (list 1 2 3 4) -1 22)
14 (set-at (list 1 2 3 4) 22 22)
```

```
'(22 2 3 4)
'()
'(1 2 3 4)
'(1 2 3 4)
```

EXOS

Question 1 _____

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))),

arbre3 : '(1 (2 3)),

arbre4 : '(1 (2 (3 4) (4 5))),

arbre5 : '(1 (2 3 (4 5 6)) (7 8)).

Exos

Solution 1

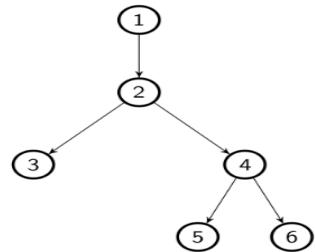


Fig. 17 – arbre1.

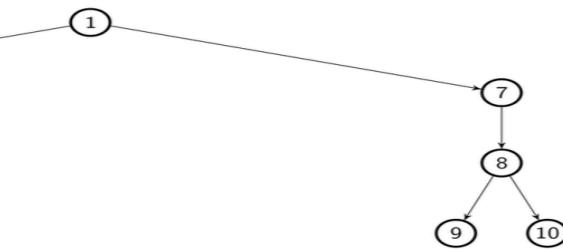
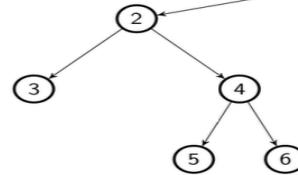


Fig. 18 – arbre2.

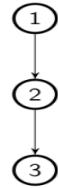


Fig. 19 – arbre3.

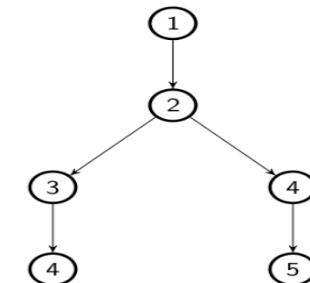


Fig. 20 – arbre4.

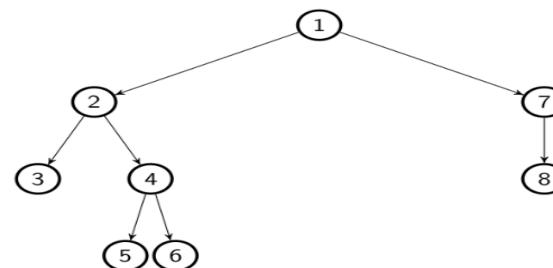


Fig. 21 – arbre5.

EXOS

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)