Programmation fonctionnelle

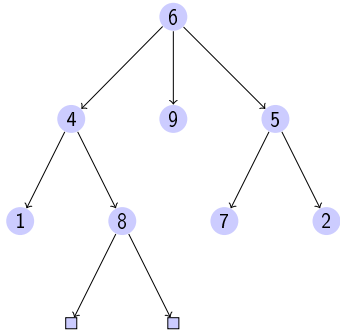
Cours n°5 : Arbres, implémentation et premières fonctions.

Licence 1 Informatique, Université Paris 8

24 Octobre 2022

Arbres

- Un arbre est une structure de données correspondant deux types d'éléments : des sommets, et des arcs reliant deux sommets.



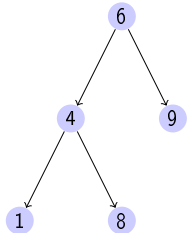
- Les rectangles noirs représentent des éléments vides, pour indiquer qu'un sommet n'a pas de successeur. Ce sont des éléments qui seront encodés par des arbres vides. On ne les représentera pas en pratique.

- Les sommets de cet arbre sont aussi appelés des **nœuds**. Chaque nœud possède des successeurs. On dit que $4, 9$ et 5 sont les successeurs, ou fils, de 6.

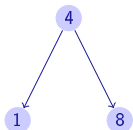
- Réciproquement, le nœud 4 a un **parent**, qui est le nœud 6. Seul le nœud 6 (celui situé tout en haut) n'admet pas de parent.

Arbres binaires

- Un arbre binaire est un arbre dans lequel chaque nœud admet au plus 2 fils. Il peut en admettre qu'un seul, dans ce cas on supposera toujours que c'est le plus à gauche.
- **Exemple :**



  - 6 est le nœud de l'arbre appelé **racine** de A,
  - la partie



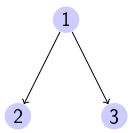
(resp. 9)

est appelée le **fils gauche** (resp. le **fils-droit**) de l'arbre A.

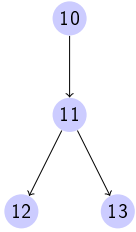
  - Les nœuds 1,8 et 9 sont appelés des **feuilles**, ce sont les nœuds qui n'admettent pas de fils, au bout des branches. Un nœud qui n'est pas une feuille est appelé nœud **non-terminal**.

Implémentation des arbres

- On va représenter les arbres en utilisant une structure de **liste** en Racket. Il y a 6 méthodes principales (et sûrement d'autres) pour implémenter un graphe.
- **(IMP1)** Considérer les nœuds non-terminaux comme des listes de taille variable et les feuilles comme des éléments.



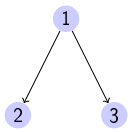
$\rightsquigarrow$ `'(1 2 3)`,



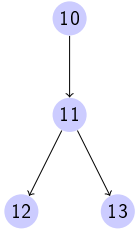
$\rightsquigarrow$ `'(10 (11 12 13))`.

Implémentation des arbres

- On va représenter les arbres en utilisant une structure de **liste** en Racket. Il y a 6 méthodes principales (et sûrement d'autres) pour implémenter un graphe.
- **(IMP1)** Considérer les nœuds non-terminaux comme des listes de taille variable et les feuilles comme des éléments.

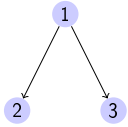


$\rightsquigarrow$ `'(1 2 3)`,

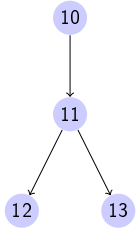


$\rightsquigarrow$ `'(10 (11 12 13))`.

- **(IMP2)** Considérer les nœuds non-terminaux comme des listes de taille variable et les feuilles comme listes à un élément.



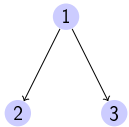
$\rightsquigarrow$ `'(1 (2) (3))`,



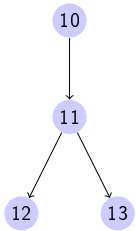
$\rightsquigarrow$ `'(10 (11 (12) (13)))`.

Implémentation des arbres 2

- **(IMP3)** Considérer les nœuds non-terminaux comme des listes de taille variable et les feuilles comme des listes de taille 3, avec un élément et deux listes vides.



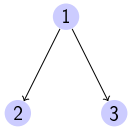
$\rightsquigarrow$ `'(1 (2 () ()) (3 () ()))`,



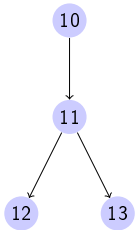
$\rightsquigarrow$ `'(10 (11 (12 () ()) (13 () ())))`.

Implémentation des arbres 2

- **(IMP3)** Considérer les nœuds non-terminaux comme des listes de taille variable et les feuilles comme des listes de taille 3, avec un élément et deux listes vides.

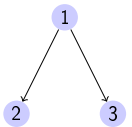


$\rightsquigarrow$ `'(1 (2 () ()) (3 () ()))`,

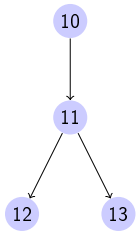


$\rightsquigarrow$ `'(10 (11 (12 () ()) (13 () ())))`.

- **(IMP4)** 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 éléments.



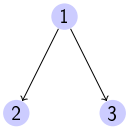
$\rightsquigarrow$ `'(1 2 3)`,



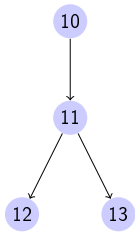
$\rightsquigarrow$ `'(10 (11 12 13) ())`.

Implémentation des arbres 3

- **(IMP5)** Considérer les nœuds non-terminaux comme des listes de taille 3 (quitte à compléter), et les feuilles comme des listes de taille 1.



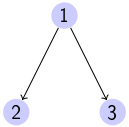
$\rightsquigarrow$ `'(1 (2) (3))`,



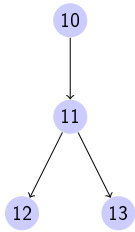
$\rightsquigarrow$ `'(10 (11 (12) (13)) ())`.

Implémentation des arbres 3

- **(IMP5)** Considérer les nœuds non-terminaux comme des listes de taille 3 (quitte à compléter), et les feuilles comme des listes de taille 1.

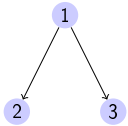


$\rightsquigarrow$ `'(1 (2) (3))`,

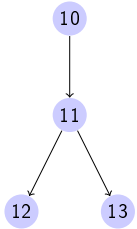


$\rightsquigarrow$ `'(10 (11 (12) (13)) ())`.

- **(IMP6)** Considérer les nœuds non-terminaux comme des listes de taille 3 (quitte à compléter), et les feuilles comme des listes de taille 3.



$\rightsquigarrow$ `'(1 (2 () ()) (3 () ()))`,



$\rightsquigarrow$ `'(10 (11 (12 () ()) (13 () ())) ())`.

Choix de l'implémentation et fonctions primitives

- Chaque représentation a ses avantages et inconvénients... l'implémentation **(IMP6)** utilise des définitions structurellement plus lourdes, mais plus homogènes et faciles à utiliser.
- Une fois choisie notre structure, nous allons devoir écrire les fonctions essentielles à la manipulation des arbres.

Choix de l'implémentation et fonctions primitives

- Chaque représentation a ses avantages et inconvénients... l'implémentation **(IMP6)** utilise des définitions structurellement plus lourdes, mais plus homogènes et faciles à utiliser.
- Une fois choisie notre structure, nous allons devoir écrire les fonctions essentielles à la manipulation des arbres.
- **Primitives de test** :
  - **arbre?** qui retourne vrai si un élément est un arbre;
  - **arbre-vide?** qui retourne vrai si l'arbre en paramètre est vide;
  - **feuille?** qui yeste si un arbre est une feuille;
  - **arbre=?** qui permet de tester l'égalité de deux arbres;

Choix de l'implémentation et fonctions primitives

- Chaque représentation a ses avantages et inconvénients... l'implémentation **(IMP6)** utilise des définitions structurellement plus lourdes, mais plus homogènes et faciles à utiliser.
- Une fois choisie notre structure, nous allons devoir écrire les fonctions essentielles à la manipulation des arbres.
- **Primitives de test :**
  - **arbre?** qui retourne vrai si un élément est un arbre;
  - **arbre-vide?** qui retourne vrai si l'arbre en paramètre est vide;
  - **feuille?** qui yeste si un arbre est une feuille;
  - **arbre=?** qui permet de tester l'égalité de deux arbres;
- **Primitives d'accès :**
  - **racine** qui prend en paramètre un arbre *a* et renvoie la valeur de son nœud racine;
  - **fils-g** qui retourne le fils gauche d'un arbre **non vide**;
  - **fils-d** qui retourne le fils droit d'un arbre **non vide**;

Choix de l'implémentation et fonctions primitives

- Chaque représentation a ses avantages et inconvénients... l'implémentation **(IMP6)** utilise des définitions structurellement plus lourdes, mais plus homogènes et faciles à utiliser.
- Une fois choisie notre structure, nous allons devoir écrire les fonctions essentielles à la manipulation des arbres.
- **Primitives de test** :
  - **arbre?** qui retourne vrai si un élément est un arbre;
  - **arbre-vide?** qui retourne vrai si l'arbre en paramètre est vide;
  - **feuille?** qui yeste si un arbre est une feuille;
  - **arbre=?** qui permet de tester l'égalité de deux arbres;
- **Primitives d'accès** :
  - **racine** qui prend en paramètre un arbre *a* et renvoie la valeur de son nœud racine;
  - **fils-g** qui retourne le fils gauche d'un arbre **non vide**;
  - **fils-d** qui retourne le fils droit d'un arbre **non vide**;
- **Primitives de construction** :
  - **arbre-vide** qui crée et retourne un arbre vide;
  - **attache-arbre** qui crée et retourne un arbre avec une valeur donnée, et deux arbres qui seront ses fils.

Fonctions primitives

- **Primitives :**

```
(define (arbre? a)
  (or (null? a)
      (and (= 3 (length a))
           (not (list? (car a)))
           (list? (cadr a))
           (arbre? (cadr a))
           (list? (caddr a))
           (arbre? (caddr a)))))
```

```
(define (arbre-vide? a)
  (null? a))
```

```
(define (feuille? a)
  (and (not (arbre-vide? a))
       (arbre-vide? (fils-g a))
       (arbre-vide? (fils-d a))))
```

```
(define (arbre=? a1 a2)
  (equal? a1 a2))
```

```
(define (racine a)
  (car a))
```

```
(define (fils-g a)
  (cadr a))
```

```
(define (fils-d a)
  (caddr a))
```

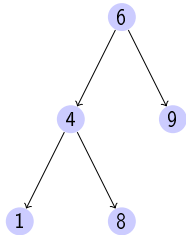
```
(define (arbre-vide )
  '())
```

```
(define (attache-arbre r a1 a2)
  (list r a1 a2))
```

- Pourquoi utiliser des primitives plutôt que les fonctions de listes ? C'est plus simple si on décide de changer l'implémentation ! On aura juste à changer les primitives et pas toutes nos fonctions sur les arbres.

Exemples

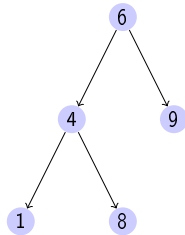
- Pour définir l'arbre




```
(define a '(6 (4 (1 () ()) (8 () ())) (9 () ())))
```

Exemples

- Pour définir l'arbre

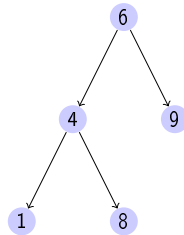



```
(define a '(6 (4 (1 () ()) (8 () ())) (9 () ())))
```

```
(arbre-vide? a)
(feuille? a)
(fils-g a)
(fils-d a)
(arbre-vide? (fils-d a))
(feuille? (fils-d a))
(arbre-vide? (fils-d (fils-d a)))
(attache-arbre 2 (fils-d a) (fils-d a))
(attache-arbre 2 (fils-g a) (arbre-vide))
```

Exemples

- Pour définir l'arbre




```
(define a '(6 (4 (1 () ()) (8 () ())) (9 () ())))
```

```
(arbre-vide? a)                          #f
(feuille? a)                             #f
(fils-g a)                               '(4 (1 () ()) (8 () ()))
(fils-d a)                               '(9 () ())
(arbre-vide? (fils-d a))                 #f
(feuille? (fils-d a))                    #t
(arbre-vide? (fils-d (fils-d a)))        #t
(attache-arbre 2 (fils-d a) (fils-d a))  '(2 (9 () ()) (9 () ()))
(attache-arbre 2 (fils-g a) (arbre-vide))  '(2 (4 (1 () ()) (8 () ())) ())
```

Attention à l'implémentation

- Notons que si on choisit une autre implémentation, par exemple **(IMP1)**, nous aurons à écrire d'autres fonctions primitives.

- Cette implémentation n'étant pas homogène, tester si un symbole *a* est un arbre est plus compliqué, en revanche on peut écrire les fonctions suivantes:

```
(define (arbre-vide? a)
  (null? a))

(define (feuille? f)
  (not (list? f)))

(define (racine a)
  (car a))

(define (fils-g a)
  (cadr a))

(define (fils-d a)
  (caddr a))
```

```
(define (arbre=? a1 a2)
  (equal? a1 a2))

(define (arbre-vide )
  '())

(define (feuille f)
  f)

(define arbre list)
```

et pour la définition d'un arbre:

```
(define a (arbre 1 (feuille 2) (feuille 3)))
```

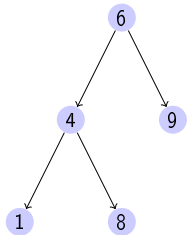
Fonctions sur des arbres

- La structure d'arbre, de par sa définition, est propice à l'utilisation de la récursivité. Pour écrire une fonction sur un arbre, on pourra ainsi l'appeler récursivement sur **son fils gauche et son fils droit**, et reconstituer le résultat final en fonction de la racine.

Fonctions sur des arbres

- La structure d'arbre, de par sa définition, est propice à l'utilisation de la récursivité. Pour écrire une fonction sur un arbre, on pourra ainsi l'appeler récursivement sur **son fils gauche et son fils droit**, et reconstituer le résultat final en fonction de la racine.

- **Un exemple très classique** : on veut écrire une fonction qui compte le nombre de nœuds d'un arbre donné.

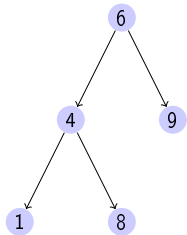


$$\text{Nb\_nœud}(a) = 1 + \text{Nb\_nœud}\left(\begin{array}{c} 4 \\ 1 \quad 8 \end{array}\right) + \text{Nb\_nœud}\left(9\right)$$

Fonctions sur des arbres

- La structure d'arbre, de par sa définition, est propice à l'utilisation de la récursivité. Pour écrire une fonction sur un arbre, on pourra ainsi l'appeler récursivement sur **son fils gauche et son fils droit**, et reconstituer le résultat final en fonction de la racine.

- **Un exemple très classique** : on veut écrire une fonction qui compte le nombre de nœuds d'un arbre donné.



$$\text{Nb\_nœud}(a) = 1 + \text{Nb\_nœud}\left(\begin{array}{c} 4 \\ 1 \quad 8 \end{array}\right) + \text{Nb\_nœud}\left(9\right)$$

- **Cas d'arrêt ?** Lorsque l'arbre est vide; il a 0 nœud, ou lorsque l'arbre est une feuille: il a 1 nœud.

Fonctions sur des arbres 2

- Nombre de nœuds :

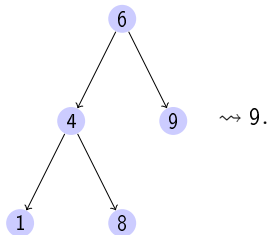
```
(define (nb-noeuds a)
  (if (arbre-vide? a)
      0
      (if (feuille? a)
          1
          (+ 1 (nb-noeuds (fils-g a)) (nb-noeuds (fils-d a))))))
```

Fonctions sur des arbres 2

- **Nombre de nœuds :**

```
(define (nb-noeuds a)
  (if (arbre-vide? a)
      0
      (if (feuille? a)
          1
          (+ 1 (nb-noeuds (fils-g a)) (nb-noeuds (fils-d a))))))
```

- **Exercice :** Définir une fonction `max-arbre` qui prend en paramètres un arbre *a* renvoie son maximum.



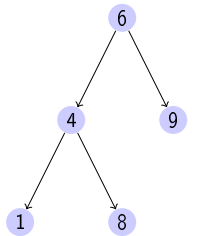
$\rightsquigarrow 9.$

Fonctions sur des arbres 2

- **Nombre de nœuds :**

```
(define (nb-noeuds a)
  (if (arbre-vide? a)
      0
      (if (feuille? a)
          1
          (+ 1 (nb-noeuds (fils-g a)) (nb-noeuds (fils-d a))))))
```

- **Exercice :** Définir une fonction `max-arbre` qui prend en paramètres un arbre *a* renvoie son maximum.



$\rightsquigarrow 9.$

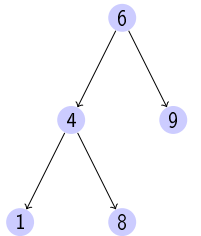
```
(define (max-arbre a)
  (if (feuille? a)
      (racine a)
      (let ((m (max (max-arbre (fils-g a)) (max-arbre (fils-d a)))))
        (if (> (racine a) m)
            (racine a)
            m))))
```

Fonctions sur des arbres 2

- **Nombre de nœuds :**

```
(define (nb-noeuds a)
  (if (arbre-vide? a)
      0
      (if (feuille? a)
          1
          (+ 1 (nb-noeuds (fils-g a)) (nb-noeuds (fils-d a))))))
```

- **Exercice :** Définir une fonction `max-arbre` qui prend en paramètres un arbre *a* renvoie son maximum.



$\rightsquigarrow 9.$

```
(define (max-arbre a)
  (if (feuille? a)
      (racine a)
      (let ((m (max (max-arbre (fils-g a)) (max-arbre (fils-d a)))))
        (if (> (racine a) m)
            (racine a)
            m))))
```

Ne fonctionne pas !!

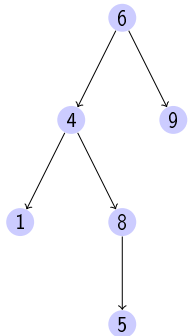
Attention aux cas d'arrêts...

- Dans la plupart des fonctions récursives sur des arbres, les deux cas d'arrêt (arbre vide et feuille) sont nécessaires.

Attention aux cas d'arrêts...

- **Dans la plupart des fonctions récursives sur des arbres, les deux cas d'arrêt (arbre vide et feuille) sont nécessaires.**
- **Exemple** : Si

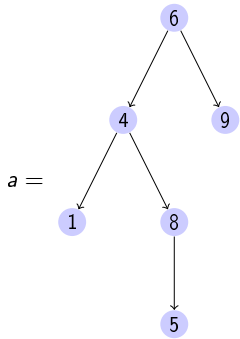
$a =$



alors en appelant récursivement sur le fils-gauche de a, puis de nouveau sur son fils-droit, on devra déterminer le nombre de feuilles de

Attention aux cas d'arrêts...

- **Dans la plupart des fonctions récursives sur des arbres, les deux cas d'arrêt (arbre vide et feuille) sont nécessaires.**
- **Exemple** : Si



alors en appelant récursivement sur le fils-gauche de a, puis de nouveau sur son fils-droit, on devra déterminer le nombre de feuilles de

- Cet arbre n'étant ni vide, ni une feuille, on va calculer son nombre de feuilles par des appels récursifs sur ses deux fils; or l'un est vide, et l'autre est une feuille: on a besoin de savoir répondre dans les deux cas !

... et aux appels récursifs !

- De plus, lorsque vous écrivez une fonction où seul l'un des deux cas d'arrêt est possible, par exemple `max-arbre`, **attention aux appels récursifs** !

- Puisque dans ce cas on ne prendra pas de cas d'arrêt où l'arbre est vide, **il faut s'assurer avant chaque appel à tester que le fils qu'on manipule n'est pas vide** !

- **Voici une version valable :**

```
(define (max-arbre a)
  (cond ((feuille? a) (racine a))
        ((arbre-vide? (fils-g a)) (max (racine a) (max-arbre (fils-d a))))
        ((arbre-vide? (fils-d a)) (max (racine a) (max-arbre (fils-g a))))
        (else (max (racine a) (max-arbre (fils-g a)) (max-arbre (fils-d a))))))
```

- **Exercice :**
  - Définir une fonction `nb-feuilles` qui compte le nombre de feuilles d'un arbre a en paramètres.
  - Définir une fonction `appartient` qui prend en paramètres un arbre a et un symbole x, et renvoyant un booléen indiquant si x appartient à a.

Solution

- **Solution des exercices précédents :**
  - Définir une fonction `nb-feuilles` qui compte le nombre de feuilles d'un arbre *a* en paramètres.

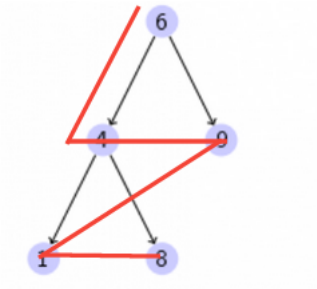
```
(define (nb-feuilles a)
  (if (arbre-vide? a)
      0
      (if (feuille? a)
          1
          (+ (nb-feuilles (fils-g a))
             (nb-feuilles (fils-d a))))))
```

  - Définir une fonction `appartient` qui prend en paramètres un arbre *a* et un symbole *x*, et renvoyant un booléen indiquant si *x* appartient à *a*.

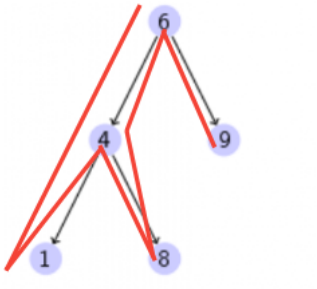
```
(define (appartient x a)
  (if (arbre-vide? a)
      #f
      (or (equal? (racine a) x) (appartient x (fils-g a))
                                (appartient x (fils-d a)))))
```

Parcours d'arbre

- Pour pouvoir accéder aux données stockées dans un arbre, il faut **parcourir** cet arbre. Il existe principalement deux types de parcours:
  - Parcours **en largeur**,
  - Parcours **en profondeur**,



Parcours en largeur



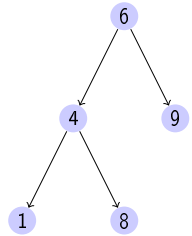
Parcours en profondeur

- **En profondeur** : on passe les nœuds en revue en commençant toujours par le même fils, puis en descendant le plus profondément possible dans l'arbre. Lorsqu'on arrive sur un arbre vide, on remonte jusqu'au noeud supérieur et on redescend dans le fils encore inexploré.
- **En largeur** : on passe en revue étage par étage: d'abord la racine, puis les racines des fils gauche et droit, puis les racines de leurs fils, etc.

Parcours en profondeur

- Il existe 3 types de parcours en profondeur, dépendant de l'ordre d'affichage des éléments:
  - Parcours **infixe** : Fils gauche, puis racine, puis fils droit.
  - Parcours **préfixe** : Racine, puis fils gauche, puis fils droit.
  - Parcours **postfixe** : Fils gauche, puis fils droit, puis racine.
- **Exemple** : Pour

$a =$



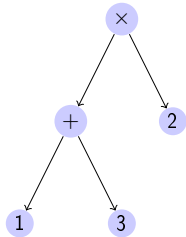
`(infixe a)` renvoie `'(1 4 8 6 9)`.

`(prefixe a)` renvoie `'(6 4 1 8 9)`.

`(postfixe a)` renvoie `'(1 8 4 9 6)`.

Exemple : opérations

- Un calcul arithmétique utilisant les opérations $+, -, \times$ et $\div$ peut être représenté par un arbre, par exemple:



- Parcours **infixe** : $((1 + 3) \times 2)$. C'est la notation mathématique usuelle.

- Parcours **préfixe** : $(\times (+ 1 3) 2)$. C'est la notation de Racket, d'où le nom de **notation préfixée**.

Parcours en profondeur infixe

- **Rappel :** Pour le parcours infixe, d'abord le fils gauche, puis la racine, puis le fils droit.

```
(define (infixe a)
        (if (arbre-vide? a)
            '()
            (append (infixe (fils-g a))
                    (list (racine a))
                    (infixe (fils-d a)))))
```