INF 421 — 05

Luc Maranget

Arbres

Luc.Maranget@inria.fr

http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/421/

Diversion

On sait…

- Les classes sont les « patrons » (cf. couture) des objets.
- Chaque objet possède ses propres composants dynamiques.
- Tous les objets partagent les même composants statiques.

Slogan!

Ce qui est **static** est à la classe, ce qui est dynamique est à l'objet.

Application du slogan

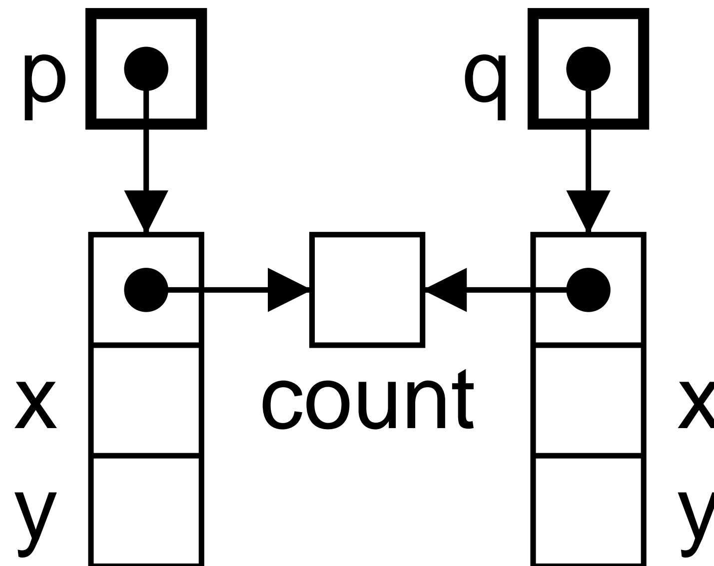
```
class Pair {
  static int count = 0 ;
  int x, y ;

  Pair (int x, int y) {
    this.x = x ; this.y = y ; count++ ;
  }

  public static void main(String [] arg) {
    Pair p = new Pair (1, 2), q = new Pair (2, 3) ;
    System.out.println(count) ;
  }
}
```

- Quelle est la notation « complète » pour `count` ? `Pair.count`.
- Qu'affiche le programme : `2`.

Implémentation (simplification)

Deux objets de la même classe `Pair`.

- Ont chacun leur propres champs `x` et `y`.
- Pointent vers le vecteur des variables de la classe `Pair`.



Cas des méthodes

Le slogan s'applique encore.

- Méthode **static**. Il n'y a même pas besoin de parler d'objet. La méthode existe, c'est tout. Exemple, `Integer.parseInt`.
- Méthode dynamique. Elle est propre à chaque objet. Exemple `stack.push(1)`.
- Dans le code d'une méthode dynamique, un objet courant existe, il s'appelle : **this**.
- Dans le code d'une méthode statique on a pas accès aux champs/méthodes dynamiques, car il n'y a *pas d'objet*.

Si c'est static, ya pas d'objet

Une petite inattention.

```
class Pair {
  private int count ;
  int x, y ;
  Pair (int x, int y) {
    this.x = x ; this.y = y ; count++ ;
  }

  static int lireCount() {
    return count ;
  }
}
```

Est-ce que ça compile ? Non ! Pourquoi ?.

La méthode statique `lireCount` fait référence à la variable non-statique count.

```
# javac Pair.java
Pair.java:10: non-static variable count
              cannot be referenced from a static context
    return count ;
           ^
```

Quelques exemples de méthodes dynamiques

- Codage objet des piles :

```
Stack s1 = new Stack (), s2 = new Stack () ;
s1.push(1) ; s2.push(2) ;
```

- Sortie standard/d'erreur, `System.out`/`System.err`, deux objets différents (rangés dans deux variables de classes. . .).

```
System.out.println("Je vais dans out") ;
System.err.println("Je vais dans err") ;
```

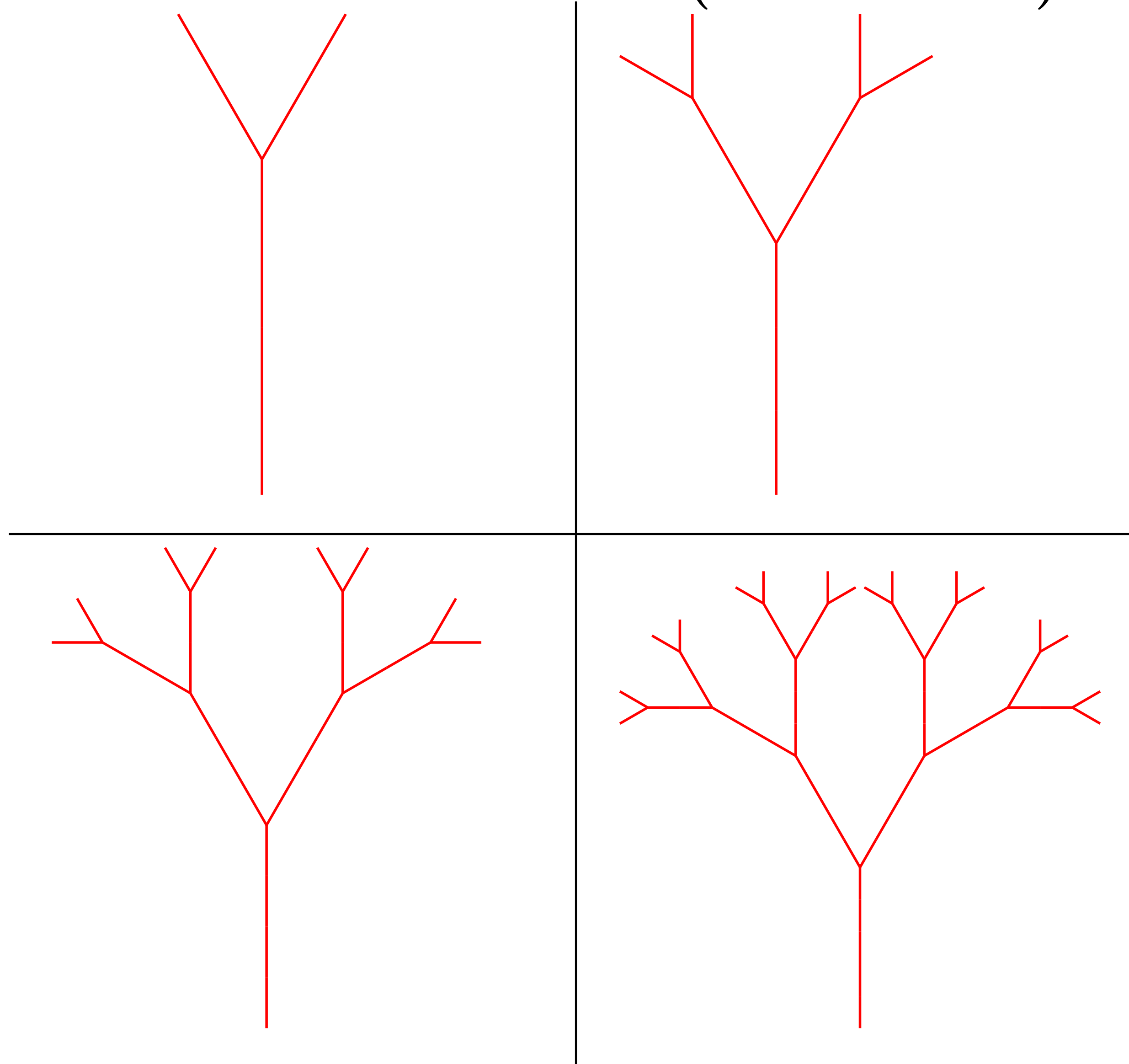
Et si on fait :

```
# java A > bonga
```

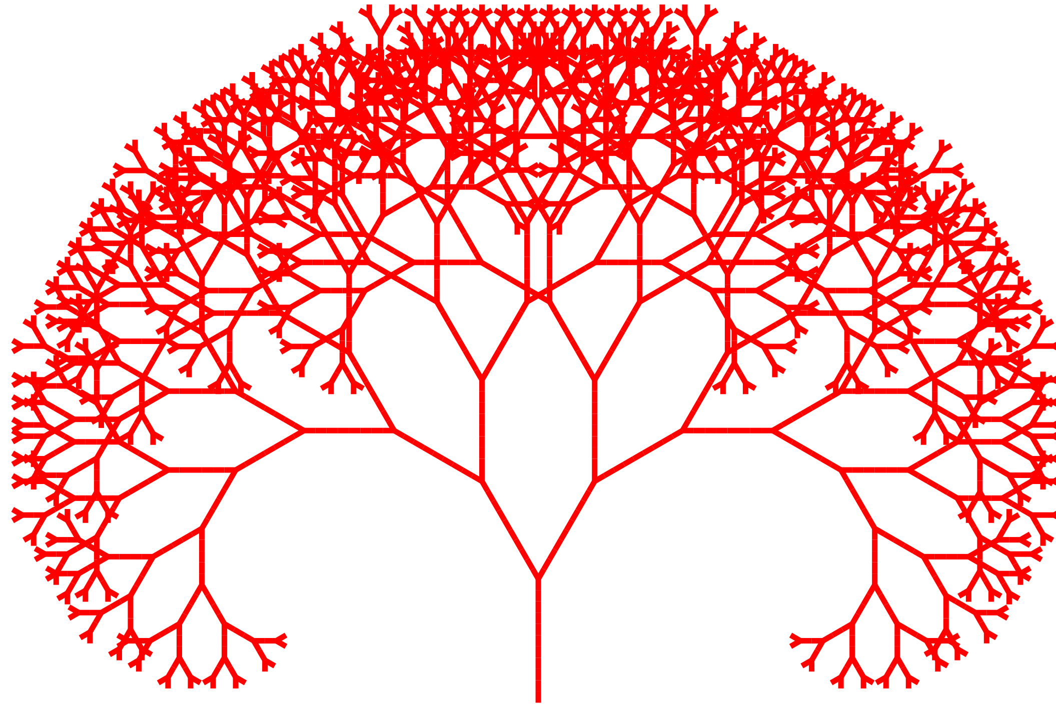
- Ça range dans le fichier `bonga` : « `Je vais dans out` ».
- Ça affiche : « `Je vais dans err` ».

Un arbre dans la nature

Structure récursive (« fractale »)

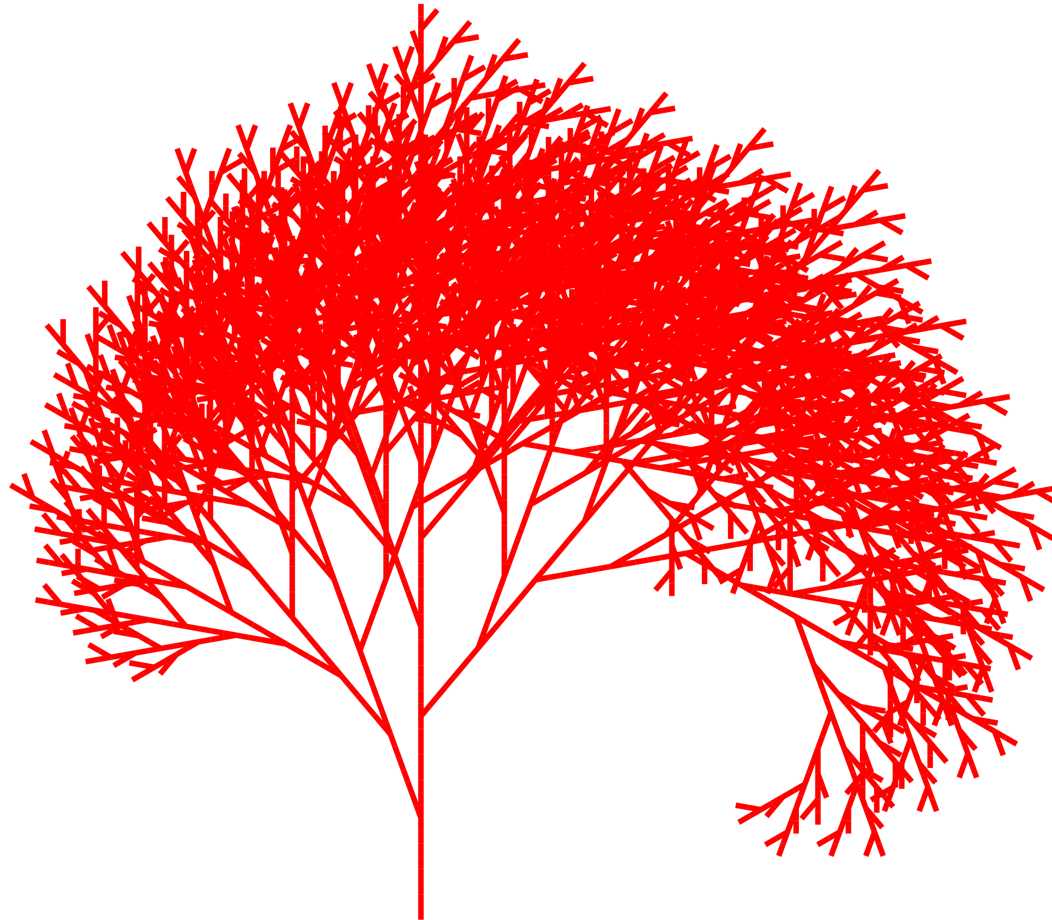


Au final. . .



T → D[-T]+T

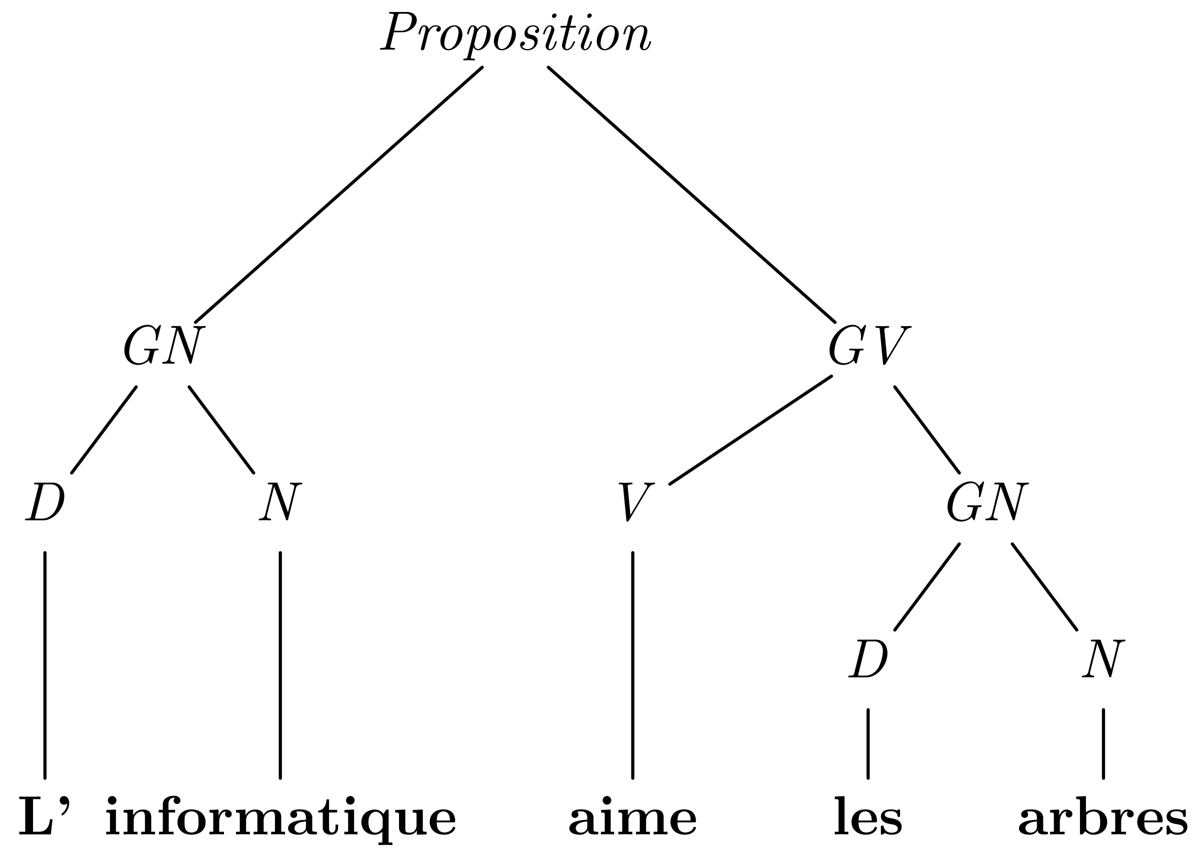
Un autre arbre, plus naturel



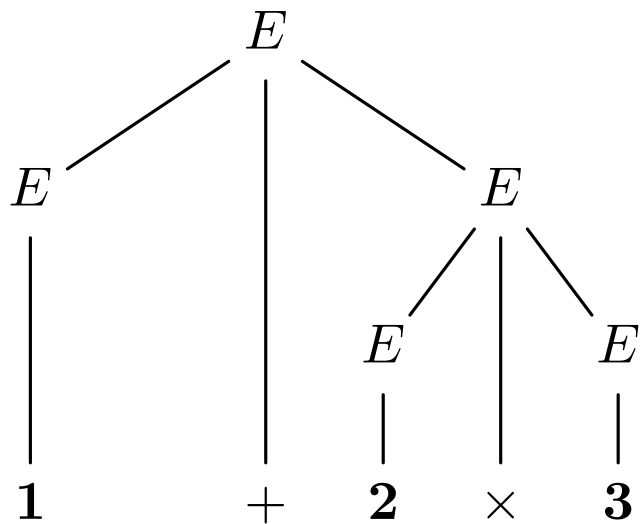
T → D[-T]D[++T]T

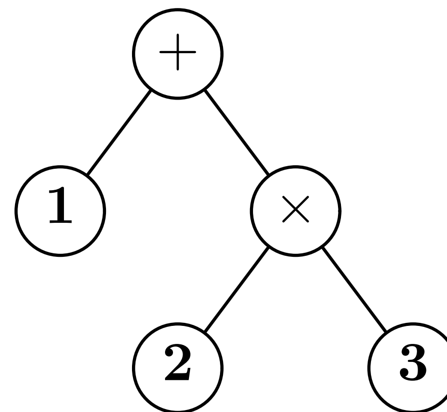
Les arbres « hiéarchie »

Les arbres structurants



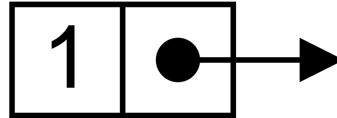
Ou plus informatiquement...



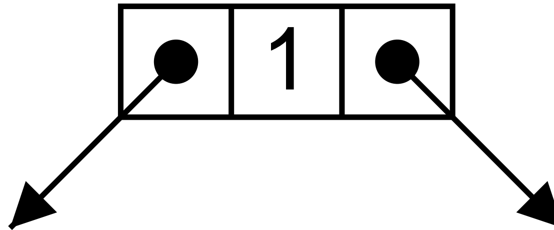


Bon et en Java ?

- La liste :



- L'arbre (binaire) :



Définition des cellules d'arbre (binaire)

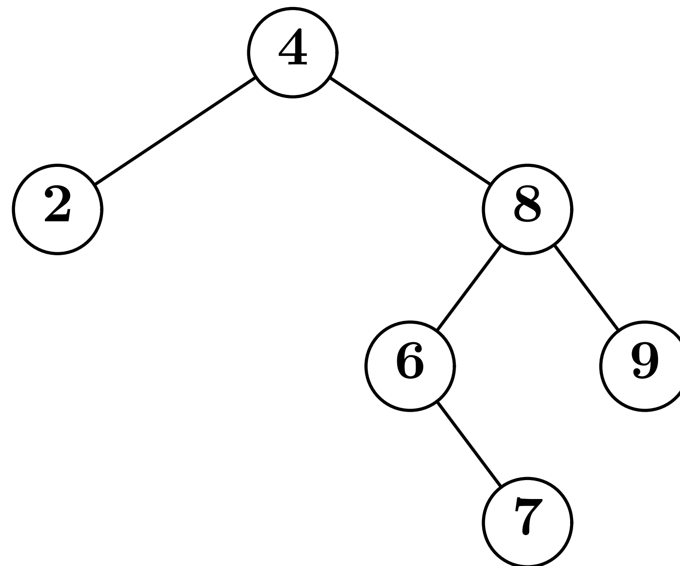
```
class Tree {
  int val ;
  Tree left, right ;

  Tree (Tree left, int val, Tree right) {
    this.left = left ;
    this.val = val ;
    this.right = right ;
  }
}
```

Remarquer

- Nos arbres ont au plus deux fils.
- On retrouve notre dangereux ami, l'arbre vide : **null**.

Fabriquons un arbre

```
new Tree (
   new Tree (null, 2, null),
   4,
   new Tree (
     new Tree (null, 6, new Tree (null, 7, null)),
     8,
     new Tree (null, 9, null)))
```



Surcharge du constructeur

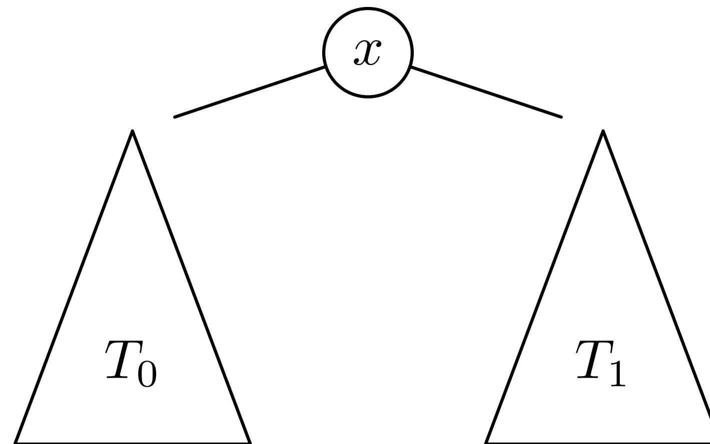
Permet de faire un peu plus concis (mais ça n'ira pas loin).

```
Tree (int val) {
  this.val = val ;
}

new Tree (
   new Tree (2),
   4,
   new Tree  (
     new Tree (null, 6, new Tree (7)),
     8,
     new Tree (9)))
```

Arbre et récursivité

Les arbres sont des structures très récursives :

- L'arbre vide est un arbre.
- Si T_0 et T_1 sont des arbres, alors (T_0, x, T_1) est un arbre.

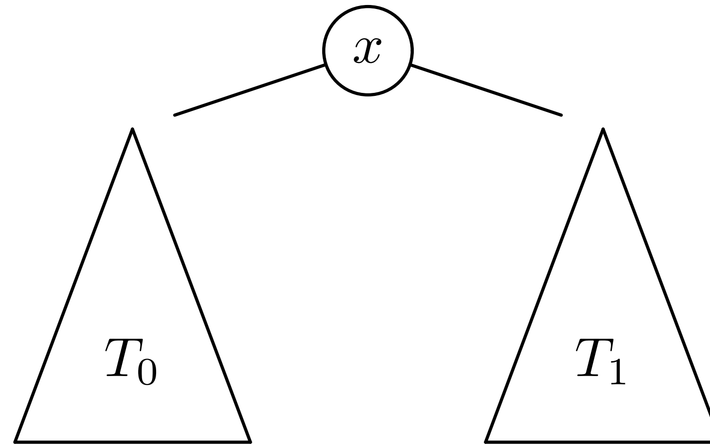


On peut voir les arbres comme solutions de cette équation.

$$T = \mathbf{null} \uplus (T \times \mathbf{int} \times T)$$

La programmation est systématiquement récursive.

Vocabulaire

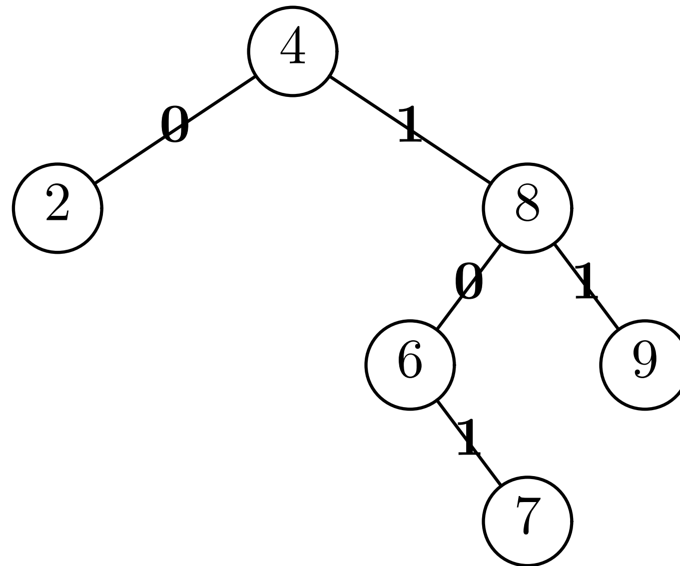


- Le « sommet » (ou « nœud ») marqué x est la *racine*.
- Les arbres T_0 et T_1 sont les *sous-arbres*.
- Les racines de T_0 et T_1 sont les *fils* (*enfants*) de la racine qui est leur *père* (*parent*).
- Si T_0 et T_1 sont l'arbre vide, alors le sommet x est une *feuille*.

En java, arbre $\sim$ référence, sommet $\sim$ cellule.

Se repérer dans un arbre

Dans une liste on parlait du k-ième élément, dans un arbre on parle de chemin.



Par exemple le chemin vers la feuille étiquetée 7 est : **1.0.1**.

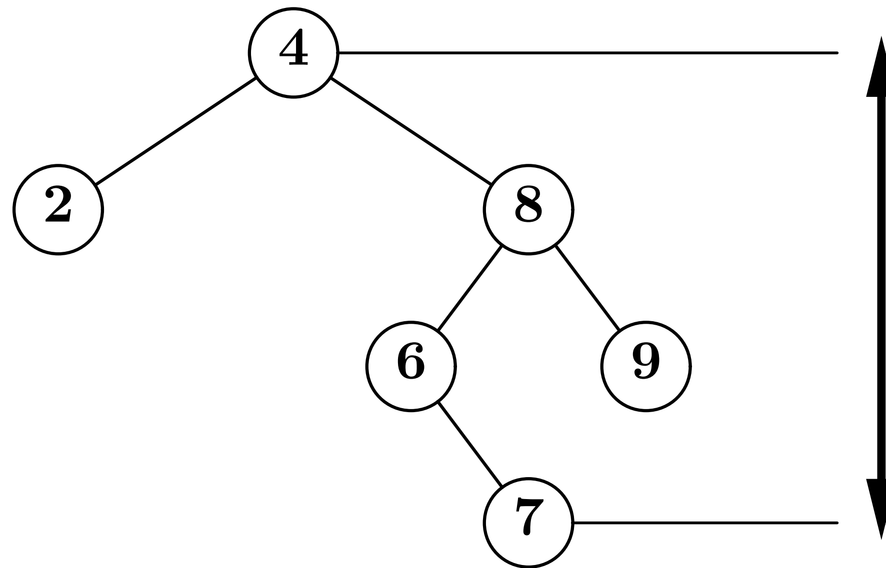
Le chemin vers la racine est : «» (rien ou Λ).

Il est également pratique de repérer les sous-arbres (ex. **0.0**).

Bref, les chemins sont tout bêtement des listes.

Hauteur

La profondeur d'un sous-arbre est la longueur du chemin qui y mène,



- La hauteur d'un arbre est la profondeur maximale de ses sous-arbres
- C'est aussi la profondeur maximale des feuilles, plus un (arbres vides).

Calcul de la hauteur

Il n'y pas le choix : approche... récursive.

$$
\begin{array}{rcl}
H(\emptyset) & = & 0 \\
H(T_0, _, T_1) & = & 1 + \max(H(T_0), H(T_1))
\end{array}
$$

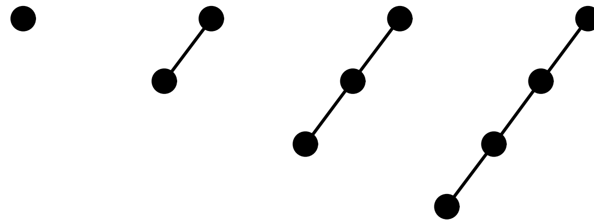
```
static int getHeight(Tree t) {
  if (t == null) {
    return 0 ;
  } else {
    return 1+Math.max(getHeight(t.left), getHeight(t.right)) ;
  }
}
```

Borner la hauteur

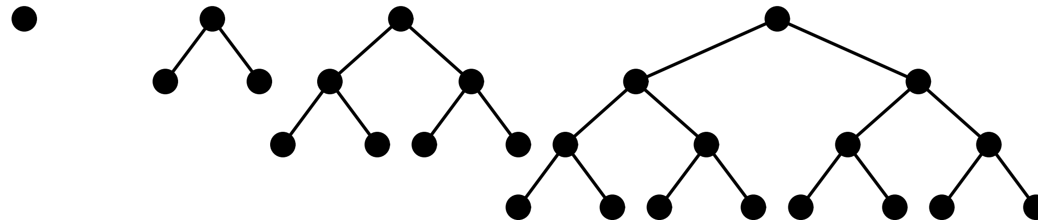
Soit un arbre de hauteur h, contenant n sommets.

$$h \leq n \leq 2^h - 1$$

- Arbres avec n minimal :



- Arbres avec n maximal :



Soit encore.

$$\log_2(n+1) \leq h \leq n$$

Trouver un sous-arbre connaissant son chemin

La définition inductive est assez claire :

$$T/\Lambda = T, \quad (T_0, _, T_1)/\mathbf{0}.C = T_0/C, \quad (T_0, _, T_1)/\mathbf{1}.C = T_1/C,$$

Un `Chemin` est une liste d'entiers.

```
static Tree subTree(Tree t, Chemin c) {
  if (c == null) {
    return t ;
  } else if (t == null) {
    throw new Error () ;
  } else if (c.val == 0) {
    return subTree(t.left, c.next) ;
  } else if (c.val == 1) {
    return subTree(t.right, c.next) ;
  } else {
    throw new Error () ;
  }
}
```

Trouver son chemin II

Du point de vue du chemin, on effectue un parcours de liste, d'où le code alternatif suivant.

```
static Tree subTree(Tree t, Chemin c) {
  for ( ; c != null ; c = c.next) {
    // Traiter d'abord l'erreur
    if (t == null) { throw new Error () ; }
    // Ici t n'est pas null
    if (c.val == 0) { t = t.left ; }
    else if (c.val == 1) { t = t.right ; }
    else { throw new Error () ; }
  }
  return t ;
}
```

De fait, la récursion était terminale (cf. amphi 02).

Arbres strictement binaires

Les sommets ont zéro ou deux fils. Définition sans l'arbre vide :

- Une feuille est un arbre.
- Si T_0 et T_1 sont des arbres, alors (T_0, x, T_1) est un arbre.

Attention : en programmation **null** demeure... (pour repérer les feuilles).

```
class Tree {
  // Comme avant.

  Tree (int x) { this.val = x; } // Constructeur des feuilles.

  boolean isLeaf() {
    return this.left == null && this.right == null ;
  }
}
```

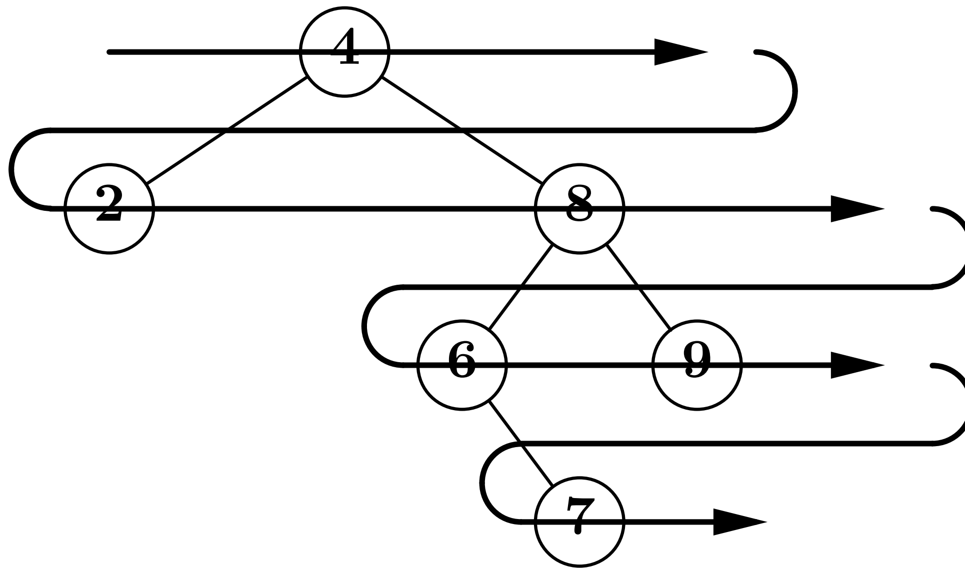
Une amusante propriété des arbres binaires (stricts)

Un arbre binaire strict possède $n+1$ feuilles et n sommets internes.

- $n = 0$, cas de la feuille (cas de base).
- $n > 0$, soit $T = (T_1, T_2)$ ($n_1 + 1$ et $n_2 + 1$ feuilles chacun),
  - Nombre de feuilles : $(n_1 + 1) + (n_2 + 1)$.
  - Nombre de sommets internes : $n_1 + n_2 + 1$, (ceux de T_1 plus ceux de T_2, plus la racine.

Parcourir un arbre

En largeur d'abord.



Si on parcourt pour afficher, on affichera donc :

[4], [2, 8], [6, 9], [7].

C'est conceptuellement assez simple (parcours « par étages » de profondeur croissante), mais un peu délicat à programmer.

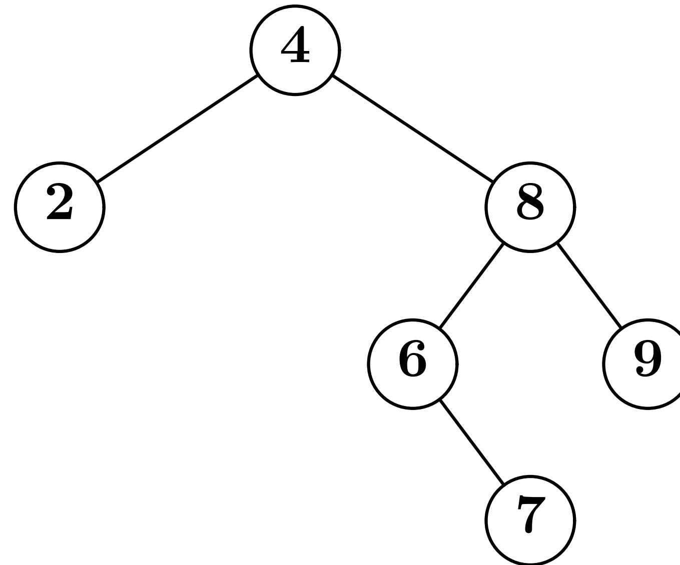
Parcours en profondeur d'abord

Conceptuellement simples, si on veut bien adopter le point de vue récursif :

- Parcourir sous-arbres par sous-arbre.
- Mais encore :
  - Préfixe : $x \to T_0 \to T_1$.
  - Infixe : $T_0 \to x \to T_1$.
  - Postfixe : $T_0 \to T_1 \to x$.

Il n'y a rien d'autre à comprendre (savoir)!

Exemple, en profondeur (postfixe)



- T_0, T_1, **4**.
- **2**, T_1, **4**.
- **2**, T_0, T_1, **8**, **4**.
- **2**, T_0, **9**, **8**, **4**.
- **2**, **7**, **6**, **9**, **8**, **4**.

Le plus simple est encore de programmer

- Si T est vide, ... ne rien faire !
- Si T est (T_0, x, T_1), ... afficher T_0, T_1 et enfin x.

```
static void postfix(Tree t) {
  if (t != null) {
    postfix(t.left) ;
    postfix(t.right) ;
    System.out.println(t.val) ;
  }
}
```

Note : généralisation de l'affichage des listes.

```
static void print(List p) {
  if (p != null) {
    print(p.next) ; System.out.prinln(t.val) ;
  }
}
```

Un problème un peu différent

Calculer la liste des étiquettes (dans l'ordre postfixe).

Solution « théorique »

$$
\begin{aligned}
P(\emptyset) &= \emptyset \\
P(T_0, x, T_1) &= P(T_0); P(T_1); x
\end{aligned}
$$

Bon nous voilà bien avancés...

```
static IntList postfix(Tree t) {
  if (t == null) {
     return null ;
  } else {
    return
      IntList.append(postfix(t.left),
        IntList.append(postfix(t.right),
          new IntList (t.val, null))) ;
  }
}
```

Débarassons nous de ces concaténations coûteuses

Généralisons (un peu) le problème. Le mission de $P(T,k)$ est renvoyer la liste des étiquettes de T suivie de la liste k.

$$\begin{aligned} P(\emptyset, k) &= k \\ P((T_0, x, T_1), k) &= P(T_0, P(T_1, (x; k))) \end{aligned}$$

```
static IntList postfix(Tree t, IntList k) {
  if (t == null) {
    return k ;
  } else {
    return
      postfix(t.left, postfix(t.right, new IntList (t.val,k))) ;
  }
}
static IntList postfix(Tree t) { // Surcharge
  return postfix(t, null) ;
}
```

Comment programmer le parcours en largeur

- Le point de vue récursif fonctionne mal, pourquoi ?
- L'affichage des sous-arbres gauche et droit se mélange !

On va donc avoir recours à un raisonnement plus global.

- Il « suffit » en fait d'afficher *tous* les sommets de profondeur k avant les sommets de profondeur $k+1$.
- Par ailleurs un sommet de profondeur $k+1$ est *toujours* fils d'un (unique) sommet de profondeur k.
- Donc on peut afficher les sommets de profondeur k et en même temps collecter leurs fils.

Programmation du parcours en largeur

```
static void bfs(Tree t) {
  TreeList next = new TreeList(t,null) ; // int k=0 ;
  while (next != null) {  // next -> sous-arbres profondeur k
    TreeList now = next ; // now -> sous-abres profondeur k
    next = null ;
    for ( ; now != null ; now = now.next) {
// pour tous les sous-arbres de profondeur k
      Tree t = now.val ;
      if (t != null) {
        System.out.println(t.val) ; // Afficher
        next = new TreeList (t.left,
                 new TreeList (t.right, next)) ;
        // Collecter les fils (next -> profondeur k+1)
      }
    } // k = k+1 => next -> profondeur k
  }
}
```

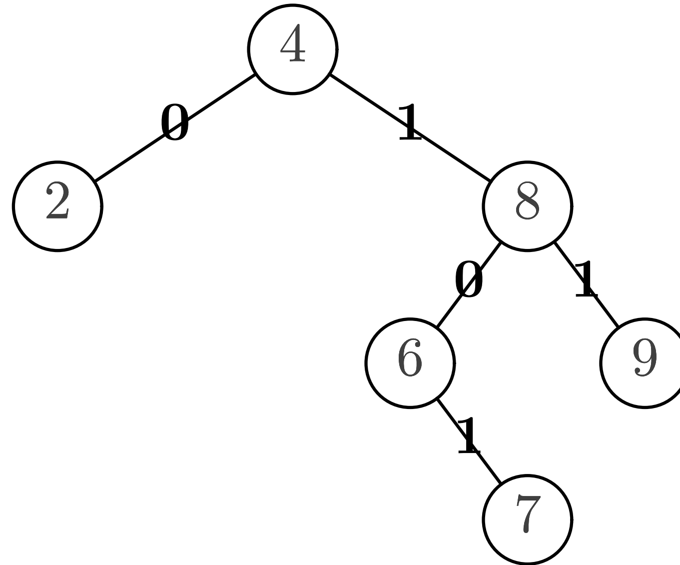
Programmation traditionnelle de la largeur d'abord

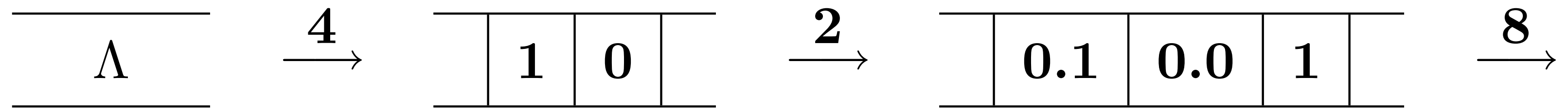
```
static void bfs(Tree t) {
  Fifo f = new Fifo () ;
  f.enfile() ;
  while (!f.isEmpty()) {
     Tree t = f.defile() ;
     if (t != null) {
       System.out.println(t.val) ;
       f.enfile(t.left) ;
       f.enfile(t.right) ;
     }
  }
}
```

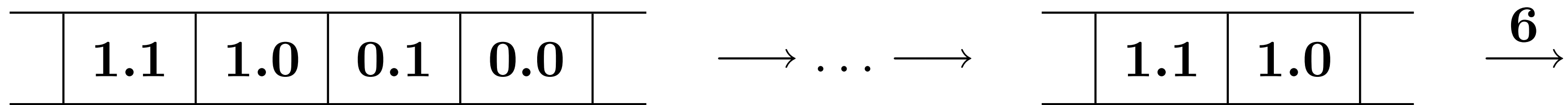
Pourquoi ça marche ?

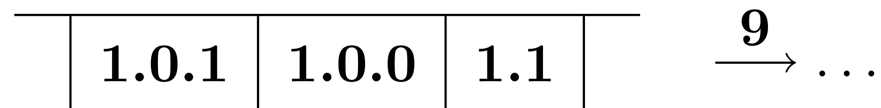
- Terminaison ? Soit $N(T)$ nombre de sommets et d'arbres vides dans un arbre (nombres de chemins)
- $\Sigma_{T \in \texttt{f}} N(T)$ décroit strictement à chaque tour de boucle.

Exemple de fonctionnement









Preuve de correction du parcours avec file

Transitions effectuées sur un état comprenant file $\mathcal{F}$ et affichage $\mathcal{A}$.

$$\mathcal{F} = F; (T_0, x, T_1)^k, \quad \mathcal{A} = A \quad \Rightarrow \quad \mathcal{F} = T_1^{k+1}; T_0^{k+1}; F, \quad \mathcal{A} = A; x^k$$
$$\mathcal{F} = F; \emptyset^k, \quad \mathcal{A} = A \quad \Rightarrow \quad \mathcal{F} = F, \quad \mathcal{A} = A$$

(T à la profondeur k, noté T^k.)

On voit alors que l'on a, en n étapes

$$\mathcal{F} = T_1^k; \cdots; T_n^k, \ \mathcal{A} = A \quad \stackrel{*}{\Rightarrow} \quad \mathcal{F} = U_1^{k+1}; \cdots; U_m^{k+1}, \ \mathcal{A} = A; x_1^k; \cdots; x_\ell^k$$

Ce qui suffit pour montrer, qu'en h grosses étapes, on a :

$$\mathcal{F} = T^0, \quad \mathcal{A} = \quad \stackrel{*}{\Rightarrow} \quad \mathcal{F} =, \quad \mathcal{A} = A$$

Où A est une séquence de sommet de profondeurs croissantes.

Et avec une pile explicite ?

```
static parcours(Tree t) {
  Stack<Tree> stack = new Stack<Tree> () ;
  stack.push(t) ;
  while (!stack.isEmpty()) {
     Tree t = stack.pop() ;
     if (t != null) {
       System.out.println(t.val) ;
       stack.push(t.right) ;
       stack.push(t.left) ;
     }
  }
}
```

C'est le parcours : préfixe !

- Il est clair que dans (T_0, x, T_1), x est affiché d'abord.
- Il est moins clair que T_0 est affiché avant T_1.

Preuve de l'affichage préfixe avec pile explicite

Transitions effectuées sur un état comprenant pile $\mathcal{S}$ et affichage $\mathcal{A}$.

$$\mathcal{S} = S; (T_0, x, T_1), \quad \mathcal{A} = A \quad \Rightarrow \quad \mathcal{S} = S; T_1; T_0, \quad \mathcal{A} = A; x$$
$$\mathcal{S} = S; \emptyset, \quad \mathcal{A} = A \quad \Rightarrow \quad \mathcal{S} = S, \quad \mathcal{A} = A$$

Notons $\mathcal{P}$ la séquence des sommets dans l'ordre préfixe :

$$\mathcal{P}(\emptyset) =, \qquad \mathcal{P}(T_0, x, T_1) = x; \mathcal{P}(T_0); \mathcal{P}(T_1)$$

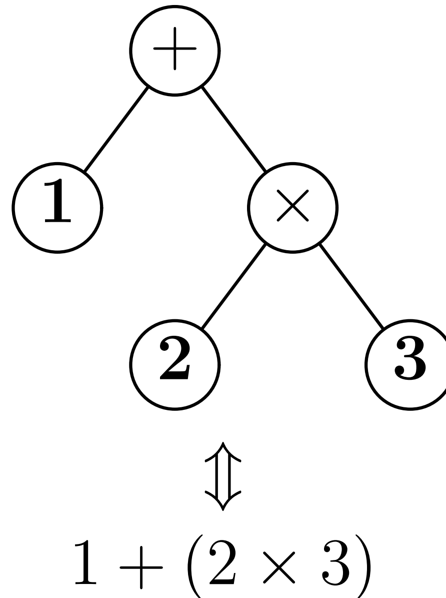
On veut montrer

$$\mathcal{S} = S; T, \quad \mathcal{A} = A \quad \stackrel{*}{\Rightarrow} \quad \mathcal{S} = S, \quad \mathcal{A} = A; \mathcal{P}(T)$$

Par induction structurelle sur T (cas de base laissé en exercice).

$$\begin{aligned}
\mathcal{S} = S; (T_0, x, T_1), \quad \mathcal{A} = A \quad & \Rightarrow \quad \mathcal{S} = S; T_1; T_0, \quad \mathcal{A} = A; x \\
& \stackrel{*}{\Rightarrow} \quad \mathcal{S} = S; T_1, \quad \mathcal{A} = A; x; \mathcal{P}(T_0) \\
& \stackrel{*}{\Rightarrow} \quad \mathcal{S} = S, \quad \mathcal{A} = A; x; \mathcal{P}(T_0); \mathcal{P}(T_1)
\end{aligned}$$

Les expressions arithmétiques



$$\Updownarrow$$

$$1 + (2 \times 3)$$

Approche récursive, une expression est :

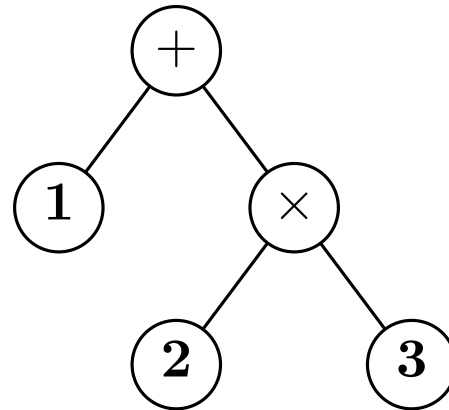
- Un entier (une feuille x).
- Une « opération » $(E_1, \oplus, E_2)$ (un sous-arbre binaire).

Inportant : Il n'y a pas d'arbre vide.

Réalisation simple

```
class Exp {
  final static int OP=0, INT=1 ;
  int nature ; // champ 'indicateur'

  int val ;   // valide si nature == INT
  char op ; Exp left, right ; // valides si nature == OP

  Exp (int val) {
    nature = INT ; this.val = val ;
  }

  Exp (Exp left, char op, Exp right) {
    nature = OP ;
    this.op = op ; this.left = left ; this.right = right ;
  }
}
```

Construction, avec les constructeurs




```
Exp e = new Exp(
    new Exp(1),
    '+',
    new Exp (new Exp(2), '*', new Exp(3))) ;
```

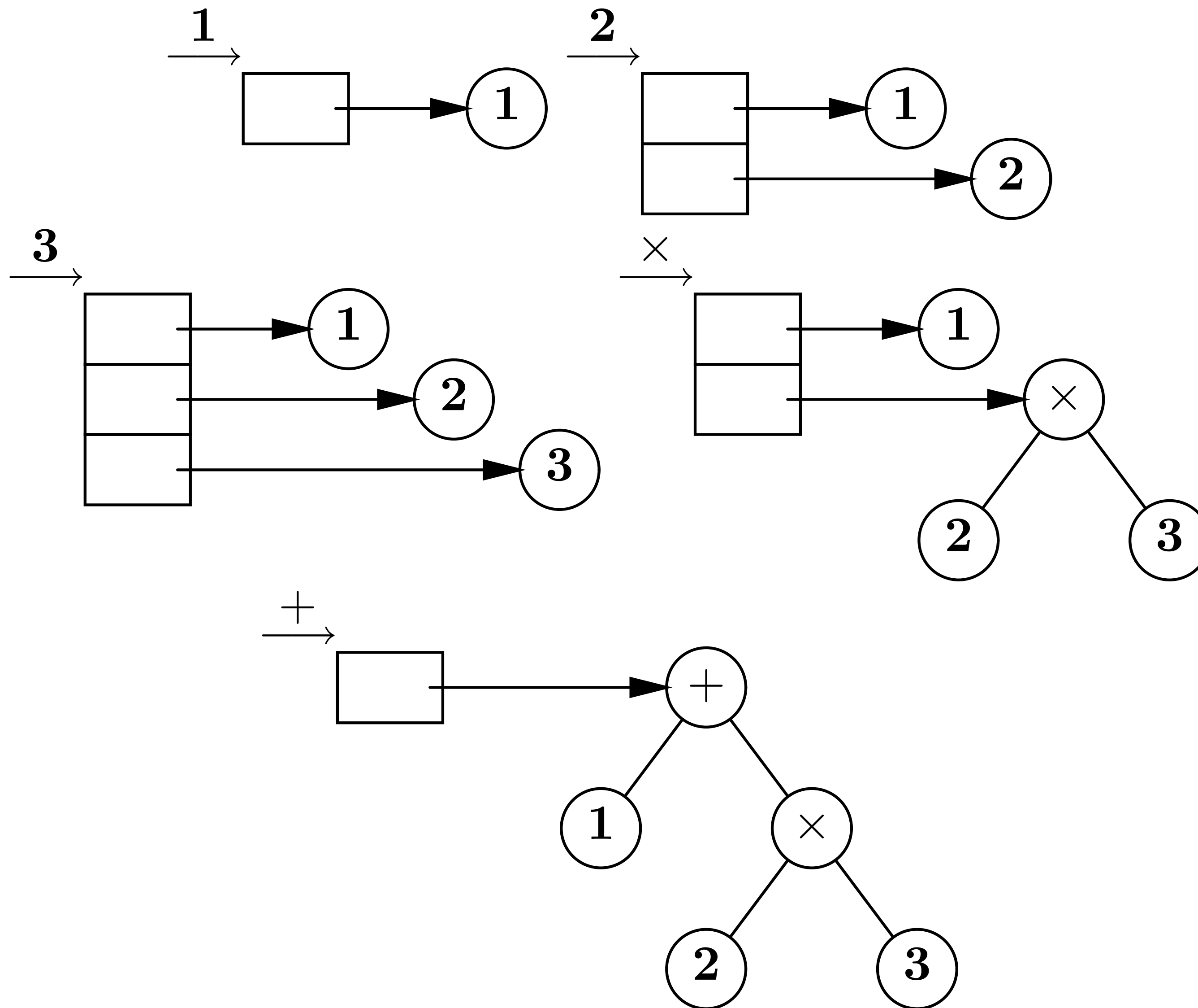
C'est lourd et en fait assez eloigné des habitudes.

Habituellement, on écrit $1 + 2 \times 3$ et on « comprend » l'arbre.

Pour le moment

- Transformer les expressions écrites linéairement (notation infixe) en arbre est trop compliqué (cf. INF 431).
- Mais nous saurons le faire avec des opérations en notation postfixe ! Avec une pile (d'expressions).
  - Un entier x : empiler la feuille **new** `Exp(`x`)`.
  - Un opérateur $\oplus$: dépiler x, dépiler y, empiler,**new** `Exp (`y`, `$\oplus$`, `x`)`.
- C'est très similaire à la calculette d'il y a quinze jours.

Exemple d'exécution du lecteur d'expressions



Réalisation du lecteur d'expressions

```
class Exp {
  public static void main (String [] arg) {
    Stack<Exp> stack = new Stack<Exp> ;
    for (int i = 0 ; i < arg.length ; i++) {
      String cmd = arg[i] ;
      if (cmd.equals("+")) {
        Exp x = stack.pop(), Exp = stack.pop() ;
        stack.push(new Exp (y, '+', x));
      } else if ...
      // Idem pour "*", "/", "-"
      } else {
        stack.push(new Exp (Integer.parseInt(cmd))) ;
      }
    }
    Exp t = stack.pop() ;
  }
}
```

Que faire de nos arbres ?

- Les évaluer,

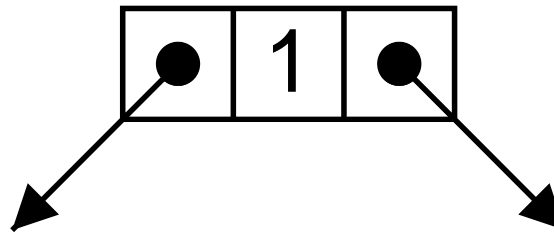
```
static int eval(Exp e) {
  if (e.nature == INT) {
    return e.val ;
  } else if (e.nature == OP) {
    switch (e.op) {
      case '+': return eval(e.left) + eval(e.right)  ;
      case '-': return eval(e.left) - eval(e.right)  ;
      case '*': return eval(e.left) * eval(e.right)  ;
      case '/': return eval(e.left) / eval(e.right)  ;
     }
  }
  throw new Error ("eval") ;
}
```

- Les afficher (sous forme infixe!).
- Les dériver…

Résumons nous

Il y a beaucoup de points de vue sur les arbres.

- Le point de vue pointeur :



- Le point de vue récursif :

$$T = \mathbf{null} \uplus (T \times \mathbf{int} \times T)$$

- Le point de vue cas particulier des graphes (cf. poly).

Un arbre est un graphe non orienté, non vide, connexe et sans circuit. Un sommet distingué est la racine.

Autrement dit, si on le dessine, on a un arbre !

Les arbres n-aires

Un sommet possède un nombre arbitraire de fils.

- L'arbre vide est un arbre.
- Si L est une liste d'arbre, alors (x, L) est un arbre.

Les diverses notions (parcours, etc.) restent naturelles.

En Java :

```
class Tree {
  int val ;
  TreeList sons ;
  Tree (int val, TreeList sons) {
    this.val = val ; this.sons = sons ;
  }
}

class TreeList { ... }
```

Soyons souples

Selon les circonstances, les listes peuvent être remplacées par des tableaux, des tableaux extensibles (`ArrayList<Tree>`) etc.

```
import java.util.* ;
class Tree {
  int val ;
  ArrayList<Tree> sons ;

  Tree (int val, ArrayList<Tree> sons) {
    this.val = val ; this.sons = sons ;
  }

  Tree (int val, Tree t1, Tree t2) {
    this.val = val ;
    this.sons = new  ArrayList<Tree> () ;
    this.sons.add(t1) ; this.sons.add(t2) ;
  }
}
```