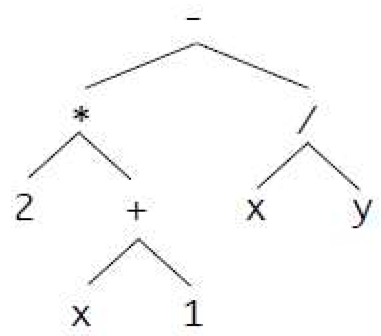
Agenda

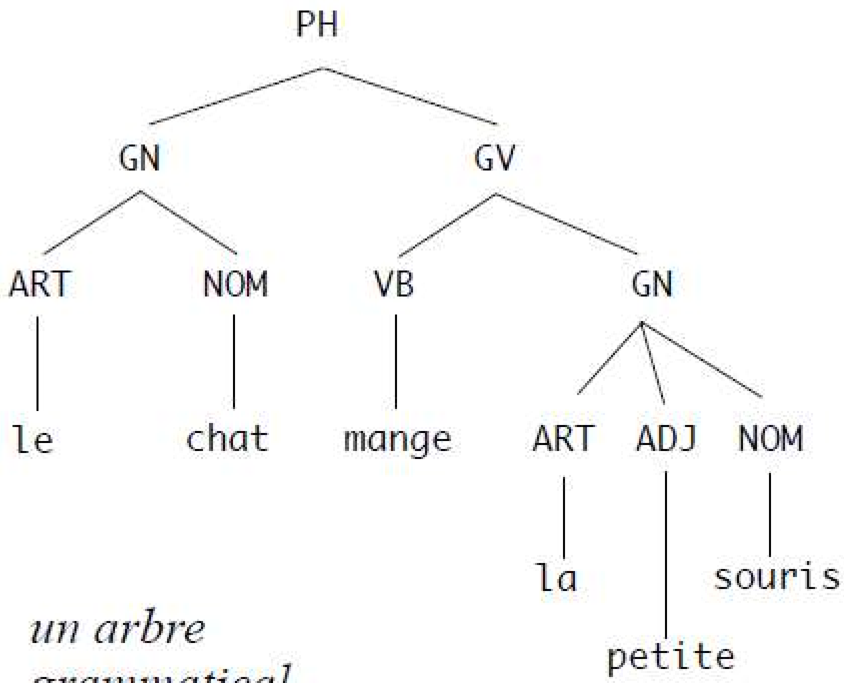
- Langage d'expressions préfixées
- Les fonctions
- Programmer avec des images / Animations
- Programmer par récurrence
- Les listes (chainées)
- Les calculs itératifs
- Type abstraits et généralisation
- **Les arbres binaires**

Divers types d'arbres

- Il y a plusieurs types d'arbres possibles, strictement binaires ou ayant un nombre variables de fils :



un arbre binaire d'expression



un arbre grammatical (1-2-3)

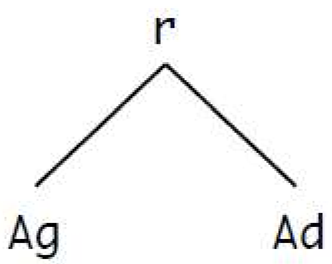
Les arbres binaires d'expressions

- Nous allons dans un premier temps centrer notre étude sur les **arbres binaires d'expressions algébriques** : analyser, transformer, compiler !

```
<arbre>   ::= <noeud> | <feuille>
<noeud>   ::= (<op> <arbre> <arbre>)
<op>      ::= + | - | * | /
<feuille> ::= VARIABLE | NOMBRE
```

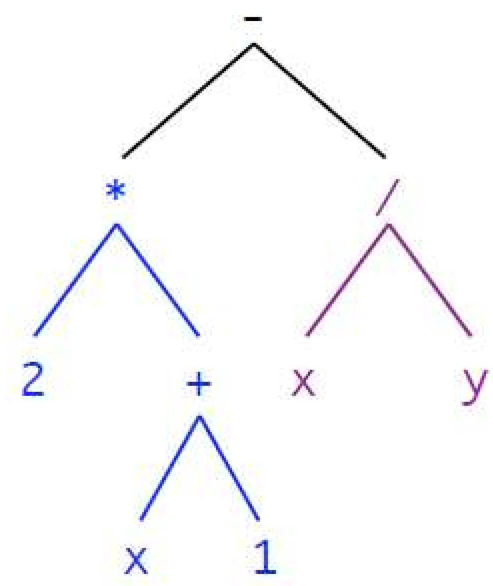
2 est un **arbre** [une *feuille*]

x est un **arbre** [une *feuille*]



est un **arbre** [un *noeud*] si :

- r est un opérateur
- Ag est un arbre
- Ad est un arbre



2*(x+1)-x/y

(- (* 2 (+ x 1)) (/ x y))

- **Le type abstrait « arbre binaire d'expression »**
- Le ***type abstrait*** est déjà dans le teachpack valrose.rkt
- Un **arbre binaire d'expression** sera représenté :
  - si c'est une ***feuille***, directement par cette feuille.
  - si c'est un ***noeud***, par une liste à 3 éléments (r Ag Ad)

```
(define (arbre r Ag Ad)
  (list r Ag Ad))
```

```
(define (feuille? obj)
  (or (number? obj)
      (and (symbol? obj) (not (operateur? obj)))))
```

```
(define (operateur? obj)
  (member obj '(+ * - /)))
```

  - • Il n'y a ***pas d'arbre vide*** dans cette théorie !

- Les trois accesseurs suivent la grammaire :

```
(define (racine A)       ; A est un noeud
  (if (feuille? A)
    (error "racine : Pas de racine pour " A)
    (first A)))
```

```
(define (fg A)       ; A est un noeud
  (if (feuille? A)
    (error "fg : Pas de fils gauche pour " A)
    (second A)))
```

```
(define (fd A)       ; A est un noeud
  (if (feuille? A)
    (error "fd : Pas de fils droit pour " A)
    (third A)))
```

• Construction d'un arbre

- Pour construire un arbre, on peut :
  - soit passer proprement par le constructeur du type abstrait :

```scheme
(define AT          ; un arbre pour les tests
  (arbre '-
         (arbre '* 2 (arbre '+ 'x 1))
         (arbre '/ 'x 'y)))
```

- soit *outrepasser le type abstrait* et utiliser directement une liste.
- C'est mal, et on le réservera uniquement aux tests !

```scheme
(define AT '(- (* 2 (+ x 1)) (/ x y)))
```

- Quoiqu'il en soit :

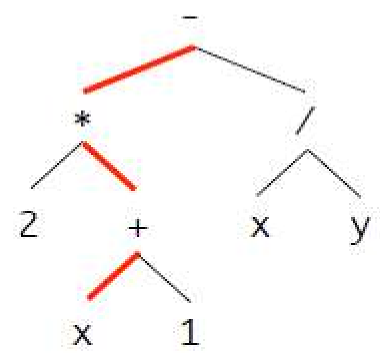
```scheme
> AT
(- (* 2 (+ x 1)) (/ x y))
> (fg AT)
(* 2 (+ x 1))
```

```scheme
> (fd (fg AT))
(+ x 1)
> (racine (fd AT))
/
```

- Quelques algorithmes de base
- **Hauteur d'un arbre**
- C'est la longueur d'un chemin de longueur maximum reliant la racine à une feuille :

```
(define (hauteur A)
  (if (feuille? A)
      0
      (+ 1 (max (hauteur (fg A)) (hauteur (fd A))))))
```

Notez la récurrence double !




```
> (hauteur AT)
3
```

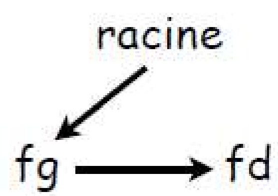
- **N.B.** Un arbre de hauteur h contient au plus 2^h feuilles [s'il est *complet*]. Inversement, s'il contient n feuilles, on s'attend à ce qu'il ait une hauteur telle que $n=2^h$, d'où $h=\log_2(n)$ en moyenne ...

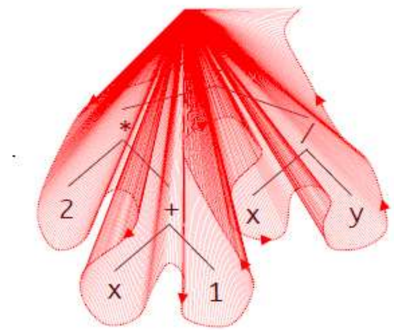
- # Présence d’une feuille
- La **présence d'une feuille** donnée x dans un arbre A

```
(define (feuille-de? x A)          ; x est-elle une feuille de A ?
  (if (feuille? A)
      (equal? x A)
      (or (feuille-de? x (fg A))
          (feuille-de? x (fd A)))))
```

- Notez le ***court-circuit*** du or qui évite d'explorer le fils droit si la feuille est trouvée à gauche !
- **DEFINITION** : Un parcours est **en profondeur** si l'un des fils est complètement exploré avant d'entamer l'exploration de l'autre !

- Le parcours ci-contre est donc un *parcours en profondeur préfixe.*



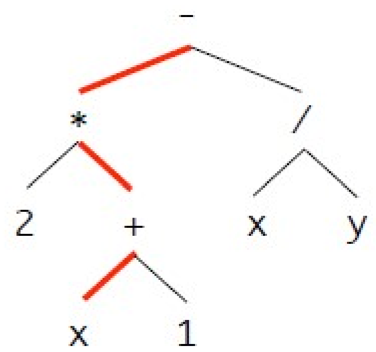


• Feuillage d'un arbre

- La **présence d'une feuille** donnée x dans un arbre A
- Calculer le **feuillage d'un arbre** revient à produire la liste plate de ses feuilles dans un parcours en profondeur préfixe :

```
(define (feuillage A)
  (if (feuille? A)
      (list A)
      (append (feuillage (fg A)) (feuillage (fd A)))))
```

```
> (feuillage AT)
(2 x 1 x y)
```

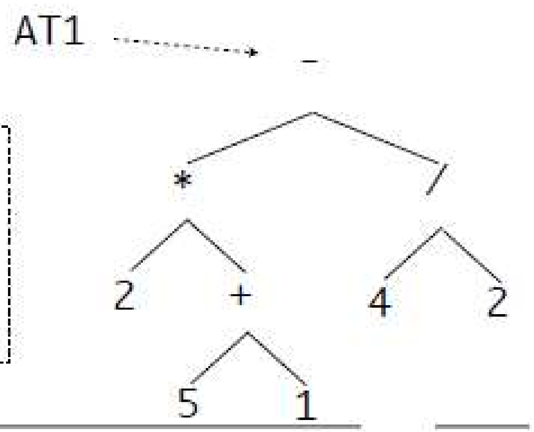


- Variante : le nombre de feuilles.

```
> (nombre-de-feuilles AT)
5
```

```
(define (nombre-de-feuilles A)
  (if (feuille? A)
      1
      (+ (nombre-de-feuilles (fg A)) (nombre-de-feuilles (fd A)))))
```

Valeur d'un arbre arithmétique



DEFINITION : Un arbre sera dit **arithmétique** si toutes ses feuilles sont des constantes. Il est **algébrique** si au moins une feuille est une variable.

```
(define (valeur A)       ; l'arbre A est arithmétique !
  (if (feuille? A)
      A
      (local [(define r (racine A))
              (define vg (valeur (fg A)))
              (define vd (valeur (fd A)))]
        (cond ((equal? r '+) (+ vg vd))
              ((equal? r '-) (- vg vd))
              ((equal? r '*) (* vg vd))
              ((equal? r '/) (/ vg vd))
              (else (error "valeur : opérateur inconnu : " (racine A)))))))
```

```
> (valeur AT1)
10
```

N.B. Une erreur grave consisterait à écrire `((racine A) vg vd)`, ce serait une *erreur de typage*. Pourquoi ?...

Complément Scheme : la construction linguistique case

• La série de tests avec equal? dans le cond de la fonction valeur n'est pas très agréable. On utilise plutôt un case :

```
(define (valeur A)                    ; A est arithmétique !
  (if (feuille? A)
      A
      (local [(define vg (valeur (fg A)))
              (define vd (valeur (fd A)))]
        (case (racine A)
          ((+) (+ vg vd))
          ((-) (- vg vd))
          ((*) (* vg vd))
          ((/) (/ vg vd))
          (else (error "valeur : opérateur inconnu : " (racine A)))))))
```

N.B. On peut prévoir *plusieurs cas en même temps* :

```
(case (racine A)                  ⇔     (local [(define x (racine A))]
   ((+ -) ...)                            (cond ((member? x '(+ -)) ...)
   ((*) ...)                                    ((equal? x '*) ...)
   etc                                          etc
```

Valeur d'un arbre algébrique

• On suppose maintenant que l'arbre est **algébrique** : il contient des variables !

• La fonction `valeur` ne peut pas calculer si elle ne possède pas les *valeurs des variables* de l'arbre. On va donc les passer en paramètre...

• Sous la forme d'une **liste de couples** (*<variable>* *<valeur>*) comme :

((x 3) (y 8) (z -1))

DEFINITION : Une telle liste de couples (*<variable>* *<valeur>*) se nomme une **A-liste**, ou liste d'associations. Les variables sont les **clés**.

• La primitive `(assoc x AL)` retourne la première association (x *valeur*) si elle existe, ou bien `false` si x n'est pas une clé :

```
> (assoc 'y '((x 3) (y 8) (z -1)))
(y 8)
```

```
> (assoc 'b '((x 3) (y 8) (z -1)))
#f
```

- Si elle n'existait pas, voici une définition de (assoc x AL) :

```
(define ($assoc x AL)
  (cond ((empty? AL) #f)
        ((equal? (first (first AL)) x) (first AL))
        (else ($assoc x (rest AL)))))
```

- En utilisant assoc, il est alors facile de modifier le traitement des feuilles et d'obtenir la nouvelle version de la fonction (valeur A AL) :

```
(define (valeur2 A AL)          ; A contient des variables dont on va chercher les valeurs dans AL
 (if (MaFeuille? A)
   (if (number? A)
     A
     (local [(define essai (assoc A AL))]     ; essai = #f ou (A v)
      (if (not (equal? essai #f))
        (second essai)
        (error 'valeur "Variable indefinie:" A))))
   (local [(define vg (valeur2 (MonFG A) AL)) (define vd (valeur2 (MonFD A) AL))]
    (case (MaRacine A)
     ((+) (+ vg vd))
     ((-) (- vg vd))
     ((*) (* vg vd))
     ((/) (/ vg vd))
     (else (error 'valeur "Operateur inconnu:" (MaRacine A)))))))

(define MaListe_Assoc '((x 3) (y 4) (z 5)))

;;(valeur2 MonAT MaListe_Assoc)
```

Production de code pour une calculette HP

• Les calculettes HP utilisent une machine à pile. La séquence de touches $\boxed{3}$ $\boxed{\text{ENTER}}$ fait entrer 3 dans la pile, au sommet. La touche $\boxed{+}$ remplace les deux éléments au sommet par leur somme.

```
> (scheme->hp '(- (* 2 (+ 5 1)) (/ 4 2)))
(2 enter 5 enter 1 enter + * 4 enter 2 enter / -)
```

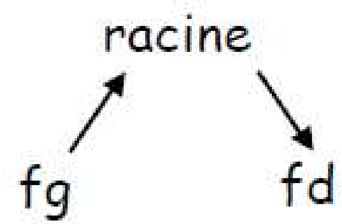
• La traduction s'obtient donc par un **parcours en profondeur postfixe.**

racine

fg → fd

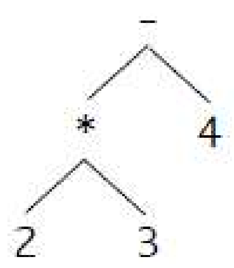
```
(define (scheme->hp A)
  (if (feuille? A)
      (list A 'enter)
      (append (scheme->hp (fg A))
              (scheme->hp (fd A))
              (list (racine A)))))
```

• Le **parcours en profondeur infixe** est utilisé pour travailler avec des expressions mathématiques. Hélas il s'agit d'une notation ambigüe, ***difficile à analyser*** [priorités d'opérateurs].

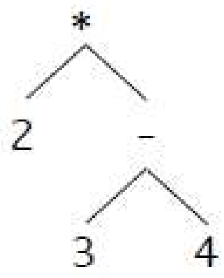


2 * 3 - 4

en langage Python :



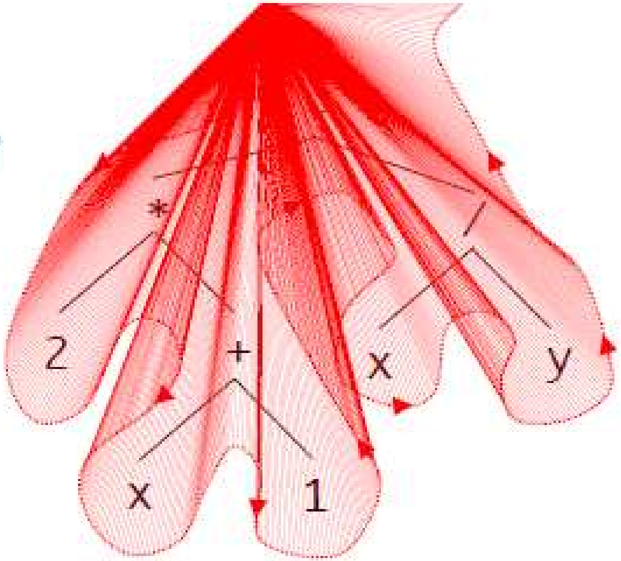
en langage APL :



• C'est parce qu'elle ne nécessite pas de priorité d'opérateurs que la notation préfixée complètement parenthésée a été choisie par LISP ! On peut bien entendu présenter les résultats comme on veut...

Parcours préfixe plat d'un arbre

• Etant donné un arbre A, on cherche à obtenir la liste de tous les éléments rencontrés lors d'un parcours en profondeur préfixe :




```
> (arbre->prefixe '(- (* 2 (+ x 1)) (/ x y)))
(- * 2 + x 1 / x y)
```

```
(define (arbre->prefixe A)     ; Arbre → Liste
  (if (feuille? A)
    (list A)
    (cons (racine A)
          (append (arbre->prefixe (fg A)) (arbre->prefixe (fd A))))))
```

• En quelque sorte, on a *enlevé les parenthèses* !

• Saurait-on les remettre : programmer la fonction réciproque ?...

La réciproque : arboriser un parcours préfixe plat !

• Peut-on enlever les parenthèses à Scheme ? Oui si l'arité de chaque fonction est bien définie. Par exemple avec des opérateurs binaires :

```
(+ * - x y z + u v)
```

parcours préfixe ↑ ↓ arboriser

bijectif !

```
(+ (* (- x y) z) (+ u v))
```

• **Arboriser un parcours préfixe plat** consiste à remettre les parenthèses, à retrouver la structure d'arbre. Il s'agit d'un cas particulier d'ANALYSE SYNTAXIQUE.

• Une bonne solution ne doit faire qu'**un seul passage sur la liste** !

• **EUREKA** : notons que **le reste d'un parcours préfixe** n'est plus un parcours préfixe, mais **débute par un parcours préfixe** ! C'est dans ce genre de situation qu'il ne faut pas hésiter à **GE-NE-RA-LI-SER**.

• Etant donnée une liste plate L **débutant par un parcours préfixe**, produire le couple (A, R) où A est l'arbre de ce parcours préfixe et R la liste restante. Exemple : L = (+ * - x y z + u v a b c d)

A = (+ (* (- x y) z) (+ u v))   R = (a b c d)

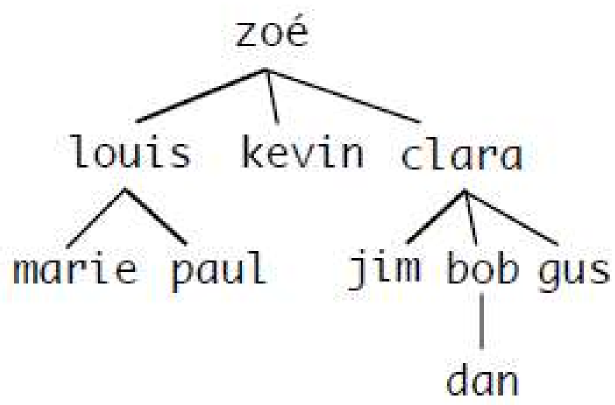
```
(define (arboriser L)                    ; retourne (A  R)
  (cond ((empty? L) (error "arboriser : Parcours incomplet !"))
        ((operateur? (first L)) ....)
        (else ....)))
```

cf TP !

```
> (arboriser '(+ * - x y z + u v a b c d))
((+ (* (- x y) z) (+ u v)) (a b c d))
```

• De manière générale, ne jamais sous-estimer la force de retourner en résultat <u>non seulement le résultat mais en plus une indication sur ce qu'il reste à faire</u> ! D'où l'intérêt des fonctions à plusieurs résultats...

Vers les arbres généraux...



- Nos arbres binaires d'expressions peuvent être généralisés : un noeud pourra accepter un nombre arbitraire de fils.

- Nous abondonnons fg et fd, et parlerons de la **forêt des fils**, une liste L contenant la suite des fils, de gauche à droite. La grammaire des arbres d'expression devient donc :

```
<arbre>   ::= <noeud> | <feuille>
<noeud>   ::= (<op> <arbre> <arbre> …)
<op>      ::= + | - | * | /
<feuille> ::= VARIABLE | NOMBRE
```

```
(define _arbre list)        ; le constructeur de noeud
(define _racine first)      ; la racine, sur un noeud seulement !
(define _foret rest)        ; la forêt des fils, sur un noeud seulement !
(define (_feuille? A) (not (list? A)))

(define ATEST '(zoé (louis marie paul) kevin (clara jim (bob dan) gus)))
```

• Exemple d'algorithme : recherche du sous-arbre de A de racine p. Il s'agit de ne pas se mélanger les pinceaux entre arbres et listes !

```
; le sous-arbre de racine p dans l'arbre A, ou bien #f
(define (sous-arbre p A)
  (cond ((_feuille? A) (if (equal? A p) A #f))
        ((equal? (_racine A) p) A)
        (else (chercher p (_foret A)))))

(define (chercher p L)          ; L est une forêt d'arbres
  (if (empty? L)
      #f
      (local [(define try (sous-arbre p (first L)))]
        (if (not (equal? try #f))
            try
            (chercher p (rest L))))))
```

```
> (sous-arbre 'kevin ATEST)
kevin
> (sous-arbre 'clara ATEST)
(clara jim (bob dan) gus)
```

```
> (sous-arbre 'bob ATEST)
(bob dan)
> (sous-arbre 'nicolas ATEST)
#f
```