Université Nice Sophia Antipolis

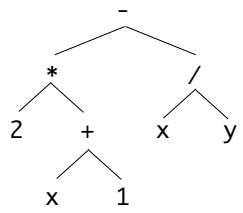
Les Arbres Binaires d'Expressions

Type abstrait
Parcours Récursif
Calcul Formel

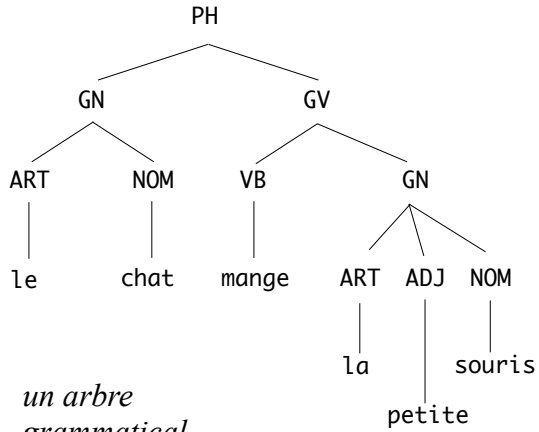
pp. 161-171
et 201-206

Divers types d'arbres

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



un arbre binaire d'expression



un arbre grammatical (1-2-3)

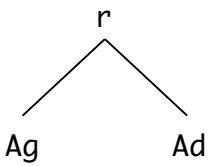
Les arbres binaires d'expressions

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

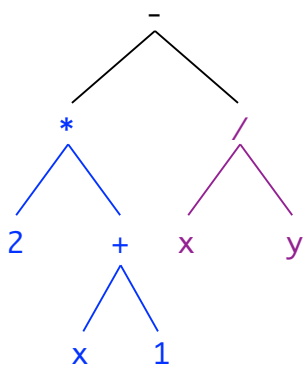
```
<arbre> ::= <noeud> | <feuille>
<noeud> ::= (<op> <arbre> <arbre>)
<op>    ::= + | - | * | /
<feuille> ::= VARIABLE | NOMBRE
```

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 '(+ * - /)))
```

- **NB** : 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 :

```
(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 fera uniquement pour les tests :

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

• Quoiqu'il en soit :

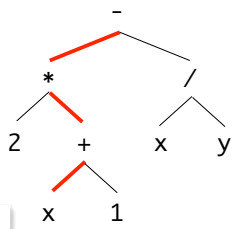
```
> AT
(- (* 2 (+ x 1)) (/ x y))
> (fg AT)
(* 2 (+ x 1))
```

```
> (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))))))
```

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

Notez la récurrence double !

N.B. Un arbre de hauteur h contient au plus 2^h feuilles [s'il est *parfaitement équilibré*]. 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

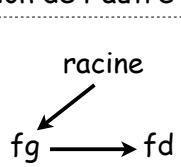
• La **présence d'une feuille** donnée x dans un arbre A.

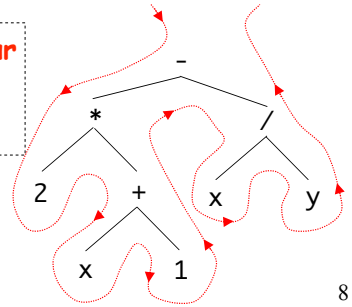
```
(define (feuille-de? x A)       ; x est une feuille de A ?
  (if (feuille? A)
      (equal? x A)
      (or (feuille-de? x (fg A))
          (feuille-de? x (fd A)))))
```

• Notez le *court-circuit* du or qui évite d'explorer le fils droit si la feuille est trouvée à gauche !

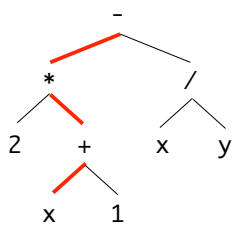
DEFINITION : Un parcours est **en profondeur** si l'un des fils est complètement exploré avant d'entamer l'exploration de l'autre !

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





Feuillage d'un arbre



• 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)
```

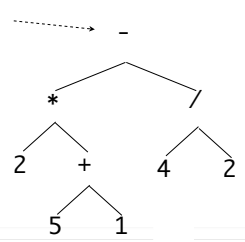
• Variante : le **nombre de feuilles**.

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

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


Valeur d'un arbre arithmétique

AT1 ⇢



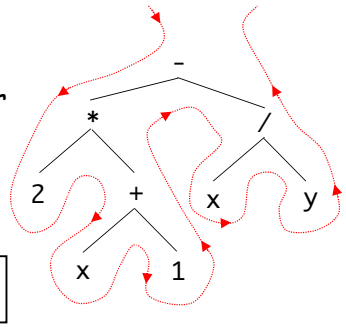
DEFINITION : Un arbre sera dit **arithmétique** si toutes ses feuilles sont des **constantes**. Il est **algébrique** si au moins une feuille est une variable.

```
(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 : " r))))))
```

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

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

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 ?

cf TD !

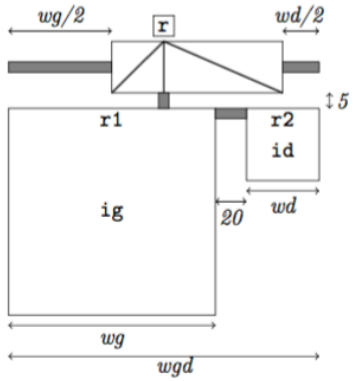
Comment dessiner un arbre ?

• **Par récurrence** pardi ! Puisque l'arbre est une **donnée récursive**...

• Il s'agit de construire une **image**. Si l'arbre est un noeud, il suffit de supposer connues l'image du fg et l'image du fd. On sera donc *ipso facto* en possession de leur dimension, et il suffira de les placer proprement en-dessous de la racine.

• Voir le livre de cours PCPS page 171 pour les détails...

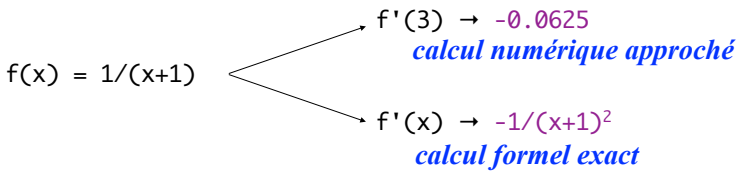
• Mais faites-le vous-même, c'est plus *rigolo* !



Mais qu'est-ce que le **CALCUL FORMEL** ?

• Le **calcul formel** [*computer algebra*] consiste à pratiquer une algorithmique **symbolique** sur des objets (ici mathématiques).

• ***Symbolique*** est opposé à ***numérique*** : on travaille sur des formes **exactes** plutôt que sur des approximations numériques !



• Les premiers gros systèmes de calcul formel furent REDUCE [1968] et MACSYMA [1970], tous deux écrits en LISP. MACSYMA existe encore, on peut le télécharger [http://maxima.sourceforge.net].

http://fr.wikipedia.org/wiki/Logiciel_de_calcul_formel

Le Problème de la **Simplification**

• C'est en fait le problème central ! Comment réduire les expressions à des formes irréductibles, de préférence canoniques [pour l'unicité] ?

• Un **évaluateur** peut calculer la valeur numérique d'un arbre :

```
(valeur '(+ (* 2 3) 5))                    → 11
(valeur '(+ (* x 2) y)) '((x 3) (y -1))) → 5
```

• Un **simplificateur** sait travailler avec des arbres *algébriques*, en présence d'**inconnues** :

```
(simplif '(+ (* x (- 5 4)) (+ 1 (* 0 y)))) → (+ x 1)
```

• Il s'agit donc bien d'un algorithme ***formel***, avec localement quelques calculs sur constantes, comme celui de `(- 5 4)`.

Algebraic simplification: a guide for the perplexed
Joel Moses, Project MAC, MIT, 1971

• Nous essayons de simplifier nos arbres binaires d'expressions. Pour être modulaire, le toplevel du simplificateur se contente d'aiguiller sur des **simplificateurs spécialisés** :

```
(define (simplif A)
  (if (feuille? A)
      A
      (case (racine A)
        ((+) (simplif+ A))
        ((-) (simplif- A))
        ((*) (simplif* A))
        ((/) (simplif/ A))
        (else (error "simplif : Op. inconnu " (racine A))))))
```

• Chaque spécialiste va implémenter ses propres règles de réduction :

```
(define (simplif+ A)              ; on sait que A est un arbre de racine +
  (local [(define Ag (simplif (fg A)))
          (define Ad (simplif (fd A)))]
    (cond ((and (number? Ag) (number? Ad)) (+ Ag Ad))
          ((equal? 0 Ag) Ad)
          ((equal? 0 Ad) Ag)                    etc, cf TP !
          (else (arbre '+ Ag Ad)))))
```

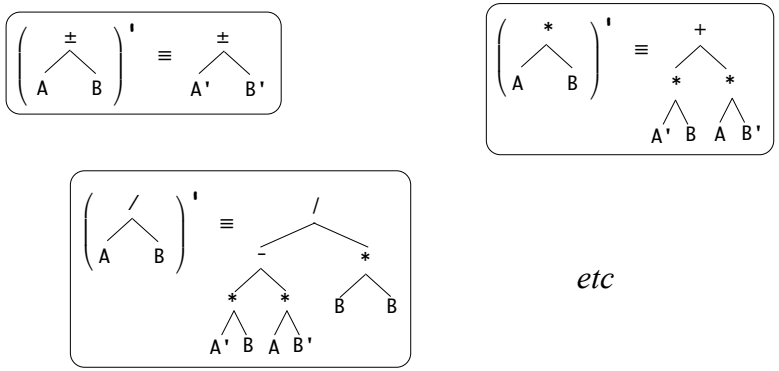

Le Problème de la **Dérivation Symbolique**

$$A = y/x \quad \rightarrow \quad \frac{\partial}{\partial x} A = -y/x^2 \quad \frac{\partial}{\partial y} A = 1/x \quad \frac{\partial}{\partial z} A = 0$$

• But : dériver un arbre d'expression `A` par rapport à une variable `v`. Ici aussi le dérivateur va aiguiller sur des **dérivateurs spécialisés**. Le résultat doit être simplifié *autant que faire se peut* car la taille des arbres dérivés croît très vite !

```
(define (diff A v)                          ; retourne ∂A/∂v
  (if (feuille? A)
      (if (equal? A v) 1 0)                 ; ∂v/∂v=1  et  ∂u/∂v=0
      (local [(define op (racine A))]
        (cond ((member op '(+ -)) (diff+- A v))
              ((equal? op '*)   (diff* A v))
              ((equal? op '/)   (diff/ A v))
              (else  (error "diff : Op. inconnu " (racine A)))))))
```

- Laissons-nous guider par les **règles de dérivation** usuelles :



etc

- Voici par exemple le spécialiste de la **dérivation d'une somme** :

```
(define (diff+- A v)                    ; A = (± Ag Ad)
  (arbre (racine A) (diff (fg A) v) (diff (fd A) v)))
```

- Notez la ***récurrence croisée*** : `diff` utilise `diff+-` et inversement !
  co-récurrence

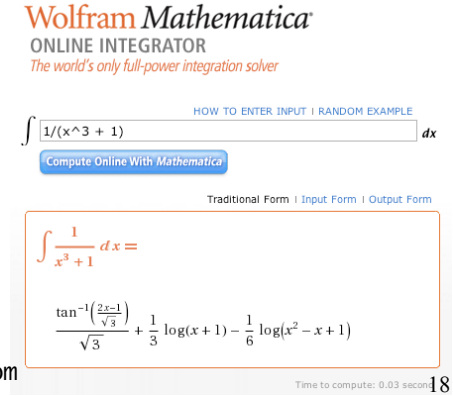
Le Problème de l'**Intégration Symbolique**

- *On parle bien de recherche d'une **primitive**, pas de calcul approché !*

- Hélas le problème est **infiniment plus difficile** ! Si la dérivation est purement mécanique, l'intégration requiert de l'expertise (**IA**)...

- Stratégie : intégration par parties, changement de variables ???...

++PROBLEME : certaines fonctions n'ont PAS de primitive exprimable comme combinaison de ***fonctions élémentaires !***



- Les logiciels de calcul formel font de leur mieux (et en général ne calculent pas comme nous)...

`http://integrals.wolfram.com`

Résoudre une équation de manière **symbolique** ?

- A l'opposé de la résolution *numérique approchée*...

- Cas simplifié : une équation **A = B** où A et B sont deux arbres binaires, la variable x n'apparaissant qu'une seule fois, seulement dans l'arbre A.

```
(define (solve A B var)
  ...)

> (solve '(/ a (- (* 2 x) 1)) 3 'x)
(/ (+ (/ 3 a) 1) 2)
```

$$\frac{a}{2x-1} = 3$$
$$2x - 1 = \frac{a}{3}$$
$$2x = \frac{a}{3} + 1$$
$$x = \frac{\frac{a}{3}+1}{2} \quad \text{STOP}$$

- C'était un problème d'examen final il y a 4 ans... Assez facile, mais il faut être soigneux, et surtout être capable de savoir si l'inconnue est dans le sous-arbre gauche ou droit de A, pour converger vers elle !

Arbres et Fonctions

- Un arbre est une **expression**, pas une **fonction** ! Comme si l'on confondait $x^2 + 1$ et $x \mapsto x^2 + 1$

- Comment ***construire la fonction associée à un arbre*** ?
  - on part d'un arbre `A`, par exemple `(* x (+ x 2))`, en la variable `x`.
  - on sait prendre sa valeur en un point x_0 avec `(valeur A AL)`, *cf TD !*
  - on sait construire une fonction avec le constructeur `lambda`.

```
(define (arbre->fonction A v)       ; v est la variable : x, y, ...
  (lambda (x0) (valeur A (list (list v x0)))))     ; cf TD !
```

```
> (define A '(* x (+ x 2)))
> (define f (arbre->fonction A 'x))
> (map (lambda (x) (list x (f x))) '(0 1 2 3 4))
((0 0) (1 3) (2 8) (3 15) (4 24))
```

N.B. On a passé la variable de l'arbre en paramètre. On pourrait aussi la trouver automatiquement (s'il n'y en a qu'une)...