1 Expression algébrique totalement parenthésée (sans variable)

Soit X un ensemble, $\mathcal{O}$ un ensemble d'applications de $X \times X$ dans X (les opérateurs binaires) et $\mathcal{F}$ un ensemble d'applications de X dans X (les opérateurs unaires encore appelés fonctions). On notera $\mathcal{P}$ l'ensemble à deux éléments constitué de la parenthèse ouvrante et de la parenthèse fermante.

Définition 1.1 *On appelle expression algébrique totalement parenthésée (abréviation EATP) le sous ensemble des suites finies d'éléments de* $X \cup \mathcal{O} \cup \mathcal{F} \cup \mathcal{P}$ *défini récursivement par*

- *si* $x \in X$, *alors la suite à un élément* x *est une EATP*
- *si* E *est une EATP et* $f \in \mathcal{F}$, *alors la suite* $f\ E$ *est une EATP*
- *si* E_1 *et* E_2 *sont deux EATP et* $f \in \mathcal{O}$ *un opérateur binaire, alors la suite* $(\ E_1\ f\ E_2\ )$ *est encore une EATP*

Exemple 1.1 $X = \mathbf{Z}$, $\mathcal{O} = \{+, -, *\}$, $\mathcal{F} = \{\pm\}$ ($\pm : x \mapsto -x$ pour le distinguer de l'opérateur binaire de différence)

- $((\pm(1+5) + (7*8)) - 3)$ est une EATP ; en effet
  - 1 et 5 sont des EATP, donc $(1+5)$ est une EATP et donc aussi $\pm(1+5)$
  - 7 et 8 sont des EATP, donc aussi $(7*8)$
  - en conséquence $(\pm(1+5) + (7*8))$ est encore une EATP
  - et donc $((\pm(1+5) + (7*8)) - 3)$ est encore une EATP
- $(2 + 3 * 5)$ n'est pas une EATP : en l'absence de règles de priorités, une telle expression algébrique courante peut aussi bien s'évaluer en $((2+3)*5)$ qu'en $(2 + (3*5))$
- $(2 + (3*))$, $(2 + 4$, $2 + 4$ ne sont pas des EATP (même si cette dernière a une signification évidente)

Une expression algébrique totalement parenthésée est définie par la *grammaire* suivante (où le symbole ::= doit se lire *est la même chose que* et le symbole | doit se lire *ou bien*) :

- nombre ::= élément de X
- opérateur ::= élément de $\mathcal{O}$
- fonction ::= élément de $\mathcal{F}$
- EATP : := nombre | fonction EATP | (EATP opérateur EATP)

ce qui peut encore se lire

- un *nombre* désigne n'importe quel élément de X
- un *opérateur* désigne n'importe quel élément de $\mathcal{O}$
- une *fonction* désigne n'importe quel élément de $\mathcal{F}$
- une EATP est soit un nombre, soit la suite formée d'une fonction et d'une EATP, soit la suite formée d'une parenthèse ouvrante, d'une EATP, d'un opérateur, d'une EATP et d'une parenthèse fermante.

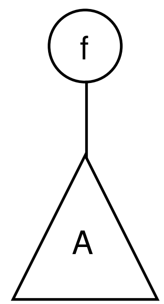
1.1 Arbre d'une EATP

Arbre binaire hétérogène vérifiant les propriétés suivantes :

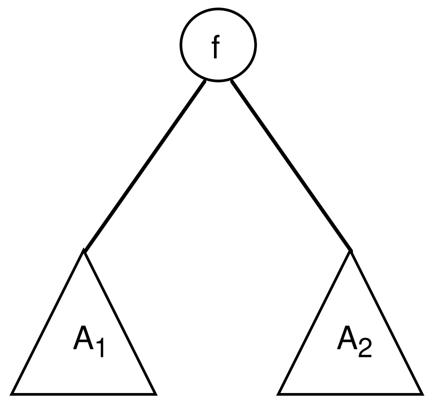
- les feuilles de l'arbre sont étiquetées par des éléments de X
- les noeuds d'ordre 2 de l'arbre sont étiquetés par des éléments de $\mathcal{O}$
- les noeuds d'ordre 1 de l'arbre sont étiquetés par des éléments de $\mathcal{F}$

Un tel arbre sera dit de type $(X, \mathcal{O}, \mathcal{F})$.

- si x est un élément de X, l'arbre associé à l'EATP x est l'arbre sans noeud dont l'unique feuille est étiquetée par x
- si E est une EATP dont l'arbre associé est A et si f est un élément de $\mathcal{F}$, alors l'arbre associé à l'EATP $f\ E$ est



- si E_1 et E_2 sont deux EATP dont les arbres associés sont A_1 et A_2 et si f est un opérateur, alors l'arbre associé à l'EATP ($E_1\ f\ E_2$) est


f
A1
A2

Théorème 1.1 *L'arbre d'une EATP est défini de manière unique et l'application qui à une EATP associe son arbre est une bijection de l'ensemble des EATP sur l'ensemble des arbres binaires hétérogènes de type* $(X, \mathcal{O}, \mathcal{F})$.

Démonstration: considérons une EATP E ; alors soit E est réduite à un élément x de X, soit elle commence par une fonction f de $\mathcal{F}$, soit elle commence par une parenthèse. Ceci montre que lorsque l'on considère une EATP, il est immédiat de savoir dans laquelle des trois situations inductives nous nous trouvons, et donc que l'arbre d'une EATP est défini de manière unique par induction. Inversement, étant donné un arbre binaire hétérogène de type $(X, \mathcal{O}, \mathcal{F})$, on lui associe de manière unique une expression algébrique de la manière suivante :

- si l'arbre est réduit à une feuille x, alors l'expression algébrique associée est x
- si l'arbre admet une racine d'ordre 1 étiquetée par $f \in \mathcal{F}$ ayant pour unique branche A, alors l'expression algébrique associée est $f\ E$ où E est l'expression algébrique associée à A
- si l'arbre admet une racine d'ordre 2 étiquetée par $f \in \mathcal{O}$ de branche gauche A_1 et de branche droite A_2, alors l'expression algébrique associée est $(\ E_1\ f\ E_2\ )$ où E_i est l'expression algébrique associée à A_i.

Traduction Caml : définir la réunion disjointe $X \cup \mathcal{F} \cup \mathcal{O} \cup \mathcal{P}$.

```
# type EATP =
       | Nombre of int
       | Fct of string
       | Op of string
       | Delim of char;;
Type EATP_elt defined.
```

Les suites d'éléments de $X \cup \mathcal{F} \cup \mathcal{O} \cup \mathcal{P}$:

```
#type EATP == EATP_elt list;; (* définition d'un synonyme *)
Type EATP defined.
```

Arbres de type $(X, \mathcal{O}, \mathcal{F})$

```
#type arbre_XOF =
             Feuille of int
          |  Noeud1 of string * arbre_XOF
          |  Noeud2 of arbre_XOF * string * arbre_XOF;;
Type arbre_XOF defined.
```

Fonction qui transforme un arbre en expression algébrique totalement parenthésée :

```
#let rec arbre_en_EATP = function
        Feuille x -> [Nombre x]
     |  Noeud1 (fonction , branche ) -> (Fct fonction)::(arbre_en_EATP branche)
     |  Noeud2 (branche_g, operateur, branche_d) ->
             (Delim `(`::(arbre_en_EATP branche_g)) @
             ((Op operateur)::(arbre_en_EATP branche_d)) @
             [Delim `)`];;
arbre_en_EATP : arbre_XOF -> EATP_elt list = <fun>
```

avec un essai :

```
#let a=
    Noeud1 ("sin", Noeud2( Noeud2( Feuille 23, "*", Feuille 2), "+", Feuille 56))
 in arbre_en_EATP a;;
- : EATP_elt list =
 [Fct "sin"; Delim `(`; Delim `(`; Nombre 23; Op "*"; Nombre 2; Delim `)`;
  Op "+"; Nombre 56; Delim `)`]
```

Procédure d'impression d'une EATP :

```
#let rec print_EATP= function
                []  -> ()
               | (Nombre x)::reste -> print_int x; print_EATP reste
               | (Op s)::reste -> print_string s; print_EATP reste
               | (Fct s)::reste -> print_string s; print_EATP reste
               | (Delim c)::reste -> print_char c; print_EATP reste;;
print_EATP : EATP_elt list -> unit = <fun>

#let a=
   Noeud1 ("sin", Noeud2( Noeud2( Feuille 23, "*", Feuille 2), "+", Feuille 56))
 in print_EATP (arbre_en_EATP a); print_newline();;
sin((23*2)+56)
- : unit = ()
```

Examiner les trois cas possibles.

Cas où l'expression est réduite à un élément de X ou commence par une fonction de $\mathcal{F}$:

```
let rec EATP_en_arbre = function
              [Nombre x] -> Feuille x
          |   (Fct fonction)::reste -> Noeud1 (fonction, EATP_en_arbre reste)
          |   (Delim `(`)::reste -> ????
          |   _ -> invalid_arg "ce n'est pas une EATP";;
```

Déterminer, dans une EATP commençant par une parenthèse ouvrante, l'opérande gauche, l'opérateur et l'opérande droite (sachant qu'elle se terminera de toute façon par une parenthèse fermante) :

Remarquons qu'aucun préfixe gauche strict d'une EATP (c'est à dire une sous-expression algébrique formée des p premiers éléments de l'expression totale) n'est lui-même une EATP :

- c'est clair si l'expression est réduite à un élément x de X, le seul préfixe strict étant la suite vide qui n'est pas une EATP
- si l'expression totale est de la forme $f\ E$ avec $f \in \mathcal{F}$, alors un préfixe strict est de la forme $f\ E'$ où E' est un préfixe strict de E, qui, par induction, n'est donc pas une EATP ; $f\ E'$ n'est donc pas une EATP
- si l'expression totale est de la forme $(\ E_1\ f\ E_2\ )$, un préfixe strict ne contient pas la parenthèse fermante et contient donc plus de parenthèses ouvrantes que de parenthèses fermantes ce qui l'empêche d'être une EATP

Dans une expression ($E_1\ f\ E_2$), E_1 est le premier préfixe de la suite $E_1\ f\ E_2$) qui soit une EATP.

Pour construire l'arbre associé à l'expression ($E_1\ f\ E_2$), il suffit donc d'essayer de construire l'arbre associé à la suite $E_1\ f\ E_2$) ; lorsque la construction s'arrête (sur une erreur puisque la suite n'est pas une EATP), on a obtenu l'arbre associé à E_1.

Retourner par un moyen quelconque la suite $f\ E_2$) ; de cette manière, la racine de l'arbre sera étiquetée par f, sa branche gauche sera l'arbre associée à E_1 précédemment calculé

La branche droite sera l'arbre associé au premier préfixe de l'expression restante qui soit une EATP, que l'on peut déterminer de la même manière en tentant de construire son arbre.

Une fois supprimés f et E_2 il ne doit rester que la parenthèse fermante, que l'on peut supprimer sans regret.

Renoncer à ne considérer que des EATP syntaxiquement correctes et considérer des suites quelconques : le but de la fonction sera

- extraire de la suite le premier préfixe qui soit une EATP,
- construire l'arbre de cette EATP
- retourner simultanément cet arbre et la fin de la suite.

Retourner la fin de la suite. Plusieurs méthodes :

1. stocker la suite dans une référence sur une liste, liste qui sera raccourcie au fur et à mesure que ses premiers éléments seront absorbés dans l'EATP préfixe (peu récursif)
2. stocker la suite dans un tableau et d'établir un indice dénotant le premier élément de la suite non encore examiné (peu récursif)
3. faire retourner à la fonction non seulement l'arbre construit, mais également le reste de la chaîne à examiner, ceci par l'intermédiaire d'un couple
4. utiliser une structure de donnée particulière, analogue à une liste qui se dévorerait elle-même au fur et à mesure qu'on en filtre la tête : c'est la structure de *flux*

Aménager notre fonction `EATP_en_arbre` pour qu'elle accepte en entrée n'importe quelle liste formée d'éléments du type `EATP_elt` et qu'elle retourne un couple formé de l'arbre de l'EATP préfixe et du reste de la liste non encore considéré :

```
#let rec analyse_EATP = function
          |   (Nombre x)::reste -> (Feuille x),reste
          |   (Fct fonction)::reste ->
                 let (arbre,reste1) = analyse_EATP reste in
                                  (Noeud1 (fonction, arbre),reste1)
          |   (Delim `(`)::reste ->
                 begin
                    match analyse_EATP reste with
                      | (arbre_g,(Op operateur)::reste1) ->
                           begin
                             match analyse_EATP reste1 with
                               | (arbre_d,(Delim `)`)::reste2) ->
                                   (Noeud2(arbre_g,operateur,arbre_d),reste2)
                               |  _ -> invalid_arg "erreur de syntaxe"
                           end
                      | _ -> invalid_arg "erreur de syntaxe"
                  end
          |   _ -> invalid_arg "erreur de syntaxe";;
analyse_EATP : EATP_elt list -> arbre_XOF * EATP_elt list = <fun>
```

2 Expressions algébriques avec priorités des opérateurs

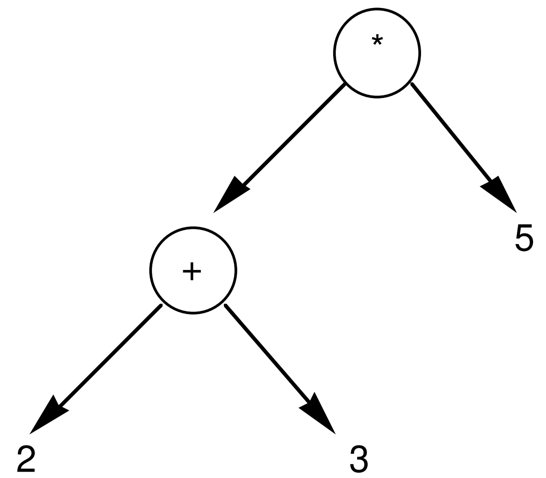
2.1 Priorités d'opérateurs

Syntaxe utilisée pour les expressions algébriques totalement parenthésées : simple, non abigue, lourde, peu lisible.

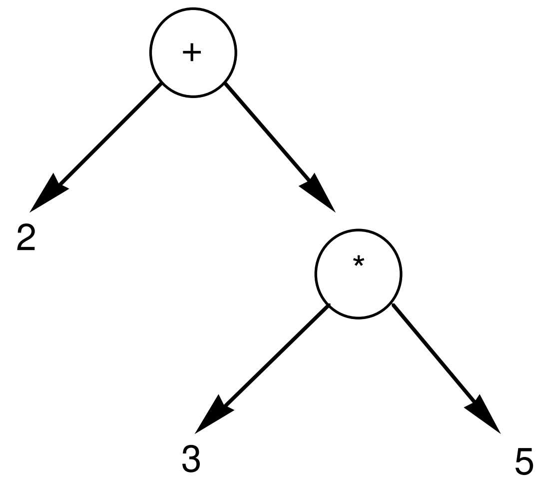
Trop de parenthèses.

Pouvons-nous nous passer de ces parenthèses ? Evidemment non, à moins que nous ne nous donnions de nouvelles règles de construction de l'arbre de l'expression algébrique (et donc de son évaluation éventuelle).

Considérons par exemple une expression comme $2 + 3 * 5$. Une calculatrice quatre opérations de bas de gamme, qui effectue les opérations au fur et à mesure que celles-ci se présentent, donnera comme résultat 25 correspondant au parenthésage implicite $(2 + 3) * 5$ et donc à l'arbre



Par contre, un mathématicien ou une calculatrice scientifique donnera la priorité à la multiplication sur l'addition et fournira comme résultat 17, correspondant au parenthésage implicite $2 + (3 * 5)$ et donc à l'arbre



Opérateur de puissance (que l'on symbolise habituellement pas l'accent circonflexe) :

doit-on interpréter l'expression algébrique $2 \wedge 3 \wedge 2$

- comme valant $512 = 2 \wedge 9 = 2 \wedge (3 \wedge 2)$ (associatif à droite)
- comme valant $64 = 8 \wedge 2 = (2 \wedge 3) \wedge 2$ (associatif à gauche)

Disposer

- de l'ensemble X,
- d'un ensemble d'opérateurs $\mathcal{O}$
- d'un ensemble de fonctions $\mathcal{F}$,
- d'une application $\pi : \mathcal{O} \rightarrow \mathbf{N}$ qui décrit la priorité de l'opérateur
- on notera $\mathcal{P}$ l'ensemble à deux éléments constitué de la parenthèse ouvrante et de la parenthèse fermante.

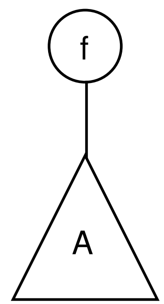
Définition 2.1 *On appelle expression algébrique le sous-ensemble des suites finies d'éléments de* $X \cup \mathcal{O} \cup \mathcal{F} \cup \mathcal{P}$ *défini récursivement par*

- *si* $x \in X$, *alors la suite à un élément* x *est une expression algébrique irréductible*
- *si* E *est une expression algébrique irréductible et* $f \in \mathcal{F}$, *alors la suite* $f\ E$ *est une expression algébrique irréductible*
- *si* E *est une expression algébrique, alors* $(\ E\ )$ *est une expression algébrique irréductible*
- *si* $E_1, \ldots, E_n$ *sont des expressions algébriques irréductibles* $(n \geq 1)$ *et* $f_1, \ldots, f_{n-1}$ *appartiennent à* $\mathcal{O}$, *alors la suite* $E_1\ f_1\ \ldots\ f_{n-1}\ E_n$ *est encore une expression algébrique*

2.2 Arbre d'une expression algébrique

Nous allons alors définir l'arbre associé à une expression algébrique de la manière suivante :

- si x est un élément de X, l'arbre associé à l'expression algébrique x est l'arbre sans noeud dont l'unique feuille est étiquetée par x
- si E est une expression algébrique irréductible dont l'arbre associé est A et si f est un élément de $\mathcal{F}$, alors l'arbre associé à l'expression algébrique $f\ E$ est



- si E est une expression algébrique, l'arbre associé à l'expression algébrique irréductible (E) est l'arbre associé à E
- si $E_1, \ldots, E_n$ sont des expressions algébriques irréductibles $(n \geq 1)$ et $f_1, \ldots, f_{n-1}$ appartiennent à $\mathcal{O}$, notons $m = \min(\pi(f_i))$ et $i = \max\{k \mid \pi(f_k) = m\}$; si A_1 est l'arbre associé à l'expression algébrique ($E_1\ f_1\ \ldots\ f_{i-1}\ E_i$) et A_2 est l'arbre associé à l'expression algébrique ($E_{i+1}\ f_{i+1}\ \ldots\ f_{n-1}\ E_n$), alors l'arbre associé à l'expression algébrique $E_1\ f_1\ \ldots\ f_{n-1}\ E_n$ est l'arbre dont la racine est étiquetée par f_i, dont la branche gauche est A_1 et la branche droite

A_2 ; ceci revient à identifier l'expression $E_1\ f_1\ \ldots\ f_{n-1}\ E_n$ et l'expression

$$(\ E_1\ f_1\ \ldots\ f_{i-1}\ E_i\ )\ f_i\ (\ E_{i+1}\ f_{i+1}\ \ldots\ f_{n-1}\ E_n\ )$$

Remarque On aurait tout aussi pu prendre $i = \min\{k \mid \pi(f_k) = m\}$;

Prendre le plus grand i revient à imposer que les opérateurs soient associatifs à gauche

Prendre le plus petit i revient à imposer que les opérateurs soient associatifs à droite.

La convention habituelle sur les opérateurs arithmétiques qui nous intéressent tout particulièrement est l'associativité à gauche (sauf parfois pour l'opérateur puissance).

Une suite $E_i\ f_i\ \dots\ f_{j-1}\ E_j$ est de niveau n si chacun des opérateurs f_k, $i \leq k \leq j-1$ a une priorité au moins égale à n

Si N est la plus grande priorité des opérateurs de $\mathcal{O}$, nous considérerons que les expressions algébriques irréductibles sont de niveau $N+1$

Expression algébrique de niveau m :

- soit une expression algébrique de niveau $m+1$ (dans le cas où elle ne contient aucun opérateur de niveau m)
- soit la juxtaposition d'une expression algébrique de niveau $m+1$, d'un opérateur f de priorité m et d'une expression algébrique de niveau m

Grammaire : (EA : expression algébrique)

- expression algébrique ::= EA_1
- EA_m ::= EA_{m+1} | $\mathrm{EA}_{m+1}\ f_m\ \mathrm{EA}_m$

Grammaire factorisée à gauche :(EA : partie droite d'expression algébrique)

- expression algébrique ::= EA_1
- EA_m ::= $\mathrm{EA}_{m+1}\ \mathrm{PDEA}_m$
- PDEA_m ::= Vide | $f_m\ \mathrm{EA}_m$

Fonction d'explorationd'une expression algébrique au niveau $m \leq N$:

- explorer l'expression au niveau $m+1$ en retournant le reste de la suite non encore exploré
- si le terme suivant n'est pas un opérateur de priorité m, c'est terminé
- sinon explorer la suite qui succède à l'opérateur au niveau m

Explorer les expressions algébriques irréductibles de niveau $N+1$

- immédiat pour les expressions réduites à un élément de X
- pour (E) ôter la parenthèse ouvrante, explorer E au niveau 1, vérifier qu'à la fin on a bien une parenthèse fermante
- pour f E où f est une fonction, ôter f et explorer le reste au niveau $N+1$ (puisque E doit être irréductible).

Parcours par niveaux d'une expression algébrique

```
type EA_elt = Nombre of float | Op of string | Fct of string | Delim of string;;

let priorite = function
            |  Op "+" -> 1
            |  Op "-" -> 1
            |  Op "*" -> 2
            |  Op "/" -> 2
            |  Op "^" -> 3
            |  _ -> invalid_arg "opérateur inconnu";;
let max_prio = 3;;

let rec explore_gauche niveau = function
       | []  -> []
       | liste ->
          if niveau <= max_prio then
            let reste = explore_gauche (niveau+1) liste
              in explore_droite niveau reste
           else
            explore_irreductible liste
and explore_irreductible = function
       | (Nombre x)::reste -> reste
```

```
       | (Delim `(`)::reste ->
              begin
               match explore_gauche 1 reste with
                 | (Delim `)`)::nouveau_reste -> nouveau_reste
                 | _ -> invalid_arg "parenthèse manquante"
              end
       | (Fct f)::reste -> explore_irreductible reste
       | _ -> invalid_arg "erreur de syntaxe" (* à suivre *)
and explore_droite niveau = function
       | ((Op f)::reste) as liste ->
                 if (priorite (Op f))=niveau
                    then explore_gauche niveau reste
                    else liste
       | liste -> liste;;
```

Impression d'une expression algébrique

```
#let rec imprime_gauche niveau = function
      | []  -> []
      | liste ->
        if niveau <= max_prio then
         let reste = imprime_gauche (niveau+1) liste
           in imprime_droite niveau reste
        else
         imprime_irreductible liste
   and imprime_irreductible = function
      | (Nombre x)::reste -> print_int x; reste
      | (Delim `(`)::reste ->
             begin
              print_char `(`;
              match imprime_gauche 1 reste with
                | (Delim `)`)::nouveau_reste ->
                       print_char `)`; nouveau_reste
                | _ -> invalid_arg "parenthèse manquante"
             end
      | (Fct f)::reste -> print_string f; imprime_irreductible reste
      | _ -> invalid_arg "erreur de syntaxe"
   and imprime_droite niveau = function
```

```
| ((Op f)::reste) as liste ->
          if priorite (Op f) = niveau
             then
               begin
                print_string f;
                imprime_gauche niveau reste
               end
             else liste
| liste -> liste;;
```

Construction de l'arbre (associatif à droite)

```
type arbre_XOF =
             Feuille of int
          |  Noeud1 of string * arbre_XOF
          |  Noeud2 of arbre_XOF * string * arbre_XOF;;


#let rec construit_gauche niveau liste =
         if niveau <= max_prio then
          match construit_gauche (niveau+1) liste with
            (arbre,reste) -> construit_droite niveau reste arbre
         else
          construit_irreductible liste
   and construit_irreductible = function
       | (Nombre x)::reste -> ((Feuille x), reste)
       | (Delim `(`)::reste ->
              begin
               match construit_gauche 1 reste with
                 | ( arbre,(Delim `)`)::nouveau_reste ) ->
                             (arbre , nouveau_reste)
                 | _ -> invalid_arg "parenthèse manquante"
              end
```

```
    | (Fct f)::reste -> begin
                match construit_irreductible reste with
                  (arbre, nouveau_reste) ->
                   ((Noeud1 (f,arbre)),nouveau_reste)
              end
    | _ -> invalid_arg "erreur de syntaxe"
and construit_droite niveau liste arbre_g = match liste with
    | ((Op f)::reste) ->
        if (priorite (Op f))=niveau
           then
            begin
              match construit_gauche niveau reste with
               (arbre_d,nouveau_reste) ->
                 (Noeud2(arbre_g,f,arbre_d),nouveau_reste)
            end
           else (arbre_g,liste)
    | _ -> (arbre_g,liste);;
```

Construction de l'arbre (associatif à gauche) : c'est la fonction d'analyse à droite qui construit l'arbre

```
let rec analyse_gauche niveau expression =
       if niveau <= max_prio then
          let  (arbre_g,reste) = analyse_gauche (niveau+1) expression in
              analyse_droite niveau reste arbre_g
       else
         analyse_irreductible expression
and analyse_irreductible = function
      | (Nombre x)::reste -> (Feuille x, reste)
      | (Fct f)::reste ->
           let (arbre,nouveau_reste) = analyse_irreductible reste
                 in (Noeud1 (f,arbre),nouveau_reste)
      | (Delim '('):reste ->
         begin
          match analyse_gauche 1 reste with
           | (arbre,(Delim ')'::nouveau_reste)) ->
                             (arbre,nouveau_reste)
           | _ -> invalid_arg "il manque une parenthèse fermante"
         end
      | _ -> invalid_arg "erreur de syntaxe"
and analyse_droite niveau expression arbre_gauche =
```

```
match expression with
 | (Op f)::reste when priorite (Op f) = niveau ->
       (* transmet un arbre partiel *)
      let (arbre_droit,reste1) = analyse_gauche (niveau+1) reste in
          analyse_droite niveau reste1
              (Noeud2(arbre_gauche,f,arbre_droit))
 | _ -> (arbre_gauche, expression);;
```

2.3 Un analyseur syntaxique arithmétique

Ensemble X : les entiers et les chaînes de caractères (ce qui permettra de travailler avec des variables formelles)

Fonctions toutes les chaînes de caractère.

```
type EATP_elt =
              Nombre of int
        |     Variable of string
        |     Plus | Moins | Mult | Div | Exp
        |     Fct of string
        |     Delim of char;;
```

Arbres de syntaxe :

```
type arbre_XOF =
             |  Feuille_entier of int
             |  Feuille_var of string
             |  Applique of string*arbre_XOF
             |  Somme of arbre_XOF*arbre_XOF
             |  Diff of arbre_XOF*arbre_XOF
             |  Prod of arbre_XOF*arbre_XOF
             |  Quot of arbre_XOF*arbre_XOF
             |  Puiss of arbre_XOF*arbre_XOF;;
```

- Analyse au niveau 1 effectuée par des fonctions comportant le suffixe PlusMoins,
- Analyse au niveau 2 effectuée par des fonctions comportant le suffixe MultDiv,
- Analyse au niveau 3 effectuée par des fonctions comportant le suffixe Puiss.

```
#let rec analyse_gauche_PlusMoins expression =
       let (arbre_g,reste) = analyse_gauche_MultDiv expression in
               analyse_droite_PlusMoins reste arbre_g
and analyse_droite_PlusMoins expression arbre_gauche =
       match expression with
        | Plus::reste ->
             let (arbre_droit,reste1) =  analyse_gauche_MultDiv reste in
                  analyse_droite_PlusMoins reste1
                                        (Somme(arbre_gauche,arbre_droit))
        | Moins::reste ->
            let (arbre_droit,reste1) = analyse_gauche_MultDiv reste in
                  analyse_droite_PlusMoins reste1
                                         (Diff(arbre_gauche,arbre_droit))
        | _ -> (arbre_gauche, expression)
     (* à suivre *)

    (* suite *)
and analyse_gauche_MultDiv expression =
      let (arbre_g,reste) = analyse_gauche_Puiss expression in
               analyse_droite_MultDiv reste arbre_g
and analyse_droite_MultDiv expression arbre_gauche =
       match expression with
        | Mult::reste ->
```

```
            let (arbre_droit,reste1) =  analyse_gauche_Puiss reste in
                  analyse_droite_MultDiv reste1 (Prod(arbre_gauche,arbre_droit))
        | Div::reste ->
           let (arbre_droit,reste1) = analyse_gauche_Puiss reste in
                  analyse_droite_MultDiv reste1 (Quot(arbre_gauche,arbre_droit))
        | _ -> (arbre_gauche, expression)
and analyse_gauche_Puiss expression =
       let (arbre_g,reste) = analyse_irreductible expression in
              analyse_droite_Puiss reste arbre_g
and analyse_droite_Puiss expression arbre_gauche =
       match expression with
        | Exp::reste ->
            let (arbre_droit,reste1) = analyse_irreductible reste in
                  analyse_droite_Puiss reste1 (Puiss(arbre_gauche,arbre_droit))
        | _ -> (arbre_gauche, expression)
and analyse_irreductible = function
      | (Nombre x)::reste -> ((Feuille_entier x), reste)
      | (Variable s)::reste -> ((Feuille_var s),reste)
      | (Fct f)::reste ->
            let (arbre,nouveau_reste) = analyse_irreductible reste in
                           (Applique (f,arbre),nouveau_reste)
      | (Delim '('):reste ->
```

```
          begin
            match analyse_gauche_PlusMoins reste with
             |  (arbre,(Delim ‘)‘::nouveau_reste)) -> (arbre,nouveau_reste)
             |  _ -> invalid_arg "il manque une parenthèse fermante"
          end
      | _ -> invalid_arg "erreur de syntaxe";;
```

La fonction d'analyse d'une expression s'écrit alors

```
#let analyse expression =
          match analyse_gauche_PlusMoins expression with
                 | (arbre,[]) -> arbre
                 | _ -> invalid_arg "erreur de syntaxe";;
analyse : EATP_elt list -> arbre_XOF = <fun>
```

avec un essai sur l'expression $1 + x - \sin(y^3)$:

```
#analyse [Delim ‘(‘; Nombre 1; Plus; Variable "x"; Moins; Fct "sin";
       Delim ‘(‘; Variable "y"; Exp; Nombre 3; Delim ‘)‘; Delim ‘)‘];;
- : arbre_XOF =
 Diff
  (Somme (Feuille_entier 1, Feuille_var "x"),
   Applique ("sin", Puiss (Feuille_var "y", Feuille_entier 3)))
```

2.4 Un analyseur syntaxique logique

Presque tous les opérateurs logiques binaires sont associatifs : le OU et le ET de manière évidente, le OUX (ou exclusif) car il correspond à l'addition dans $\mathbf{Z}/2\mathbf{Z}$, l'EQUIV (équivalence logique) de manière évidente puisque $(a \iff b) \iff c$ prend la valeur vraie si et seulement si un nombre impair de variables prennent la valeur vraie. Le seul opérateur binaire non associatif est l'implication, mais il est bien risqué d'écrire sans parenthèses $a \Rightarrow b \Rightarrow c$.

Garder sans risque l'associativité à droite.

Types de base :

```
#type logique_elt =
                Booléen of bool
             |  Var of string
             |  Non | Ou | Et | Oux | Impl | Ssi
             |  Delim of char;;

type arbre_logique = Vide
             |  Feuille_bool of bool
             |  Feuille_var of string
             |  Neg of arbre_logique
             |  Disj of arbre_logique*arbre_logique (* disjonction *)
             |  Conj of arbre_logique*arbre_logique  (* conjonction *)
             |  DisjEx of arbre_logique*arbre_logique (* disjonction exclusive *)
             |  Implication of arbre_logique*arbre_logique (* implication *)
             |  Equiv of arbre_logique*arbre_logique;; (* équivalence logique *)
Type logique_elt defined.
#Type arbre_logique defined.
```

Priorités habituelles par ordre croissant : l'équivalence, l'implication, le OU exclusif, le OU et enfin le ET. Ceci nous conduit aux fonctions suivantes :

```
#let rec construit_gauche_Ssi liste =
         let (arbre,reste) = construit_gauche_Impl liste in
            construit_droite_Ssi reste arbre
and construit_droite_Ssi liste arbre_g = match liste with
       | Ssi::reste ->
           let (arbre_d,nouveau_reste) = construit_gauche_Ssi reste in
                    (Equiv(arbre_g,arbre_d),nouveau_reste)
       | _ -> (arbre_g,liste)
and construit_gauche_Impl liste =
         let (arbre,reste) = construit_gauche_Oux liste in
            construit_droite_Impl reste arbre
and construit_droite_Impl liste arbre_g = match liste with
       | Impl::reste ->
           let (arbre_d,nouveau_reste) = construit_gauche_Impl reste in
                  (Implication(arbre_g,arbre_d),nouveau_reste)
       | _ -> (arbre_g,liste)
and construit_gauche_Oux liste =
         match construit_gauche_Ou liste with
            (arbre,reste) -> construit_droite_Oux reste arbre
and construit_droite_Oux liste arbre_g = match liste with
```

```
      | Oux::reste ->
          let (arbre_d,nouveau_reste) = construit_gauche_Oux reste in
                (DisjEx(arbre_g,arbre_d),nouveau_reste)
      | _ -> (arbre_g,liste)
and construit_gauche_Ou liste =
        let (arbre,reste) = construit_gauche_Et liste in
             construit_droite_Ou reste arbre
and construit_droite_Ou liste arbre_g = match liste with
      | Ou::reste ->
          let (arbre_d,nouveau_reste) = construit_gauche_Ou reste in
                (Disj(arbre_g,arbre_d),nouveau_reste)
      | _ -> (arbre_g,liste)
and construit_gauche_Et liste =
        let (arbre,reste) = construit_irreductible liste in
              construit_droite_Et reste arbre
and construit_droite_Et liste arbre_g = match liste with
      | Et::reste ->
          let (arbre_d,nouveau_reste) = construit_gauche_Et reste in
                 (Conj(arbre_g,arbre_d),nouveau_reste)
      | _ -> (arbre_g,liste)
and construit_irreductible = function
      | (Booléen x)::reste -> ((Feuille_bool x), reste)
```

```
     | (Var s)::reste -> ((Feuille_var s), reste)
     | (Delim `(`)::reste ->
            begin
             match construit_gauche_Ssi reste with
               | ( arbre,(Delim `)`)::nouveau_reste ) ->
                           (arbre , nouveau_reste)
               | _ -> invalid_arg "parenthèse manquante"
            end
     | Non::reste ->
         let (arbre, nouveau_reste) = construit_irreductible reste in
                          (Neg(arbre),nouveau_reste)
     | _ -> invalid_arg "erreur de syntaxe";;
```

Fonction générale d'analyse syntaxique d'une expression logique :

```
#let analyse_logique expression =
            match construit_gauche_Ssi expression with
                 | (arbre,[]) -> arbre
                 | _ -> invalid_arg "erreur de syntaxe";;
analyse_logique : logique_elt list -> arbre_logique = <fun>
```

Essai pour l'expression logique $a \vee b \iff (a \vee c) \wedge b$:

```
#let expr = [ Var "a"; Ou; Var "b"; Ssi;
              Delim `(`; Var "a"; Ou; Var "c"; Delim `)`; Et; Var "b"]
       in analyse_logique expr;;
- : arbre_logique =
 Equiv
  (Disj (Feuille_var "a", Feuille_var "b"),
   Conj (Disj (Feuille_var "a", Feuille_var "c"), Feuille_var "b"))
```

3 Manipulations formelles d'expressions

3.1 Simplification

Règles de simplifications :

- $x + 0 = 0 + x = x$
- $x - 0 = x$
- $x * 0 = 0 * x = 0$ et $x * 1 = 1 * x = x$
- $x/1 = x$ et $0/x = 0$ (où nous poserons par convention que $0/0 = 0$)
- $x^1 = x$, $1^x = 1$, $x^0 = 1$ (où nous poserons par convention que $0^0 = 1$) et $0^x = 0$ si $x \neq 0$ (convention justifiée au moins si $x > 0$).

Très faciles à faire sur l'arbre de l'expression :

- une feuille est déjà simplifiée
- pour simplifier un noeud unaire, on se contente de simplifier la branche (on pourrait faire mieux en appliquant des règles comme $\sin(0) = 0$ ou $\cos(0) = 1$)
- pour simplifier un noeud binaire, on commence par simplifier les deux branches puis on applique les règles de simplification relatives à l'opérateur correspondant
- on effectue au passage les opérations licites sur les entiers : somme, différence, produit

```
#let rec simplifie =  function
        |  Vide -> Vide
        |  Somme (arbre_g,arbre_d) ->
                simplifie_somme (Somme(simplifie arbre_g,simplifie arbre_d))
        |  Diff (arbre_g,arbre_d) ->
                simplifie_diff (Diff(simplifie arbre_g,simplifie arbre_d))
        |  Prod (arbre_g,arbre_d) ->
                simplifie_prod (Prod(simplifie arbre_g,simplifie arbre_d))
        |  Quot (arbre_g,arbre_d) ->
                simplifie_quot (Quot(simplifie arbre_g,simplifie arbre_d))
        |  Puiss (arbre_g,arbre_d) ->
                simplifie_puiss (Puiss(simplifie arbre_g,simplifie arbre_d))
        |  Applique(f,arbre) -> Applique(f,simplifie arbre)
                (* à modifier si l'on souhaite des règles spécifiques *)
        |  arbre -> arbre
and simplifie_somme = function
        |  Somme (arbre_g,Feuille_entier 0) -> arbre_g
        |  Somme (Feuille_entier 0, arbre_d) -> arbre_d
        |  Somme (Feuille_entier x,Feuille_entier y) ->
                                                 Feuille_entier (x+y)
        |  arbre -> arbre
and simplifie_diff = function
```

```
         |  Diff (arbre_g,Feuille_entier 0) -> arbre_g
         |  Diff (Feuille_entier x,Feuille_entier y) ->
                                 Feuille_entier (x-y)
         |  arbre -> arbre
and simplifie_prod = function
         |  Prod (arbre_g,Feuille_entier 1) -> arbre_g
         |  Prod (Feuille_entier 1, arbre_d) -> arbre_d
         |  Prod (arbre_g,Feuille_entier 0) -> Feuille_entier 0
         |  Prod (Feuille_entier 0, arbre_d) -> Feuille_entier 0
         |  Prod (Feuille_entier x,Feuille_entier y) ->
                                        Feuille_entier (x * y)
         |  arbre -> arbre
and simplifie_quot = function
         |  Quot (arbre_g,Feuille_entier 1) -> arbre_g
         |  Quot (Feuille_entier 0, arbre_d) -> Feuille_entier 0
         |  arbre -> arbre
     (* à suivre *)

   (* suite *)
and simplifie_puiss = function
         |  Puiss (arbre_g,Feuille_entier 1) -> arbre_g
         |  Puiss (Feuille_entier 1, arbre_d) -> Feuille_entier 1
         |  arbre -> arbre;;
```

3.2 Evaluation d'une expression

Donner des valeurs aux variables. Comme pour le dictionnaire de l'analyse lexicale, nous supposerons que les valeurs des variables sont transmises dans un **environnement** qui consiste en une liste de couples dont le premier terme est le nom de la variable et le second est un nombre réel.

Disposer d'une liste de fonctions Caml qui réalisent l'évaluation en nombre flottants des fonctions abstraites que nous avons définies : liste de couples dont le premier terme est la chaîne de caractères représentant la fonction abstraite et dont le second terme est la fonction Caml correspondante, par exemple

```
["sin",sin ; "cos",cos ; "log", ln ; ... ]
```

Fonction d'évaluation :

```
let evaluation liste_fonctions environnement =
     let rec evalue = function
          |  Feuille_entier n -> float_of_int n
          |  Feuille_var s ->
                         begin
                           try assoc s environnement with
                             Not_found ->
                                invalid_arg ("variable "^s^" sans affectation")
                         end
          |  Applique(f,branche) ->
                         begin
                           try
                              let f_Caml= assoc f liste_fonctions in
                                          f_Caml (evalue branche)
                             with Not_found ->
                                 invalid_arg ("fonction "^f^" non évaluable")
                           end
          |  Somme(branche_g,branche_d) ->
                      (evalue branche_g) +. (evalue branche_d)
          |  Diff(branche_g,branche_d) ->
                      (evalue branche_g) -. (evalue branche_d)
```

```
             |  Prod(branche_g,branche_d) ->
                          (evalue branche_g) *. (evalue branche_d)
             |  Quot(branche_g,branche_d) ->
                          (evalue branche_g) /. (evalue branche_d)
             |  Puiss(branche_g,branche_d) ->
                          power (evalue branche_g) (evalue branche_d)
             |  _ -> invalid_arg "évaluation impossible"
 in evalue;;
```

Essai :

```
#let arbre = analyse(lexeur ["sin",Fct "sin"] "x^2+2*x-sin(3^y)");;
arbre : arbre_XOF =
 Diff
  (Somme
    (Puiss (Feuille_var "x", Feuille_entier 2),
     Prod (Feuille_entier 2, Feuille_var "x")),
   Applique ("sin", Puiss (Feuille_entier 3, Feuille_var "y")))
#evaluation [] ["x",1.2 ; "y", 2.2] arbre;;
Uncaught exception: Invalid_argument "fonction sin non évaluable"
#evaluation ["moins",(function x-> -x); "sin",sin] ["x",1.2 ; "y", 2.2] arbre;;
- : float = 4.8167616648
```

3.3 Dérivation formelle

Dériver par induction structurelle une expression algébrique E dépendant d'un certain nombre de variables (chaînes de caractères).

Soit x l'une de ces variables. Calculer la dérivée de E par rapport à la variable x.

Soit donc E' la dérivée de E par rapport à x. Nous noterons A l'arbre de E et A' l'arbre de E'.

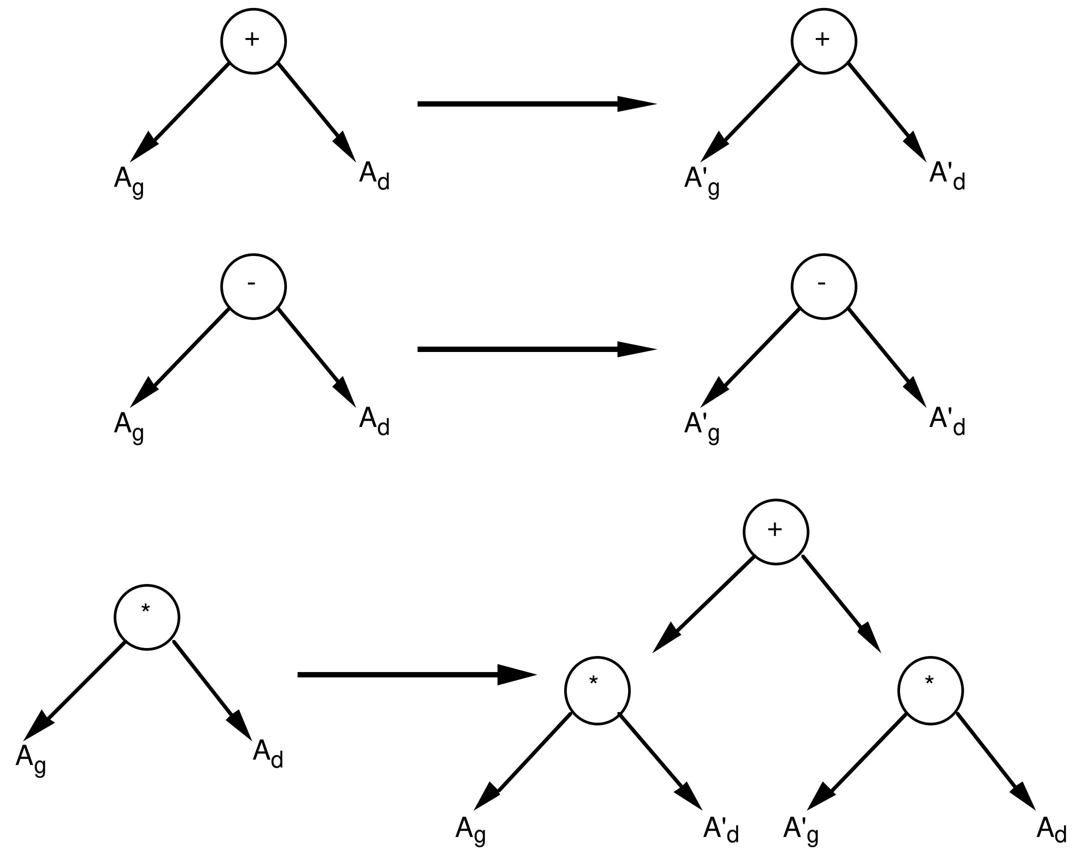
Si A est réduit à une feuille, c'est que E est soit un nombre soit une variable. Alors, en appliquant les règles évidentes $\frac{\partial}{\partial x}n = 0$, $\frac{\partial}{\partial x}y = 0$, $\frac{\partial}{\partial x}x = 1$ on obtient :

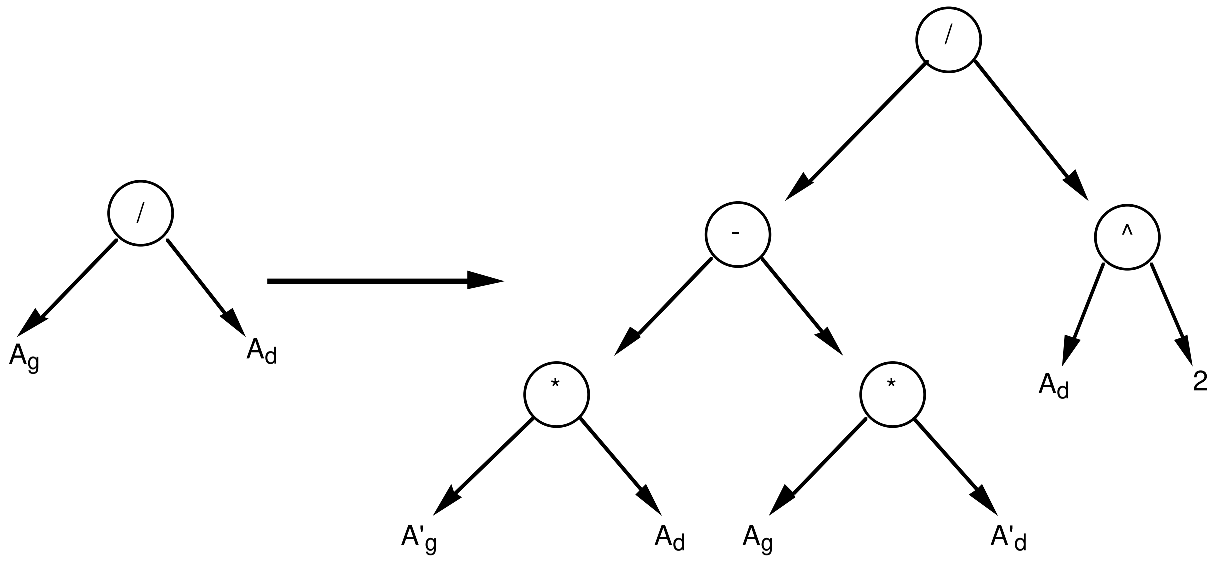
- si E est soit un nombre n, soit une variable y différente de x, A' est l'arbre réduit à la feuille numérique 0
- si E est la variable x, A' est l'arbre réduit à la feuille numérique 1.

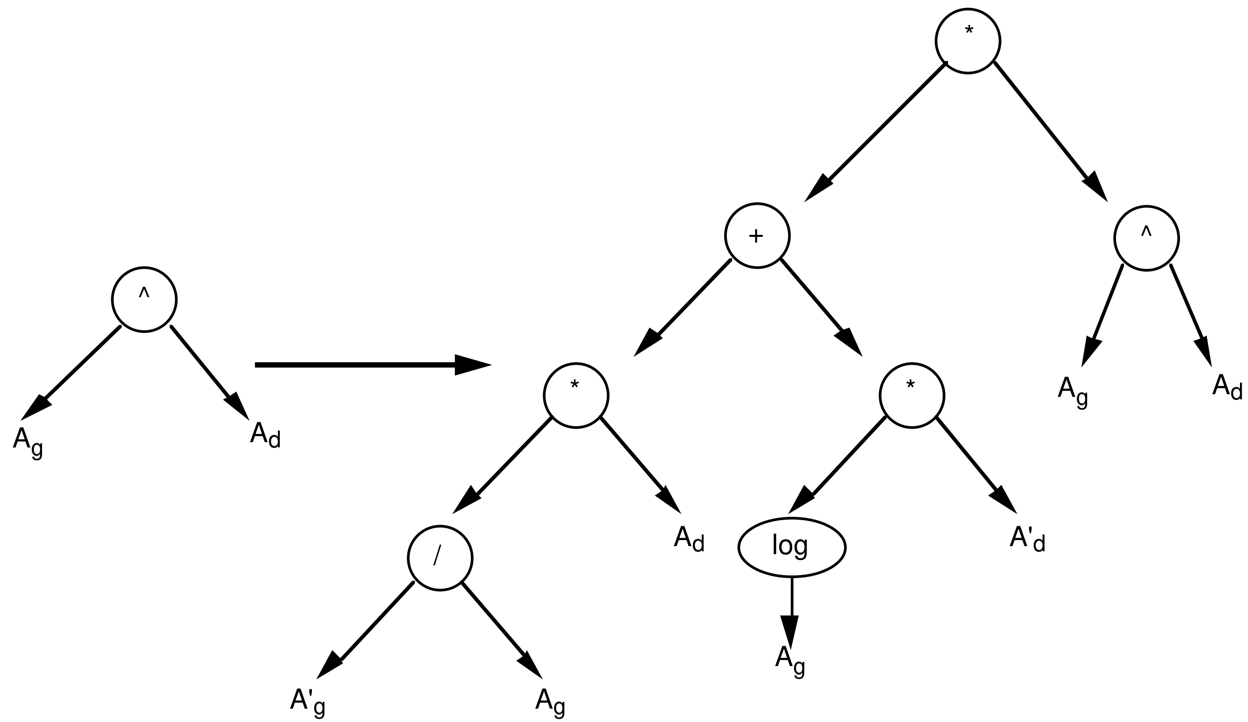
Si la racine de A est l'un des opérateurs $+, -, *, /, \hat{}$, les règles de dérivation

- $(u+v)' = u' + v'$
- $(u-v)' = u' - v'$
- $(vu)' = u'v + uv'$
- $(u/v)' = (u'v - vu')/v^2$
- $(u\hat{}v)' = \big((u'/u)v + \log(u)v'\big)(u\hat{}v)$

permettent d'énoncer les règles suivantes pour la construction de l'arbre de l'expression dérivée, en notant A_g et A_d les branches gauche et droite issues de la racine de A (c'est à dire les arbres de u et v) ainsi que A'_g et A'_d les *dérivées* de ces branches (c'est à dire les arbres de u' et v') :


+
A_g
A_d
+
A'_g
A'_d
-
A_g
A_d
-
A'_g
A'_d
*
A_g
A_d
+
*
A_g
A'_d
*
A'_g
A_d

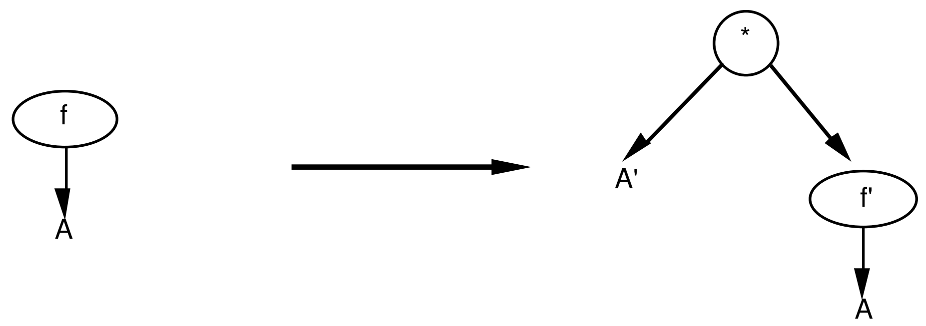

/
A_g
A_d
/
-
^
*
*
A_d
2
A'_g
A_d
A_g
A'_d


^
Ag
Ad
*
+
^
Ag
Ad
*
*
/
Ad
log
A'd
A'g
Ag
Ag

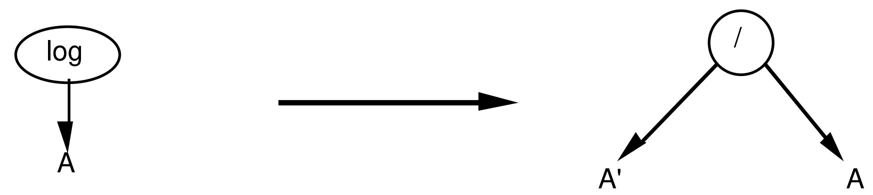
Si la racine de l'arbre est une fonction f dont la dérivée f' appartient encore à $\mathcal{F}$, la formule

$$f(u)' = u'f'(u)$$

se traduit simplement sur les arbres par



Si la dérivée de la fonction f n'est pas dans $\mathcal{F}$, il faut travailler au cas par cas ; par exemple dans le cas de la fonction logarithme, la formule $(\log(u))' = u'/u$ se traduit par



Traduire tout ceci en Caml à l'aide de la fonction suivante :

```
#let rec derive arbre x =
          match arbre with
                |  Vide -> Vide
                |  Feuille_entier n -> Feuille_entier 0
                |  Feuille_var s when s=x -> Feuille_entier 1
                |  Feuille_var _ -> Feuille_entier 0
                |  Applique ("moins", branche) ->
                           Applique("moins",(derive branche x))
                |  Applique ("exp", branche) ->
                           Prod((derive branche x), Applique("exp",branche))
                |  Applique ("log", branche) ->
                           Quot((derive branche x),branche)
                |  Applique ("sin", branche) ->
                           Prod((derive branche x), Applique("cos",branche))
                |  Applique ("cos", branche) ->
                           Applique("moins",
                                  Prod((derive branche x),
                                        Applique("sin",branche)))
                |  Applique (_, branche) ->
                                  invalid_arg "opérateur non prévu"
                |  Somme (branche_g,branche_d) ->
```

```
                     Somme((derive branche_g x),(derive branche_d x))
        |  Diff (branche_g,branche_d) ->
                     Diff((derive branche_g x),(derive branche_d x))
(* à suivre *)

  (* suite *)
        |  Prod (branche_g,branche_d) ->
                     Somme(  Prod((derive branche_g x),branche_d),
                             Prod(branche_g,(derive branche_d x)) )
        |  Quot (branche_g,branche_d) ->
                     Quot(Diff( Prod((derive branche_g x),branche_d),
                                Prod(branche_g,(derive branche_d x)) ),
                          Puiss(branche_d,Feuille_entier 2))
        |  Puiss (branche_g,Feuille_entier n) ->
                          Prod( Feuille_entier n,
                                Prod((derive branche_g x),
                                     Puiss(branche_g,Feuille_entier (n-1))) )
        |  Puiss (branche_g,branche_d) as tout ->
                Prod(Somme( Quot(Prod((derive branche_g x),branche_d),
                                 branche_g),
                            Prod(Applique("log",branche_g),
                                 (derive branche_d x)) ),
                      tout);;
```

Les arbres obtenus demandent à être simplifiés : beaucoup de sous-expressions du type $x + 0$ ou $1 * x$. Résultat du calcul de $\dfrac{d}{dx}(x^x + \cos(x^2))$

```
#let a  = analyse (lexeur_algébrique "x^x+cos(x^2)");;
a : arbre_arithmétique =
 Somme (Puiss (Feuille_var "x", Feuille_var "x"),
   Applique ("cos", Puiss (Feuille_var "x", Feuille_entier 2)))
#derive a "x";;
- : arbre_arithmétique =
 Somme
  (Prod (Somme
      (Quot (Prod (Feuille_entier 1, Feuille_var "x"), Feuille_var "x"),
       Prod (Applique ("log", Feuille_var "x"), Feuille_entier 1)),
     Puiss (Feuille_var "x", Feuille_var "x")),
   Applique ("moins",
     Prod
      (Prod (Feuille_entier 1, Puiss (Feuille_var "x", Feuille_entier 1)),
       Applique ("sin", Puiss (Feuille_var "x", Feuille_entier 2)))))
```

ce qui correspond à la formule

$$\Big(\frac{1 * x}{x} + \log(x) * 1\Big) * x^x + \Big(-\big((1 * x^1) * \sin(x^2)\big)\Big)$$

Dérivation simplifiée :

```
#simplifie (derive a "x");;
- : arbre_arithmétique =
 Somme
  (Prod
    (Somme
      (Quot (Feuille_var "x", Feuille_var "x"),
       Applique ("log", Feuille_var "x")),
     Puiss (Feuille_var "x", Feuille_var "x")),
   Applique
    ("moins",
     Prod
      (Feuille_var "x",
       Applique ("sin", Puiss (Feuille_var "x", Feuille_entier 2)))))
```

ce qui correspond à la formule

$$\left(\frac{x}{x} + \log(x)\right) * x^x + \left(- \left(x * \sin(x^2)\right)\right)$$

C'est déjà un peu plus raisonnable.