Informatique MP
TP
ARBRES EXPRESSIONS ALGEBRIQUES

Antoine MOTEAU
antoine.moteau@wanadoo.fr

2000, dernière rédaction : 24 Juillet 2003

. . . /ExpAlg-C.tex (2003)
. . . /ExpAlg-C.tex Compilé le dimanche 11 mars 2018 à 11h 23m 12s avec LaTeX.
Compilable avec LaTeX, PDFLaTeX, LuaLaTeX, XeLaTeX.

Ordre du TP : 3-ième tiers d'année (TP)

Eléments utilisés :

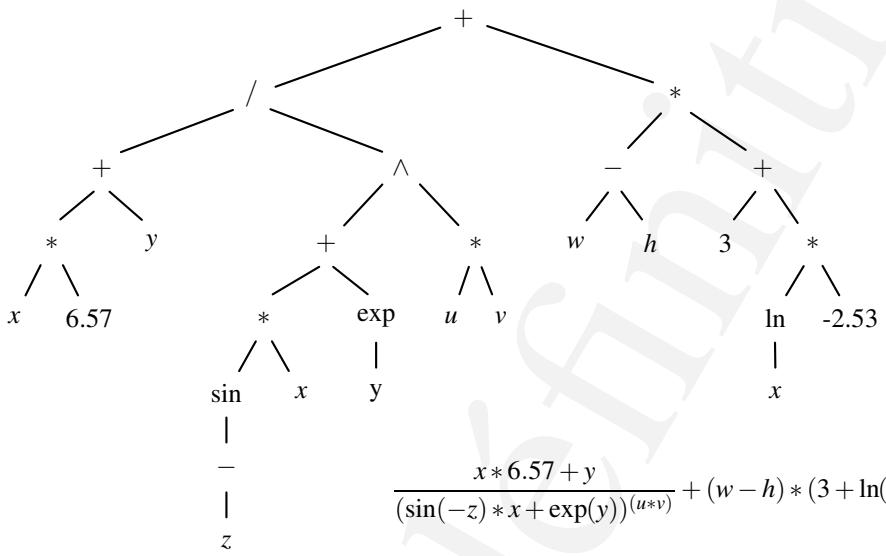
- arbres
- flux (un peu conséquent)
-

Documents relatifs au TP :

- Texte du TP : .tex, .dvi, .pdf
- Squelette de programme Caml : xxxx-S.ml
- Programme corrigé Caml : xxxx-C.ml

ARBRES D'EXPRESSIONS ALGEBRIQUES (sur $\mathbb{R}$)

Une expression algébrique, peut se représenter par un arbre :



$$\frac{x*6.57+y}{(\sin(-z)*x+\exp(y))^{(u*v)}}+(w-h)*(3+\ln(x)*-2.53)$$

La représentation arborescente d'une expression algébrique permet

- d'obtenir diverses représentations de l'expression, par exemple, pour l'arbre ci-dessus :
  - une EATP (Expression Algébrique infixe Totalement Parenthésée)

    $$((((x*6.57)+y)/(((\sin(-(z))*x)+\exp(y))\wedge(u*v)))+((w-h)*(3+(\ln(x)*-2.53))))$$

  - une EANTP (Expression Algébrique infixe Non Totalement parenthésée)

    $$(x*6.57+y)/(\sin(-z)*x+\exp(y))\wedge(u*v)+(w-h)*(3+\ln(x)*-2.53)$$

    où les opérateurs obéissent aux règles de priorité usuelles.
  - une forme post-fixe RPN (expression en Notation Polonaise Inverse (Reverse))

    $$x\uparrow 6.57*y+z\ \textbf{chs}\ \ \textbf{sin}\ x*y\ \textbf{exp}\ +u\uparrow v*\wedge/w\uparrow h-3\uparrow x\ \textbf{ln}\ 2.53\ \textbf{chs}\ *+*+$$

    où $\uparrow$ est l'instruction d'empilement ("Enter"), **chs** l'instruction de changement de signe.
- la réalisation du calcul formel : évaluation, dérivation, simplifications, etc ...

Objectif.

Dans ce TP, on se propose d'exploiter la représentation arborescente des expressions algébriques (dans le cadre réel), pour réaliser certaines opérations de calcul formel.

La construction directe ou la lecture directe d'un arbre n'étant pas pratique, on utilisera, comme intermédiaire, la représentation d'une expression algébrique sous forme de chaîne de caractères. Cela conduit donc, en préalable, à la réalisation de ressources de conversion :

- Construction d'un arbre à partir d'une chaîne représentant une EATP, EANTP, ERPN.
- Construction d'une chaîne représentant une EATP, EANTP, ERPN, à partir d'un arbre.

Documents.

Les éléments de ce TP sont initialement dans le dossier MPx/documents d'échange/TP-Info/ExpAlg/, et vous devez copier tout le dossier ExpAlg/ dans votre dossier personnel (voir explorateur).

Ensuite, vous trouverez normalement dans votre dossier ExpAlg/

- ce texte, au format .pdf : ExpAlg-S.pdf,
- la bibliothèque BibEAlg (ressources prédéfinies), représentée par les fichiers
  - BibEAlg.mli (source Caml de l'interface, lisible au format texte)
  - BibEAlg.zi (compilation Caml de l'interface, illisible)
  - BibEAlg.zo (compilation Caml des fonctions de la bibliothèque, illisible)
- un fichier Caml non entièrement exécutable, à compléter : ExpAlg-S.ml,
  comportant des ressources déjà écrites et des éléments partiellement écrits, à compléter.

Partie 1 : Informations, ressources

1 Généralités, organisation du document

On traite exclusivement les expressions algébriques usuelles du domaine réel.

Une expression algébrique est fournie initialement sous forme d'une chaîne de caractères (ou d'une liste de caractères ou d'un flux de caractères), le plus souvent en notation infixe (notation algébrique standard), qui peut être totalement parenthésée (EATP) ou non (ENATP), et qui comporte des

- valeurs constantes (nombres entiers ou décimaux si on est dans le domaine réel),
- opérateurs binaires (`^` * / + - . etc . . .)
- opérateurs unaires (-) (qui est noté comme le - binaire)
- parenthèses (en notation infixe)
- fonctions d'une seule variable (exp, ln, sin, cos , etc . . .)
- variables ou constantes nommées (autres identificateurs, de une ou plusieurs lettres)

Remarque. Il n'y a pas de distinction entre variable et fonction : une variable est une fonction sans argument qui renvoie la valeur (ou l'identificateur) associée à son nom.

Eléments de Caml : **flux (`stream`), pour information.**

Les ressources pré-écrites (bibliothèque BibEAlg) utilisent des objets du type `'a stream` (ou a' flux), définis par Caml, qui sont particulièrement adaptés à l'objectif de transformation des expressions EA(N)TP en arbres.
Les objets du type `'a stream` s'apparentent à des listes, mais avec des particularités :

- accès uniquement aux éléments de tête, par filtrage
- lecture destructrice
- . . .

Ressources : **création d'un ensemble de ressources organisées en bibliothèque (BibEAlg).**

La création de l'arbre associé à une EANTP représentée initialement sous forme de flux (ressource `stream` de Caml) de caractères, s'effectuera en deux étapes :

- première étape : analyse lexicale : identifier et séparer les constituants,
- deuxième étape : analyse syntaxique : organiser l'expression en construisant l'arbre.

Ensuite, on introduit des éléments de confort :

- représentation graphique d'un arbre d'expression algébrique
- ~~conversion d'un arbre en (chaîne de caractère) EATP ou EANTP ou RPN~~ (reporté dans le TP)
- . . .

Puis on définit un ensemble de ressources pour les calculs algébriques usuels, appliquées au cas des expressions algébriques arborescentes du domaine réel, mettant en jeu des constantes, des variables et des fonctions d'une seule variable.

Remarque. Uniquement pour l'évaluation numérique, pour ne pas introduire un système (complexe) de gestion de variables, on se limitera aux quatre seules variables x, y, z et t.

Enfin, on donne des informations sur les modalités de création d'un module de bibliothèque (ici BibEAlg) et on termine par la documentation :

- Interface du module de bibliothèque
- Description des fonctions exportées par la bibliothèque :
  - simple texte, avec description des éléments et de leur mode d'emploi,
  - éventuellement fichier d'aide style Window.

Réalisations (TP) :

Après la mise en place de fonctions de conversion d'un arbre en EATP, en EANTP ou en ERPN, l'essentiel du TP consiste en l'exploitation des ressources, fournies par la bibliothèque BibEAlg, pour réaliser des fonctions du calcul formel comme

- des simplifications (avec un succès non toujours bien probant)
- la dérivation (par rapport à une variable)
- . . .

2 ANALYSE LEXICALE (EANTP, EATP, RPN).

On analyse un flux de caractères qui représente l'expression algébrique brute, en groupant et séparant ses divers constituants, pour produire un flux de lexèmes (première mise en forme de l'expression algébrique, avec reconnaissance et séparation de constituants).

2.1 Type des constituants.

```
type lexeme =
 |Mot of string                 (* noms de fonctions, variables (alphabet a .. z) *)
 |Symbole of char               (* opérateurs ^ * / + -  et parenthèses ( ) *)
 |Constante_entiere of int      (* exemple 256789 *)
 |Constante_flottante of float  (* exemple 123.756 *)
;;
```

On introduit aussi quelques éléments de confort (facilités pour l'édition) :

```
let string_of_lexeme = function
 |Mot m                -> m                  |Symbole c             -> string_of_char c
 |Constante_entiere i  -> string_of_int i    |Constante_flottante f  -> string_of_float f
;;
(* string_of_lexeme : lexeme -> string = <fun> *)

let print_lexeme lex = print_string (string_of_lexeme lex)
;;
(* print_lexeme : lexeme -> unit = <fun> *)
```

2.2 Mise en place des fonctions d'analyse lexicale

On procède du particulier au général.

2.2.1 Ressources de lecture de lexèmes

On a besoin de ressources permettant

- d'évacuer les séparateurs usuels d'un texte (espace, tabulation, fin de ligne, fin de page)
- de lire le reste d'un entier (après déclenchement par le premier chiffre)
- de lire le reste d'un mot (après déclenchement par un premier caractère)
- de lire le reste de la partie fractionnaire d'un décimal (après déclenchement par le point décimal .)

```
let rec blancs_from_char_stream = function
  | [< '( ' ' | '\t' | '\n' ); blancs_from_char_stream r >] -> r
  | [< >] -> ()
;;
(* blancs_from_char_stream : char stream -> unit = <fun> *)

let rec int_from_char_stream n = function   (* n = accu *)
  | [< '( '0'..'9' as c); (int_from_char_stream (10 * n + int_of_char c - 48)) r >] -> r
  | [< >] -> n
and     string_from_char_stream m = function   (* m = accu *)
  | [<'('A'..'Z' |'a'..'z' |'à'|'â'|'ä'|'é'|'è'|'ê'|'ë'|'î'|'ï'|'ô'|'ö'|'ù'|'û'|'ü'|'ç' as c);
        (string_from_char_stream (m ^ (string_of_char c) ))  r >]  -> r
  | [< >] -> m
and     frac_from_char_stream fr = function   (* fr = accu *)
  | [< ' ( '0' .. '9' as c);
        (frac_from_char_stream (float_of_int (int_of_char c - 48))) r >] -> fr +. r /. 10.0
  | [< >] -> fr
;;
(* int_from_char_stream : int -> char stream -> int = <fun> *)
(* string_from_char_stream : string -> char stream -> string = <fun> *)
(* frac_from_char_stream : float -> char stream -> float = <fun> *)
```

2.2.2 Lecture d'un lexème, après déclenchement

Le premier caractère lu (déclencheur) identifie la nature du lexème. Après déclenchement, il faut poursuive la lecture du reste du lexème (si ce n'est pas un Symbole).

```
let rec lexeme_from_char_stream flux =
  blancs_from_char_stream flux;    (* filtrage "blanc" en tête *)
  match flux with
    | [<'('A'..'Z'|'a'..'z'| 'à'|'â'|'ä'|'é'|'è'|'ê'|'ë'|'î'|'ï'|'ô'|'ö'|'ù'|'û'|'ü'|'ç' as c);
        (string_from_char_stream (string_of_char c)) m >] -> Mot m
    | [< '('0'..'9' as c); (int_from_char_stream (int_of_char c - 48)) n; (suite n) r >] -> r
    | [< ' c >] -> Symbole c  (* pas forcément sensé *)

and suite n = function
  | [< ' '.' ; (frac_from_char_stream (float_of_int n)) r >] -> Constante_flottante r
  | [< >] -> Constante_entiere n
;;
(* lexeme_from_char_stream : char stream -> lexeme = <fun>
   suite : int -> char stream -> lexeme = <fun>
*)
```

Tests : (on crée artificiellement et facilement un flux de caractères à l'aide de `stream_of_string`)

```
lexeme_from_char_stream (stream_of_string " un test");;       (*  Mot "un" *)
lexeme_from_char_stream (stream_of_string " 123un test");;    (* Constante_entiere 123 *)
lexeme_from_char_stream (stream_of_string " 123.45un test");; (* Constante_flottante 123.45 *)
```

2.2.3 Analyse lexicale : transformation d'un flux de caractères en un flux de lexèmes.

```
let rec lexeme_stream_of_char_stream = function
  | [< lexeme_from_char_stream a; lexeme_stream_of_char_stream r >] -> [< 'a ; r >]
  | [< >] -> [< >]
;;
(* lexeme_stream_of_char_stream : char stream -> lexeme stream = <fun> *)
```

3 ANALYSE SYNTAXIQUE, CONSTRUCTION DE L'ARBRE.

A partir d'une expression algébrique sous forme d'un flux de lexèmes, on se propose de construire un arbre binaire de lexème qui représente l'expression algébrique.

On n'a pas fait de différence entre lexèmes variable ou constante nommée ou fonction : ce sont toutes des fonctions avec ou sans argument. La différence se fera au moment de la construction de l'arbre (analyse syntaxique) et sera repérée dans l'arbre par une situation particulière pour le nœud concerné.

3.1 Structure de l'arbre.

On utilise un type d'arbre binaire général, sous forme de produit :

```
type 'a arbre =
 | Vide
 | N of  ('a arbre) *  'a * ( 'a arbre)
;;
```

que l'on particularisera comme `lexeme arbre`, les opérateurs binaires ayant deux fils non vides, les opérateurs unaires et fonctions n'ayant que le fils gauche valide (par convention) et les constantes ou variables n'ayant pas de fils valide.

On aurait pu utiliser un type d'arbre plus spécialisé, avec plusieurs type de nœuds ou de feuilles, comme :

```
type 'a arbre2 =
 | F of 'a                                (* feuille à fruit = variable ou constante *)
 | N1 of   'a * ( 'a arbre2)              (* noeud opérateur unaire ou fonction *)
 | N2 of  ('a arbre2) *  'a * ( 'a arbre2);:  (* noeud opérateur binaire *)
```

3.2 Construction d'un arbre de lexèmes depuis un flux de lexèmes EANTP

On se place dans le cas d'un flux de lexèmes représentant une EANTP, où l'on gère la notion de priorité entre lexèmes selon les règles usuelles, les parenthèses n'étant utilisées que pour forcer les priorités (par exemple argument de fonction). Ce processus s'appliquera aussi aux flux de lexèmes représentant une EATP, puisqu'une EATP est un cas particulier d'EANTP.

```
let priorité_lexeme = function                              (* priorités usuelles *)
 |Symbole '('                                    -> 6 (* parenthèse forcée *)
 |(Constante_entiere i |Constante_flottante f) -> 5
 |Mot m                                          -> 4 (* variable, constante nommée, fonction *)
 |Symbole '^'                                    -> 3
 |(Symbole '*' | Symbole '/' )                   -> 2
 |(Symbole '+' | Symbole '-' )                   -> 1 (* - binaire *)
 | _                                             -> 0 (* 0 ou failwith "Opérateur invalide" *)
;;
(* priorité_lexeme : lexeme -> int = <fun> *)
```

Remarque. Le "-" unaire, considéré comme fonction (mais écrit comme Symbole), sera traité au cas par cas.

Constructeurs élémentaires (allégement d'écriture ultérieure)

```
let arbre_of_int n = N(Vide, Constante_entiere n, Vide)
and arbre_of_float r = N(Vide, Constante_flottante r, Vide)
;;
(* arbre_of_int : int -> lexeme arbre = <fun>
   arbre_of_float : float -> lexeme arbre = <fun>  *)
```

Pour la construction, l'algorithme suivant procède du général au particulier.

```
let rec arbre_of_lexeme_stream = function
   |[< expression_non_vide e >] -> supprime_par e

and expression_non_vide  = function
   |[< expression_simple_signee e; (suite_expression e) e' >] -> e'

and expression_simple_signee  = function     (* -e , +e , e  avec e non vide *)
 | [< 'Symbole '-' ; expression_simple_signee e >] ->     (* - unaire *)
     begin match e with
           | N(Vide, Constante_entiere n, Vide)   -> arbre_of_int (-n)      (* raccourci *)
           | N(Vide, Constante_flottante r, Vide) -> arbre_of_float (-. r)  (* raccourci *)
           | N(e', Symbole '-', Vide) -> e'                        (* répétition - unaire *)
           | _  -> N(e, Symbole '-', Vide)       (* N(e, Mot "-", Vide) était acceptable *)
     end
 | [< ' Symbole '+' ; expression_simple_signee e >] -> e   (* + unaire : ignoré  *)
 | [< expression_simple_non_signee e >]
        -> if e = Vide then failwith "Argument absent" else e

and expression_simple_non_signee = function
 | [<  expression_parenthesee e >]                      -> e
 | [< ' Mot m as c ; expression_simple_non_signee e >] ->
     begin match e with
           | Vide                    -> N(Vide, c, Vide)  (* Variable ou constante nommée *)
           | N(g, Symbole '(', Vide) -> N(g, c, Vide)     (* Fonction : c(g) *)
           | _ -> failwith "Incorrect"
     end
 | [< ' (Constante_entiere a as n)>]   -> N(Vide, n, Vide)
 | [< ' (Constante_flottante a as r)>] -> N(Vide, r, Vide)
 | [<  >]  -> Vide

and expression_parenthesee = function  (* Avec marquage temporaire de Priorité impérative *)
 |[< 'Symbole '('; expression_non_vide e; 'Symbole ')' >] -> N(e, Symbole '(', Vide)
```

```
and  suite_expression e = function
   |[< '(Symbole '+' | Symbole '-' | Symbole '*'| Symbole '/'| Symbole '^' as s );
          (expression_droite e s) e' >] -> e'
   |[< expression_simple_non_signee e'>] ->
                              if e' <> Vide then failwith "Opérateur absent" else e
   |[<  >]  -> e

and expression_droite e s = function
   |[<expression_parenthesee e' ; (suite_expression e') ee >] -> insere e s ee
   |[<expression_non_vide e' >] -> insere e s e'

and insere e s = function   (* insertion e s dans N(g,p,d)=fin expression, selon priorités *)
  | N(g, p, Vide) as e' -> N(e, s, e')   (* spécial priorité maxi: fonct, var, cte *)
  | N(g, p, d) when priorité_lexeme s >= priorité_lexeme p      -> N( insere e s g, p, d )
  | N(g, p, d) as e' when priorité_lexeme s < priorité_lexeme p -> N(e, s, e')
  | _ -> failwith "Erreur : insertion impossible"

and supprime_par = function  (* suppression finale des parenthèses forçant les priorités *)
  | Vide -> Vide
  | N(e, Symbole '(' , Vide) -> supprime_par e
  | N(g, p, d) -> N( supprime_par g, p, supprime_par d )
;;
(* arbre_of_lexeme_stream : lexeme stream -> lexeme arbre = <fun>
   expression_non_vide : lexeme stream -> lexeme arbre = <fun>
   expression_simple_signee : lexeme stream -> lexeme arbre = <fun>
   expression_simple_non_signee : lexeme stream -> lexeme arbre = <fun>
   expression_parenthesee : lexeme stream -> lexeme arbre = <fun>
   suite_expression : lexeme arbre -> lexeme stream -> lexeme arbre = <fun>
   expression_droite : lexeme arbre -> lexeme -> lexeme stream -> lexeme arbre = <fun>
   insere : lexeme arbre -> lexeme -> lexeme arbre -> lexeme arbre = <fun>
   supprime_par : lexeme arbre -> lexeme arbre = <fun> *)
```

Remarque. Seuls les Symboles sont explicitement pris parmi "+", "-", "×", "\", "∧".
Les noms de variables ou de fonctions sont alphabétiques quelconques (l'analyse est seulement syntaxique).
Une **analyse sémantique** ultérieure pourrait examiner si les noms de variables ou de fonctions, contenus dans un arbre, sont des noms acceptables dans un contexte donné.

3.3 Création d'un arbre de lexèmes depuis un flux de lexèmes RPN.

Non détaillé, mais c'est plus facile, puisqu'il n'y a pas de parenthèses ni de priorités à gérer.

3.4 Utilitaires sur les arbres d'expressions algébriques, test

3.4.1 Raccourci de construction (arbre de lexèmes, depuis une chaîne EANTP).

```
let arbre_of_string s =
          arbre_of_lexeme_stream (lexeme_stream_of_char_stream (stream_of_string s))
;;
(* arbre_of_string : string -> lexeme arbre = <fun> *)
```

3.4.2 Dessin d'un arbre de lexèmes (tant bien que mal . . .), test

On récupère une fonction de dessin déjà utilisée dans un précédent TP :
`(* dessine_arbre : ('a -> string) -> 'a arbre -> unit = <fun> *)`
que l'on particularise au cas des `lexeme arbre` :

```
let dessin_la = dessine_arbre (string_of_lexeme)        (* "_la" pour "_lexeme_arbre" *)
;;
(* dessin_la : lexeme arbre -> unit = <fun> *)

dessin_la (arbre_of_string "x*y+(z*x)^(u*v)+w");;  (* exemple pour test *)
```

4 Evaluation, opérations usuelles

On pourrait avoir un **dictionnaire des variables**, dans lequel chaque variable est accompagnée d'une valeur d'affectation ou d'une indication de non affectation ... et, de même, on pourrait avoir un **dictionnaire des fonctions usuelles** où chaque fonction est accompagné d'un "mode" de calcul ....
Ici, l'objectif n'étant pas de gérer des dictionnaires, on se place dans une **situation simple** :

1. Les variables connues sont supposées être uniquement les quatre variables de nom "x", "y", "z", "t", et on ne dispose que d'une constante nommée, connue, qui est de nom "pi".
   Comme les variables connues peuvent être affectées indifféremment de valeur entière, de valeur flottante ou de leur identificateur (non affectation), il faut introduire un type particulier :

   ```
   type variable =   Z of int        (* Entier relatif *)
                   | R of float      (* Réel (en fait décimal limité, rationnel) *)
                   | S of string     (* Chaine, réduite à "x", "y", "z" ou "t" *)
   ;;
   let pi = R (4.0 *. atan(1.0));;  (* constante nommée *)
   let x = ref( S "x") and y = ref( S "y") and z = ref( S "z") and t = ref( S "t");;
   ```

   On pourra ainsi, par exemple, faire les affectations suivantes :

   ```
   x := R 1.53;;    (* affecter la valeur float 1.53 à "la variable" "x" *)
   x := S "x";;     (* AVEC PRECAUTION DE CORRESPONDANCE: dé-affecter la variable "x" *)
   t := Z 5;;       (* affecter la valeur entière 5 à "la variable" "t" *)
   x := S "y";;     (* serait une substitution. A EVITER ! *)
   ```

   On définit aussi la transformation d'une variable en un lexeme (Constante valeur de variable ou Mot nom de variable , selon que la variable est affectée ou non) :
   `value lexeme_of_variable : variable -> lexeme`

   Remarque. Les variables et constantes connues sont représentées par les variables Caml de même nom (x, y, z, t, pi), ce qui interdit d'utiliser ces identificateurs à d'autres fins dans le programme Caml.
2. Les fonctions connues sont limitées à "exp", "ln", "sin", "cos", "tan", "asin", "acos", "atan", "abs", "ent", "sqrt" (et "-" unaire) et doivent pouvoir s'évaluer indifféremment sur des lexèmes Constantes entières ou floats.
3. Les opérateurs binaires sont "+", "-" (binaire), "$\times$", "$\backslash$", "$\wedge$" et doivent pouvoir s'évaluer indifféremment sur des lexèmes Constantes entières ou floats.
4. **Les autres identificateurs, variables ou fonctions, non connus, seront admis mais non évalués.**

Pour accepter des arguments numériques qui soient, indifféremment, des entiers ou des flottants, on envisage de transformer un argument numérique (lexème constante) dans le type flottant, par la fonction :
`value float_of_lexeme_cte : lexeme -> float`
D'autre part, chaque fois qu'un sous-arbre pourra être évalué numériquement, il est succeptible d'être remplacé par un arbre réduit à sa valeur numérique v : N(Vide, Constante_xxxx v, Vide).

4.1 fonctions de bas niveau (usage interne seulement).

Ces fonctions sont à usage interne seulement **(privé)**, c'est à dire normalement inconnues de l'utilisateur : en effet, elles ne s'appliquent qu'à certains cas de lexèmes, sans gestion de toutes les erreurs, et leur liste de paramètres est peu explicite. Elles ne doivent donc être appelées que dans des conditions bien précises.
Les erreurs non gérées devront être "rattrapées" à l'issue des appels à ces fonctions.

`value lexeme_of_lexeme_variable : lexeme -> lexeme`
lexeme_of_lexeme_variable a, où a est un lexème "variable" (parmi "pi", "x", "y", "z", "t"), renvoie le lexème Constante numérique ou le lexème Mot "chaine" associé à la variable.
Renvoie l'erreur "Variable inconnue" si a ne correspond pas à "pi", "x", "y", "z", "t".

```
value ajoute_lexeme_cte    : lexeme * lexeme -> lexeme
value soustrait_lexeme_cte : lexeme * lexeme -> lexeme
value multiplie_lexeme_cte : lexeme * lexeme -> lexeme
value divise_lexeme_cte    : lexeme * lexeme -> lexeme
value power_lexeme_cte     : lexeme * lexeme -> lexeme
```
Evaluation des opérations usuelles, entre des lexèmes Constante, avec adaptation de type numérique éventuel.

`value operateur_bi_lexeme_cte : lexeme -> lexeme -> lexeme -> lexeme`
operateur_bi_lexeme_cte g d op où g et d sont des lexèmes Constante et op est un lexème Symbole d'opérateur, renvoie le lexème Constante évalué au calcul correspondant à l'opération op entre g et d.
Renvoie l'erreur "Opérateur inconnu" si op ne correspond pas à "+", "-", "×, "\, "∧".

```
value moins_lc : lexeme -> lexeme    (* moins unaire *)
value valeur_absolue_lc : lexeme -> lexeme    value partie_entiere_lc : lexeme -> lexeme
value sqrt_lc : lexeme -> lexeme
value exp_lc  : lexeme -> lexeme              value ln_lc   : lexeme -> lexeme
value sin_lc  : lexeme -> lexeme              value asin_lc : lexeme -> lexeme
value cos_lc  : lexeme -> lexeme              value acos_lc : lexeme -> lexeme
value tan_lc  : lexeme -> lexeme              value atan_lc : lexeme -> lexeme
```

Renvoient un lexème Constante, résultat de l'évaluation de fonctions usuelles sur un lexème Constante.

`value operateur_mono_lexeme_cte : lexeme -> lexeme -> lexeme`
operateur_mono_lexeme_cte arg f où arg est un lexème Constante et f est un lexème Mot fonction ou le lexème Symbole '-' (moins unaire), renvoie le lexème Constante évalué au calcul correspondant à f(arg).
Renvoie l'erreur "Fonction inconnue" si f ne correspond pas à une des fonctions connues (ou à - unaire).

4.2 fonctions de niveau intermédiaire, exportées.

Ces fonctions appellent les fonctions précédentes et "récupèrent" leurs erreurs éventuelles.

```
value is_arbre_zero : lexeme arbre -> bool
is_arbre_un : lexeme arbre -> bool
```

is_arbre_zero a renvoie true si et seulement si la racine de a est réduite à un lexème Constante 0 ou 0.0 .
is_arbre_zero a renvoie true si et seulement si la racine de a est réduite à un lexème Constante 1 ou 1.0 .

```
value prefix +? (* value ajoute_arbre *)    : lexeme arbre * lexeme arbre -> lexeme arbre
value prefix -? (* value soustrait_arbre *) : lexeme arbre * lexeme arbre -> lexeme arbre
value prefix *? (* value multiplie_arbre *) : lexeme arbre * lexeme arbre -> lexeme arbre
value prefix /? (* value divise_arbre *)    : lexeme arbre * lexeme arbre -> lexeme arbre
value prefix ^? (* value power_arbre *)     : lexeme arbre * lexeme arbre -> lexeme arbre
```

Ces opérations renvoient un arbre, résultat de l'opération entre les deux arguments. Si les deux arguments sont réduits à des racines lexème Constante, le résultat est l'arbre évalué, réduit à une racine lexème Constante.

> La prise en compte des cas particuliers où un des arguments a une racine lexème Constante qui est "zéro" ou "un", permet de réduire éventuellement (selon les règles arithmétiques dans $\mathbb{R}$) l'arbre résultat.
> A priori, ce sont les seules simplifications effectuées automatiquement .

On introduit également les fonctions usuelles appliquées à un arbre. Par exemple,

`value sin_arbre : lexeme arbre -> lexeme arbre;;`
sin_arbre u renvoie l'arbre correspondant à sin(u). Si u est réduit à une racine lexème Constante, le résultat est l'arbre évalué, réduit à une racine lexème Constante.

Les deux fonctions suivantes (peu sûres) doivent rester à usage interne bien contrôlé :

```
operateur_bi_arbre : lexeme arbre -> lexeme arbre -> lexeme -> lexeme arbre
operateur_mono_arbre : lexeme arbre -> lexeme -> lexeme arbre
```

operateur_bi_arbre g d op lorsque op est un lexème Symbole d'opérateur binaire connu, renvoie l'arbre binaire résultat de cette opération entre g et d, selon les règles définies ci-dessus.

operateur_mono_arbre g f lorsque f est un lexème Mot fonction ou le lexème Symbole '-' (moins unaire), renvoie l'arbre binaire résultat de la fonction f sur g. Si g est réduit à une racine lexeme Constante et si f est une fonction connue, le résultat est un arbre réduit à une racine lexeme Constante dont la valeur est l'évaluation de f sur g.

Dans les deux cas, si l'opérateur est inconnu, le résultat est l'arbre formel correspondant.

4.3 Evaluation d'un arbre

`value evalue_arbre : lexeme arbre -> lexeme arbre`
evalue_arbre a renvoie un arbre où tous les calculs numériques possibles connus ont été réalisés, y compris ceux avec les variables affectées de valeur et ceux résultant de la présence de 0 ou de 1. Les variables, opérateurs ou fonctions inconnus restent non évalués.

5 Mise en place du module de bibliothèque BibEAlg

Comme l'éditeur CMD-Caml a une capacité limitée (32 Ko), tout ne tiendra pas dans un seul fichier.

- Les définitions de type : lexème, arbre, variable, option
- les fonctions d'analyse lexicale
- les fonctions de l'analyse syntaxique (construction de l'arbre)
- les fonctions de l'évaluation
- ressources de dessin et quelques autres ...

sont organisées en une bibliothèque "BibEAlg", composée des fichiers BibEAlg.mli (texte), BibEAlg.zi (compilé), BibEAlg.zo (compilé) et éventuellement BibEAlg.hlp (documentation), tous situés dans un même dossier à préciser et tous de même nom primaire.

étape 1 : Compilation initiale
Tous ces éléments Caml introduits précédement sont initialement dans le fichier BibEAlg.ml
Le fichier BibEAlg.ml est compilé, ce qui permet de relever explicitement les types de fonctions.

étape 2 : Création des sources fichier d'interface (.mli) et fichier de code (.ml) (de même nom primaire)

- Les types de données publics (à exporter) sont exclus de BibEAlg.ml au profit de BilEAlg.mli
- Les types des fonctions publiques (à exporter) sont inclus comme déclarations dans le fichier BilEAlg.mli (réponse de type fournie par Caml, précédée de "value" et privée de l'information finale "= <fun>" ou "= ()")
- Les types des variables et constantes publiques (à exporter) sont inclus comme déclarations dans le fichier BilEAlg.mli (exemple : `value pi : variable;;`).

étape 3 : Création des fichiers compilés (tous fichiers, de même nom primaire, dans un même dossier) :

- BibEAlg.zi (compilation de l'interface .mli)
- BibEAlg.zo (compilation des objets de .ml)

a) Supprimer les fichiers BibEAlg.zi et BibEAlg.zo s'il existent déjà (nettoyage).
b) Compiler par le menu "file compile" de CamlWin, **dans cet ordre** :

- en premier, BibEalg.mli (ce qui produit le fichier BibEalg.zi)
- ensuite, BibEalg.ml (ce qui produit le fichier BibEalg.zo)
  Remarque. Si BibEalg.zi n'existe pas, la compilation de BibEalg.ml produira ce fichier BibEalg.zi, avec toutes les définitions utilisée dans BibEalg.ml.
  Ainsi, pour pouvoir contrôler le contenu de BibEalg.zi, il faut créer un fichier BibEalg.mli ne contenant que les définitions que l'on veut exporter et le compiler en premier.

étape 4 : Création de la documentation

- Le fichier BibEalg.mli est le fichier source d'interface
- On peut créer un fichier d'aide Windows : BibEalg.hlp (avec HcWin, compilateur d'aide Windows) et/ou un fichier de texte simple pour la description des fonctions : BibEalg.txt
- Un fichier du genre Lisezmoi.txt pour informations générales (licence, conditions de vente, ...).

étape 5 : Fournir un exemple simple d'utilisation, par exemple dans un fichier test.ml :

```
#directory "R:/InfoMp/Tp-Info/ExpAlg";;          (* A ADAPTER *) (* "barre" renversée *)
load_object "bibealg";;  (* minuscules ! *)
#open "bibealg";;        (* minuscules ! *)

let a = arbre_of_string "(x*6.57+y)/(sin(-z)*x+exp(y))^(u*v)+(w-h)*(3+ln(x)*-2.53)";;
dessin_la a;;
```

étape 6 : Installation de la bibliothèque
Ici l'installation se résume à la simple copie des fichiers BibEalg.mli, BibEalg.zi, BibEalg.zo, BibEalg.hlp, ..., dans un dossier utilisateur, par exemple `R:/InfoMp/Tp-Info/ExpAlg/` .

étape 7 : tester (fonctions testées avant l'étape 1) mais il est préférable de re-tester ...

6 Interface BibEAlg.mli

```
type lexeme =
 |Mot of string                (* noms de fonctions, de variables (dans l'alphabet a .. z ) *)
 |Symbole of char              (* opérateurs ^ * / + -  et parenthèses ( ) *)
 |Constante_entiere of int     (* exemple 256789 *)
 |Constante_flottante of float (* exemple 123.756 *)
;;
type 'a arbre = | Vide
                | N of  ('a arbre) * 'a * ('a arbre)
;;
type variable =   Z of int        (* Entier relatif (de Z) *)
                | R of float      (* Float (décimal limité), réel (de R) *)
                | S of string     (* Chaine (String) *)
;;
type option = normal | trig | expln      (* options pour la simplification *)
;;
value pi : variable;;                    (* seules "constantes" et variables connues *)
value x : variable ref;; value y : variable ref;;
value z : variable ref;; value t : variable ref;;

value lowcase_of_char : char -> char;;
value lowcase_of_string : string -> string;;

value string_of_lexeme : lexeme -> string;;
value print_lexeme : lexeme -> unit;;
value priorité_lexeme : lexeme -> int;;

value dessin_la : lexeme arbre -> unit;;          (* représentation graphique *)

value arbre_of_int : int -> lexeme arbre;;        (* arbres constantes élémentaires *)
value arbre_of_float : float -> lexeme arbre;;

value arbre_of_string : string -> lexeme arbre;;  (* chaîne EANATP -> arbre de lexèmes *)

value is_arbre_zero : lexeme arbre -> bool;;
value is_arbre_un : lexeme arbre -> bool;;

value prefix +? : lexeme arbre -> lexeme arbre -> lexeme arbre;;
value prefix -? : lexeme arbre -> lexeme arbre -> lexeme arbre;;
value prefix *? : lexeme arbre -> lexeme arbre -> lexeme arbre;;
value prefix /? : lexeme arbre -> lexeme arbre -> lexeme arbre;;
value prefix ^? : lexeme arbre -> lexeme arbre -> lexeme arbre;;

value minus_arbre : lexeme arbre -> lexeme arbre;;  (* moins unaire *)
value exp_arbre : lexeme arbre -> lexeme arbre;;    (* seules fonctions connues *)
value ln_arbre : lexeme arbre -> lexeme arbre;;
value sqrt_arbre : lexeme arbre -> lexeme arbre;;
value sin_arbre : lexeme arbre -> lexeme arbre;;
value cos_arbre : lexeme arbre -> lexeme arbre;;
value tan_arbre : lexeme arbre -> lexeme arbre;;
value asin_arbre : lexeme arbre -> lexeme arbre;;
value acos_arbre : lexeme arbre -> lexeme arbre;;
value atan_arbre : lexeme arbre -> lexeme arbre;;
value abs_arbre : lexeme arbre -> lexeme arbre;;
value int_arbre : lexeme arbre -> lexeme arbre;;    (* partie entière *)

value evalue_arbre : lexeme arbre -> lexeme arbre;;
```

Partie 2 : Le TP

7 TP sur les arbres d'expressions algébriques

7.1 Transformation d'un arbre en une chaîne RPN

Pour obtenir une chaîne en Notation Polonaise Inverse (RPN ou PI), il suffit de parcourir l'arbre dans l'ordre "gauche, droite, père", à l'aide d'une fonction auxiliaire qui possède un accumulateur (initialement vide).

Dans la construction de l'expression de N(g,p,d), le résultat obtenu pour l'accumulation avec la construction de g est passé comme accumulateur pour la construction de d et ce dernier résultat est suivi de l'expression de p (string_of_lexeme p) puis d'un caractère vide (" ") pour "aérer" (et séparer), **mais**

- il faut introduire la touche "Enter" (à représenter par "|") lorsque g est une constante ou variable et d est non vide, pour séparer g de d,
- il faut gérer spécialement le "-" unaire (à éditer comme "chs"),
- et il y a le problème des constantes négatives ("-123" devrait s'éditer comme "123 chs").

```
(* ERPN_of_arbre : lexeme arbre -> string = <fun> *)                    (* A ECRIRE *)

ERPN_of_arbre (arbre_of_string "(x*6.57+y)/(sin(-z)*x+exp(y))^(u*v)+(w-h)*(3+ln(x)*-2.53)");;
renvoie  "x |6.57 * y + z chs sin x * y exp + u |v * ^ / w |h - 3 |x ln 2.53 chs * + * + ".
```

7.2 Transformation d'un arbre en une chaîne Algébrique Parenthésée

On utilise un parcours en profondeur "infixe" (gauche, père, droit). Cependant, comme il faut mettre les parenthèses, il est préférable de l'écrire directement sans utiliser de parcours générique :

7.2.1 Forme Totalement Parenthésée (EATP)

C'est assez facile, puisque l'on met systématiquement les parenthèses autour

- du fils d'une fonction (y compris le "-" unaire),
- d'un bloc (g, opérateur, d),

et pas de parenthèses autour d'une constante, constante nommée ou variable.

```
(* EATP_of_arbre : lexeme arbre  -> string = <fun> *)                    (* A ECRIRE *)
```

7.2.2 Forme Non Totalement Parenthésée (EANTP)

Ici, c'est plus délicat : il faut prendre en compte les priorités entre opérateurs pour introduire ou ne pas introduire de parenthèses.
On utilisera une fonction auxiliaire qui renvoie le couple : (priorité de la racine, chaîne de l'arbre).

- Pour les opérateurs binaires, selon la comparaison de la priorité de la racine par rapport à un fils, on entoure ou non ce fils par des parenthèses.
- Les parenthèses seront systématiques autour de l'argument d'une fonction (y compris "-" unaire).

```
(* EANTP_of_arbre : lexeme arbre -> string = <fun> *)                    (* A ECRIRE *)
```

7.3 Expressions équivalentes ou égales

Deux expressions sont

- équivalentes ("formellement égales") si elles prennent simultanément les mêmes valeurs quelques soient les valeurs des variables (si on sait faire l'évaluation complète),
- seront "structurellement égales" si elles conduisent à des arbres structurellement identiques (et elles seront alors équivalentes).

Cependant, la différence structurelle ne suffit pas pour décider de la non équivalence.
Si l'égalité structurelle (même noeuds et mêmes feuilles aux mêmes endroits) est facile à établir, par contre, décider de l'équivalence n'est pas chose aisée.
Par exemple `"(x+y)*x"` et `"x*y+x^2"` ne donnent pas naissance aux mêmes arbres, bien que qu'elles soient "équivalentes" (dans le domaine $\mathbb{R}[x,y]$) et même si on développe complétement deux expressions équivalentes, on n'obtient pas forcément le même arbre. (par exemple `"x+(y+z)"` et `"(y+z)+x"` ou encore `"x+2*y"` et `"y+(x+y)"` dans $\mathbb{R}[x,y,z]$).

> En fait, on peut démontrer qu'aucun système fini de règles ne permet de transformer deux expressions équivalentes en une même troisième expression.

On se contentera ici de la fonction :

`value prefix =? (egal_arbre) : lexeme arbre -> lexeme arbre -> bool;;  (* A ECRIRE *)`
a =? b (ou egal_arbre a b) est vrai si et seulement si les arbres a et b sont "quasi" structurellement égaux ("quasi" : on tiendra compte de la commutativité de "+" et de "*") mais les calculs connus sont non effectués.

7.4 Simplifications (application de formules).

On applique les formules de simplification directement à des arbres d'expression algébrique (cela permet d'enchaîner les simplifications sans être obligé de reconstruire un arbre à chaque fois) et en final, on pourra renvoyer l'EANTP sous forme de chaîne.
Remarque. L'évaluation prend déjà en compte quelques simplifications opératoires dues aux valeurs 0 et 1.

Dans le domaine trigonométrique (trig), (a et b étant des arbres), on pourra transformer :

les formes `"cos(a)^2 + sin(a)^2"` en des formes `"1"`,

les formes `"sin(a) * cos(b) + cos(a) * sin(b)"` en des formes `"sin(a+b)"` etc..

Dans le domaine exponentiel et logarithme (expln), (a et b étant des arbres), on pourra transformer :

les formes `"ln(a) + ln(b)"` en `"ln(a*b)"` et les formes `"exp(a) * exp(b)"` en `"exp(a+b)"` ,

les formes `"exp(ln(a))"` et `"ln(exp(a))"` en `"a"` etc..

Cependant, on sera encore limité par le problème du test de l'équivalence de sous-arbres pour pouvoir appliquer "finement" les formules de simplification. Il faudra se limiter (pour l'instant) aux égalité structurelles.

- Par exemple, avec la formule `"cos(a)^2 + sin(a)^2 = 1"`, `"cos(x+2*y)^2 + sin(x+2*y))^2"` se simplifiera en `"1"`, mais la transformation de `"cos(x+2*y)^2 + sin(y+(x+y))^2"` en `"1"` et celle de `"sin(a)^2 + cos(a)^2"` en `"1"` peuvent échouer (il n'y a pas de miracles).
- D'autre part, il est parfois nécessaire d'appliquer des formules "à l'envers" avant de simplifier ...

7.4.1 Comparaison à des arbres "modèles"

Pour accomplir une simplification dans un arbre u, on utilise un **arbre modèle** (de la formule à utiliser) que l'on compare à l'arbre à simplifier.

1. On crée une **liste de correspondance** entre les "variables" du modèle et les sous-arbres correspondant dans u :
   `value association_arbre_model:lexeme arbre->lexeme arbre->(lexeme*lexeme arbre) list;;`
2. Dans la liste d'association (variable du modèle, sous-arbre de l'arbre à simplifier), lorsque les égalités structurelles montre la parfaite correspondance avec le modèle, on applique la formule de simplification.

Pour chaque formule, on aura une fonction de simplification selon le modèle de cette formule. Par exemple :

`value model_trig1 : lexeme arbre -> lexeme arbre;; (* cos(a)^2+sin(a)^2 *)`
model_trig1 u compare l'arbre u avec le pré-modèle $\cos^2 a + \sin^2 b$ puis, si d'après la liste d'association, les sous-arbres de u correspondant aux variables a et b sont (structurellement) égaux, renvoie l'arbre constant égal à 1 et, sinon renvoie u.

Remarque. C'est un pneu lourd, mais il n'y a pas de choses (à peu près simples) à faire dans ce contexte.

7.4.2 Simplifications (élémentaires), par catégories

Les simplifications se font par catégories d'expressions et seront "orientées" dans l'une des catégories usuelles par la spécification d'un paramètre :

`type option = normal | trig | expln;;  (* option de simplification *)`

1. fonctions spécialisées (de bas niveau) :

   `value simplify_arbre_normal : lexeme arbre -> lexeme arbre;;`
   simplify_arbre_normal a, renvoie l'arbre déduit de a par remplacement de sous-structures de a selon quelques règles algébriques usuelles (au niveau de la racine). (on se limite à quelques formules)

   `value simplify_arbre_trig : lexeme arbre -> lexeme arbre;;`
   simplify_arbre_trig a, renvoie l'arbre déduit de a par remplacement de sous-structures de a selon des formules de trigonométrie usuelles. (on se limite à quelques formules)

   `value simplify_arbre_expln : lexeme arbre -> lexeme arbre;;`
   simplify_arbre_expl a, renvoie l'arbre déduit de a par remplacement de sous-structures de a selon des formules usuelles relatives à $\wedge$, exp et Ln. (on se limite à quelques formules)

2. fonction de simplification avec option :

   `value simplify_arbre : lexeme arbre -> option -> lexeme arbre;;`
   simplify_arbre a opt, où opt est une option parmi (normal, trig ou expln) renvoie l'arbre déduit de a par les simplifications indiquées par opt.

3. Extension aux formules sous forme de chaînes EANTP :

   `value simplify : string -> option -> string;;`
   simplify expr opt, où expr est une chaîne représentant une expression EANTP et opt est une option de simplification parmi (normal, trig ou expln), renvoie la chaîne de l'expression EANTP déduite de l'expression de expr par les simplifications indiquées par opt.

Exemples :

Si, dans le modèle trigonométrique on a introduit la méthode qui simplifie, de gauche à droite, selon la formule $\cos^2 a + \sin^2 b = 1$, alors `simplify "cos(x+y)^2+sin(x+y)^2" trig;;` renvoie `"1"`.

Si, dans le modèle exponentiel on a introduit la méthode qui simplifie, de gauche à droite, selon la formule $e^a * e^b = e^{a+b}$, alors `simplify "exp(a) * exp(2*b)" expln;;` renvoie `"exp(a+2*b)"`.

Travail à effectuer : Introduire de nouvelles formules pour les options **trig** et **expln**
(on pourra s'inspirer de celles qui sont déjà écrites) et faire des exemples de test ...

7.5 Dérivation : dérivations partielles

La dérivation s'apparente à l'application de formules dans un arbre :

- diff(a,x) donne 0 (si a est une constante ou une variable autre que x ou une formule sans x)
- diff(x,x) donne 1
- diff$(a*b,x)$ donne diff$(a,x)*b+a*$diff$(b,x))$
- diff$(\ln(a),x)$ donne diff$(a,x)/a$
- etc ... (voir formulaire de dérivation)

Remarques.

Il faudra ensuite simplifier pour prendre en compte le grand nombre de valeurs 0 et 1 introduites.

diff(pi, pi) donne 1, diff(ln, ln) donne 1, diff(ln(ln), ln) donne $1/\ln$ (comme avec Maple !).

```
value diff_arbre : string -> lexeme arbre -> lexeme arbre;;        (* A ECRIRE *)
```

diff_arbre v a, où v est une chaîne représentant un nom de variable ("u", "v", "ln" ... par exemple) renvoie l'arbre déduit de a par dérivation par rapport à la variable de nom v.
On prendra en compte **toutes** les formules relatives aux opérateurs $(+,-,\times,/,\wedge,-(unaire))$ et aux fonctions connues ici (exp, ln, sqrt, sin, cos, tan, asin, acos, atan, abs, mais pas int).

Remarque. Dans un souci de clarification, on pourra décomposer, selon la nature de la racine :

```
diff_variable : string -> lexeme arbre -> lexeme arbre;;
diff_operateur_bi : string -> lexeme arbre -> lexeme arbre;;
diff_operateur_mono : string -> lexeme arbre -> lexeme arbre;;
```

En final, on a la dérivation d'une expression algébrique, donnée sous forme de chaîne :

```
value diff : string -> string -> string;;                          (* A ECRIRE *)
```

diff expr v, où expr est une chaîne EANTP et v est un nom de variable valide sous la forme "nom" (sans espace), renvoie la chaîne EANTP de la dérivée par rapport à v de expr.

Par exemple,

```
x := S "x";; y := S "y";; z := S "z";; t := S "t";;   (* "unassign" *)
diff "y*y + x*y+z*ln(x+y^2) + y^2 * b^2" "y";;   (* --> "y+y+x+z*2*y/(x+y^2)+2*y*b^2" *)
diff "y*y + x*y+z*ln(x+y^2)+ y^2 * b^2" "b";;    (* --> "y^2+2*b" *)
```

$< \mathscr{FIN} >$ TP expAlg (énoncé) suite = squelette, ...

8 Squelette de programme (à compléter)

```
#directory "F:/MPx/TP-info/ExpAlg";;  (* A ADAPTER Attention / *)
load_object "BibEAlg";;  (* respect de la casse !! *)
#open "BibEAlg";;        (* respect de la casse !! *)

(*=== 3.4. Construction et dessin: Exemple ===*)
let a = arbre_of_string "(x*6.57+y)/(sin(-z)*x+exp(y))^(u*v)+(w-h)*(3+ln(x)*-2.53)";; dessin_la a;;

(*===  4.3. Evaluation (tests) ===*)
let b = arbre_of_string "(x+y)*z+t+ln(raz)";; dessin_la b;;

x := R 1.5333;; y := S "y";; z := Z 5;; t := S "t";; let b1 = evalue_arbre b;; dessin_la b1;;
x := R 1.5;; y := R 2.5;; z := Z 5;; t := Z 1;; let b1 = evalue_arbre b;; dessin_la b1;;
x := Z 3;; y := Z 7;; z := S "z";; t := S "t";; let b2 = evalue_arbre b;; dessin_la b2;;

(*=== 7. EDITIONS ===*)
(*=== 7.1.1. Edition sous forme polonaise inverse (RPN, post-fixe) *)
let ERPN_of_arbre = aux ""
 where rec aux accu = function     (* postfixe:  gauche, droit, père *)
     A_ECRIRE ()
;;
(* ERPN_of_arbre : lexeme arbre -> string = <fun> *)

(*=== 7.1.2. Edition sous forme EATP ====*)
let rec EATP_of_arbre = function      (* avec ( ) *)
    A_ECRIRE ()
;;
(* EATP_of_arbre : lexeme arbre -> string = <fun> *)

(*=== 7.2.3. Edition sous forme EANTP ===*)
let EANTP_of_arbre a = snd( EANTP a)
 where rec EANTP = function      (* avec réduction des () par priorités *)
    A_ECRIRE ()
;;
(* EANTP_of_arbre : lexeme arbre -> string = <fun> *)

ERPN_of_arbre a;;  EATP_of_arbre a;;   EANTP_of_arbre a;;

(*== Complément : Evaluation expression chaine ===*)
let evalue a = EANTP_of_arbre( evalue_arbre (arbre_of_string a))
;;
(* evalue : string -> string = <fun> *)

evalue "(x*6.57+y)/(sin(-z)*x+exp(y))^(u*v)+(w-h)*(3+ln(x)*-2.53)";;

(*== 7.3. EGALITE STRUCTURELLE ===*) (* plus commutativité de + et x *)
let rec prefix =? x y = egal_arbre x y
and egal_arbre  = fun
    A_ECRIRE ()
;;
(* =? : lexeme arbre -> lexeme arbre -> bool = <fun>
   egal_arbre : lexeme arbre -> lexeme arbre -> bool = <fun>   *)

let a1 = arbre_of_string "(x+y)+z";; let a2 = arbre_of_string "z+(x+y)";;
let a3 = arbre_of_string "y+(z+x)";;
a1 =? a2;;  (* true  *)
a1 =? a3;;  (* false *)

(*== 7.4. SIMPLIFICATIONS ====*)
x := S "x";; y := S "y";; z := S "z";; t := S "t";;            (* "unassign"  *)

(*=== simplifications algébriques simples, au premier niveau ===*)
```

```
let simplify_arbre_normal0 a = evalue_arbre (simp_norm0 a)   (* normal0 : à améliorer dans le futur *)
  where rec simp_norm0 = function
    | N(g, Symbole '-', Vide) -> let g' = simp_norm0 g in     (* suppression des - unaires cumulés *)
           begin match g' with
                 | N(g'', Symbole '-', Vide) -> g''
                 | _ -> N(g', Symbole '-', Vide)
           end
    | N(g, Mot m, Vide)    -> let g' = simp_norm0 g in N(g', Mot m, Vide)
    | N(g, Symbole '+', d) -> let g' = simp_norm0 g and d' = simp_norm0 d in
                              if g' =? d' then (arbre_of_int 2) *? d' else N(g', Symbole '+', d')
    | N(g, Symbole '-', d) -> let g' = simp_norm0 g and d' = simp_norm0 d in
                              if g' =? d' then arbre_of_int 0 else N(g', Symbole '-', d')
    | N(g, Symbole '*', d) -> let g' = simp_norm0 g and d' = simp_norm0 d in
                              if g' =? d' then d ^? arbre_of_int 2 else N(g', Symbole '*', d')
    | N(g, Symbole '/', d) -> let g' = simp_norm0 g and d' = simp_norm0 d in
                              if g' =? d' then arbre_of_int 1 else N(g', Symbole '/', d')
    | N(g, Symbole '^', d) -> let g' = simp_norm0 g and d' = simp_norm0 d in N(g', Symbole '^', d')
    | a -> a
;;
(* simplify_arbre_normal0 : lexeme arbre -> lexeme arbre = <fun> *)

(*==== Comparaison à un modèle -> liste d'association --*)
let association_arbre_model model u =  let l = ref [] in  if aux model u then !l else []
  where rec aux = fun
  | Vide _ -> false
  | u (N(Vide, Mot m, Vide)) -> l := (Mot m, u)::!l; true
  | (N(Vide, p, Vide)) (N(Vide, p', Vide)) -> p = p'
  | (N(g,p,Vide)) (N(g',p',Vide)) -> (p = p') & (aux g g')
  | (N(xg,xp,xd)) (N(yg, yp,yd)) when xp = yp ->
     begin match xp with
             (Symbole '+' | Symbole '*') -> ((aux xg yg) & (aux xd yd)) or ((aux xg yd) & (aux xd yg))
           | _ ->  (aux xg yg) & (aux xd yd)
     end
  | _  _ -> false
;;
(* association_arbre_model : lexeme arbre -> lexeme arbre  -> (lexeme * lexeme arbre) list = <fun> *)

(*==== Modèles normal plus élaboré .... --*)

(*==== Modèles trigo --*)
let model_trig1 u =
  let l = association_arbre_model u (arbre_of_string "sin(a)^2+cos(b)^2")
  in try let a = assoc (Mot "a") l and b = assoc (Mot "b") l in if a =? b then arbre_of_int 1 else u
     with Not_found -> u
;;
(* model_trig1 : lexeme arbre -> lexeme arbre = <fun> *)

let model_trig2 u =
 let l = association_arbre_model u (arbre_of_string "sin(a)*cos(b)+sin(c)*cos(d)")
  in try let a = assoc (Mot "a") l and b = assoc (Mot "b") l
         and c = assoc (Mot "c") l and d = assoc (Mot "d") l
         in if a =? d & b =? c then N( N(a, Symbole '+', b), Mot "sin", Vide) else u
     with Not_found -> u
;;
(* model_trig2 : lexeme arbre -> lexeme arbre = <fun> *)

let u = arbre_of_string "sin(x+y)^2+cos(x+y)^2";;  EANTP_of_arbre (model_trig1 u);;
let u = arbre_of_string "cos(x+y)^2+sin(x+y)^2";;  EANTP_of_arbre (model_trig1 u);;
let u = arbre_of_string "sin(x+y)*cos(z)+sin(z)*cos(x+y)";; EANTP_of_arbre (model_trig2 u);;
let u = arbre_of_string "sin(x+y)*cos(z)+cos(x+y)*sin(z)";; EANTP_of_arbre (model_trig2 u);;
(*--------------*)
let models_trig u = model_trig1 (model_trig2 u)     (* composition des modèles de simplifications *)
;;
(* models_trig : lexeme arbre -> lexeme arbre = <fun> *)
```

```
let simplify_arbre_trig a = evalue_arbre (simp_trig a)
  where rec simp_trig = function
    | N(Vide, p, Vide) as u -> evalue_arbre u
    | N(g, p, Vide) -> let g' = evalue_arbre (simp_trig g) in models_trig (N(g', p, Vide))
    | N(g, p, d)    -> let g' = evalue_arbre (simp_trig g) and d' = evalue_arbre (simp_trig d)
                       in models_trig (N(g', p, d'))
    | u -> u
;;
(* simplify_arbre_trig : lexeme arbre -> lexeme arbre = <fun> *)

let u = arbre_of_string "(sin(x+y)*cos(z)+cos(x+y)*sin(z))^2+cos(x+y+z)^2";;
EANTP_of_arbre (simplify_arbre_trig u);;

(*==== Modèles exp et Ln ---*)
let model_expln1 u = let l = association_arbre_model u (arbre_of_string "exp(a)*exp(b)")
  in try let a = assoc (Mot "a") l and b = assoc (Mot "b") l in N(N(a, Symbole '+', b), Mot "exp", Vide)
     with Not_found -> u
and model_expln2 u = let l = association_arbre_model u (arbre_of_string "ln(a)+ln(b)")
  in try let a = assoc (Mot "a") l and b = assoc (Mot "b") l in N(N(a, Symbole '*', b), Mot "ln", Vide)
     with Not_found -> u
and model_expln3 u = let l = association_arbre_model u (arbre_of_string "ln(exp(a))")
  in try assoc (Mot "a") l
     with Not_found -> u
and model_expln4 u = let l = association_arbre_model u (arbre_of_string "exp(ln(a))")
  in try assoc (Mot "a") l
     with Not_found -> u
;;
(* model_expln1 : lexeme arbre -> lexeme arbre = <fun> *)
(* model_expln2 : lexeme arbre -> lexeme arbre = <fun> *)
(* model_expln3 : lexeme arbre -> lexeme arbre = <fun> *)
(* model_expln4 : lexeme arbre -> lexeme arbre = <fun> *)

let u = arbre_of_string "exp(x+y)*exp(z)";;   EANTP_of_arbre (model_expln1 u);;
let u = arbre_of_string "ln(x+y)+ln(z)";;     EANTP_of_arbre (model_expln2 u);;
let u = arbre_of_string "ln(exp(sin(x)+y))";; EANTP_of_arbre (model_expln3 u);;

(*-------------*)
let models_expln u =      (* composition des modèles de simplifications *)
  model_expln4( model_expln3 (model_expln1 (model_expln2 u)))
;;
(* models_expln : lexeme arbre -> lexeme arbre = <fun> *)

let simplify_arbre_expln a = evalue_arbre (simp_expln a)
where rec simp_expln = function
  | N(Vide, p, Vide) as u -> evalue_arbre u
  | N(g, p, Vide) -> let g' = evalue_arbre (simp_expln g)
                     in models_expln (N(g', p, Vide))
  | N(g, p, d) -> let g' = evalue_arbre (simp_expln g) and d' = evalue_arbre (simp_expln d)
                  in models_expln (N(g', p, d'))
  | u -> u
;;
(* simplify_arbre_expln : lexeme arbre -> lexeme arbre = <fun> *)

let u = arbre_of_string "exp(exp(ln(x+y))*ln(exp(z)))*exp(t)";; dessin_la u;;
EANTP_of_arbre (simplify_arbre_expln u);;

(*===== Simplification selon options --*)
let simplify_arbre u = function
  | normal -> simplify_arbre_normal0 u  (* simplify_arbre_normal u *)
  | trig   -> simplify_arbre_trig u
  | expln  -> simplify_arbre_expln u
;;
(* simplify_arbre : lexeme arbre -> option -> lexeme arbre = <fun> *)
```

```
(*--- Simplification selon options, sous forme de chaines ===*)
let simplify a opt= EANTP_of_arbre (simplify_arbre (arbre_of_string a) opt)
;;
(* simplify : string -> option -> string = <fun> *)

simplify "((x+y)+z) - (y+(z+x))" normal;;
simplify "(sin(x+y)*cos(z)+cos(x+y)*sin(z))^2+cos(x+y+z)^2" trig;;
simplify "exp(exp(ln(x+y))*ln(exp(z)))*exp(t)" expln;;

(*== 7.5. DERIVATION ===*)
let rec diff_arbre var = function
  | Vide -> Vide
  | N(Vide, Mot s, Vide) as e -> diff_variable var (evalue_arbre e)
  | N(Vide, m, Vide) -> arbre_of_int 0
  | N(g, p, Vide) as e -> evalue_arbre (diff_operateur_mono var e)
  | e   -> evalue_arbre (diff_operateur_bi var e)

and diff_variable var = function
    A_ECRIRE ()

and diff_operateur_bi var = function
  | N( u, p, v ) as e  -> let du = diff_arbre var u and dv = diff_arbre var v in
           begin match p with
                    A_ECRIRE ()
           end
  | _ -> failwith "Erreur impossible"

and diff_operateur_mono var = function
 | N(x, (Symbole '-' as g), Vide) -> minus_arbre (diff_arbre var x)
 | N(x, Mot s, Vide) as u ->  let dx = diff_arbre var x in
        begin try match lowcase_of_string s with
                      A_ECRIRE ()
                    | _   -> failwith "Fonction inconnue"  (* inutile si vérifié *)
              with Failure s -> failwith s
        end
 | _ -> failwith "Fonction attendue"
;;
(* diff_arbre : string -> lexeme arbre -> lexeme arbre = <fun>
   diff_variable : string -> lexeme arbre -> lexeme arbre = <fun>
   diff_operateur_bi : string -> lexeme arbre -> lexeme arbre = <fun>
   diff_operateur_mono : string -> lexeme arbre -> lexeme arbre = <fun>
*)

(*--------- adaptation au cas string ----------*)
let diff a v =
 match arbre_of_string v with
  | N(Vide, Mot m, Vide) -> if v = m then EANTP_of_arbre (diff_arbre m (arbre_of_string a))
                            else failwith "variable incorrecte"
  | _ -> failwith "variable incorrecte"
;;
(* diff : string -> string -> string = <fun> *)

x := S "x";; y := S "y";; z := S "z";; t := S "t";;  (* "unassign" *)
let s = "y*y + x*y+z*ln(x+y^2) + y^2 * b^2";;
diff s "y";;  diff s "b ";;   diff s "b";;
```

9 Corrigé

```
#directory "H:/MPx/TP-info/ExpAlg";;  (* A ADAPTER Attention / *)
load_object "BibEAlg";;  (* respect de la casse !! *)
#open "BibEAlg";;        (* respect de la casse !! *)

(*=== 3.4. Construction et dessin: Exemple ===*)
(*=== 4.3. Evaluation (tests) ====*)
let b = arbre_of_string "(x+y)*z+t";; dessin_la b;;
(*-- affectation partielle --*)
x := R 1.5333;; y := S "y";; z := Z 5;; t := S "t";; let b1 = evalue_arbre b;; dessin_la b1;; dessin_la b;;
(*-- affectation totale  --*)
x := R 1.5;; y := R 2.5;; z := Z 5;; t := Z 1;; let b1 = evalue_arbre b;; dessin_la b1;;
x := Z 3;; y := Z 7;; z := S "z";; t := S "t";; let b2 = evalue_arbre b;; dessin_la b2;;

(*== 7.1. Edition sous forme polonaise inverse (post-fixe) ===*)
let ERPN_of_arbre = aux ""
 where rec aux accu = function     (* postfixe:  gauche, droit, père *)
  |Vide     -> accu
  |N(Vide,e,Vide) -> accu ^
    begin match e with
           | Constante_entiere c   when c < 0  -> (string_of_lexeme (Constante_entiere (-c))) ^ " chs "
           | Constante_flottante c when c < 0. -> (string_of_lexeme (Constante_flottante (-. c))) ^ " chs "
           | _ -> string_of_lexeme e ^ " "
    end
  |N(g,e,d) -> let sep = match g, d with
                         | N(Vide, _, Vide), Vide  -> ""
                         | N(Vide, _, Vide), _     -> (char_for_read (char_of_int 124))
                         | _                       -> ""
               in let op = if (d = Vide) & (e = Symbole '-') then "chs" else (string_of_lexeme e)
                  in let x = (aux accu g) ^ sep in (aux x d) ^ op ^ " "
;;
(* ERPN_of_arbre : lexeme arbre -> string = <fun> *)

(*== 7.2.1. Edition sous forme EATP *)
let rec EATP_of_arbre = function      (* avec ( ) *)
  | Vide -> ""
  | N(Vide,p,Vide) -> string_of_lexeme p
  | N(g, p, Vide)  -> let f = string_of_lexeme p and x = EATP_of_arbre g  in f ^ "(" ^ x ^ ")"
  | N(g, p, d )    -> let op = string_of_lexeme p and x = EATP_of_arbre g and y = EATP_of_arbre d
                      in "(" ^ x ^ op ^ y ^ ")"
;;
(* EATP_of_arbre : lexeme arbre -> string = <fun> *)

(*== 7.2.2. Edition sous forme EANTP *)
let EANTP_of_arbre a = snd( EANTP a)
 where rec EANTP = function      (* avec réduction des () par priorités *)
  | Vide  -> (0, "")
  | N(Vide, p, Vide)  -> (priorité_lexeme p, string_of_lexeme p)
  | N(g, p, Vide ) when p = Symbole '-' -> let op = string_of_lexeme p  and (k,x) = EANTP g
                                           in (4, op ^ (if k < 4 then "(" ^ x ^ ")" else x))
  | N(g, p, Vide) -> let f = string_of_lexeme p  and (k,x) = EANTP g
                     in (4, f ^ "(" ^ x ^ ")" )
  | N(g, p, d)  -> let h = priorité_lexeme p and op = string_of_lexeme p
                   and (ka,a) = EANTP g and (kb,b) = EANTP d
                   in let wa = (if h > ka then "(" ^ a ^ ")" else a) ^ op
                   in (h, wa^ (if h > kb then  "(" ^ b ^ ")" else b))
;;
(* EANTP_of_arbre : lexeme arbre -> string = <fun> *)

let s = "(x*6.57+y)/(sin(-z)*x+exp(y))^(u*v)+(w-h)*(3+ln(x)*(-2.53))";;
let a = arbre_of_string s;; dessin_la a;;
ERPN_of_arbre a;; EATP_of_arbre a;; EANTP_of_arbre a;; s;;

(*== Complément : Evaluation expression chaine *)
let evalue a = EANTP_of_arbre( evalue_arbre (arbre_of_string a));;
(* evalue : string -> string = <fun> *)
```

```
x;; y;; z;; t;; evalue "(x*6.57+y)/(sin(-z)*x+exp(y))^(u*v)+(w-h)*(3+ln(x)*-2.53)";;

(*== 7.3. EGALITE STRUCTURELLE *) (* plus commutativité de + et x *)
let rec prefix =? x y = egal_arbre x y
and egal_arbre  = fun
 | Vide Vide -> true
 | (N(xg,xp,Vide)) (N(yg,yp,Vide)) -> (xp = yp) & (egal_arbre xg yg )
 | (N(xg,xp,xd)) (N(yg, yp,yd)) when xp = yp ->
     begin match xp with
           (Symbole '+' | Symbole '*') -> ( (egal_arbre xg yg) & (egal_arbre xd yd) ) or
                                          ( (egal_arbre xg yd) & (egal_arbre xd yg) )
           | _ ->  (egal_arbre xg yg) & (egal_arbre xd yd)
     end
 | _  _ -> false
;;
(* =? : lexeme arbre -> lexeme arbre -> bool = <fun>
   egal_arbre : lexeme arbre -> lexeme arbre -> bool = <fun>  *)

let a1 =arbre_of_string "(x+y)+z";; let a2 =arbre_of_string "z+(x+y)";; let a3 =arbre_of_string "y+(z+x)";;
a1 =? a2;;  (* true  *)   a1 =? a3;;  (* false *)  a1 -? a3;;

(*== 7.4. SIMPLIFICATIONS *)
x := S "x";; y := S "y";; z := S "z";; t := S "t";;           (* "unassign"  *)

(*=== simplifications algébriques simples, au premier niveau *)
let simplify_arbre_normal0 a = evalue_arbre (simp_norm0 a)
 where rec simp_norm0 = function
  | N(g, Symbole '-', Vide) -> let g' = simp_norm0 g in   (* suppression des - unaires cumulés *)
            begin match g' with
                  | N(g'', Symbole '-', Vide) -> g''
                  | _                         -> N(g', Symbole '-', Vide)
            end
  | N(g, Mot m, Vide)    -> let g' = simp_norm0 g in N(g', Mot m, Vide)
  | N(g, Symbole '+', d) -> let g' = simp_norm0 g  and d' = simp_norm0 d in
                            if g' =? d' then (arbre_of_int 2) *? d' else N(g', Symbole '+', d')
  | N(g, Symbole '-', d) -> let g' = simp_norm0 g and d' = simp_norm0 d in
                            if g' =? d' then arbre_of_int 0 else N(g', Symbole '-', d')
  | N(g, Symbole '*', d) -> let g' = simp_norm0 g and d' = simp_norm0 d in
                            if g' =? d' then d ^? arbre_of_int 2 else N(g', Symbole '*', d')
  | N(g, Symbole '/', d) -> let g' = simp_norm0 g and d' = simp_norm0 d in
                            if g' =? d' then arbre_of_int 1 else N(g', Symbole '/', d')
  | N(g, Symbole '^', d) -> let g' = simp_norm0 g and d' = simp_norm0 d in N(g', Symbole '^', d')
  | a -> a
;;
(* simplify_arbre_normal0 : lexeme arbre -> lexeme arbre = <fun> *)

(*==== Comparaison à un modèle -> liste d'association --*)
let association_arbre_model model u = let l = ref [] in if aux model u then !l else []
 where rec aux = fun
  | Vide _ -> false
  | u (N(Vide, Mot m, Vide)) -> l := (Mot m, u)::!l; true
  | (N(Vide, p, Vide)) (N(Vide, p', Vide)) -> p = p'
  | (N(g,p,Vide)) (N(g',p',Vide)) -> (p = p') & (aux g g')
  | (N(xg,xp,xd)) (N(yg, yp,yd)) when xp = yp ->
     begin match xp with
           (Symbole '+' | Symbole '*') -> ((aux xg yg) & (aux xd yd)) or ((aux xg yd) & (aux xd yg))
          | _ ->  (aux xg yg) & (aux xd yd)
     end
  | _  _ -> false
;;
(* association_arbre_model : lexeme arbre -> lexeme arbre -> (lexeme * lexeme arbre) list = <fun> *)
```

```
(*==== Modèles "normal" ... non rédigé --*)

(*==== Modèles "trigo" --*)
let model_trig1 u = let l = association_arbre_model u (arbre_of_string "sin(a)^2+cos(b)^2")
  in try let a = assoc (Mot "a") l and b = assoc (Mot "b") l
          in if a =? b then arbre_of_int 1 else u
     with Not_found -> u
and model_trig2 u = let l = association_arbre_model u (arbre_of_string "sin(a)*cos(b)+sin(c)*cos(d)")
  in try let a = assoc (Mot "a") l and b = assoc (Mot "b") l
         and c = assoc (Mot "c") l and d = assoc (Mot "d") l
         in if a =? d & b =? c then N( N(a, Symbole '+', b), Mot "sin", Vide) else u
     with Not_found -> u
;;
(* model_trig1 : lexeme arbre -> lexeme arbre = <fun> *)
(* model_trig2 : lexeme arbre -> lexeme arbre = <fun> *)

let u = arbre_of_string "sin(x+y)^2+cos(x+y)^2";; EANTP_of_arbre (model_trig1 u);;
let u = arbre_of_string "cos(x+y)^2+sin(x+y)^2";; EANTP_of_arbre (model_trig1 u);;
let u = arbre_of_string "sin(x+y)*cos(z)+sin(z)*cos(x+y)";; EANTP_of_arbre (model_trig2 u);;
let u = arbre_of_string "sin(x+y)*cos(z)+cos(x+y)*sin(z)";; EANTP_of_arbre (model_trig2 u);;

(*--------------*)
let models_trig u =  model_trig1 (model_trig2 u)         (* composition des modèles de simplifications *)
;;
(* models_trig : lexeme arbre -> lexeme arbre = <fun> *)

let simplify_arbre_trig a = evalue_arbre (simp_trig a)
  where rec simp_trig = function
   | N(Vide, p, Vide) as u -> evalue_arbre u
   | N(g, p, Vide) -> let g' = evalue_arbre (simp_trig g) in models_trig (N(g', p, Vide))
   | N(g, p, d) -> let g' = evalue_arbre (simp_trig g) and d' = evalue_arbre (simp_trig d)
                   in models_trig (N(g', p, d'))
   | u -> u
;;
(* simplify_arbre_trig : lexeme arbre -> lexeme arbre = <fun> *)

let u = arbre_of_string "(sin(x+y)*cos(z)+cos(x+y)*sin(z))^2+cos(x+y+z)^2";;
dessin_la u;; EANTP_of_arbre (simplify_arbre_trig u);;

(*==== Modèles "exp" et "Ln" ---*)
let model_expln1 u = let l = association_arbre_model u (arbre_of_string "exp(a)*exp(b)")
  in try let a = assoc (Mot "a") l and b = assoc (Mot "b") l
         in N(N(a, Symbole '+', b), Mot "exp", Vide)
     with Not_found -> u
and model_expln2 u = let l = association_arbre_model u (arbre_of_string "ln(a)+ln(b)")
  in try let a = assoc (Mot "a") l and b = assoc (Mot "b") l
         in N(N(a, Symbole '*', b), Mot "ln", Vide)
     with Not_found -> u
and model_expln3 u = let l = association_arbre_model u (arbre_of_string "ln(exp(a))")
  in try assoc (Mot "a") l
     with Not_found -> u
and model_expln4 u = let l = association_arbre_model u (arbre_of_string "exp(ln(a))")
  in try assoc (Mot "a") l
     with Not_found -> u
;;
(* model_expln1 : lexeme arbre -> lexeme arbre = <fun> *)
(* model_expln2 : lexeme arbre -> lexeme arbre = <fun> *)
(* model_expln3 : lexeme arbre -> lexeme arbre = <fun> *)
(* model_expln4 : lexeme arbre -> lexeme arbre = <fun> *)

let u = arbre_of_string "exp(x+y)*exp(z)";;  EANTP_of_arbre (model_expln1 u);;
let u = arbre_of_string "ln(x+y)+ln(z)";;     EANTP_of_arbre (model_expln2 u);;
let u = arbre_of_string "ln(exp(sin(x)+y))";; EANTP_of_arbre (model_expln3 u);;

(*-------------*)
let models_expln u =                          (* composition des simplifications *)
  model_expln4( model_expln3 (model_expln1 (model_expln2 u)))
```

```
;;
(* models_expln : lexeme arbre -> lexeme arbre = <fun> *)

let simplify_arbre_expln a = evalue_arbre (simp_expln a)
  where rec simp_expln = function
   | N(Vide, p, Vide) as u -> evalue_arbre u
   | N(g, p, Vide) -> let g' = evalue_arbre (simp_expln g) in models_expln (N(g', p, Vide))
   | N(g, p, d) -> let g' = evalue_arbre (simp_expln g) and d' = evalue_arbre (simp_expln d)
                   in models_expln (N(g', p, d'))
   | u -> u
;;
(* simplify_arbre_expln : lexeme arbre -> lexeme arbre = <fun> *)

let u = arbre_of_string "exp(exp(ln(x+y))*ln(exp(z)))*exp(t)";;
dessin_la u;; EANTP_of_arbre (simplify_arbre_expln u);;

(*===== Simplification selon options --*)
let simplify_arbre u = function
  | normal -> simplify_arbre_normal0 u    (* simplify_arbre_normal u *)
  | trig   -> simplify_arbre_trig u
  | expln  -> simplify_arbre_expln u
;;
(* simplify_arbre : lexeme arbre -> option -> lexeme arbre = <fun> *)

(*--- Simplification selon options, sous forme de chaines ===*)
let simplify a opt= EANTP_of_arbre (simplify_arbre (arbre_of_string a) opt)
;;
(* simplify : string -> option -> string = <fun> *)

simplify "((x+y)+z) - (y+(z+x))" normal;;
simplify "(sin(x+y)*cos(z)+cos(x+y)*sin(z))^2+cos(x+y+z)^2" trig;;
simplify "exp(exp(ln(x+y))*ln(exp(z)))*exp(t)" expln;;

(*== 7.5. DERIVATION ====*)
let rec diff_arbre var = function
  | Vide -> Vide
  | N(Vide, Mot s, Vide) as e -> diff_variable var (evalue_arbre e)
  | N(Vide, m, Vide) -> arbre_of_int 0
  | N(g, p, Vide) as e -> evalue_arbre (diff_operateur_mono var e)
  | e   -> evalue_arbre (diff_operateur_bi var e)

and diff_variable var = function
  | N(Vide, Mot s, Vide) as e when (s = var) -> arbre_of_int 1
  | _ -> arbre_of_int 0

and diff_operateur_bi var = function
  | N( u, p, v ) as e  ->
    let du = diff_arbre var u and dv = diff_arbre var v in
    begin match p with
        | Symbole `+` -> du +? dv
        | Symbole `-` -> du -? dv
        | Symbole `*` -> (du *? v ) +? (u *? dv)
        | Symbole `/` -> ( du /? v ) -? ( (u *? dv ) /? (v ^? (arbre_of_int 2) ))
        | Symbole `^` -> ((u ^? v) *? ( (ln_arbre u) *? dv))
                         +? (v *? (u ^? (v -? arbre_of_int 1)) *? du)
        | _ -> failwith "Opérateur inconnu"
    end
  | _ -> failwith "Erreur impossible"
```

```
and diff_operateur_mono var = function
 | N(x, (Symbole '-' as g), Vide) -> minus_arbre (diff_arbre var x)
 | N(x, Mot s, Vide) as u ->
   let dx = diff_arbre var x in
   begin
     try match lowcase_of_string s with
          |"exp"  -> u *? dx
          |"ln"   -> dx /? x
          |"sqrt" -> dx /? (arbre_of_int 2 *? u )
          |"sin"  -> (cos_arbre x) *? dx
          |"cos"  -> minus_arbre ( (sin_arbre x) *? dx )
          |"tan"  -> dx /? ( (cos_arbre x) ^? (arbre_of_int 2) )
          |"asin" -> dx /? sqrt_arbre ( (arbre_of_int 1) -? (x ^? (arbre_of_int 2) ) )
          |"acos" -> minus_arbre (dx /? sqrt_arbre ( (arbre_of_int 1) -? (x ^? (arbre_of_int 2) ) ))
          |"atan" -> dx /? ( (arbre_of_int 1) +? (x ^? (arbre_of_int 2) ) )
          |"abs" -> dx *? ( x /? u )
(*          |"int" -> arbre_of_int 0  *)
          | _   -> failwith "Fonction inconnue"  (* inutile si vérifié *)
     with Failure s -> failwith s
   end
 | _ -> failwith "Fonction attendue"
;;
(* diff_arbre : string -> lexeme arbre -> lexeme arbre = <fun>
   diff_variable : string -> lexeme arbre -> lexeme arbre = <fun>
   diff_operateur_bi : string -> lexeme arbre -> lexeme arbre = <fun>
   diff_operateur_mono : string -> lexeme arbre -> lexeme arbre = <fun>
*)

(*--------- adaptation au cas string ----------*)
let diff a v =
 match arbre_of_string v with
  | N(Vide, Mot m, Vide) -> if v = m then EANTP_of_arbre (diff_arbre m (arbre_of_string a))
                            else failwith "variable incorrecte"
  | _ -> failwith "variable incorrecte"
;;
(* diff : string -> string -> string = <fun> *)

x := S "x";; y := S "y";; z := S "z";; t := S "t";;  (* "unassign" *)
let s = "y*y + x*y+z*ln(x+y^2) + y^2 * b^2";;
diff s "y";; diff s "b ";; diff s "b";; simplify (diff s "y") normal;;

(***********************************)
(* Fin de TP ExpAlg-C  (Corrigé)   *)
(***********************************)
```

10 Compléments éventuels

10.1 Evaluation d'une ERPN sous forme de chaîne à l'aide d'une pile

10.2 Evaluation d'une EATP sous forme de chaîne à l'aide d'une pile

Sans passer par un arbre, on transforme une chaîne EATP en une EATP de lexèmes puis, à l'aide d'une pile, en une forme RPN de lexèmes que l'on peut évaluer à l'aide d'une pile.

10.3 Arbres d'expressions booléennes. Formes normales

Après adaptation des arbres d'expressions algèbriques au domaine des expressions booléennes, on pourrait mettre en place la transformation d'une expression booléenne sous forme

- normale conjonctive
- normale disjonctive

Bonne idée de TP pour l'an prochain ...

10.4 Représentation d'une expression algébrique sous forme "dessinée" normale

$$\frac{x\times 6.57+y}{\big(\sin(-z)\times x+\exp(y)\big)^{u\times v}}+(w-h)\times\big(3+\ln(x)\times -2.53\big)$$

est plus agréable à lire que :

$$(x*6.57+y)/(\sin(-z)*x+\exp(y))\wedge(u*v)+(w-h)*(3+\ln(x)*-2.53)$$

10.5 Automates : expression algébrique d'un langage, construction d'un automate

Voir Cours/automates/ConstrB1.ml ...

$< \mathscr{F}\mathscr{I}\mathscr{N} >$ **TP ExpAlg** (énoncé + squelette + corrigé).