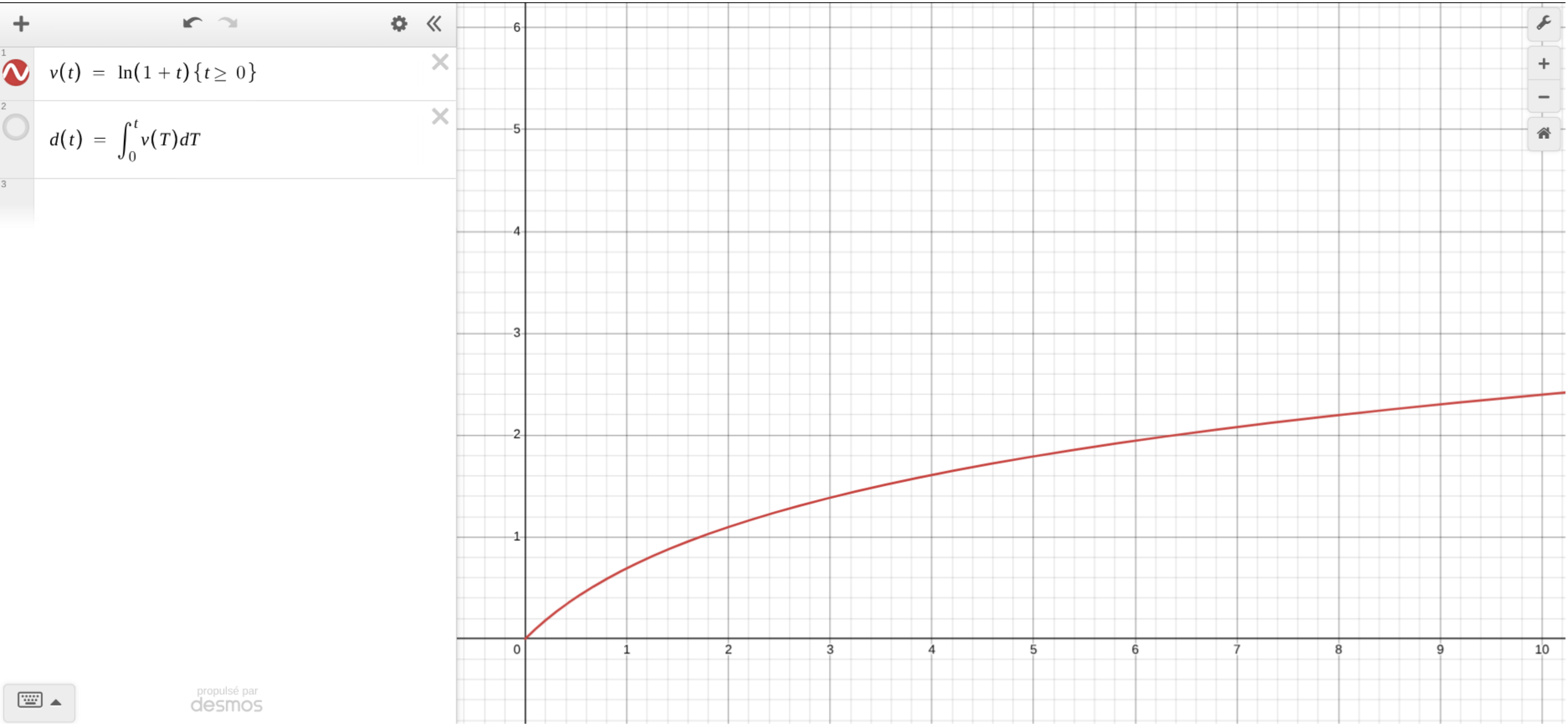
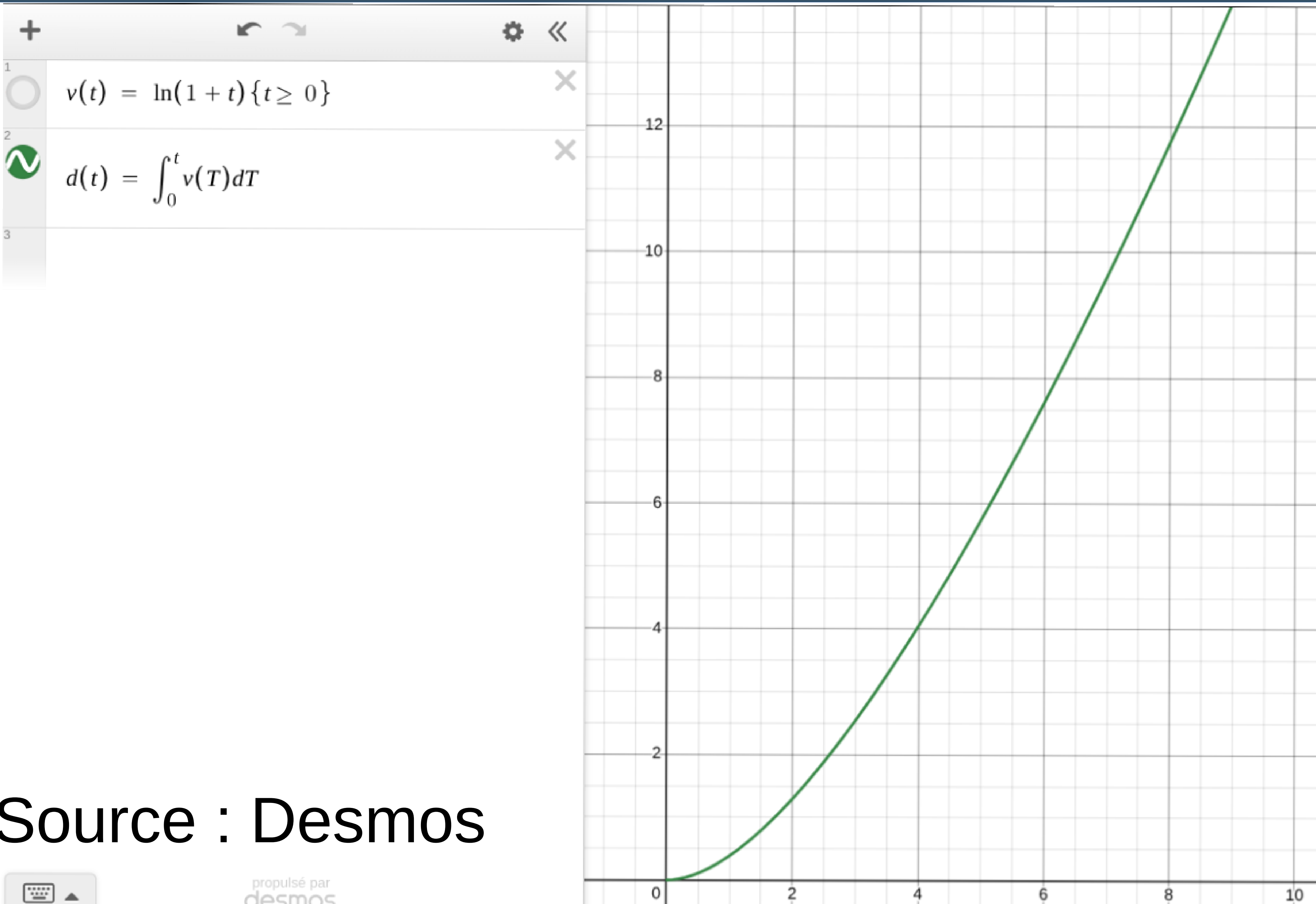
Procédure de primitivation de fonctions rationnelles et logarithmiques par le biais de l'algorithme de Risch

Introduction



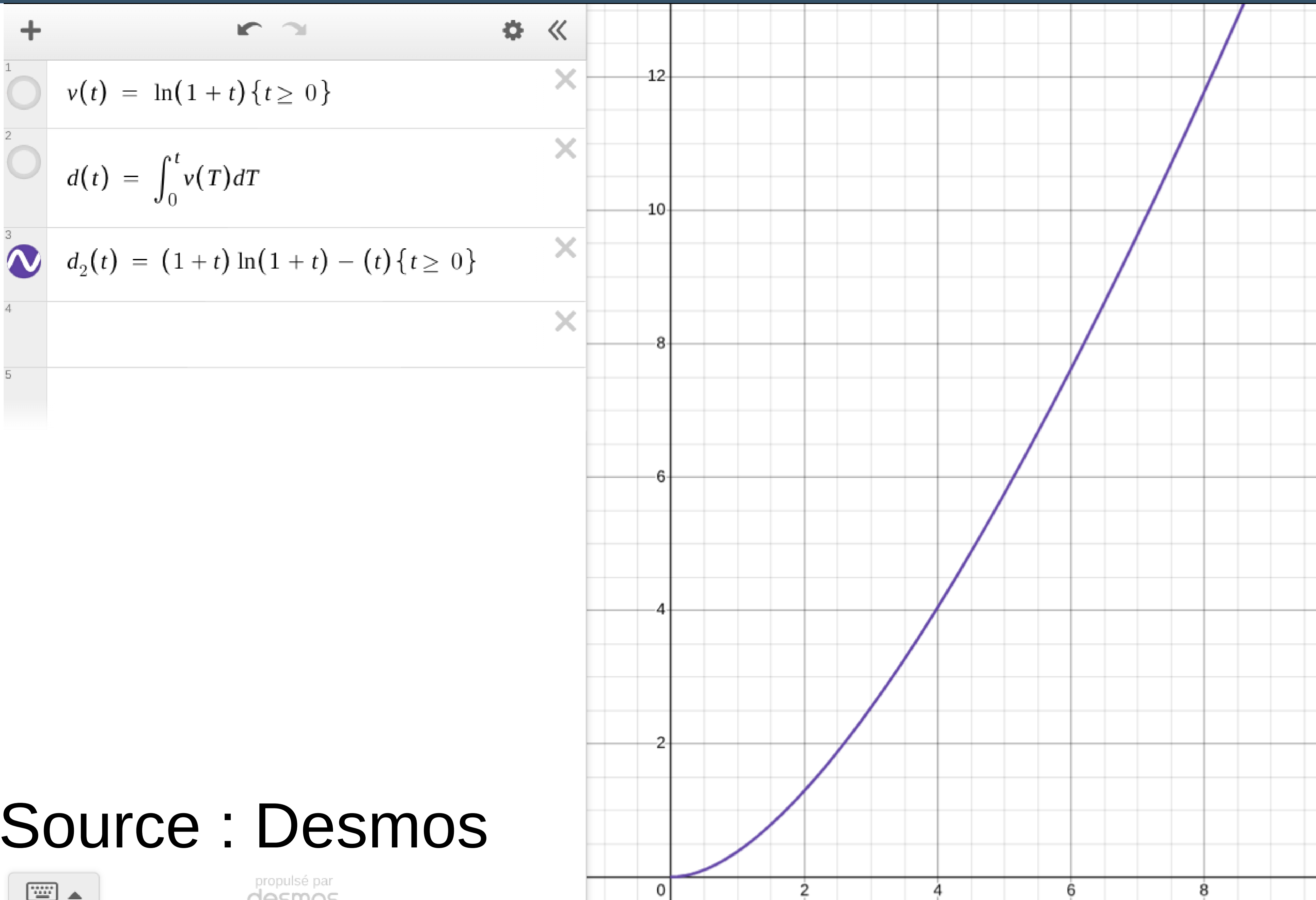
Source : Desmos

Introduction



Source : Desmos

Introduction



Source : Desmos

Problématique

Dans quelle mesure la mise en place d’une procédure de recherche de primitive exacte est-elle pertinente ?

Sommaire

- Les fonctions
- La théorie
- L’algorithme de Risch
- Les problèmes liés aux systèmes de calcul formel
- Résultats
- Les calculatrices actuelles
- Conclusions
- Annexe

Les fonctions

Des fonctions élémentaires

$$\frac{7x^5 + 4x^4 - 9x^2 + 1}{x^4 + 9}$$

$$\frac{e^{\tan x}}{1 + x^2} \sin\left(\sqrt{1 + (\ln x)^2}\right)$$

Des fonctions non élémentaires

$$\mathrm{li}(x) = \int_0^x \frac{\mathrm{d}t}{\ln(t)}.$$

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, \mathrm{d}t.$$

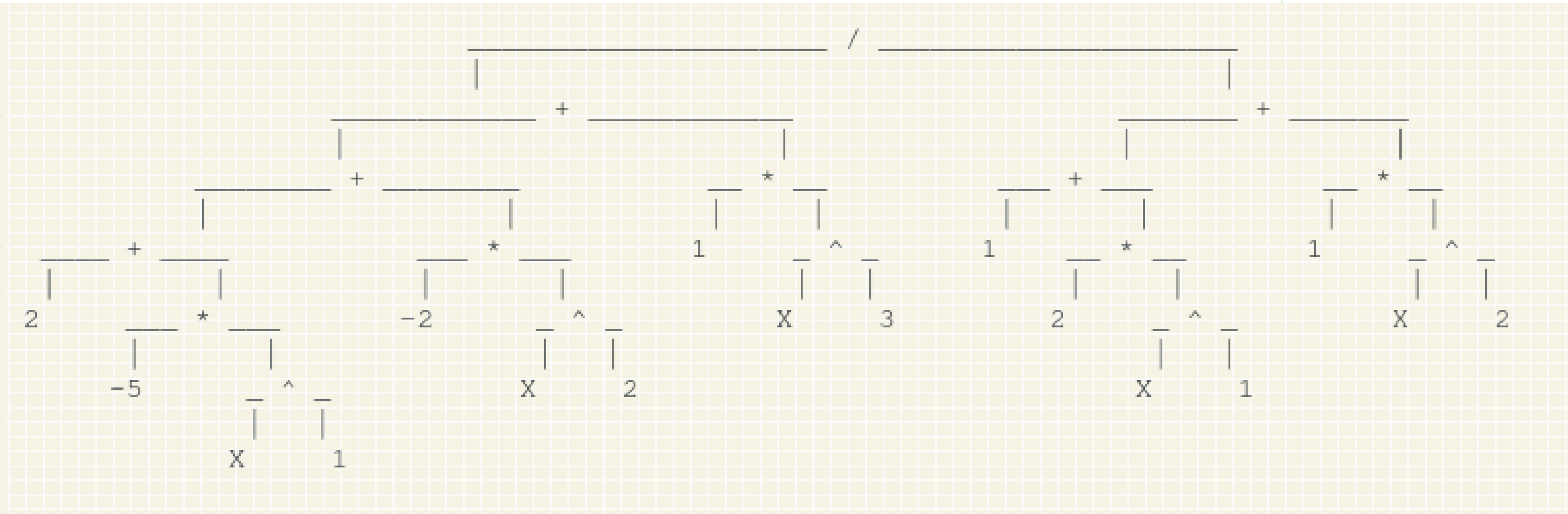
Les fonctions

Type des fonctions élémentaires de base

```
(* type constante exacte ancien type constante complexe = { re : float; im : float;};; *)
(* type rationnel et extensions algebriques *)
type constante = Q of {a : int; b : int;} | E of {nom : string; approx : float};;


(* type des fonctions élémentaires de base *)
type f_elem = Exp | Ln | X | Z | C of constante;;

(* type contenant les opérations élémentaires *)
type op_elem = Plus | Moins | Fois | Divise | Puissance | Compose;;

(* type contenant la structure des fonctions : a symbolic tree *)
type ast_elem =
  | Arg0 of f_elem
  | Arg2 of (ast_elem * op_elem * ast_elem)
  | Abstrait of fonction_abstraite
  | P of (ast_elem * (ast_elem array))
  | F of (ast_elem * (ast_elem array) * (ast_elem array))

(* type fonction abstraite nom;fonction;deriver n ieme;n le nombre de dérivation depuis fonction *)
and fonction_abstraite = {nom: string; fonction: ast_elem; d_fonction: ast_elem; etat_derive: int};;
```

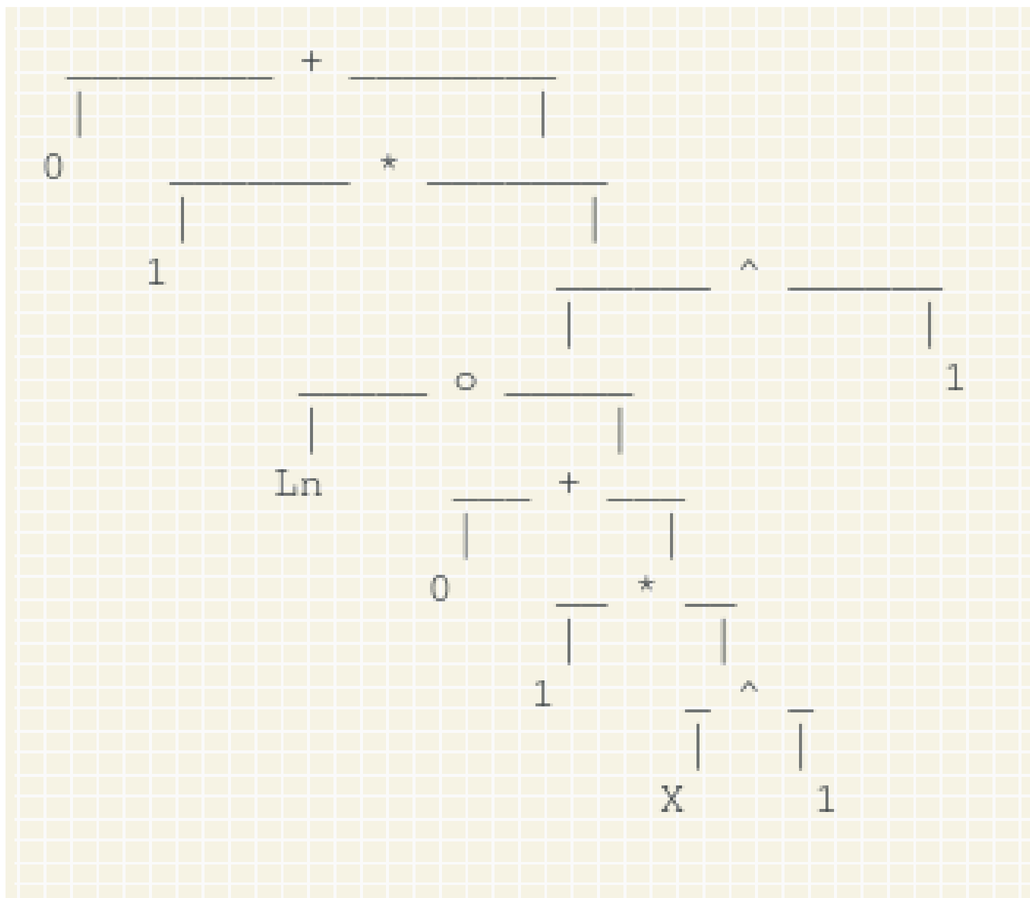
Les fonctions

```
let f2 = F (Arg0 X,
            [|(ast_const 2 1);(ast_const (-5) 1);(ast_const (-2) 1);(ast_const 1 1)|],
            [|(ast_const 1 1);(ast_const 2 1);(ast_const 1 1)|])
in
```



Les fonctions

```
let ln = P (Arg2 (Arg0 Ln,
                  Compose,
                  P (Arg0 X, [|ast_const 0 1;ast_const 1 1|])),
            [|ast_const 0 1;ast_const 1 1|])
;;
```



La théorie

Un corps différentiel est un corps commutatif F muni d'une application D (ou ∂ ou $\cdot'$) de F dans F tel que les conditions suivantes soient satisfaites :

(a) $\forall u, v \in F,\ D(u+v) = D(u) + D(v)$

(b) $\forall u, v \in F,\ D(u \cdot v) = D(u) \cdot v + D(v) \cdot u$

D est appelé une dérivation ou un opérateur différentiel.

On note $Con(F)$ le noyau de D, soit $Con(F) = \{c \in F \,/\, D(c) = 0\}$. $Con(F)$ est appelé le corps des constantes et est un sous-corps de F.

La théorie

Soit $\theta \in G$ où G est une extension de corps différentiel de F.

Alors θ est logarithmique sur F si et seulement si il existe un élément $u \in F$ tel que $D(\theta) = \frac{D(u)}{u}$.
On note alors $\theta = \log(u)$.

Et θ est exponentiel sur F si et seulement si il existe un élément $u \in F$ tel que $D(u) = \frac{D(\theta)}{\theta}$.
On note alors $\theta = \exp(u)$.

La théorie

Théorème de Liouville-Rosentlicht :

Soit F et G une extension élémentaire de F de même corps de constantes.

Soient $f \in F$ et $g \in G$, supposons que $g = \int f$ (i.e. $D(g) = f$).

Alors il existe $v_0, v_1, \ldots, v_n \in F$ et $c_1, \ldots, c_n \in Con(F)$ tel que,

$$f = v_0' + \sum_{i=1}^{n} c_i \frac{v_i'}{v_i}$$

autrement dit tel que,

$$\int f = v_0 + \sum_{i=1}^{n} c_i \cdot ln(v_i)$$

La théorie

Factorisation sans carré

$$P(X) = \prod_{k=1}^{n} A_k^k(X)$$

(où pour tout $k \in [|1;n|]$, A_k est soit un polynôme sans carré soit 1. De plus tous les polynômes sans carré différent de 1 sont premiers entre eux.)

Un exemple :

Factorisation de $P(X) = 4X^4 - 36X^2 + 16X + 48$

Factorisation par les racines : $P(X) = 4(X+1)(X+3)(X-2)^2$

Factorisation sans carré : $P(X) = 4(X^2+4X+3)(X-2)^2$

Algorithm 1 Risch

```
procedure RISCH(f, ext :: lst)    ▷ avec f ∈ 𝕂[X][ext :: lst] et ext la dernière extension

    f ← normalise(f, ext)

    if type(ext) = rationnel then
        return integration_rationnel(f)

    else if type(ext) = logarithmique then
        return integration_logarithmique(f, lst)    ▷ appel à Risch récursif dans ce cas

    else
        return Non_implémenté
    end if

end procedure
```

L’algorithme de Risch

Algorithm 2 Méthode d’Hermite

procedure HERMITE($F(\theta)$)
    $P \leftarrow partie\_polynomial(F)$
    $G \leftarrow partie\_fraction(F)$
    $FSC \leftarrow factorisation\_sans\_carré(G)$
    $DSC \leftarrow décomposition\_sans\_carré(G, FSC)$
    $FI, FL \leftarrow 0, 0$

    for $H \in DSC$ **do**
        $A, Q \leftarrow H = \dfrac{A}{Q^i}$
        if Si $i = 1$ **then**
            $FL \leftarrow FL + H$

        else
            $i \leftarrow i - 1$
            $S, T \leftarrow euclide\_étendu(Q, A)$ ▷ $S \cdot Q + T \cdot Q' = A$ possible car $Q \wedge Q' = 1$
            $FI \leftarrow FI - \dfrac{T}{i \cdot Q^i}$ ▷ Intégration par partie
            $DSC \leftarrow \dfrac{i \cdot S + T'}{i \cdot Q^i} :: DSC$
        end if
    end for
    return (P, FI, FL)
end procedure

L'algorithme de Risch

Algorithm 3 Méthode de Rothstein Trager

procedure ROTHSTEIN-TRAGER($F(\theta) = \dfrac{A(\theta)}{B(\theta)}$) ▷ B unitaire et sans carré et $deg(A) < deg(B)$

    $R(Z), réussite \leftarrow partie\_primitive(résultante(A(\theta) - Z \cdot B(\theta)', B(\theta)))$
    $[R_1; \ldots; R_n] \leftarrow factorisation(R(Z))$ ▷ Si possible

    if $réussite$ **then**
        $L \leftarrow 0$
        for $i = 1$ to n **do**
            $[\alpha_1; \ldots; \alpha_{deg(R_i)}] \leftarrow racines(R_i)$
            for $j = 1$ to $deg(R_i)$ **do**
                $L \leftarrow L + \alpha_j \cdot ln(PGCD(A(\theta) - \alpha_j \cdot B(\theta)', B(\theta)))$
            end for
        end for
        return L
    else
        return Primitive non élémentaire
    end if

end procedure

Intégration de polynômes logarithmiques récursivement

Soit θ une extension logarithmique et

$$f = \sum_{i=0}^{n} p_i \theta^i$$

Alors, si $\int f$ élémentaire dans $\mathbb{K}[\theta]$

$$\sum_{i=0}^{n} p_i \theta^i = \left(\sum_{i=0}^{n+1} q_i \theta^i \right)' + \sum_{j=0}^{m} c_j \frac{v_j'}{v_j} \iff \begin{cases} 0 = q_{n+1}' \\ p_n = (n+1) q_{n+1} \theta' + q_n' \\ \quad \vdots \\ p_1 = 2 q_2 \theta' + q_1' \\ p_0 = q_1 \theta' + q_0' + \sum_{j=0}^{m} c_j \frac{v_j'}{v_j} \end{cases}$$

Avec $q_0, \ldots, q_{n+1}, v_0, \ldots, v_m \in \mathbb{K}$

Intégration de polynômes logarithmiques récursivement

Si $f = ln(x) = \theta$

On a en supposant $\int f$ élémentaire : $\theta = (\lambda\theta^2 + \mu\theta + \nu)' + \sum_{j=0}^{m} c_j \frac{v_j'}{v_j}$

L'algorithme de Risch

Intégration de polynômes logarithmiques récursivement

Si $f = ln(x) = \theta$

On a en supposant $\int f$ élémentaire : $\theta = (\lambda\theta^2 + \mu\theta + \nu)' + \sum_{j=0}^{m} c_j \frac{v_j'}{v_j}$

$$\text{Donc} \begin{cases} 0 = \lambda' \\ 1 = 2\lambda\theta' + \mu' \\ 0 = \mu\theta' + \nu' + \sum_{j=0}^{m} c_j \frac{v_j'}{v_j} \end{cases}$$

L’algorithme de Risch

Intégration de polynômes logarithmiques récursivement

$$\text{Donc} \begin{cases} \lambda = 0 \\ 1 = \mu' \\ 0 = \mu\theta' + \nu' + \sum_{j=0}^{m} c_j \frac{v_j'}{v_j} \end{cases}$$

Intégration de polynômes logarithmiques récursivement

$$\text{Donc} \begin{cases} \lambda = 0 \\ \mu = x \\ 0 = 1 + \nu' + \sum_{j=0}^{m} c_j \frac{v_j'}{v_j} \end{cases}$$

L'algorithme de Risch

Intégration de polynômes logarithmiques récursivement

$$\text{Donc} \begin{cases} \lambda = 0 \\ \mu = x \\ \nu = -x \\ \sum_{j=0}^{m} c_j \cdot ln(v_j) = 0 \end{cases}$$

L'algorithme de Risch

Intégration de polynômes logarithmiques récursivement

Si $f = ln(x) = \theta$

On a en supposant $\int f$ élémentaire : $\theta = (\lambda\theta^2 + \mu\theta + \nu)' + \sum_{j=0}^{m} c_j \frac{v'_j}{v_j}$

Donc $\int f = \lambda\theta^2 + \mu\theta + \nu + \sum_{j=0}^{m} c_j \cdot ln(v_j) = x \cdot ln(x) - x$

Les problèmes liés aux systèmes de calcul formel

Problèmes de détection des zéros, de factorisation, simplification et représentation

Si
$$\begin{cases} 1+x \neq x+1 \\ (1+x)^2 \neq 1+2x+x^2 \\ x-x \neq 0 \\ F(x) = \frac{1+x}{1} \neq 1+x = P(x) \end{cases}$$

Alors que choisir ? $(1+x)^{42}$ ou $1+42x+\ldots+x^{42}$

Les problèmes liés aux systèmes de calcul formel

Les choix faits :

```
(* Simplifie l'addition *)
| Arg2 (Arg0 (C (Q {a = 0; b = 1})), Plus, f4) -> f4
| Arg2 (f3 , Plus, Arg0 (C (Q {a = 0; b = 1}))) -> f3
| Arg2 (Arg0 (C x), Plus, Arg0 (C y)) -> Arg0 (C (add_constante x y))
| Arg2 (f1, Plus, f2) when egal_ast f1 f2 -> Arg2 (Arg0 (C (Q {a = 2; b = 1})), Fois, f1)
| Arg2 (P (x,a), Plus, P (y,b)) when egal_ast x y -> add_poly (P (x,a)) (P (y,b))
| Arg2 (F (x,a,b), Plus, F (y,c,d)) when egal_ast x y -> add_frac (F (x,a,b)) (F (y,c,d))
(* Simplifie la soustraction *)
| Arg2 (Arg0 (C (Q {a = 0; b = 1})), Moins, f4) -> Arg2 (ast_minus_un, Fois, f4)
| Arg2 (f3, Moins, Arg0 (C (Q {a = 0; b = 1}))) -> f3
| Arg2 (Arg0 (C x), Moins, Arg0 (C y)) -> Arg0 (C (minus_constante x y))
| Arg2 (f3, Moins, Arg0 (C x)) -> Arg2 (f3, Plus, Arg0 (C (neg_constante x)))
| Arg2 (f1, Moins, f2) when egal_ast f1 f2 -> ast_null
| Arg2 (P (x,a), Moins, P (y,b)) when egal_ast x y -> minus_poly (P (x,a)) (P (y,b))
| Arg2 (F (x,a,b), Moins, F (y,c,d)) when egal_ast x y -> minus_frac (F (x,a,b)) (F (y,c,d))
(* Simplifie la multiplication *)
| Arg2 (Arg0 (C (Q {a = 0; b = 1})) ,Fois , f4) -> ast_null
| Arg2 (f3 ,Fois , Arg0 (C (Q {a = 0; b = 1}))) -> ast_null
| Arg2 (Arg0 (C x) ,Fois , Arg0 (C y)) -> Arg0 (C (mult_constante x y))
| Arg2 (Arg0 (C (Q {a = 1; b = 1})) ,Fois , f4) -> f4
| Arg2 (f3 ,Fois , Arg0 (C (Q {a = 1; b = 1}))) -> f3
| Arg2 (P (_, [|Arg0 (C (Q {a = 1; b = 1}))|]) ,Fois , f4) -> f4
| Arg2 (f3 ,Fois , P (_, [|Arg0 (C (Q {a = 1; b = 1}))|])) -> f3
| Arg2 (f1, Fois, f2) when egal_ast f1 f2 -> Arg2 (f1, Puissance, Arg0 (C (Q {a = 2; b = 1})))
```

Les problèmes liés aux systèmes de calcul formel

Les choix faits :

```ocaml
and is_zero_ast ast_arbre =
  (* test de manière imprécise si une expression est nul, rique d'échec massif du à l'implémentation *)
  let fonction = build (apt_of_ast ast_arbre) in
  let compte = ref 0 in
  let test = ref true in
  while !compte < int_of_float (10.**3.) && !test do
    compte := !compte + 1;
    let z1 = E {nom = "z1" ; aprox = Random.float (707.48)} in
    let f1 = fonction z1 in

    (*
      OCaml's floating-point numbers follow the IEEE 754 standard, using double precision (64 bits) numbers.
      binary64  Double precision    1.80*10^308 max
      donc le float max en évaluation pour ne pas obtenir nan avec exp et inférieur à 707.48
    *)

    let valeur = match abs_constante f1 with | Q q -> c_q_to_e (Q q) | E e -> e.aprox in

    if valeur > seuil_zero
      then (
        test := false
      )
  done;
  !test && (not (appartient_f_elem ast_arbre X && appartient_f_elem ast_arbre Z))
```

Les problèmes liés aux systèmes de calcul formel

Une gestion des nombres compliquée

```
(* type constante exacte ancien type constante complexe = { re : float; im : float;};; *)
(* type rationnel et extensions algebriques *)
type constante = Q of {a : int; b : int;} | E of {nom : string; approx : float};;
```

```
(* seuil auquel on considère qu'un nombre est nul*)
let seuil_zero = 1e-10 ;;

(* seuil max *)
let seuil_max_int = max_int/100;;

exception Int_overflow;;
```

Les problèmes liés aux systèmes de calcul formel

Les problèmes de priorités des opérateurs

$1 + (2 \times 3) = 7$ $(1 + 2) \times 3 = 9$

```
1+(x+((x^3)+(x^5)))
```

Résultats

Algorithme de Risch

```
ln(x)
```

```
          -------------- + --------------
          |                             |
 ---- + ----                    -------- * --------
 |         |                    |                 |
0     --- * ---          --- + ---               --- ^ ---
      |       |          |       |               |       |
     -1      _ ^ _      0    __ * __         __ o __      1
             |   |           |     |         |     |
            X     1         1     _ ^ _     Ln      X
                                  |   |
                                 X     1
```

Résultats

Tests sur 1000 fractions rationnelles avec des pôles de degrés différents (et comptés avec multiplicité)

Avec un seul pôle

```
[Running] ocaml "/home/arthur/Code/TIPE_mania/Risch.ml"
1000    Nombre de test
0       Nombre d’échecs
1000    Nombre de réussite

[Done] exited with code=0 in 181.238 seconds
```

Avec deux pôles

```
[Running] ocaml "/home/arthur/Code/TIPE_mania/Risch.ml"
1000    Nombre de test
138     Nombre d’échecs
27      Nombre de réussite

[Done] exited with code=0 in 312.923 seconds
```

Résultats

Avec cinq pôles maximum

```
[Running] ocaml "/home/arthur/Code/TIPE_mania/Risch.ml"
1000
505
258

[Done] exited with code=0 in 312.391 seconds
```

Nombre de test
Nombre d'échecs
Nombre de réussite

1) 170385721
2) 10780810
3) 44647356086950400

1) Nombre d'appels à la fonction de simplification de constante

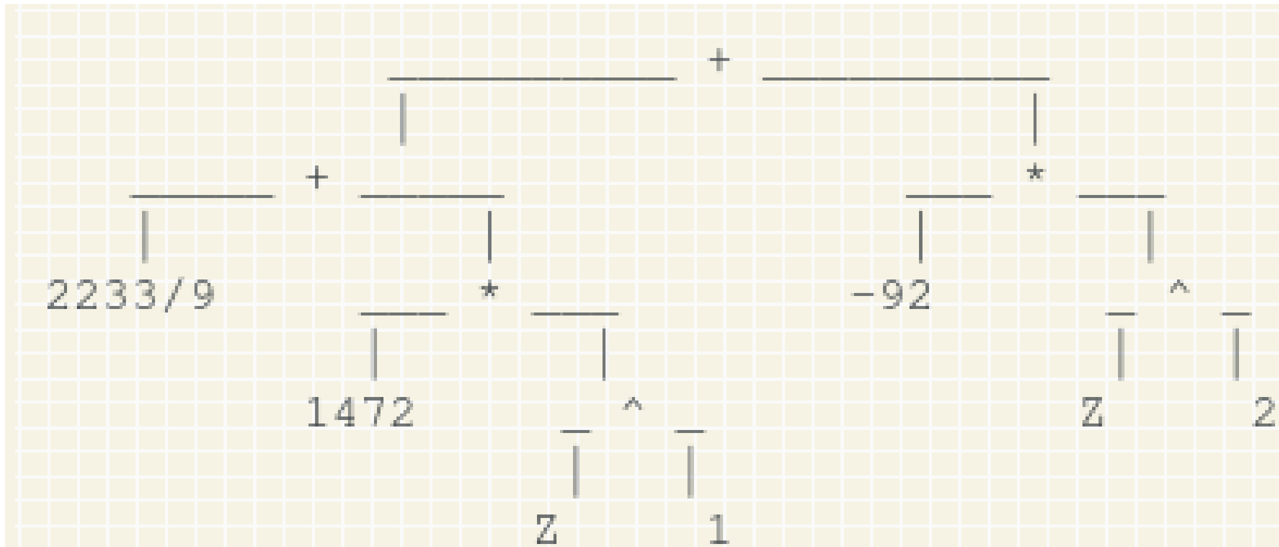
2) Nombre d'appels à la fonction de simplification de fonction

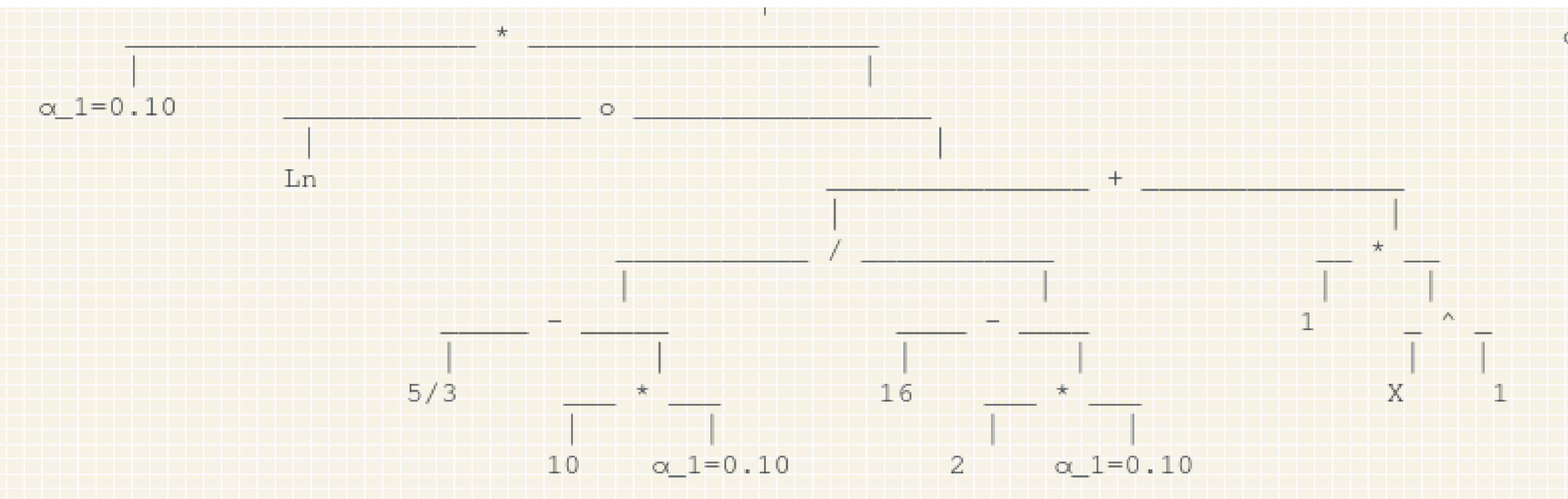
3) Plus grand int enregistré

Et un échec est un Int_overflow

Une réussite non vérifiable est :

```
5
- + (16) * ((X) ^ (1))
3
-------------------------------------------
2 + (10) * ((X) ^ (1)) + (1) * ((X) ^ (2))
```


+
+
*
2233/9
*
-92
^
1472
^
Z
2
Z
1


*
α_1=0.10
o
Ln
+
/
*
-
-
1
^
5/3
*
16
*
X
1
10
α_1=0.10
2
α_1=0.10

Résultats

Algorithme de Risch

```
1/(ln(x))

Pas de primitive elementaire
```

WolframAlpha

$$\int \frac{1}{\ln(x)}\, d\,x$$

NATURAL LANGUAGE MATH INPUT

Indefinite integral

$$\int \frac{1}{\log(x)}\, dx = \mathrm{li}(x) + \text{constant}$$

Résultats

Algorithme de Risch

```
ln(ln(x))
```

```
Pas de primitive elementaire
```

WolframAlpha

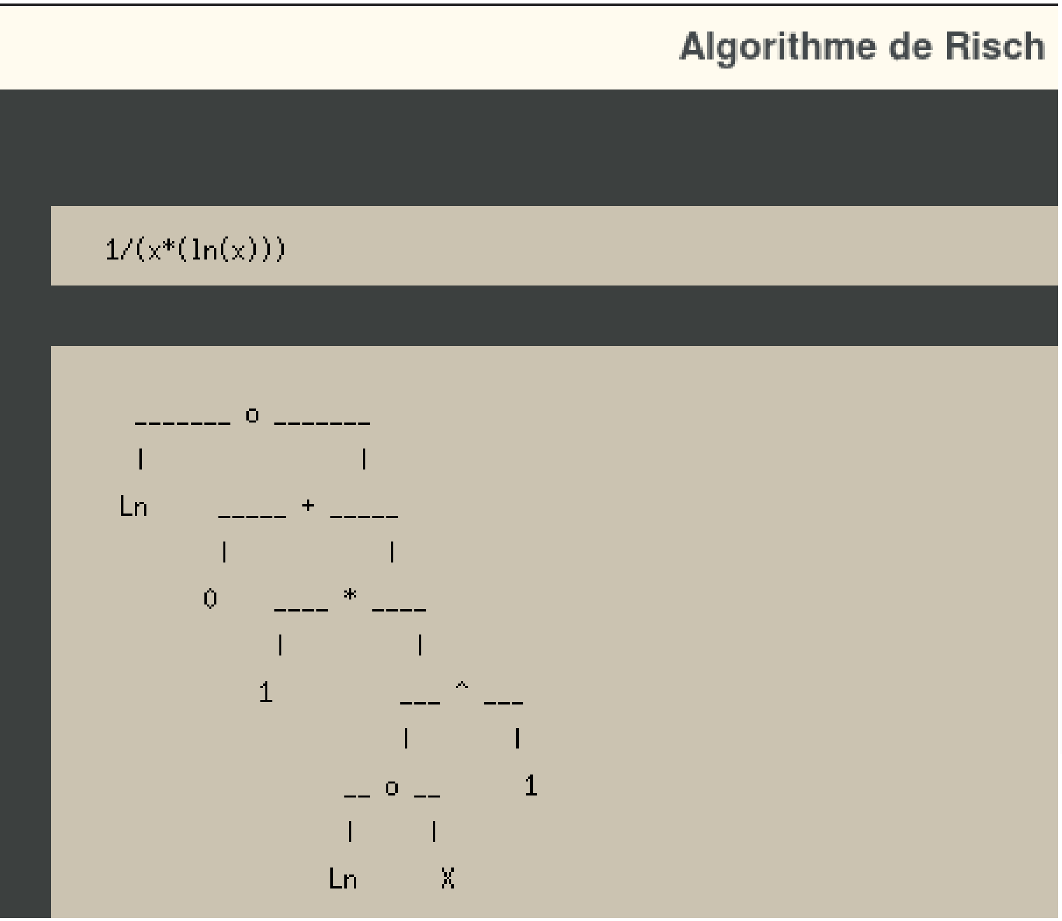
$\int \ln(\ln(x))\, dx$

NATURAL LANGUAGE | MATH INPUT

Indefinite integral

$$\int \log(\log(x))\, dx = x \log(\log(x)) - \mathrm{li}(x) + \text{constant}$$

Résultats



Algorithme de Risch
1/(x*(ln(x)))
_______ o _______
| |
Ln   _____ + _____
        | |
        0   ____ * ____
             | |
             1   ___ ^ ___
                        | |
                     __ o __   1
                     | |
                     Ln   X

WolframAlpha
$\int \frac{1}{x*\ln(x)} \, d\,x$
NATURAL LANGUAGE
MATH INPUT
Indefinite integral
$\int \frac{1}{x \log(x)} \, dx = \log(\log(x)) + \text{constant}$

Résultats

```
Ln o (0 + (1) * ((X) ^ (1))) + Ln o (---- + (1) * ((Ln o (0 + (1) * ((X) ^ (1)))) ^ (1)))
                                    ( X                                                 )

           ___________________ + ___________________
           |                                       |
 _____ o _____               ____________ o ____________
 |           |               |                         |
Ln     ___ + ___            Ln         __________ + __________
       |       |                       |                     |
      0    __ * __                  _ / _      _______ * _______
           |     |                  |   |      |               |
          1     _ ^ _              -1    X    1               ______ ^ ______
                |   |                                         |             |
               X     1                              _____ o _____            1
                                                    |           |
                                                   Ln     ___ + ___
                                                          |       |
                                                         0    __ * __
                                                              |     |
                                                             1     _ ^ _
                                                                   |   |
                                                                  X     1
```

Résultats

WolframAlpha

$$\int \frac{1+\ln(x)}{x\ln(x)-1}\, d\, x$$

NATURAL LANGUAGE | MATH INPUT

Indefinite integral

$$\int \frac{1+\log(x)}{x\log(x)-1}\, dx = \log\big(x\log(x)-1\big) + \text{constant}$$

(assuming a complex-valued logarithm)

```
1 + (1) * ((Ln o (0 + (1) * ((X) ^ (1)))) ^ (1))
----------------------------------------------------------------------------------------------------------------
1 + ((2) * (X)) * ((Ln o (0 + (1) * ((X) ^ (1)))) ^ (1)) + ((X) ^ (2)) * ((Ln o (0 + (1) * ((X) ^ (1)))) ^ (2))


( )   (              (              ( (2) * (X)    ))                                                                              )
( )   (              (              ( -----------  ))                                                                              )
( )   ( (2) * (X)    ( (2) * (X)    ( (X) ^ (2)    ))                                                                              )
( )   ( ----------- - (----------- - (-------------))                                                                              )
( )   ( (X) ^ (2)    ( (X) ^ (2)    ( 2            ))     -1                                                                       )
( )   ( ----------------------------------------------- + -----------                                                              )
( )   ( (X) ^ (2)                                         (X) ^ (2)                                                                )
( )   ( ---------------------------------------------------------------                                                            )
( )   ( -1                                                                                                                         )
(-1) * (----------------------------------------------------------------------------------------------------------------------------)
( )   (                                            ( ((2) * ((X) ^ (2)) - (((2) * (X)) ^ (2)) )   ( )   )                          )
( )   (                                            ( (-----------------------------------------) * (2)  )                          )
( )   ( (2) * ((X) ^ (2)) - (((2) * (X)) ^ (2))    ( ((X) ^ (4)                               )   ( )   )   1                      )
( )   ( ----------------------------------------- - (---------------------------------------------------) + ---------------------  )
( )   ( (X) ^ (4)                                  ( 4                                                  )   0 + (1) * ((X) ^ (1))  )
---------------------------------------------------------------------------------------------------------------------------------------
             ( (2) * (X)    )
             ( -----------  )
(2) * (X)    ( (X) ^ (2)    )
----------- - (-------------) + (1) * ((Ln o (0 + (1) * ((X) ^ (1)))) ^ (1))
(X) ^ (2)    ( 2            )
```

Résultats

WolframAlpha

$$\int \frac{1+\ln(x)}{(1+x\ln(x))^2}\, d x$$

NATURAL LANGUAGE

MATH INPUT

Indefinite integral

$$\int \frac{1+\log(x)}{(1+x\log(x))^2}\, dx = -\frac{1}{x\log(x)+1} + \text{constant}$$

(assuming a complex-valued logarithm)

Les calculatrices actuelles

- Recherches de résultats en mémoire
- Approximation par des sommes de Riemann ou des variantes.
- Utilisation d’une procédure de recherche de primitive
- Transformation à l’aide des outils mathématiques connus et d’heuristiques

Conclusion

- Un système de calcul formel nécessite un travail immense pour des opérations élémentaires
- La représentation des objets mathématiques joue un rôle capital dans les calculs
- Il est nécessaire de trouver un juste milieu entre exactitude, précision et les complexités algorithmiques

Annexe : Code

```
(* --------------- Entête --------------- *)

(* #require bigdecimal;; *)
#use "topfind";;
#require "graphics";;

(* seuil auquel on considére qu'un nombre est nul*)
let seuil_zero = 1e-10 ;;

(* seuil max *)
let seuil_max_int = max_int/100;;

exception Int_overflow;;


(* pour des statistiques simplification constante 0 - simplification ast 1 - int max 2 *)
let appel_tab = [|0;0;0|];;
Random.self_init ();;





(* --------------- Déclaration des types --------------- *)



(* type constante exacte ancien type constante complexe = { re : float; im : float;};; *)
(* type rationnel et extensions algebriques *)
type constante = Q of {a : int; b : int;} | E of {nom : string; approx : float};;


(* type des fonctions élémentaires de base *)
type f_elem = Exp | Ln | X | Z | C of constante;;

(* type contenant les opérations élémentaires *)
type op_elem = Plus | Moins | Fois | Divise | Puissance | Compose;;

(* type contenant la structure des fonctions : a symbolic tree *)
type ast_elem =
  | Arg0 of f_elem
  | Arg2 of (ast_elem * op_elem * ast_elem)
  | Abstrait of fonction_abstraite
  | P of (ast_elem * (ast_elem array))
```

```
  | F of (ast_elem * (ast_elem array) * (ast_elem array))

(* type fonction abstraite nom;fonction;deriver n ieme;n le nombre de dérivation depuis fonction *)
and fonction_abstraite = {nom: string; fonction: ast_elem; d_fonction: ast_elem; etat_derive: int};;


(* implémentation pratique apt = a praticopratique tree *)
type apt = Fonction of (constante -> constante) | Node of (apt * ((constante -> constante) -> (constante -> constante) -> 
 constante -> constante) * apt);;


(* type extension de corps différentiel, Exp Ln X ou algebric *)
type corps_diff = Xe | Ext of (f_elem * fonction_abstraite * corps_diff);;


(* type pour echec renvoie vide ou branche non implementer *)
type option = Res of ast_elem | Null | Notimplementederror;;


let file = open_out "output.txt";;

(*voir verbatim pour fichier *)






(* --------------- Déclaration des Constantes et Ast fondamentaux --------------- *)



let c_zero = Q {a = 0; b = 1};;
let c_un = Q {a = 1; b = 1};;
let c_minus_un = Q {a = -1; b = 1};;
let c_const nom den = Q {a = nom; b =  den};;
let c_ext ext a = E {nom = ext; approx = a};;


let ast_null = Arg0 (C (Q {a = 0; b = 1}));;
let ast_un = Arg0 (C (Q {a = 1; b = 1}));;
let ast_minus_un = Arg0 (C (Q {a = -1; b = 1}));;


let ast_const a b = assert (b <> 0); Arg0 (C (c_const a b));;
```

```
let ast_x = Arg0 X;;
let ast_Z = Arg0 Z;;
let ast_ln = Arg0 Ln;;
let ast_exp = Arg0 Exp;;


let ast_plus ast_1 ast_2 = Arg2 (ast_1, Plus, ast_2);;
let ast_moins ast_1 ast_2 = Arg2 (ast_1, Moins, ast_2);;
let ast_fois ast_1 ast_2 = Arg2 (ast_1, Fois, ast_2);;
let ast_divise ast_1 ast_2 = Arg2 (ast_1, Divise, ast_2);;
let ast_puissance ast_1 ast_2 = Arg2 (ast_1, Puissance, ast_2);;
let ast_compose ast_1 ast_2 = Arg2 (ast_1, Compose, ast_2);;





(* --------------- Déclaration des fonctions --------------- *)



(* rajouter les nombres négatifs *)
type token = Pl | Mo | Pu | Di | Fo | Num of int | Log | Expo | PaG | PaD | XX ;;


let tok_of_string s = match s with
  | "+" -> [Pl]
  | "-" -> [Mo]
  | "/" -> [Di]
  | "^" -> [Pu]
  | "*" -> [Fo]
  | "(" -> [PaG]
  | ")" -> [PaD]
  | "ln" -> [Log]
  | "exp" -> [Expo]
  | "x" -> PaG::XX::[PaD]
  | _ -> ( match int_of_string_opt s with
    | Some i -> PaG::(Num i)::[PaD]
    | None -> failwith "tok_of_string : caractère non reconnu"
  )
;;
```

```

let analyse_lexicale (texte:string) =
  let re = Str.regexp "\\+\\|-\\|/\\|\\^\\|\\*\\|(\\|)\\|ln\\|exp\\|x\\|[0-9]+" in (* regexp *)
  let rec tokenize texte i = (* renvoie la liste de tokens pour texte, pris à partir de l'indice i *)
    if i = String.length texte then
      []
    else
      try
        let j = Str.search_forward re texte i in
        let str = Str.matched_string texte in
        if str = " " || str = "\n" then
          tokenize texte (j + String.length str)
        else
          (tok_of_string str) @ (tokenize texte (j + String.length str))
      with
        Not_found -> []
  in
  tokenize texte 0
;;

let rec extract_paranthese lst i =
    match lst with
      | x::llst -> (
          match x with
          | PaD -> if i-1 = 0 then [],llst else (let lint,lext = extract_paranthese llst (i-1) in x::lint,lext)
          | PaG -> (let lint,lext = extract_paranthese llst (i+1) in x::lint,lext)
          | _ -> (let lint,lext = extract_paranthese llst i in x::lint,lext)
        )
      | [] -> failwith "extract_parenthese : mal parenthéser"
;;


let rec parse lst = match lst with
  | PaG::llst -> (
      let lint,lext = extract_paranthese llst 1 in
      match lext with
      | [] -> parse lint
      | Pl::ls -> Arg2 (parse lint, Plus, parse ls)
      | Pu::ls -> Arg2 (parse lint, Puissance, parse ls)
      | Mo::ls -> Arg2 (parse lint, Moins, parse ls)
      | Fo::ls -> Arg2 (parse lint, Fois, parse ls)
      | Di::ls -> Arg2 (parse lint, Divise, parse ls)
      | _ -> failwith "parse : mauvais syntaxe"
    )
```

```
  | [Num i] -> ast_const i 1

  | [XX] -> Arg0 X

  | Log::llst -> Arg2 (Arg0 Ln, Compose, parse llst)

  | Expo::llst -> Arg2 (Arg0 Exp, Compose, parse llst)

  | _ -> failwith "parse : fail"
;;





(* -------------- Fonctions utilitaires pour l'affichage -------------- *)



let print_debug n = Printf.fprintf file " endroit a regarder indice %d \n" n
;;


let get_constante_string (c:constante) = match c with
  | Q rationnel -> (Printf.sprintf "%d" rationnel.a) ^ (if rationnel.b = 1 then "" else "/" ^ (Printf.sprintf "%d" rationnel.b))
  | E extension -> (Printf.sprintf "%s" extension.nom) ^ "=" ^ (Printf.sprintf "%.2f" extension.approx)
;;


let rec repete n c = if n <= 0 then "" else c ^ (repete (n-1) c)
;;


(* cree un bloc correspondant à une valeur ainsi que les deux blocs des sous-arbres *)
let fusionne v (l1, b1) (l2, b2) =
  let nb_lignes = max (Array.length b1) (Array.length b2) in
  let block = Array.make (nb_lignes + 2) "" in
  let entete = repete ((l1+1) / 2) " " ^ repete ((l1+l2+2) / 4) "_" ^ v ^ repete ((l1+l2+2) / 4) "_" ^ repete ((l2+1) / 2 ) " " in
  let largeur = String.length entete in
  block.(0) <- entete;
  block.(1) <- repete ((l1+1) / 2) " " ^ "|" ^ repete (largeur - (l1+1) / 2 - (l2+1) / 2 - 2) " " ^ "|" ^ repete ((l2+1) / 2) " ";
```

```
  assert (String.length block.(0) = String.length block.(1));
  for i = 0 to nb_lignes - 1 do
    if (i < Array.length b1 && i < Array.length b2) then
      block.(i+2) <- b1.(i) ^ repete (largeur - l1 - l2) " " ^ b2.(i)
    else if i < Array.length b1 then
      block.(i+2) <- b1.(i) ^ repete (largeur - l1) " "
    else if i < Array.length b2 then
      block.(i+2) <- repete (largeur - l2) " " ^ b2.(i)
  done;
  (largeur, block)
;;


let rec block_of_ast_arbre ast_arbre =

  let ast_of_poly_array_t poly indet =
    (* transforme un ast P en ça version ast *)
    let f = ref poly.(0) in
    for i = 1 to Array.length poly - 1 do
      f := Arg2 (!f, Plus, Arg2(poly.(i), Fois, Arg2 (indet, Puissance, ast_const i 1)))
    done;
    !f
  in

  match ast_arbre with
    | Arg0 x ->
        (match x with
         | C x ->
          (match x with
            | Q rationnel -> let s = get_constante_string x in (String.length s, [|s|])
            | E extension -> let s = (Printf.sprintf "%s" extension.nom) ^ "=" ^ (Printf.sprintf "%.2f" extension.approx)
             in (String.length s, [|s|])
          )
         | X -> (3, [|" X "|])
         | Z -> (3, [|" Z "|])
         | Ln -> (3, [|" Ln"|])
         | Exp -> (3, [|"Exp"|])
        )

    | Arg2 (f1, oper, f2) ->
        let str = ( match oper with
            | Plus -> " + "
            | Moins ->  " - "
            | Fois ->  " * "
```

```
            | Divise ->  " / "
            | Puissance ->  " ^ "
            | Compose ->  " o "
          ) in fusionne str (block_of_ast_arbre f1) (block_of_ast_arbre f2)

    | Abstrait f1 -> (String.length f1.nom, [|f1.nom|])

    | P (x,p) -> block_of_ast_arbre (ast_of_poly_array_t p x)

    | F (x,a,b) -> fusionne " / " (block_of_ast_arbre (ast_of_poly_array_t a x)) (block_of_ast_arbre (ast_of_poly_array_t b x))
;;


let print_ast_arbre ast_arbre =
  let (largeur, block) = block_of_ast_arbre ast_arbre in
  for i = 0 to Array.length block - 1 do
    (* print_endline block.(i) *)
    Printf.fprintf file "%s\n" block.(i)
  done
;;


let print_ast_arbre_graphics ast_arbre : unit =
  let (largeur, block) = block_of_ast_arbre ast_arbre in
  let x = Graphics.current_x () in
  for i = 0 to Array.length block - 1 do
    Graphics.moveto x (Graphics.current_y () - 20);
    Graphics.draw_string (Printf.sprintf "%s" block.(i));
  done;
  ()
;;


let rec block_of_ast_arbre_lineaire ast_arbre =

  let ast_of_poly_array_t poly indet =
    (* transforme un ast P en ça version ast *)
    let f = ref poly.(0) in
    for i = 1 to Array.length poly - 1 do
      f := Arg2 (!f, Plus, Arg2(poly.(i), Fois, Arg2 (indet, Puissance, ast_const i 1)))
    done;
    !f
  in
```

```

  match ast_arbre with
    | Arg0 x ->
        (match x with
         | C x ->
          (match x with
            | Q rationnel -> get_constante_string x
            | E extension -> extension.nom ^ "=" ^ (string_of_float extension.approx)
          )

         | X -> "X"
         | Z -> "Z"
         | Ln -> "Ln"
         | Exp -> "Exp"
        )

    | Arg2 (f1, oper, f2) ->
        let str = ( match oper with
            | Plus -> " + "
            | Moins ->  " - "
            | Fois ->  " * "
            | Divise ->  " / "
            | Puissance ->  " ^ "
            | Compose ->  " o "
          ) in "(" ^ (block_of_ast_arbre_lineaire f1) ^ ")" ^ str ^ "(" ^ (block_of_ast_arbre_lineaire f2) ^ ")"

    | Abstrait f1 -> f1.nom

    | P (x,p) -> block_of_ast_arbre_lineaire (ast_of_poly_array_t p x)

    | F (x,a,b) -> (
      let s1 =  (block_of_ast_arbre_lineaire (ast_of_poly_array_t a x)) ^ "\n" in
      let s2 = "\n" ^ (block_of_ast_arbre_lineaire (ast_of_poly_array_t b x)) in
      s1 ^ (String.make (max (String.length s1)  (String.length s2)) '_') ^ s2
    )
;;


let print_ast_arbre_lineaire ast_arbre = Printf.fprintf file "%s\n" (block_of_ast_arbre_lineaire ast_arbre)
;;


let print_ast ast =

```

```
  let ast_of_poly_array_t poly indet =
    (* transforme un ast P en ça version ast *)
    let f = ref poly.(0) in
    for i = 1 to Array.length poly - 1 do
      f := Arg2 (!f, Plus, Arg2(poly.(i), Fois, Arg2 (indet, Puissance, ast_const i 1)))
    done;
    !f
  in

  let rec explore_haut f =
    match f with
    | Arg0 x ->
      (match x with
       | C x -> (
          match x with
          | Q rationnel -> (if rationnel.b = 1 then (0,1) else (-1,2))
          | E extension -> (0,1)
        )
       | _ -> (0,1)
      )

    | Arg2 (f1, oper, f2) ->
      let (l1,h1) = explore_haut f1 in
      let (l2,h2) = explore_haut f2 in
      (
        match oper with
          | Divise -> (l2-h2,h1-l1+1)
          | _ ->  (min l1 l2, max h1 h2)
      )

    | Abstrait f1 -> (0,1)

    | P (x,p) -> explore_haut (ast_of_poly_array_t p x)

    | F (x,a,b) -> (
        explore_haut (Arg2 ((ast_of_poly_array_t a x),Divise,(ast_of_poly_array_t b x)))
      )
  in

  let (l,h) = explore_haut ast in

  let array_print = Array.make (h-l) " " in

  let rise_up_to n i =
```

```
    let m = String.length array_print.(i) in
    if m < n then array_print.(i) <- array_print.(i) ^ (String.make (n-m-1) ' ')
  in

  let rec aux_ast_arbre f haut =

    let ind = haut - l in (* indice correspondant dans le tableau *)

    match f with
      | Arg0 x ->
          (match x with
           | C x -> (
               match x with
               | Q rationnel -> (
                 if rationnel.b = 1
                   then array_print.(ind) <- array_print.(ind) ^ (Printf.sprintf "%d" rationnel.a)
                 else
                   (
                   rise_up_to (String.length array_print.(ind)) (ind+1);
                   rise_up_to (String.length array_print.(ind)) (ind-1);
                   let n = (max (String.length (Printf.sprintf "%d" rationnel.a)) (String.length (Printf.sprintf "%d" rationnel.b))) in
                   array_print.(ind+1) <- array_print.(ind+1) ^ (Printf.sprintf "%d" rationnel.a) ^ (String.make (n - (String.length (Printf.sprintf "%d" rationnel.a))) ' ');
                   array_print.(ind) <- array_print.(ind) ^ (String.make n '-');
                   array_print.(ind-1) <- array_print.(ind-1) ^ (Printf.sprintf "%d" rationnel.b) ^ (String.make (n - (String.length (Printf.sprintf "%d" rationnel.b))) ' ');
                   )
                 )

               | E extension -> array_print.(ind) <- array_print.(ind) ^ (Printf.sprintf "%s" extension.nom) ^ "=" ^ (Printf.sprintf "%.2f" extension.approx);
            )

           | X -> array_print.(ind) <- array_print.(ind) ^ "X";
           | Z -> array_print.(ind) <- array_print.(ind) ^ "Z";
           | Ln -> array_print.(ind) <- array_print.(ind) ^ "Ln";
           | Exp -> array_print.(ind) <- array_print.(ind) ^ "Exp";
          )

      | Arg2 (f1, oper, f2) -> (
        match oper with
          | Plus -> (
              aux_ast_arbre f1 haut;
```

```
                array_print.(ind) <- array_print.(ind) ^ " + ";
                aux_ast_arbre f2 haut;
              )

            | Divise ->  (
                let d1,m1 = explore_haut f1 in
                let d2,m2 = explore_haut f2 in

                let n = String.length array_print.(ind) in

                for i = ind-m2 to ind-d1+1 do
                  if i <> ind then rise_up_to (n + 1) i;
                done;
                aux_ast_arbre f1 (haut-d1+1);
                aux_ast_arbre f2 (haut-m2);

                array_print.(ind) <- array_print.(ind) ^ (String.make (max (String.length array_print.(ind-d1+1)) (String.
                length array_print.(ind-m2)) - n + 2) '-');
              )

            | Fois | Puissance ->  (
                let d1,m1 = explore_haut f1 in
                let d2,m2 = explore_haut f2 in
                let d,m = (min d1 d2,max m1 m2) in

                let n = ref ((String.length array_print.(ind)) + 1) in
                for i = ind+d to ind+m-1 do
                  rise_up_to !n i;
                  array_print.(i) <- array_print.(i) ^ "("
                done;

                aux_ast_arbre f1 haut;

                n := (String.length array_print.(ind));
                for i = ind+d to ind+m-1 do
                  rise_up_to !n i;
                  array_print.(i) <- array_print.(i) ^ ")"
                done;

                array_print.(ind) <- array_print.(ind) ^ (
                  match oper with
                  | Fois -> " * "
                  | Puissance -> " ^ "
                  | _ -> failwith "print_ast : n'arrive pas"
```

```
                );

                for i = ind+d to ind+m-1 do
                  rise_up_to (!n+4) i;
                  array_print.(i) <- array_print.(i) ^ "("
                done;

                aux_ast_arbre f2 haut;

                n := (String.length array_print.(ind));
                for i = ind+d to ind+m-1 do
                  rise_up_to !n i;
                  array_print.(i) <- array_print.(i) ^ ")"
                done;
              )

            | Moins | Compose -> (
              let d,m = explore_haut f2 in
              aux_ast_arbre f1 haut;
              array_print.(ind) <- array_print.(ind) ^ (
                match oper with
                | Moins -> " - "
                | Compose -> " o "
                | _ -> failwith "print_ast : n'arrive pas"
              );
              let n = String.length array_print.(ind) in
              for i = ind+d to ind+m-1 do
                rise_up_to n i;
                array_print.(i) <- array_print.(i) ^ "("
              done;
              aux_ast_arbre f2 haut;
              let n = String.length array_print.(ind)+1 in
              for i = ind+d to ind+m-1 do
                rise_up_to n i;
                array_print.(i) <- array_print.(i) ^ ")"
              done;
            )
          )

        | Abstrait f1 -> array_print.(ind) <- array_print.(ind) ^ f1.nom;

        | P (x,p) -> aux_ast_arbre (ast_of_poly_array_t p x) haut;

        | F (x,a,b) -> aux_ast_arbre (Arg2 ((ast_of_poly_array_t a x),Divise,(ast_of_poly_array_t b x))) haut;
```

```
  in

  aux_ast_arbre ast 0;
  Printf.fprintf file "\n";
  for i = h-l-1 downto 0 do
    Printf.fprintf file "%s\n" array_print.(i)
  done;
  Printf.fprintf file "\n";
;;





(* --------------- Dérivé --------------- *)



let rec derive ast_arbre =
  (* Dérive *)
  match ast_arbre with

  | Arg0 f -> (match f with
      | C c -> ast_null
      | X -> ast_un
      | Z -> ast_un
      | Exp -> Arg0 Exp
      | Ln -> Arg2 (ast_un, Divise, Arg0 X)
    )

  | Arg2 (f1, oper, f2) -> (match oper with
      | Compose -> Arg2 (derive f2, Fois, Arg2 (derive f1, Compose, f2))
      | Puissance -> Arg2 (Arg2 (Arg2 (Arg2 (Arg0 Ln, Compose, f1), Fois, Arg2 (f1, Puissance, f2)), Fois, derive f2), Plus
                    , Arg2 (f2 , Fois, Arg2(derive f1 , Fois, Arg2(f1, Puissance, Arg2 (f2 , Moins, ast_un)))))
      | Plus -> Arg2 (derive f1, Plus, derive f2)
      | Moins -> Arg2 (derive f1, Moins, derive f2)
      | Fois -> Arg2 (Arg2 (derive f1, Fois, f2), Plus, Arg2 (f1, Fois, derive f2))
      | Divise -> Arg2 (Arg2 (Arg2 (derive f1, Fois, f2), Moins, Arg2 (f1, Fois, derive f2)), Divise, Arg2 (f2, Fois, f2))
    )

  | Abstrait f -> Abstrait {nom = f.nom; fonction = f.fonction ; d_fonction = derive f.d_fonction; etat_derive = f.etat_derive + 1}
```

```

  | P (x,p) -> let p2 = Array.make (Array.length p) ast_null in
    (
      for i = 0 to Array.length p - 2 do
        p2.(i) <- Arg2 (derive p.(i), Plus, ast_fois (ast_fois p.(i+1) (ast_const (i+1) 1)) (derive x));
      done;
      if Array.length p - 1 >= 0
        then p2.(Array.length p - 1) <- derive p.(Array.length p - 1)
    );
    P (x, p2)

  | F (x,a,b) ->
    (
      let a1 = minus_poly (mult_poly (derive (P (x,a))) (P (x,b))) (mult_poly (derive (P (x,b))) (P (x,a))) in
      let b1 = mult_poly (P (x,b)) (P (x,b)) in
      F (x,poly_array a1,poly_array b1)
    )


and derive_n ast n =
  let f = ref ast in
  for i = 1 to n do
    f := derive !f;
  done;
  !f


and ast_abs str f n =
  (* Crée un ast abstrait qui nécessite la dérivé de f donc la fonction dérive avant*)
  let fd = ref f in
  for i = 0 to n-1 do
    fd := derive !fd
  done;
  {nom = str; fonction = f ; d_fonction = !fd; etat_derive = n}




(* --------------- Fonctions d'operations sur le type constante --------------- *)



and c_q_to_e (c:constante) = match c with
```

```
  | Q q -> (float_of_int q.a) /. (float_of_int q.b)
  | E e -> e.approx


and add_constante (c1:constante) (c2:constante) = match (c1,c2) with
  (* addition *)
  | Q q1,Q q2 -> (
    if (abs q1.a > seuil_max_int || abs q1.b > seuil_max_int) then raise Int_overflow;
    if (abs q2.a > seuil_max_int || abs q2.b > seuil_max_int) then raise Int_overflow;
    simplifie_constante (Q {a = q1.a * q2.b + q2.a * q1.b; b = q1.b * q2.b})
  )

  | E e1,E e2 -> E {nom = e1.nom ^ "+" ^ e2.nom; approx = e1.approx +. e2.approx}
  | Q q,E e | E e,Q q -> E {nom = e.nom ^ "+" ^ (string_of_int q.a) ^ "/" ^ (string_of_int q.b); approx = e.approx +. c_q_to_e (Q q) }


and neg_constante (c1:constante) = match c1 with
  (* negation *)
  | Q q -> if (abs q.a > seuil_max_int || abs q.b > seuil_max_int) then raise Int_overflow; Q {a = - q.a; b = q.b}
  | E e -> E {nom = "-" ^ e.nom; approx = -.e.approx}


and minus_constante (c1:constante) (c2:constante) = (* c1 - c2 *)
  (* soustraction *)
  simplifie_constante (add_constante c1 (neg_constante c2))


and mult_constante (c1:constante) (c2:constante) = match (c1,c2) with
  (* multiplication *)
  | Q q1,Q q2 -> (
    if (abs q1.a > seuil_max_int || abs q1.b > seuil_max_int) then raise Int_overflow;
    if (abs q2.a > seuil_max_int || abs q2.b > seuil_max_int) then raise Int_overflow;
    simplifie_constante (Q {a = q1.a * q2.a; b = q1.b * q2.b})
  )

  | E e1,E e2 -> E {nom = e1.nom ^ ")*(" ^ e2.nom; approx = e1.approx *. e2.approx}
  | Q q,E e | E e,Q q -> (
    if (abs q.a > seuil_max_int || abs q.b > seuil_max_int) then raise Int_overflow;
    E {nom = e.nom ^ ")*(" ^ (string_of_int q.a) ^ "/" ^ (string_of_int q.b); approx = e.approx *. c_q_to_e (Q q) }
  )


and inv_constante (c1:constante) = match c1 with
```

```
  (* fonction inverse *)
  | Q q -> assert (q.a <> 0); if (abs q.a > seuil_max_int || abs q.b > seuil_max_int) then raise Int_overflow;  Q {a = q.b; b = q.a}
  | E e -> assert (e.approx <> 0.); E {nom = "1/(" ^ e.nom; approx = 1./.e.approx}


and div_constante (c1:constante) (c2:constante) = (* c1 / c2 *)
  (* division *)
  simplifie_constante (mult_constante c1 (inv_constante c2))


and pow_constante_ent (c1:constante) (n:int) = match c1 with
  (* puissance entiere *)
  | Q q ->
    (
      if (abs q.a > seuil_max_int || abs q.b > seuil_max_int) then raise Int_overflow;
      let c,d = ref 1,ref 1 in
      for i = 1 to n do c := !c * q.a; d := !d * q.b done;
      if n >= 0 then Q {a = !c; b = !d}
      else Q {a = !d; b = !c}
    )
  | E e -> E {nom = e.nom ^ "^" ^ (string_of_int n); approx = e.approx ** (float_of_int n)}


and exp_cste (c1:constante) = match c1 with
  (* exponentielle *)
  | Q q -> E {nom = "exp("^(string_of_int q.a) ^ "/" ^ (string_of_int q.b)^")"; approx = exp (c_q_to_e (Q q))}
  | E e -> E {nom = "exp("^e.nom^")"; approx = exp e.approx}


and log_cste (c1:constante) = match c1 with
  (* logarithme *)
  | Q q -> E {nom = "ln("^(string_of_int q.a) ^ "/" ^ (string_of_int q.b)^")"; approx = log (c_q_to_e (Q q))}
  | E e -> E {nom = "ln("^e.nom^")"; approx = log e.approx}


and pow_constante (c1:constante) (c2:constante) = match (c1,c2) with
  (* puissance *)
  | Q q1,Q q2 ->
    (
      if (abs q1.a > seuil_max_int || abs q1.b > seuil_max_int) then raise Int_overflow;
      if (abs q2.a > seuil_max_int || abs q2.b > seuil_max_int) then raise Int_overflow;
      let c,d = match (simplifie_constante c2) with | Q q -> q.a,q.b | _ -> failwith "" in
      if d = 1 then (
```

```
          pow_constante_ent c1 c
        ) else (
          E {nom = "p ration"; approx = c_q_to_e (Q q1) ** c_q_to_e (Q q2)}
        )
      )
    | E e1,E e2 -> E {nom = e1.nom ^ ")^(" ^ e2.nom; approx = e1.approx ** e2.approx}
    | Q q,E e -> E {nom = (string_of_int q.a) ^ "/" ^ (string_of_int q.b) ^ ")^(" ^ e.nom; approx = c_q_to_e (Q q) ** e.approx}
    | E e,Q q -> E {nom = e.nom ^ ")^(" ^ (string_of_int q.a) ^ "/" ^ (string_of_int q.b); approx = e.approx ** c_q_to_e (Q q) }


  and abs_constante (c1:constante) = match c1 with
    | Q q -> if (abs q.a > seuil_max_int || abs q.b > seuil_max_int) then raise Int_overflow; Q {a = abs q.a; b = abs q.b}
    | E e -> E {nom = "|"^e.nom^"|"; approx = abs_float e.approx}


  and pgcd_entier a b =
    let c = ref (abs a) in
    let d = ref (abs b) in
    while !d <> 0 do
      let r = !c mod !d in
      c := !d;
      d := r;
    done;
    !c


  and signum a = if a = 0 then 0 else if a > 0 then 1 else -1


  and simplifie_constante (c1:constante) = match c1 with
    | Q q ->
      (
      if (abs q.a > seuil_max_int || abs q.b > seuil_max_int) then raise Int_overflow;
      appel_tab.(0) <- appel_tab.(0) + 1;
      if max q.a q.b > appel_tab.(2) then appel_tab.(2) <- max q.a q.b;
      let c = pgcd_entier q.a q.b in if c <> 0 then Q {a = (signum q.a) * (signum q.b) * (abs q.a) / c; b = (abs q.b) / c} else c1
      )
    | E e -> E {nom = "s("^(String.sub e.nom 0 3)^")"; approx = e.approx}
```

```


(* -------------- Fonctions de calcul de fonction -------------- *)



and ast_of_poly_array poly indet =
  (* transforme un ast P en ça version ast *)
  let f = ref poly.(0) in
  for i = 1 to Array.length poly - 1 do
    f := Arg2 (!f, Plus, Arg2(poly.(i), Fois, Arg2 (indet, Puissance, ast_const i 1)))
  done;
  !f


and ast_of_frac_array a b indet =
  (* transforme un ast F en ça version ast *)
  let ast_a = ast_of_poly_array a indet in
  let ast_b = ast_of_poly_array b indet in
  Arg2 (ast_a, Divise, ast_b)


and remplace_Abstrait f =
  (* remplace les parties abstraites *)
  match f with

  | Arg0 felem -> f

  | Arg2 (f1, oper, f2) -> Arg2 (remplace_Abstrait f1, oper, remplace_Abstrait f2)

  | Abstrait f -> f.d_fonction

  | P (x,p) ->
    (
      let p1 = Array.make (Array.length p) p.(0) in
      for i = 0 to Array.length p - 1 do
        p1.(i) <- remplace_Abstrait p.(i)
      done;
      P (remplace_Abstrait x, p1)
    )

  | F (x,a,b) ->
    (
      let pa = Array.make (Array.length a) a.(0) in
```

```
      let pb = Array.make (Array.length b) b.(0) in
      for i = 0 to Array.length a - 1 do
        pa.(i) <- remplace_Abstrait a.(i)
      done;
      for i = 0 to Array.length b - 1 do
        pb.(i) <- remplace_Abstrait b.(i)
      done;
      F (remplace_Abstrait x, pa, pb)
    )


and ast_arg_of_ast_arbre f =
  (* simplifie f en arg0 et arg2 pour la transformation en apt *)
  let fa = remplace_Abstrait f in

  match fa with

  | Arg0 felem -> fa

  | Arg2 (f1, oper, f2) -> Arg2 (ast_arg_of_ast_arbre f1, oper, ast_arg_of_ast_arbre f2)

  | Abstrait f1 -> ast_arg_of_ast_arbre (derive_n f1.fonction f1.etat_derive)

  | P (x,p) ->
    (
      let p1 = Array.make (Array.length p) p.(0) in
      for i = 0 to Array.length p - 1 do
        p1.(i) <- ast_arg_of_ast_arbre p.(i)
      done;
      ast_of_poly_array p1 (ast_arg_of_ast_arbre x)
    )

  | F (x,a,b) ->
    (
      let pa = Array.make (Array.length a) a.(0) in
      let pb = Array.make (Array.length b) b.(0) in
      for i = 0 to Array.length a - 1 do
        pa.(i) <- ast_arg_of_ast_arbre a.(i)
      done;
      for i = 0 to Array.length b - 1 do
        pb.(i) <- ast_arg_of_ast_arbre b.(i)
      done;
      ast_of_frac_array a b (ast_arg_of_ast_arbre x)
    )
```

```
and apt_of_ast ast_arbre =
  let ast = ast_arg_of_ast_arbre ast_arbre in
  match ast with
  | Arg0 f_elem ->
      ( match f_elem with
        | X -> Fonction (fun x -> x)
        | Z -> Fonction (fun x -> x)
        | C c -> Fonction (fun x -> c)
        | Exp -> Fonction (fun x -> exp_cste x)
        | Ln -> Fonction (fun x -> log_cste x)
      )

  | Arg2 (f1, oper, f2) ->
      Node (apt_of_ast f1 ,
            (match oper with
             | Plus -> (fun f g -> (fun x -> add_constante (f x) (g x)))
             | Moins -> (fun f g -> (fun x -> minus_constante (f x) (g x)))
             | Fois -> (fun f g -> (fun x -> mult_constante (f x) (g x)))
             | Divise -> (fun f g -> (fun x -> div_constante (f x) (g x)))
             | Puissance -> (fun f g -> (fun x -> pow_constante (f x) (g x)))
             | Compose -> (fun f g -> (fun x -> f (g x))))
           , apt_of_ast f2 )

  | P (x, poly) -> apt_of_ast (ast_of_poly_array poly x)

  | Abstrait f -> apt_of_ast f.d_fonction

  | _ -> failwith "N'arrive pas sauf bug avant le match"


and build apt_arbre =
  (* Construit une fonction ocaml permettant d'évaluer la fonction *)
  match apt_arbre with
  | Fonction f -> f
  | Node (f1, fc, f2) -> (fun x -> fc (build f1) (build f2) x )


and evalue_ast ast_arbre z = build (apt_of_ast ast_arbre) z


and appartient_f_elem ast_arbre elem = match ast_arbre with
```

```
    | Arg0 felem -> (
      match felem,elem with
        | C a, C b -> true
        | X,X | Z,Z | Ln,Ln | Exp,Exp -> true
        | _ -> false
      )

    | Arg2 (f1, oper, f2) -> (appartient_f_elem f1 elem) || (appartient_f_elem f2 elem)

    | Abstrait f1 -> appartient_f_elem f1.d_fonction elem

    | P (x,p) -> (
        let res = ref (appartient_f_elem x elem) in
        for i = 0 to Array.length p - 1 do
          res := !res || (appartient_f_elem p.(i) elem)
        done;
        !res
      )

    | F (x,a,b) -> (
      let res = ref (appartient_f_elem x elem) in
      for i = 0 to Array.length a - 1 do
        res := !res || (appartient_f_elem a.(i) elem)
      done;
      for i = 0 to Array.length b - 1 do
        res := !res || (appartient_f_elem b.(i) elem)
      done;
      !res
      )


  and is_zero_ast ast_arbre =
    (* test de manière imprécise si une expression est nul, rique d'échec massif du à l'implémentation *)
    let fonction = build (apt_of_ast ast_arbre) in
    let compte = ref 0 in
    let test = ref true in
    while !compte < int_of_float (10.**3.) && !test do
      compte := !compte + 1;
      let z1 = E {nom = "z1" ; approx = Random.float (707.48)} in
      let f1 = fonction z1 in

      (*
        OCaml's floating-point numbers follow the IEEE 754 standard, using double precision (64 bits) numbers.
        binary64  Double precision    1.80*10^308 max
```

```ocaml
      donc le float max en évaluation pour ne pas obtenir nan avec exp et inférieur à 707.48
    *)

    let valeur = match abs_constante f1 with | Q q -> c_q_to_e (Q q) | E e -> e.approx in

    if valeur > seuil_zero (*|| abs_constante f2 > seuil_zero || abs_constante f3 > seuil_zero || abs_constante f4 > seuil_zero*)
      then (
        (* (Printf.fprintf file "z1 : %f,%f, z2 : %f,%f, z3 : %f,%f, z4 : %f,%f, tour de boucle %d \n" f1.re f1.im f2.re f2.im f3.re f3.im f4.re f4.im !compte);  *)
        test := false
      )
  done;
  !test && (not (appartient_f_elem ast_arbre X && appartient_f_elem ast_arbre Z))


and is_const_ast ast_arbre =
  (* test de manière imprécise si une expression est nul, rique d'échec massif du à l'implémentation *)
  let fonction = build (apt_of_ast ast_arbre) in
  let compte = ref 0 in
  let test = ref true in
  let v = match abs_constante (fonction  (E {nom = "e"; approx = 2.781})) with | Q q -> c_q_to_e (Q q) | E e -> e.approx in

  while !compte < int_of_float (10.**3.) && !test do
    compte := !compte + 1;
    let z1 = E {nom = "z1" ; approx = Random.float (707.48)} in
    let f1 = fonction z1 in
    let valeur = match abs_constante f1 with | Q q -> c_q_to_e (Q q) | E e -> e.approx in

    if abs_float (valeur -. v) > 10.*.seuil_zero (*|| abs_constante f2 > seuil_zero || abs_constante f3 > seuil_zero || abs_constante f4 > seuil_zero*)
      then test := false
  done;
  (!test,fonction (Q {a = 1; b = 1}))





(* --------------- Vérifie une égalité entre deux ast --------------- *)
```

```
and egal_ast1 ast_1 ast_2 = match (ast_1,ast_2) with
  | Arg0 f_elm1, Arg0 f_elm2 -> f_elm1 = f_elm2
  | Arg2 (f1, oper1, f2),Arg2 (f3, oper2, f4) -> (egal_ast1 f1 f3) && (oper1 = oper2) && (egal_ast1 f2 f4)
  | Abstrait f1,Abstrait f2 -> egal_ast1 f1.d_fonction f2.d_fonction
  | _,_ -> false


and egal_ast2 ast_1 ast_2 = is_zero_ast (Arg2 (ast_1, Moins, ast_2))


and egal_ast ast_1 ast_2 = egal_ast1 ast_1 ast_2 || egal_ast2 ast_1 ast_2





(* --------------- Début de simplification / formatage --------------- *)


and simplifie_ast ast_arbre = appel_tab.(1) <- appel_tab.(1) + 1; match ast_arbre with
  (* tente de simplifier quelques élément d'un ast *)
  | Arg0 f -> (match f with
      | X -> ast_arbre
      | Z -> ast_arbre
      | C c -> Arg0 (C (simplifie_constante c))
      | Exp -> ast_arbre
      | Ln -> ast_arbre
    );

  | Arg2 (f1, oper, f2) ->
      let g1 = (if is_zero_ast f1 then ast_null else simplifie_ast f1) in
      let g2 = (if is_zero_ast f2 then ast_null else simplifie_ast f2) in
      let ast_simp = Arg2(g1, oper, g2) in
      (match ast_simp with
        (* Simplifie la compositon *)
        | Arg2 (Arg0 (C x), Compose, f4) -> Arg0 (C x)
        | Arg2 (f3 , Compose, Arg0 (C x)) -> Arg0 (C (build (apt_of_ast f3) x))
        (* | Arg2 (f3 , Compose, Arg0 X) -> f3 *)
        | Arg2 (Arg0 X, Compose, f4) -> f4
        | Arg2 (Arg2 (Arg0 (C (Q {a = 1; b = 1})),Divise,Arg0 X), Compose, f4) -> Arg2(ast_un,Divise,f4)
        | Arg2 (P (Arg0 X,[|Arg0 (C (Q {a = 0; b = 1}));Arg0 (C (Q {a = 1; b = 1}))|]),Compose,f4) -> f4
        (* Simplifie la puissance *)
        | Arg2 (Arg0 (C x), Puissance, Arg0 (C y)) -> Arg0 (C (pow_constante x y))
        | Arg2 (f3, Puissance, Arg0 (C (Q {a = 0; b = 1}))) -> ast_un
```

```
          | Arg2 (f3, Puissance, Arg0 (C (Q {a = 1; b = 1}))) -> f3
          | Arg2 (Arg0 (C (Q {a = 1; b = 1})), Puissance, f4) -> ast_un
          | Arg2 (Arg2 (f3, Puissance, f4), Puissance, f5) -> Arg2 (f3, Puissance, Arg2 (f4, Fois,f5))
          | Arg2 (Arg2 (Arg0 Exp, Compose, f4), Puissance, f5) -> Arg2 (Arg0 Exp, Compose, Arg2 (f4, Fois,f5))
          | Arg2 (P (x,a), Puissance, Arg0 (C (Q {a = n; b = 1}))) -> puissance_poly_ent (P (x,a)) n
          | Arg2 (F (x,a,b), Puissance, Arg0 (C (Q {a = n; b = 1}))) -> simplifie_ast (F (x, poly_array (puissance_poly_ent (P (x,a)) n), poly_array (puissance_poly_ent (P (x,b)) n)))
          (* Simplifie l'addition *)
          | Arg2 (Arg0 (C (Q {a = 0; b = 1})), Plus, f4) -> f4
          | Arg2 (f3 , Plus, Arg0 (C (Q {a = 0; b = 1}))) -> f3
          | Arg2 (Arg0 (C x), Plus, Arg0 (C y)) -> Arg0 (C (add_constante x y))
          | Arg2 (f1, Plus, f2) when egal_ast f1 f2 -> Arg2 (Arg0 (C (Q {a = 2; b = 1})), Fois, f1)
          | Arg2 (P (x,a), Plus, P (y,b)) when egal_ast x y -> add_poly (P (x,a)) (P (y,b))
          | Arg2 (F (x,a,b), Plus, F (y,c,d)) when egal_ast x y -> add_frac (F (x,a,b)) (F (y,c,d))
          (* Simplifie la soustraction *)
          | Arg2 (Arg0 (C (Q {a = 0; b = 1})), Moins, f4) -> Arg2 (ast_minus_un, Fois, f4)
          | Arg2 (f3, Moins, Arg0 (C (Q {a = 0; b = 1}))) -> f3
          | Arg2 (Arg0 (C x), Moins, Arg0 (C y)) -> Arg0 (C (minus_constante x y))
          | Arg2 (f3, Moins, Arg0 (C x)) -> Arg2 (f3, Plus, Arg0 (C (neg_constante x)))
          | Arg2 (f1, Moins, f2) when egal_ast f1 f2 -> ast_null
          | Arg2 (P (x,a), Moins, P (y,b)) when egal_ast x y -> minus_poly (P (x,a)) (P (y,b))
          | Arg2 (F (x,a,b), Moins, F (y,c,d)) when egal_ast x y -> minus_frac (F (x,a,b)) (F (y,c,d))
          (* Simplifie la multiplication *)
          | Arg2 (Arg0 (C (Q {a = 0; b = 1})) ,Fois , f4) -> ast_null
          | Arg2 (f3 ,Fois , Arg0 (C (Q {a = 0; b = 1}))) -> ast_null
          | Arg2 (Arg0 (C x) ,Fois , Arg0 (C y)) -> Arg0 (C (mult_constante x y))

          | Arg2 (Arg0 (C (Q {a = 1; b = 1})) ,Fois , f4) -> f4
          | Arg2 (f3 ,Fois , Arg0 (C (Q {a = 1; b = 1}))) -> f3
          | Arg2 (P (_, [|Arg0 (C (Q {a = 1; b = 1}))|]) ,Fois , f4) -> f4
          | Arg2 (f3 ,Fois , P (_, [|Arg0 (C (Q {a = 1; b = 1}))|])) -> f3
          | Arg2 (f1, Fois, f2) when egal_ast f1 f2 -> Arg2 (f1, Puissance, Arg0 (C (Q {a = 2; b = 1})))
          | Arg2 (P (x,a), Fois, P (y,b)) when egal_ast x y -> mult_poly (P (x,a)) (P (y,b))
          | Arg2 (F (x,a,b), Fois, F (y,c,d)) when egal_ast x y -> mult_frac (F (x,a,b)) (F (y,c,d))
          | Arg2 (f1, Fois, Arg2 (Arg0 (C (Q {a = 1; b = 1})), Divise, f2)) | Arg2 (Arg2 (Arg0 (C (Q {a = 1; b = 1})), Divise, f2), Fois, f1) -> simplifie_ast (Arg2 (f1, Divise, f2))
          (* Simplifie la division *)
          | Arg2 (f3 ,Divise , Arg0 (C (Q {a = 0; b = 1}))) -> (* print_ast_arbre f3; f3 *) failwith "Div par 0 dans simplifie"
          | Arg2 (Arg0 (C (Q {a = 0; b = 1})) ,Divise , f4) -> ast_null
          | Arg2 (Arg0 (C x) ,Divise , Arg0 (C y)) -> Arg0 (C (div_constante x y))
          | Arg2 (f3 ,Divise , Arg0 (C (Q {a = 1; b = 1}))) -> f3
          | Arg2 (f1, Divise, f2) when egal_ast f1 f2 -> ast_un
          | Arg2 (f1, Divise, Arg2 (Arg0 (C (Q {a = 1; b = 1})), Divise, f2)) -> simplifie_ast (Arg2 (f1, Fois, f2))
```

```
        | Arg2 (P (x,a), Divise, P (y,b)) when egal_ast x y -> simplifie_ast (polys_to_frac (P (x,a)) (P (y,b)))
        | Arg2 (F (x,a,b), Divise, F (y,c,d)) when egal_ast x y -> divise_frac (F (x,a,b)) (F (y,c,d))

        (* Simplifié au max *)
        | _ -> ast_simp
      )

  | P (x,a) -> (
    (* if Array.length a = 0 then ast_null
    else if Array.length a = 1 then simplifie_ast a.(0)
    else *) ajuste_poly (P (x,a))
  )

  | F (x,a,b) -> ajuste_frac ast_arbre

  | _ -> ast_arbre




(* --------------- Matrices et determinant cramer et det --------------- *)



and det_c mat =
  (* calcul d'un détérminant scalaire *)
  let n = Array.length mat in
  assert(n = Array.length mat.(0));
  if n = 1
    then mat.(0).(0)
  else
    (
      let mat2 = Array.make_matrix (n-1) (n-1) c_zero in
      let res = ref c_zero in
      for i = 0 to n-1 do
        for j = 1 to n-1 do
          for k = 0 to n-1 do
            if k < i
              then mat2.(j-1).(k) <- mat.(j).(k)
            else if k > i
              then mat2.(j-1).(k-1) <- mat.(j).(k)
            else ()
          done;
```

```
        done;
        if not (is_zero_ast (Arg0 (C mat.(0).(i)))) then (
          res := add_constante !res (mult_constante (if i mod 2 = 0 then c_un else c_minus_un) (mult_constante (det_c mat2) mat.(0).(i)));
        )
      done;
      !res
    )


and det_ast mat =
  (* calcul d'un détérminant scalaire *)
  let n = Array.length mat in
  assert(n = Array.length mat.(0));
  if n = 1
    then mat.(0).(0)
  else
    (
      let mat2 = Array.make_matrix (n-1) (n-1) ast_null in
      let res = ref ast_null in
      for i = 0 to n-1 do
        for j = 1 to n-1 do
          for k = 0 to n-1 do
            if k < i
              then mat2.(j-1).(k) <- mat.(j).(k)
            else if k > i
              then mat2.(j-1).(k-1) <- mat.(j).(k)
            else ()
          done;
        done;

      if not (is_zero_ast mat.(0).(i)) then (
        res := Arg2 (!res, Plus, Arg2 (Arg0 (if i mod 2 = 0 then C c_un else C c_minus_un), Fois, (Arg2 (det_ast mat2, Fois, mat.(0).(i)))));
      )
      done;
      simplifie_ast !res
    )


and systeme_cramer famille_colonne colonne =
  (* Résout un systeme de Cramer de type constante *)
  let n1 = Array.length colonne in
  let n2 = Array.length famille_colonne in
```

```ocaml
    assert (n2 > 0);
    let n3 = Array.length famille_colonne.(n2-1) in
    assert (n1 = n2 && n2 = n3);

    let colonne_res = Array.make n1 c_zero in
    let det_p = det_c famille_colonne in

    for k = 0 to n1 -1 do
      let mat_k = Array.make_matrix n1 n1 c_zero in

      for i = 0 to n1 - 1 do
        for j = 0 to n1 - 1 do
          if j <> k
            then mat_k.(i).(j) <- famille_colonne.(i).(j)
          else mat_k.(i).(j) <- colonne.(i)
        done;
      done;
      (* Array.iter (fun tab -> Array.iter (fun x -> Printf.fprintf file "(%f,%f);" x.re x.im) tab; rintf.fprintf file "\n") mat_k; *)
      colonne_res.(k) <- div_constante (det_c mat_k) det_p;
    done;
    colonne_res



  and systeme_cramer_log famille_colonne colonne =
    (* Résout un systeme de Cramer de type constante *)
    let n1 = Array.length colonne in
    let n2 = Array.length famille_colonne in
    assert (n2 > 0);
    let n3 = Array.length famille_colonne.(n2-1) in
    assert (n1 = n2 && n2 = n3);

    let colonne_res = Array.make n1 ast_null in
    let det_p = det_ast famille_colonne in

    for k = 0 to n1 -1 do
      let mat_k = Array.make_matrix n1 n1 ast_null in

      for i = 0 to n1 - 1 do
        for j = 0 to n1 - 1 do
          if j <> k
            then mat_k.(i).(j) <- famille_colonne.(i).(j)
          else mat_k.(i).(j) <- colonne.(i)
```

```
        done;
      done;
      (* Array.iter (fun tab -> Array.iter (fun x -> Printf.fprintf file "(%f,%f);" x.re x.im) tab; rintf.fprintf file
      "\n") mat_k; *)
      colonne_res.(k) <- Arg2((det_ast mat_k),Divise, det_p);
    done;
    colonne_res





  (* --------------- Fonctions d'operation sur polynome --------------- *)



  and poly_copie poly =
    (* copie un polynome pour eviter les effets de bords *)
    match poly with
      | P (x,p) -> (
        let n = Array.length p in
        let pc = Array.make n p.(0) in
        for i = 0 to n-1 do
          pc.(i) <- p.(i)
        done;
        P (x, pc)
      )
      | _ -> failwith "poly_copie : pas un polynome"


  and poly_indet poly =
    (* renvoie l'indeterminer du polynome *)
    match poly with
      | P (x,p) -> x
      | _ -> failwith "poly_indet : pas un polynome"


  and poly_array poly =
    (* renvoie l'array du polynome *)
    match poly with
      | P (x,p) -> p
      | _ -> failwith "poly_array : pas un polynome"

```

```
and deg_poly (poly:ast_elem) =
  (* renvoie le degré d'un polynome *)
  match poly with
  | P (x,p) ->
      (
        let n = ref (Array.length p) in

        while !n > 0 && is_zero_ast p.(!n-1) do
          n := !n - 1;
        done;

        if !n <= 0 then min_int else !n-1
      )

  | _ -> failwith "deg_poly : pas un polynome"


and ajuste_poly (poly:ast_elem) =
  (* remet le polynôme à une taille correspondant à son degré *)
  match poly with
  | P (x,p) -> (
    let degre = deg_poly poly in

    (* print_debug degre; *)

    if degre < 0
      then P (x, [|ast_null|])
    else
      (
        let new_poly = P (x, Array.make (degre+1) ast_null) in
        match new_poly with
          | P (y,np) -> (
            for i = 0 to degre do
              np.(i) <- simplifie_ast p.(i);
            done;
            P (y,np))
          | _ -> failwith "ajuste_poly : impossible case"
      )
    )

  | _ -> failwith "ajuste_poly : pas un polynome"


and poly_unitaire poly =
```

```
    (* renvoie le polynome unitaire associé et son ancien coeff dominant *)
    match poly with
      | P (x,p) -> (
        let n = Array.length p - 1 in
        let pu = Array.make (n+1) p.(0) in
        for i = 0 to n do
          pu.(i) <- simplifie_ast (ast_divise p.(i) p.(n));
        done;
        (P (x,pu),p.(n))
      )
      | _ -> failwith "poly unitaire : pas un polynome"


  and add_poly (poly1:ast_elem) (poly2:ast_elem) =
    (* addition *)
    match (poly1,poly2) with
    | P (x,p1),P (y,p2) when egal_ast x y -> (
        let n = max (deg_poly poly1) (deg_poly poly2) in

        if n < 0 then
          P (x, [|ast_null|])

        else
          (
          let res_poly_array = Array.make (n+1) ast_null in
          for i = 0 to (deg_poly poly1) do
            res_poly_array.(i) <- p1.(i)
          done;
          for i = 0 to (deg_poly poly2) do
            res_poly_array.(i) <- Arg2 (res_poly_array.(i), Plus, p2.(i))
          done;
          ajuste_poly (P (x, res_poly_array))
          )
      )

    | _,_ -> failwith "add_poly : pas des polynômes ou polynome d'indet diff"


  and neg_poly (poly:ast_elem) =
    (* négation *)
    match poly with
    | P (x,p) -> (
        let n = deg_poly poly in

```

```
      if n >= 0
        then
          (
          let res_poly_array = Array.make (n+1) p.(0) in
          for i = 0 to n do
            res_poly_array.(i) <- Arg2 (ast_minus_un, Fois, p.(i))
          done;
          (P (x, res_poly_array))
          )
        else poly
    )

  | _ -> failwith "neg_poly : pas un polynôme"


and minus_poly (poly1:ast_elem) (poly2:ast_elem) =
  (* soustraction *)
  match (poly1,poly2) with
  | P (x,p1),P (y,p2) when egal_ast x y -> (
      let n = max (deg_poly poly1) (deg_poly poly2) in

      if n < 0 then
        P (x, [|ast_null|])

      else
        (
        let res_poly_array = Array.make (n+1) ast_null in

        for i = 0 to (deg_poly poly1) do
          res_poly_array.(i) <- p1.(i)
        done;
        for i = 0 to (deg_poly poly2) do
          res_poly_array.(i) <- Arg2 (res_poly_array.(i), Moins, p2.(i))
        done;

        ajuste_poly (P (x, res_poly_array))
        )
    )

  | _,_ -> failwith "minus_poly : pas des polynômes ou polynome d'indet diff"


and mult_poly (poly1:ast_elem) (poly2:ast_elem) =
  (* multiplication *)
```

```
    match (poly1,poly2) with
    | P (x,p1),P (y,p2) when egal_ast x y ->
      (
        let n = (deg_poly poly1) + (deg_poly poly2) in

        if n < 0 then
          P (x, [|ast_null|])

        else
          (
          let res_poly_array = Array.make (n+1) ast_null in
          for i = 0 to (deg_poly poly1) do
            for j = 0 to (deg_poly poly2) do
              res_poly_array.(i + j) <- Arg2 (res_poly_array.(i + j), Plus, Arg2 (p1.(i) ,Fois ,p2.(j) ))
            done;
          done;
          ajuste_poly (P (x, res_poly_array))
          )
      )

      | P (x,p1), ast when not (egal_ast x ast)->
        (
          let n = deg_poly poly1 in

          if n < 0 then
            P (x, [|ast_null|])

          else
            (
            let res_poly_array = Array.make (n+1) ast_null in
            for i = 0 to n do
              res_poly_array.(i) <- Arg2 (p1.(i), Fois, ast);
            done;
            ajuste_poly (P (x, res_poly_array))
            )
        )

      | ast,P (x, p2) when not (egal_ast x ast)->
        (
          let n = deg_poly poly2 in

          if n < 0 then
            P (x, [|ast_null|])

```

```
        else
          (
          let res_poly_array = Array.make (n+1) ast_null in
          for i = 0 to n do
            res_poly_array.(i) <- Arg2 (p2.(i), Fois, ast)
          done;
          ajuste_poly (P (x, res_poly_array))
          )
      )

    | _,_ -> failwith "mult_poly : pas des polynômes ou polynome d'indet diff"


and mult_poly_scal (poly:ast_elem) (lambda:constante) =
  (* multiplication *)
  match poly with
  | P (x,p) ->
    (
      let pi = Array.make (Array.length p) p.(0) in
      for i = 0 to Array.length p - 1 do
        pi.(i) <- Arg2 (Arg0 (C lambda), Fois, p.(i));
      done;
      ajuste_poly (P (x,pi))
    )

  | _ -> failwith "mult_poly_scal : pas de polynome"


and puissance_poly_ent (poly:ast_elem) (n:int) =
  (* puissance non optimisé / naive *)
  match poly with
  | P (x,p) ->
    (
      let pres = ref (P (x, [|ast_un|])) in
      for i = 0 to n-1 do
        pres := mult_poly !pres poly
      done;
      ajuste_poly (!pres)
    )
  | _ -> failwith "puissance_poly_ent : pas un polynome"


and div_euclid_poly (poly1:ast_elem) (poly2:ast_elem) =
  (* division euclidienne *)
```

```
    let poly1_a = (ajuste_poly poly1) in
    let poly2_a = (ajuste_poly poly2) in

    match (poly1_a,poly2_a) with

    | P (x,p1),P (y,p2) when egal_ast x y ->
      (
        let d1 = deg_poly poly1_a in
        let d2 = deg_poly poly2_a in

        if d1 < d2 || d2 < 0
          then (P (x,[|ast_null|]) , poly1_a)

        else if d2 = 0
          then (ajuste_poly (mult_poly poly1_a (Arg2 (ast_un, Divise, p2.(0)))) , P (x,[|ast_null|]))

        else
          (
          let quotient_poly_array = Array.make (d1 - d2 + 1) ast_null in
          let reste_poly_array = ref (poly_array (poly_copie poly1_a)) in

          for i = (d1 - d2) downto 0 do
            if i + d2 < Array.length !reste_poly_array
              then begin

                let quotient_poly_array_t = Array.make (d1 - d2 + 1) ast_null in
                quotient_poly_array.(i) <- simplifie_ast (Arg2 (!reste_poly_array.(i + d2), Divise, simplifie_ast p2.(d2)));
                quotient_poly_array_t.(i) <- simplifie_ast (Arg2 (!reste_poly_array.(i + d2), Divise, simplifie_ast p2.(d2 )));
                let poly_soustrait = ajuste_poly (mult_poly poly2_a  (P (x,quotient_poly_array_t))) in
                reste_poly_array := poly_array (ajuste_poly (minus_poly (P (x,!reste_poly_array)) poly_soustrait));

              end;
          done;

          (ajuste_poly (P (x,quotient_poly_array)) ,ajuste_poly (P (x,!reste_poly_array)))
          )
      )

    | _,_ -> failwith "div_euclid_poly : pas des polynômes ou polynome d'indet diff"


  and modulo_poly (poly1:ast_elem) (poly2:ast_elem) =
  (* modulo sur les polynomes poly1 mod poly2*)
```

```
    match (div_euclid_poly poly1 poly2) with
    | quotient,reste -> reste


  and pgcd_poly (poly1:ast_elem) (poly2:ast_elem) =
    (* pgcd / algorithme d'euclide *)
    match (poly1,poly2) with
    | P (x,p1),P (y,p2) -> (

      if not (egal_ast x y) then failwith "pgcd_poly : indet differente";

      let poly_r = ref (ajuste_poly (modulo_poly poly1 poly2)) in
      let poly_p = ref poly1 in
      let poly_q = ref poly2 in

      while not (is_zero_ast !poly_r) do

        poly_p := !poly_q;
        poly_q := !poly_r;
        poly_r := ajuste_poly (modulo_poly !poly_p !poly_q);

      done;

      match poly_unitaire (ajuste_poly !poly_q) with | a,b -> a )

    | _,_ -> failwith "pgcd_poly : pas un couple de polynômes"


  and pgcd_poly_algebrique (poly1:ast_elem) (poly2:ast_elem) =
    (* pgcd / algorithme d'euclide *)
    match (poly1,poly2) with
    | P (x,p1),P (y,p2) -> (

      if not (egal_ast x y) then failwith "pgcd_poly : indet differente";

      let poly_p = ref poly1 in
      let poly_q = ref poly2 in

      while not (deg_poly !poly_q < 1) do
        let poly_r = modulo_poly !poly_p !poly_q in
        poly_p := !poly_q;
        poly_q := poly_r;
      done;

```

```
      match poly_unitaire (ajuste_poly !poly_p) with | a,b -> a )

    | _,_ -> failwith "pgcd_poly : pas un couple de polynômes"


  and poly_bezout (poly1:ast_elem) (poly2:ast_elem) =
    (* renvoie s et t deux polys tels que poly1 * s + t * poly2 = poly1 ^ poly2 *)
    match (poly1,poly2) with
    | P (x,p1),P (y,p2) when egal_ast x y -> (

      let c = ref poly1 in
      let c1 = ref (P (x,[|ast_un|])) in
      let c2 = ref (P (x,[|ast_null|])) in

      let d = ref poly2 in
      let d1 = ref (P (x,[|ast_null|])) in
      let d2 = ref (P (x,[|ast_un|])) in

      while not (is_zero_ast !d) do
        let quotient,reste = div_euclid_poly !c !d in

        let q = ajuste_poly quotient in

        let r1 = ajuste_poly (minus_poly !c1 (mult_poly q !d1)) in
        let r2 = ajuste_poly (minus_poly !c2 (mult_poly q !d2)) in

        c := !d;
        c1 := !d1;
        c2 := !d2;

        d := reste;
        d1 := r1;
        d2 := r2;
      done;

      let uc = match !c with
        | P (x,tab) -> ( tab.(Array.length tab - 1)
        )
        | _ -> failwith "poly_bezout : pas un poly bug"
      in

      let s = mult_poly !c1 (simplifie_ast (ast_divise ast_un uc)) in
      let t = mult_poly !c2 (simplifie_ast (ast_divise ast_un uc)) in

```

```
      (ajuste_poly s,ajuste_poly t)
      )

    | _,_ -> failwith "poly_bezout : pas un couple de polynômes"


  and poly_extend_euclidean (poly1:ast_elem) (poly2:ast_elem) (poly3:ast_elem) =
    (* page 13 transcendantal integration bronstein suppose poly1 ^ poly2 = 1 *)
    match (poly1,poly2,poly3) with
    | P (x,p1),P (y,p2),P (z,p3) when egal_ast x y && egal_ast y z -> (
      let s,t = poly_bezout poly1 poly2 in
      if (not (egal_ast s ast_null)) && deg_poly (mult_poly poly3 s) >= deg_poly poly2 then (
        let q,r = div_euclid_poly (mult_poly poly3 s) poly2 in
        (ajuste_poly r,ajuste_poly (add_poly (mult_poly poly3 t) (mult_poly q poly1)))
      ) else (
        (s,t)
      )
      )
    | _,_,_ -> failwith "poly_extend_euclidean : entré non valide"


  and frac_indet frac =
    (* renvoie l'indeterminer de la fraction rationnelle *)
    match frac with
      | F (x,a,b) -> x
      | _ -> failwith "frac_indet : pas une fraction rationnelle"


  and frac_nom frac =
    (* renvoie l'array du nominateur de la fraction rationnelle *)
    match frac with
      | F (x,a,b) -> a
      | _ -> failwith "frac_nom : pas une fraction rationnelle"


  and frac_denom frac =
    (* renvoie l'array du dénominateur de la fraction rationnelle *)
    match frac with
      | F (x,a,b) -> b
      | _ -> failwith "frac_denom : pas une fraction rationnelle"


  and deg_frac (frac:ast_elem) =
    (* renvoie le degré d'un polynome *)
```

```
    match frac with
    | F (x,a,b) ->
        (
          let pa = P (x, a) in
          let pb = P (x, b) in
          deg_poly pa - deg_poly pb
        )

    | _ -> failwith "deg_frac : pas une fraction rationnelle "


  and ajuste_frac (frac:ast_elem) =
    (* remet la fraction rationnelle à une taille correspondant à son degré *)
    match frac with
    | F (x,a,b) ->
      (
        assert (Array.length b > 0);

        let pa = ajuste_poly (P (x, a)) in
        let pb = ajuste_poly (P (x, b)) in
        let pgcd = mult_poly (ajuste_poly (pgcd_poly pa pb) ) (if Array.length b = 1 then P (x,b) else P (x,[|ast_un|]))  in
        let pa2,pra = div_euclid_poly pa pgcd in
        let pb2,prb = div_euclid_poly pb pgcd in

        (F (x,poly_array (ajuste_poly pa2), poly_array (ajuste_poly pb2)))
      )

    | _ -> failwith "ajuste_frac : pas une fraction rationnelle"


  and add_frac (frac1:ast_elem) (frac2:ast_elem) =
    (* addition *)
    match (frac1,frac2) with
    | F (x,pa1,pb1),F (y,pa2,pb2) when egal_ast x y ->
      (
        let nom = match add_poly (mult_poly (P (x,pa1)) (P (x,pb2))) (mult_poly (P (x,pa2)) (P (x,pb1))) with | P (a,b) -> b | _ -> failwith "erreur" in
        let denom = match mult_poly (P (x,pb1)) (P (x,pb2)) with | P (a,b) -> b | _ -> failwith "erreur" in
        ajuste_frac (F (x, nom, denom))
      )

    | _,_ -> failwith "add_frac : pas des fractions rationnelles ou frac d'indet diff"


```

```
and neg_frac (frac:ast_elem) =
  (* négation *)
  match frac with
  | F (x,a,b) ->
    (
      let a1 = match neg_poly (P (x,a)) with | P (i,k) -> k | _ -> failwith "erreur" in
      F (x, a1, b)
    )

  | _ -> failwith "neg_frac : pas une fraction rationnelle"


and minus_frac (frac1:ast_elem) (frac2:ast_elem) =
  (* soustraction *)
  match (frac1,frac2) with
  | F (x,pa1,pb1),F (y,pa2,pb2) when egal_ast x y ->
    (
      add_frac frac1 (neg_frac frac2)
    )

  | _,_ -> failwith "minus_frac : pas des fractions rationnelles ou frac d'indet diff"


and mult_frac (frac1:ast_elem) (frac2:ast_elem) =
  (* multiplication *)
  match (frac1,frac2) with
  | F (x,pa1,pb1),F (y,pa2,pb2) when egal_ast x y ->
    (
      let nom = match mult_poly (P (x,pa1)) (P (x,pa2)) with | P (a,b) -> b | _ -> failwith "erreur" in
      let denom = match mult_poly (P (x,pb1)) (P (x,pb2)) with | P (a,b) -> b | _ -> failwith "erreur" in
      F (x, nom, denom)
    )

  | _,_ -> failwith "mult_frac : pas des fractions rationnelles ou frac d'indet diff"


and puissance_frac_ent (frac:ast_elem) (n:int) =
  (* puissance non optimisé / naive *)
  match frac with
  | F (x,a,b) ->
    (
      let pres = ref (F (x, [|ast_un|], [|ast_un|])) in
      for i = 0 to n-1 do
        pres := mult_frac !pres frac
```

```
      done;
      ajuste_frac (!pres)
    )
  | _ -> failwith "puissance_frac_ent : pas une fraction"


and mult_frac_scal (frac:ast_elem) (lambda:constante) =
  (* multiplication *)
  match frac with
  | F (x,pa,pb) ->
    (
      let nom = match (mult_poly_scal (P (x,pa)) lambda) with | P (y,p) -> p | _ -> failwith "bug" in
      F (x, nom, pb)
    )

  | _ -> failwith "mult_frac_scal : pas de fraction rationnelle"


and inv_frac (frac:ast_elem) =
  (match frac with | F (x,a,b) -> assert (not (is_zero_ast frac)); F (x, b, a)| _ -> failwith "inv_frac : pas une fraction rationnel")


and divise_frac (frac1:ast_elem) (frac2:ast_elem) =
  (* division *)
  match (frac1,frac2) with
  | F (x,pa1,pb1),F (y,pa2,pb2) when egal_ast x y -> mult_frac frac1 (inv_frac frac2)
  | _,_ -> failwith "divise_frac : pas des fractions rationnelles ou frac d'indet diff"


and compose_frac (frac1:ast_elem) (frac2:ast_elem) =
  (* composition de fraction *)
  match (frac1,frac2) with
  | F (x,pa1,pb1),F (y,pa2,pb2) when egal_ast x y -> (
    let ft1 = ref (F (x, [|pa1.(0)|], [|ast_un|])) in (* partie haute *)
    let ft2 = ref (F (x, [|pb1.(0)|], [|ast_un|])) in (* partie basse *)

    for i = 1 to Array.length pa1 - 1 do
      ft1 := add_frac !ft1 (mult_frac (F (x, [|pa1.(i)|], [|ast_un|])) (puissance_frac_ent frac2 i))
    done;

    for i = 1 to Array.length pb1 - 1 do
      ft2 := add_frac !ft2 (mult_frac (F (x, [|pb1.(i)|], [|ast_un|])) (puissance_frac_ent frac2 i))
    done;
```

```

    divise_frac !ft1 !ft2
  )
  | _,_ -> failwith "compose_frac : pas des fractions rationnelles ou frac d'indet diff"


and polys_to_frac (poly1:ast_elem) (poly2:ast_elem) =
  (* fait une fraction rationnelle à partir de deux polynomes *)
  match (poly1,poly2) with
  | P (x,p1),P (y,p2) when egal_ast x y ->
    (
      assert (not (is_zero_ast poly2));
      F (x, p1, p2)
    )

  | _,_ -> failwith "polys_to_frac : pas des polynomes ou frac d'indet diff"
;;


let decomp_en_polynome_sans_carre polynome =
  (* algorithme de yun pour la décomposition *)

  let derive_theta poly = match poly with
    | P (x,a) -> (
        let n = deg_poly poly in
        if n <= 0 then P (x,[|ast_null|]) else (
          let b = Array.make n ast_null in
          for i = 0 to n-1 do
            b.(i) <- simplifie_ast (Arg2 (ast_const (i+1) 1, Fois, a.(i+1)))
          done;
          P (x,b)
        )
      )
    | _ -> failwith "decomp en poly sans carré"
  in

  let poly,u = poly_unitaire polynome in
  (* version unitaire du polynome *)
  let i = ref 1 in
  let factorisation = ref [] in
  let b = ref (ajuste_poly (derive_theta poly)) in
  let c = ref (ajuste_poly (pgcd_poly poly !b)) in
  let w = ref (ajuste_poly poly) in

```

```
  if not (egal_ast !c (P (poly_indet !c, [|ast_un|])))
    then (

      let w1,w2 = div_euclid_poly poly !c in
      w := w1;  assert (is_zero_ast w2);
      let y1,y2 = div_euclid_poly !b !c in
      let y = ref (ajuste_poly y1) in  assert (is_zero_ast y2);
      let z = ref (ajuste_poly (minus_poly !y (derive_theta !w))) in

      while (not (is_zero_ast !z)) do

        let g = pgcd_poly !w !z in
        factorisation := (g,!i)::(!factorisation);
        i := !i + 1;
        let w3,w4 = div_euclid_poly !w g in
        w := ajuste_poly w3;  assert (is_zero_ast w4);
        let y3,y4 = div_euclid_poly !z g in
        y := ajuste_poly y3;  assert (is_zero_ast y4);
        z := ajuste_poly (minus_poly !y (derive_theta !w));
      done;
    )
  ;

  factorisation := (!w,!i)::(!factorisation);

  let facto_array = (Array.of_list !factorisation) in
  Array.sort (fun (p1,j1) (p2,j2) -> j1-j2 ) facto_array;
  Array.map (fun (x,j) -> (simplifie_ast x,j)) facto_array
;;


let calcul_des_fractions_partielles fraction decomp_en_polynome_sans_carre =
  (* nb la fraction d'entré vérifier deg A < def B*)
  let get_0 (a,b) = a in
  let get_1 (a,b) = b in

  let const_of_ast ast = match ast with
    | Arg0 f -> ( match f with
      | C c -> c
      | _ -> failwith "calcul_des_fractions_partielles : pas une cste"
    )
    | _ -> failwith "calcul_des_fractions_partielles : pas un ast"
  in

```

```
    let coeff poly k = match poly with
      | P (x,tab) -> tab.(k)
      | _ -> failwith "calcul_des_fractions_partielles : pas un polynome"
    in

    let poly_x_k indet k =
      let p = Array.make (k+1) ast_null in
      p.(k) <- ast_un;
      P (indet,p)
    in

    let fa = frac_nom fraction in
    let fb = frac_denom fraction in
    let fx = frac_indet fraction in

    let n = Array.length decomp_en_polynome_sans_carre in
    let m = ref 0 in

    for i = 0 to n-1 do
      m := !m + (deg_poly (get_0 decomp_en_polynome_sans_carre.(i))) * (get_1 decomp_en_polynome_sans_carre.(i))
    done;

    let mat_partielles = Array.make_matrix !m !m c_zero in
    let array_poly = Array.make !m c_zero in

    let list_decomp = ref [] in

    for i = 0 to n-1 do
      if deg_poly (get_0 decomp_en_polynome_sans_carre.(i)) > 0 then
        (
          for j = 1 to get_1 decomp_en_polynome_sans_carre.(i) do
            let q,r = div_euclid_poly (P (fx,fb)) (puissance_poly_ent (get_0 decomp_en_polynome_sans_carre.(i)) j) in
            assert (is_zero_ast r);
            list_decomp := (q,deg_poly (get_0 decomp_en_polynome_sans_carre.(i)))::(!list_decomp)
          done;
        )
    done;

    let tab_decomp = Array.of_list (List.rev !list_decomp) in
    let repere = ref 0 in

    for i = 0 to Array.length tab_decomp - 1 do
      for j = get_1 tab_decomp.(i) - 1 downto 0 do
        for k = 0 to deg_poly (get_0 tab_decomp.(i)) do
```

```
        mat_partielles.(k+j).(!repere) <- const_of_ast (coeff (get_0 tab_decomp.(i)) k)
      done;
      repere := !repere + 1;
    done;
  done;

  for i = 0 to deg_poly (P (fx,fa)) do
    array_poly.(i) <- const_of_ast (coeff (P (fx,fa)) i)
  done;

  let mat_res = systeme_cramer mat_partielles array_poly in
  let liste_res = ref [] in
  repere := 0;

  for i = 0 to Array.length tab_decomp - 1 do
      let p = ref (P (fx,[|ast_null|])) in
      for j = get_1 tab_decomp.(i) - 1 downto 0 do
        p := add_poly !p (mult_poly_scal (poly_x_k fx j) mat_res.(!repere));
        repere := !repere + 1
      done;
      liste_res := (!p)::!liste_res
  done;

  let array_res = Array.of_list (List.rev !liste_res) in

  let mat_decomp = Array.make_matrix n !m (P (fx, [|ast_null|])) in
  repere := 0;

  for i = 0 to n-1 do
    if deg_poly (get_0 decomp_en_polynome_sans_carre.(i)) <> 0 then (
      for j = 0 to ((get_1 decomp_en_polynome_sans_carre.(i))-1) do
        mat_decomp.(i).(j) <- array_res.(!repere); (* mettre un marker*)
        repere := !repere + 1;
      done;
    )
  done;

  mat_decomp
;;


let calcul_des_fractions_partielles_log fraction decomp_en_polynome_sans_carre =
  (* nb la fraction d'entré vérifier deg A < def B*)
  let get_0 (a,b) = a in
```

```
    let get_1 (a,b) = b in

    let coeff poly k = match poly with
      | P (x,tab) -> tab.(k)
      | _ -> failwith "calcul_des_fractions_partielles : pas un polynome"
    in

    let poly_x_k indet k =
      let p = Array.make (k+1) ast_null in
      p.(k) <- ast_un;
      P (indet,p)
    in

    let fa = frac_nom fraction in
    let fb = frac_denom fraction in
    let fx = frac_indet fraction in

    let n = Array.length decomp_en_polynome_sans_carre in
    let m = ref 0 in

    for i = 0 to n-1 do
      m := !m + (deg_poly (get_0 decomp_en_polynome_sans_carre.(i))) * (get_1 decomp_en_polynome_sans_carre.(i))
    done;

    let mat_partielles = Array.make_matrix !m !m ast_null in
    let array_poly = Array.make !m ast_null in

    let list_decomp = ref [] in

    for i = 0 to n-1 do
      if deg_poly (get_0 decomp_en_polynome_sans_carre.(i)) > 0 then
        (
          for j = 1 to get_1 decomp_en_polynome_sans_carre.(i) do
            let q,r = div_euclid_poly (P (fx,fb)) (puissance_poly_ent (get_0 decomp_en_polynome_sans_carre.(i)) j) in
            assert (is_zero_ast r);
            list_decomp := (q,deg_poly (get_0 decomp_en_polynome_sans_carre.(i)))::(!list_decomp)
          done;
        )
    done;

    let tab_decomp = Array.of_list (List.rev !list_decomp) in
    let repere = ref 0 in

    for i = 0 to Array.length tab_decomp - 1 do
```

```
      for j = get_1 tab_decomp.(i) - 1 downto 0 do
        for k = 0 to deg_poly (get_0 tab_decomp.(i)) do
          mat_partielles.(k+j).(!repere) <- (coeff (get_0 tab_decomp.(i)) k)
        done;
        repere := !repere + 1;
      done;
    done;

    for i = 0 to deg_poly (P (fx,fa)) do
      array_poly.(i) <- (coeff (P (fx,fa)) i)
    done;

    let mat_res = systeme_cramer_log mat_partielles array_poly in
    let liste_res = ref [] in
    repere := 0;

    for i = 0 to Array.length tab_decomp - 1 do
        let p = ref (P (fx,[|ast_null|])) in
        for j = get_1 tab_decomp.(i) - 1 downto 0 do
          p := add_poly !p (mult_poly (poly_x_k fx j) mat_res.(!repere));
          repere := !repere + 1
        done;
        liste_res := (!p)::!liste_res
    done;

    let array_res = Array.of_list (List.rev !liste_res) in
    let mat_decomp = Array.make_matrix n !m (P (fx, [|ast_null|])) in
    repere := 0;

    for i = 0 to n-1 do
      if deg_poly (get_0 decomp_en_polynome_sans_carre.(i)) <> 0 then (
        for j = 0 to ((get_1 decomp_en_polynome_sans_carre.(i))-1) do
          mat_decomp.(i).(j) <- array_res.(!repere); (* mettre un marker*)
          repere := !repere + 1;
        done;
      )
    done;

    mat_decomp
  ;;


  let rec frac_reconnait (fonction:ast_elem) (indet:ast_elem) =
    if egal_ast fonction indet then F (indet, [|ast_null;ast_un|], [|ast_un|]) (* voir Z = X quand reconnait faire
```

```
    attention *)
    else ( match fonction with
      | Arg0 f -> F (indet, [|fonction|], [|ast_un|])

      | Arg2 (f1, oper, f2) ->
        ( let frac1,frac2 = ajuste_frac (frac_reconnait f1 indet),ajuste_frac (frac_reconnait f2 indet) in
          match oper with
            | Plus -> add_frac frac1 frac2
            | Moins -> minus_frac frac1 frac2
            | Fois -> mult_frac frac1 frac2
            | Divise -> divise_frac frac1 frac2
            | Puissance -> (match frac2 with
                | F (x, [|Arg0 (C (Q {a = n; b = 1}))|], [|Arg0 (C (Q {a = 1; b = 1}))|]) -> puissance_frac_ent frac1 n
                | _ -> F (indet, [|Arg2 (f1, Puissance, f2)|], [|ast_un|])
              )
            | Compose -> compose_frac frac1 frac2
        )

      | Abstrait f -> frac_reconnait f.d_fonction indet

      | P (indet_x,poly_a) ->
        ( if egal_ast indet_x indet then F (indet,poly_a, [|ast_un|])
          else frac_reconnait (ast_of_poly_array poly_a indet_x) indet
        )

      | F (indet_x,poly_a,poly_b) ->
        ( if egal_ast indet_x indet then F (indet,poly_a,poly_b)
          else frac_reconnait (ast_of_frac_array poly_a poly_b indet_x) indet
        )
    )
  ;;


  let poly_reconnait (fonction:ast_elem) (indet:ast_elem) =
    match ajuste_frac (frac_reconnait fonction indet) with
    | F (x,poly,[|Arg0 (C (Q {a = 1; b = 1}))|]) -> P (x, poly)
    | _ -> failwith "poly_reconnait : n'arrive pas"
  ;;


  let resultant (poly1:ast_elem) (poly2:ast_elem) =
    (* résultante *)
    match (poly1,poly2) with
    | P (x,p1),P (y,p2) when egal_ast x y ->
```

```
    (
      let d1 = deg_poly poly1 in
      let d2 = deg_poly poly2 in

      let mat = Array.make_matrix (d1+d2) (d1+d2) ast_null in

      for i = 0 to d2 - 1 do
        for j = 0 to d1 do
          mat.(i).(i+j) <- p1.(d1-j)
        done;
      done;

      for i = 0 to d1 - 1 do
        for j = 0 to d2 do
          mat.(i+d2).(i+j) <- p2.(d2-j)
        done;
      done;

      det_ast mat
    )

  | _,_ -> failwith "polys_to_frac : pas des polynomes ou frac d'indet diff"
;;


let poly_factorisation poly =

  match poly with
  | P (x,p) ->
    (
      decomp_en_polynome_sans_carre poly
    )
  | _ -> failwith "poly_factorisation : pas un poly"
;;


let rec subsitue_abs ast alpha = match ast with
  | Abstrait a -> Arg0 (C alpha)
  | Arg2 (f1, oper, f2) -> Arg2 (subsitue_abs f1 alpha, oper, subsitue_abs f2 alpha)
  | P (x,a) -> P (x,Array.map (fun x -> subsitue_abs x alpha) a)
  | _ -> ast
;;
```

```
let partie_primitive poly = match poly with
  | P (x,a) -> (
    if deg_poly poly < 0 then Res (P (x, [|ast_const 0 1|]))
    else if deg_poly poly = 0 then Res (P (x, [|ast_const 1 1|]))
    else (
      let v = ref ast_un in
      for i = deg_poly poly downto 0 do
        if not (is_zero_ast a.(i)) then v := a.(i)
      done;
      let b = poly_array (simplifie_ast (mult_poly poly (Arg2 (ast_un, Divise, !v)))) in
      let res = ref true in
      for i = 0 to deg_poly poly do
        let bc,v = is_const_ast b.(i) in
        if not bc then res := false
        else b.(i) <- Arg0 (C v)
      done;
      if !res then Res (P (x,b)) else Null
    )
  )
  | _ -> failwith "partie_primitive : pas un poly"
;;





(* --------------- Integration --------------- *)


(* On détermine les extensions nécessaire à la primitivation et
on modifie la fonction en remplacant les fonctions par leurs extensions *)
let rec determiner_extension_aux fonction = match fonction with

  | Arg2 (f1, oper, f2) -> ( match f1,oper with
      | (Arg0 Ln,Compose) -> (1,fonction)::(determiner_extension_aux f2)
      | (Arg0 Exp,Compose) -> (2,fonction)::(determiner_extension_aux f2)
      | _ -> (determiner_extension_aux f1) @ (determiner_extension_aux f2)
    )

  | Abstrait f -> determiner_extension_aux f.fonction

  | P (x,a) -> (
    let res = ref (determiner_extension_aux x) in
    for i = 0 to Array.length a - 1 do
```

```
          res := !res @ (determiner_extension_aux a.(i))
        done;
        !res
        )

    | F (x,a,b) -> (
        let res = ref (determiner_extension_aux x) in
        for i = 0 to Array.length a - 1 do
          res := !res @ (determiner_extension_aux a.(i))
        done;
        for i = 0 to Array.length b - 1 do
          res := !res @ (determiner_extension_aux b.(i))
        done;
        !res
        )

    | _ -> []
;;


let determiner_extension fonction =
  let extension = ref (Ext (X,{nom = "X";fonction = Arg0 X;d_fonction = Arg0 X;etat_derive = 0},Xe)) in
  let lst_temp = ref [] in

  List.iter
    (
      fun (x,y) ->
      if not (List.exists (fun (a,b) -> a = x && egal_ast y b) !lst_temp)
      then (lst_temp := (x,y)::!lst_temp;
            extension := (Ext (if x = 1 then (Ln,{nom = "Ln";fonction = y;d_fonction = y;etat_derive = 0},!extension) 
            else (Exp,{nom = "Exp";fonction = y;d_fonction = y;etat_derive = 0},!extension))))

    )
    (List.sort (fun (a,b) (c,d) -> compare a c) (List.rev (determiner_extension_aux fonction)));
  !extension
;;


let rec appartient_ext ext elem =
  (* vérifie si un élément est inclus dans une extension *)
  match ext with
  | Xe -> false
  | Ext (f_elem,fonction_abstraite,extt) -> if egal_ast elem (Abstrait fonction_abstraite) then true else appartient_ext 
  extt elem
```

```
;;


let rec inclus_ext ext1 ext2 =
  (* vérifie si l'extension deux est inclus dans la premiere *)
  match ext2 with
  | Xe -> true
  | Ext (f_elem,fonction_abstraite,extt2) -> (appartient_ext ext1 (Abstrait fonction_abstraite)) && (inclus_ext ext1 extt2)
;;


let rec print_extension extension = match extension with
  | Xe -> ()

  | Ext (f_elem,f,ext_suiv) -> (print_ast (Abstrait f); print_ast f.fonction ; print_extension ext_suiv)
;;


(* On a f(theta_n) = a(theta_b)/b(theta_n) on normalise et on rend f unitaire par rapport à un *)
let rec normalise fonction theta =

  if egal_ast fonction theta.fonction then fonction

  else (
    match fonction with

    | Arg2 (f1, oper, f2) -> Arg2 (normalise f1 theta, oper, normalise f2 theta)

    | Abstrait f -> normalise f.d_fonction theta

    | P (x,a) -> (
      let indet = normalise x theta in
      let aa = Array.make (Array.length a) ast_null in
      for i = 0 to Array.length a - 1 do
        aa.(i) <- normalise a.(i) theta
      done;
      P (indet,aa)
      )

    | F (x,a,b) -> (
      let indet = normalise x theta in
      let aa = Array.make (Array.length a) ast_null in
      for i = 0 to Array.length a - 1 do
```

```
        aa.(i) <- normalise a.(i) theta
      done;
      let bb = Array.make (Array.length b) ast_null in
      for i = 0 to Array.length b - 1 do
        bb.(i) <- normalise b.(i) theta
      done;
      F (indet,aa,bb)
      )

    | _ -> fonction
  )
;;



(* prend en parametre une fraction rationnel / un polynome et renvoie la partie rationnel intégré,
la partie polynomiale non intégrés et la partie logarithmique non intégré *)
let hermite_method fonction =
  (* voir p503 computer algebra *)

  let get_0 (a,b) = a in
  (* let get_1 (a,b) = b in *)

  match fonction with
  | F (Arg0 X,a,b) ->
    (
      let part_poly,part_frac_nom = div_euclid_poly (P (Arg0 X, a)) (P (Arg0 X, b)) in
      let part_frac_den,u = poly_unitaire (P (Arg0 X, b)) in
      let frac_red = polys_to_frac (mult_poly part_frac_nom (P (Arg0 X,[|Arg2(ast_un,Divise,u)|]))) part_frac_den in
      if (not (is_zero_ast frac_red)) then (
        let decomp_sans_carre = decomp_en_polynome_sans_carre part_frac_den in (* tableau de polynome *)
        let frac_partielles = calcul_des_fractions_partielles frac_red decomp_sans_carre in (* matrice carré avec nb *)

        let part_rationnelle = ref (F (Arg0 X, [|ast_null|], [|ast_un|])) in
        let part_int = ref (F (Arg0 X, [|ast_null|], [|ast_un|])) in

        let nb_carre = Array.length decomp_sans_carre in

        for i = 0 to nb_carre - 1 do

          part_int := ajuste_frac (add_frac !part_int (polys_to_frac frac_partielles.(i).(0) (get_0 decomp_sans_carre.(i))));

          for j = 2 to i+1 do
```

```
            let n = ref j in
            while !n > 1 do
              (* solve (s*q[i] + t*q[i]' = r[i,n]) à l'aide bezout car pgcd (q[i],q[i]') = 1 car decomp sans carre *)
              let (ps,pt) =
                match poly_extend_euclidean (get_0 decomp_sans_carre.(i)) (ajuste_poly (derive (get_0 decomp_sans_carre.(i)))) frac_partielles.(i).(!n-1) with
                  | p1,p2 ->

                    let pp3,u3 = poly_unitaire frac_partielles.(i).(!n-1) in

                    (ajuste_poly (mult_poly p1 (P (Arg0 X, [|u3|]))),ajuste_poly (mult_poly p2 (P (Arg0 X, [|u3|]))))
              in (* résout une équation pour trouver les polynomes ...*)

              n := !n - 1;

              let ft = polys_to_frac (mult_poly_scal pt (Q {a = 1; b = !n})) (puissance_poly_ent (get_0 decomp_sans_carre.(i)) !n) in

              part_rationnelle := ajuste_frac (minus_frac !part_rationnelle ft);

              frac_partielles.(i).(!n-1) <- add_poly ps (mult_poly_scal (derive pt) (Q {a = 1; b = !n})); (* r[i,n] <- s + t'/n *)


            done;

            part_int := ajuste_frac (add_frac !part_int (polys_to_frac frac_partielles.(i).(0) (get_0 decomp_sans_carre.(i))));
            assert (deg_frac !part_int < 0);
          done;
        done;

        (ajuste_frac !part_rationnelle,ajuste_poly part_poly,ajuste_frac !part_int)
      )
      else
        (F (Arg0 X,[|ast_null|],[|ast_un|]),ajuste_poly part_poly, F (Arg0 X,[|ast_null|],[|ast_un|]))
    )

  | P (Arg0 X,p) -> (F (Arg0 X, [|ast_null|], [|ast_un|]),fonction,F (Arg0 X, [|ast_null|], [|ast_un|]))
  | _ -> failwith "Fonction ne correspondant pas à hermite"
;;
```

```
let rosthein_trager_method fonction =
  (* p507 intégration de la partie log de la fraction rationnel monic et square free *)
  let get_0 (a,b) = a in
  let get_c ast = match ast with | Arg0 (C c) -> c | _ -> E {nom = "erreur"; approx = 0.} in

  match fonction with
    | F (x,a,b) when deg_frac fonction < 0 ->
      (
        let pa = P (x,a) in
        let pb = P (x,b) in
        let pdb = ajuste_poly (derive pb) in

        let res_z = ajuste_poly (poly_reconnait (resultant (minus_poly pa (mult_poly (P (x,[|ast_Z|])) pdb)) pb) ast_Z) in

        print_ast_arbre res_z;

        let tab_facteur = poly_factorisation res_z in
        let part_log = ref ast_null in
        let id_alpha = ref 0 in

        for i = 0 to Array.length tab_facteur - 1 do
          let p,x = match get_0 tab_facteur.(i) with | P (x,p) -> (p,x) | _ -> ([||],ast_null) in
          if deg_poly (get_0 tab_facteur.(i)) = 1
            then (
              let c = mult_constante (div_constante (get_c p.(0)) (get_c p.(1))) (c_const (-1) 1) in (* c = -c0 ou c0 racine*)
              let v = pgcd_poly (minus_poly pa (mult_poly_scal pdb c)) pb in
              part_log := Arg2 (!part_log, Plus, Arg2 (Arg0 (C c), Fois, Arg2 (Arg0 Ln, Compose, v)))
            )
          else (
            let c = {nom = "α_"; fonction = Arg0 Exp; d_fonction = Arg0 Ln; etat_derive = 0} in
            let v = pgcd_poly_algebrique (minus_poly pa (mult_poly pdb (Abstrait c))) pb in
            print_ast_arbre (pgcd_poly (minus_poly pa (mult_poly_scal pdb (c_const (1) 1))) pb);

            for i = 0 to deg_poly (get_0 tab_facteur.(i)) - 1 do
              id_alpha := !id_alpha + 1;
              part_log := Arg2 (!part_log, Plus,
                           Arg2 (Arg0 (C (E {nom = "α_"^(string_of_int !id_alpha); approx = 0.1}))), Fois,
                           Arg2 (Arg0 Ln, Compose, subsitue_abs v (E {nom = "α_"^(string_of_int !id_alpha); approx = 0.1}))));
            done;
          )

```

```
        done;
        !part_log
      )
    | _ -> failwith "rosthein_trager_method : pas une fraction rationnelle ou degré incohérent"
;;


let integration_polynome fonction =
  (* intégre un polynome d'indéterminé X a coefficient dans Q *)
  match fonction with
  | P (Arg0 X, p1) ->
      ( let primitive = Array.make (Array.length p1 + 1) (ast_null) in
        for i = 1 to (Array.length p1) do
          match p1.(i-1) with
          | Arg0 (C c) -> primitive.(i) <- simplifie_ast (Arg2 (p1.(i-1), Divise, ast_const i 1))
          | _ -> failwith "pas un polynome à coeff dans C"
        done;
        ajuste_poly (P (Arg0 X,primitive))
      )

  | _ -> failwith "integration_polynome : pas un polynome"
;;


let integration_rationnel fonction =
  (* intégre les fonctions d'indéterminé x à coefficient dans C *)
  match fonction with
    | F (Arg0 X,a,b) -> (

      let d = ajuste_frac fonction in

      if deg_frac d >= 0 then (
        let f,g,h = hermite_method d in

        let partie_rationnel = f in
        let partie_poly = ajuste_poly (integration_polynome g) in
        let partie_log = if is_zero_ast h then ast_null else (rosthein_trager_method h) in

        (Arg2 (partie_rationnel, Plus, Arg2 (partie_poly, Plus, partie_log)))
      ) else (
        if is_zero_ast d then ast_null else (rosthein_trager_method d)
      )
    )

```

```
    | P (x,p) -> ajuste_poly (integration_polynome fonction)

    | _ -> (
      failwith "integration rationnel pas une fraction rationnel"
    )
;;




(* primitive si possible la fonction sinon renvoie Null *)
let rec risch fonction extensions =
  (* On détermine les extensions nécessaire à la primitivation et
  on modifie la fonction en remplacant les fonctions par leurs extensions theta_i *)

  (* Recupère la derniere extension *)
  let type_ext = ref X in
  let theta_n = ref ({nom = "BL"; fonction = Arg0 X; d_fonction = Arg0 X; etat_derive = 0}) in
  let corps_diff = ref Xe in

  (
    match extensions with
    | Xe -> ()
    | Ext (t_ext, theta, reste_corps) -> begin type_ext := t_ext; theta_n := theta; corps_diff := reste_corps end
  );

  if ( match !type_ext with
      | X | Ln -> false
      | _ -> true )
  then Notimplementederror

  else (
    (* On a f(theta_n) = a(theta_b)/b(theta_n) on normalise et on rend f unitaire par rapport à theta_n *)
    let frac_f = match ajuste_frac (frac_reconnait (normalise fonction !theta_n) !theta_n.fonction) with
      | F (indet,a,[|Arg0 C (Q {a = 1; b = 1})|]) -> P (indet, a)
      | x -> x
    in

    (* print_ast frac_f; *)

    match !type_ext with
    | X -> Res (integration_rationnel frac_f)
    | Ln -> log_case frac_f !corps_diff
```

```
    | Exp -> exp_case frac_f !corps_diff
    | _ -> Null (* erreur inutile déjà prévenu *)
  )


and log_case fonction extensions =

  let hermite_method_log f1 =
    (* voir p503 computer algebra *)

    let get_0 (a,b) = a in
    (* let get_1 (a,b) = b in *)

    match f1 with
    | F (x,a,b) ->
      (

        assert (
          match x with
          | Arg2 (Arg0 Ln, Compose, _) -> true
          | _ -> false
        );

        let part_poly_z,part_frac_nom_z = div_euclid_poly (P (Arg0 Z, a)) (P (Arg0 Z, b)) in
        let part_frac_den,u = poly_unitaire (P (Arg0 Z, b)) in
        let frac_red = polys_to_frac (mult_poly part_frac_nom_z (P (Arg0 Z,[|Arg2(ast_un,Divise,u)|]))) part_frac_den in

        if (not (is_zero_ast frac_red)) then (

          let decomp_sans_carre_z = decomp_en_polynome_sans_carre part_frac_den in (* tableau de polynome *)
          let frac_partielles_z = calcul_des_fractions_partielles_log frac_red decomp_sans_carre_z in (* matrice carré avec nb *)

          (*transformer en ln les Z*)
          let part_poly = match part_poly_z with | P (Arg0 Z,a) -> P (x,a) | _ -> failwith "hermite_log_poly" in
          let decomp_sans_carre = Array.map (fun (y,i) -> match y with | P (Arg0 Z,a) -> (P (x,a),i) | _ -> failwith "hermite_log") decomp_sans_carre_z in (* tableau de polynome *)
          let frac_partielles = Array.make_matrix (Array.length frac_partielles_z) (Array.length frac_partielles_z.(0)) ast_null in (* matrice carré avec nb *)
          for i = 0 to (Array.length frac_partielles_z) - 1 do
            for j = 0 to Array.length frac_partielles_z.(0) - 1 do
              frac_partielles.(i).(j) <- match frac_partielles_z.(i).(j) with | P (Arg0 Z,a) -> P (x,a) | _ -> failwith "hermite_log"
            done;
```

```
          done;

          let part_rationnelle = ref (F (x, [|ast_null|], [|ast_un|])) in
          let part_int = ref (F (x, [|ast_null|], [|ast_un|])) in

          let nb_carre = Array.length decomp_sans_carre in

          for i = 0 to nb_carre - 1 do

            part_int := ajuste_frac (add_frac !part_int (polys_to_frac frac_partielles.(i).(0) (get_0 decomp_sans_carre.(i))));

            for j = 2 to i+1 do

              let n = ref j in
              while !n > 1 do

                let (ps,pt) =
                  match poly_extend_euclidean (get_0 decomp_sans_carre.(i)) (ajuste_poly (derive (get_0 decomp_sans_carre.(i)))) frac_partielles.(i).(!n-1) with
                    | p1,p2 ->
                      let pp3,u3 = poly_unitaire frac_partielles.(i).(!n-1) in
                      (ajuste_poly (mult_poly p1 (P (Arg0 X, [|u3|]))),ajuste_poly (mult_poly p2 (P (Arg0 X, [|u3|]))))
                in (* résout une équation pour trouver les polynomes ...*)

                n := !n - 1;

                let ft = polys_to_frac (mult_poly_scal pt (Q {a = 1; b = !n})) (puissance_poly_ent (get_0 decomp_sans_carre.(i)) !n) in

                part_rationnelle := ajuste_frac (minus_frac !part_rationnelle ft);

                frac_partielles.(i).(!n-1) <- add_poly ps (mult_poly_scal (derive pt) (Q {a = 1; b = !n})); (* r[i,n] <- s + t'/n *)

              done;

              part_int := ajuste_frac (add_frac !part_int (polys_to_frac frac_partielles.(i).(0) (get_0 decomp_sans_carre.(i))));

              assert (deg_frac !part_int < 0);
            done;
          done;

```

```
            (ajuste_frac !part_rationnelle,ajuste_poly part_poly,ajuste_frac !part_int)
          ) else (
            (ast_null,P (x,poly_array part_poly_z),ast_null)
          )
        )

      | P (x,p) -> (F (x, [|ast_null|], [|ast_un|]),f1,F (x, [|ast_null|], [|ast_un|]))

      | _ -> failwith "Fonction ne correspondant pas à hermite_log"
    in

    let integration_polynome_log f1 =
      match f1 with
      | P (x, p1) -> (
        assert (
          match x with
          | Arg2 (Arg0 Ln, Compose, _) -> true
          | _ -> false
        );

        let l = deg_poly f1 in
        let dtheta = derive x in
        let ppoly = Array.make (l+2) ast_null in

        let valide = ref true in

        for i = l downto 0 do

          ppoly.(i) <- (match risch (simplifie_ast (Arg2 (p1.(i),Moins,Arg2(ast_const (i+1) 1,Fois,Arg2(ppoly.(i+1),Fois,dtheta))))) extensions with
              | Res f -> (* print_ast f; *) if (not (inclus_ext extensions (determiner_extension f)) && i <> 0) then valide := false; f
              | x -> valide := false ; ast_null
            )
        done;

        if !valide then Res (P (x, ppoly)) else Null
        )

      | _ -> failwith "integration_polynome_log"
    in

    let rosthein_trager_method_log f1 =
      (* p507 intégration de la partie log de la fraction rationnel monic et square free *)
```

```
      let get_0 (a,b) = a in
      let get_c ast = match ast with | Arg0 (C c) -> c | _ -> E {nom = "erreur"; approx = 0.} in

      match f1 with
        | F (x,a,b) when deg_frac f1 < 0 ->
          (
            assert (
              match x with
              | Arg2 (Arg0 Ln, Compose, _) -> true
              | _ -> false
            );

            let pa = P (x,a) in
            let pb = P (x,b) in
            let pdb = ajuste_poly (derive pb) in

            let res_z = ajuste_poly (poly_reconnait (resultant (minus_poly pa (mult_poly (P (x,[|ast_Z|])) pdb)) pb) ast_Z) in

            print_ast_arbre res_z;

            (*
              vérifier si la partie primitive de res_z est dans Q[x] p25 bronstein
              sinon n'est pas élémentaire
            *)
            match partie_primitive res_z with
              | Null | Notimplementederror -> Null
              | Res poly_z -> (
                let tab_facteur = poly_factorisation poly_z in

                let part_log = ref ast_null in
                let id_alpha = ref 0 in

                for i = 0 to Array.length tab_facteur - 1 do
                  let p,x = match get_0 tab_facteur.(i) with | P (x,p) -> (p,x) | _ -> ([||],ast_null) in
                  if deg_poly (get_0 tab_facteur.(i)) = 1
                    then (
                      let c = mult_constante (div_constante (get_c p.(0)) (get_c p.(1))) (c_const (-1) 1) in (* c = -c0 ou c0 racine*)
                      let v = pgcd_poly (minus_poly pa (mult_poly_scal pdb c)) pb in
                      part_log := Arg2 (!part_log, Plus, Arg2 (Arg0 (C c), Fois, Arg2 (Arg0 Ln, Compose, v)))
                    )
                  else (
                    let c = {nom = "α_"; fonction = Arg0 Exp; d_fonction = Arg0 Ln; etat_derive = 0} in
```

```
                  let v = pgcd_poly_algebrique (minus_poly pa (mult_poly pdb (Abstrait c))) pb in

                  print_ast_arbre (pgcd_poly (minus_poly pa (mult_poly_scal pdb (c_const (1) 1))) pb);

                  (* Lazard-Rioboo-Trager algorithm remove pgcd issue *)
                  for i = 0 to deg_poly (get_0 tab_facteur.(i)) - 1 do
                    id_alpha := !id_alpha + 1;
                    part_log := Arg2 (!part_log, Plus,
                                 Arg2 (Arg0 (C (E {nom = "α_"^(string_of_int !id_alpha); approx = 0.1})), Fois,
                                 Arg2 (Arg0 Ln, Compose, subsitue_abs v (E {nom = "α_"^(string_of_int !id_alpha); approx = 0.1})))));
                  done;
                )

              done;
              Res !part_log
            )
        )
      | _ -> failwith "rosthein_trager_method : pas une fraction rationnelle ou degré incohérent"
  in

  let r,p,l = hermite_method_log fonction in

  let pi = if not (is_zero_ast p) then integration_polynome_log p else Res ast_null in

  let pl = if not (is_zero_ast l) then rosthein_trager_method_log l else Res ast_null in

  match pl,pi with
  | Res pll,Res pii  -> Res (simplifie_ast (Arg2 (simplifie_ast pii, Plus, Arg2 (simplifie_ast r, Plus, simplifie_ast pll))))
  | Null,Null | Null,_ | _,Null -> Null
  | Notimplementederror,Notimplementederror | _,Notimplementederror | Notimplementederror,_ -> Notimplementederror


and exp_case fonction extension_list =
  (* pas implémenter peut être dans un futur proche *)
  Notimplementederror
;;





let ast_of_string str =
```

```
  (* input un string , lit caractère par caractère utilise les () pour chaque expression ex : ((X) + [1.01] / ((exp) o
  ((X) ^ [2/1])) *)
  simplifie_ast (parse (analyse_lexicale str))
;;


let jolie_affichage () =
  Graphics.set_color (Graphics.rgb 203 195 177);
  Graphics.fill_rect 0 0 800 800;
  Graphics.set_color (Graphics.rgb 60 64 63);
  Graphics.fill_rect 0 0 50 800;
  Graphics.fill_rect 0 750 800 50;
  Graphics.fill_rect 0 0 800 50;
  Graphics.fill_rect 750 0 50 800;
  Graphics.fill_rect 0 690 800 25;
  Graphics.set_color Graphics.black;
;;


let interface () =
  Graphics.open_graph " 800x800";
  Graphics.set_window_title "Algorithme de Risch";
  Graphics.moveto 75 725;
  jolie_affichage ();
  (* Graphics.set_text_size 10; ne fonctionne pas *)

  let input_str = ref "" in
  let input_char = ref (Graphics.read_key ()) in
  let x = Graphics.current_x () in

  while !input_char <> '!' do
    input_str := !input_str ^ (String.make 1 !input_char);
    Graphics.clear_graph ();
    jolie_affichage ();
    Graphics.draw_string !input_str;
    Graphics.moveto x (Graphics.current_y ());
    input_char := (Graphics.read_key ());
  done;

  let fonction = ast_of_string !input_str in

  print_ast_arbre fonction;

  Graphics.moveto x (Graphics.current_y () - 50);
```

```
    (match risch fonction (determiner_extension fonction) with
      | Res f -> (print_ast f; print_ast_arbre f; print_ast_arbre_graphics f)
      | _ -> (Printf.fprintf file "Pas de primitive elementaire\n\n\n"; Graphics.draw_string "Pas de primitive elementaire" 
       ));

    Graphics.read_key ()
  ;;







  (* --------------- --------------- Test et assertions --------------- --------------- *)




  let mass_test nb =

    let echec = ref 0 in
    let reussite = ref 0 in

    let print_test ast =

      print_ast ast;

      match risch ast (determiner_extension ast) with
      | Res f -> (
          print_ast_arbre f;
          if egal_ast ((frac_reconnait (derive f) (Arg0 X))) ast
            then (reussite := !reussite + 1; Printf.fprintf file "\n\n Réussite total ! \n\n ")
          else Printf.fprintf file "\n\n Echec \n\n"
        )
      | _ -> Printf.fprintf file "\n\n Echec \n\n"

    in

    let randint a b = (Random.int (b+1-a)) + a in

    let retest = Array.make nb ast_null in

    for i = 0 to nb-1 do

```

```
    let n = randint 2 6 in
    let m = randint 2 5 in
    let pa = Array.make n ast_null in
    let pb = Array.make m ast_null in
    for j = 0 to n - 1 do
      pa.(j) <- simplifie_ast (ast_const (randint 1 20) (randint 1 7));
    done;
    for j = 0 to m - 2 do
      pb.(j) <- simplifie_ast (ast_const (randint 1 10) (randint 1 1));
    done;
    pb.(m-1) <- (ast_const 1 1);

    assert (not (is_zero_ast (P (Arg0 X, pa))));
    assert (not (is_zero_ast (P (Arg0 X, pb))));

    let frac = F (Arg0 X, pa, pb) in

    retest.(i) <- frac;

    print_ast_arbre frac;

    try print_test frac with
      Int_overflow -> echec := !echec + 1; retest.(i) <- ast_null

  done;

  print_int nb;
  print_newline ();
  print_int !echec;
  print_newline ();
  print_int !reussite;
  print_newline ();

  for i = 0 to nb-1 do
    print_debug i;
    if not (is_zero_ast retest.(i)) then print_test retest.(i);
    print_debug i;
  done;
;;


let print_stat () =
  print_debug appel_tab.(0);
  print_debug appel_tab.(1);
```

```
  print_debug appel_tab.(2)
;;





(*
  mass_test 1000;;
  Printf.fprintf file "\n\n\n\n";;
*)



interface ();;



print_stat ();;






(* ---------------------------------------------   FIN DU FICHIER   -------------------------------------- *)


close_out file;;
```