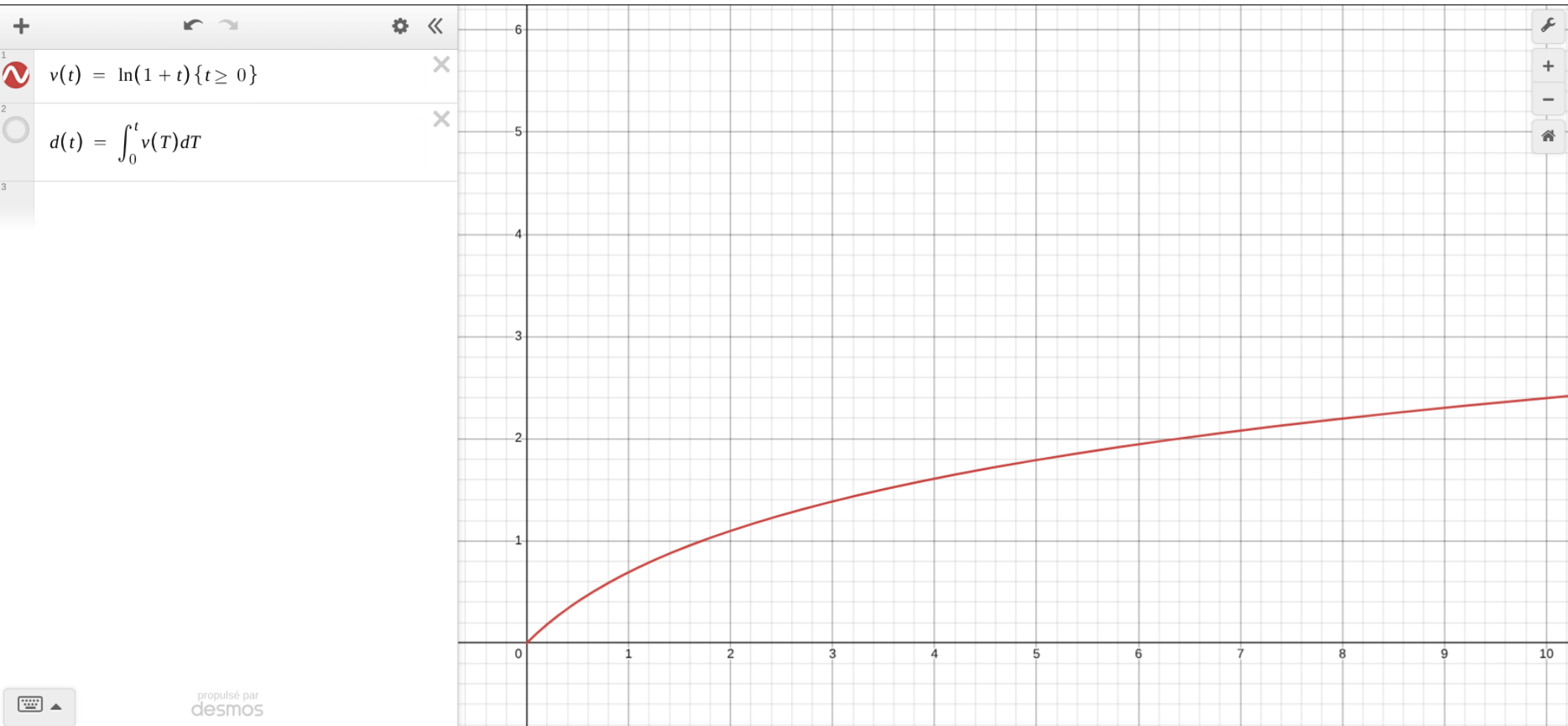


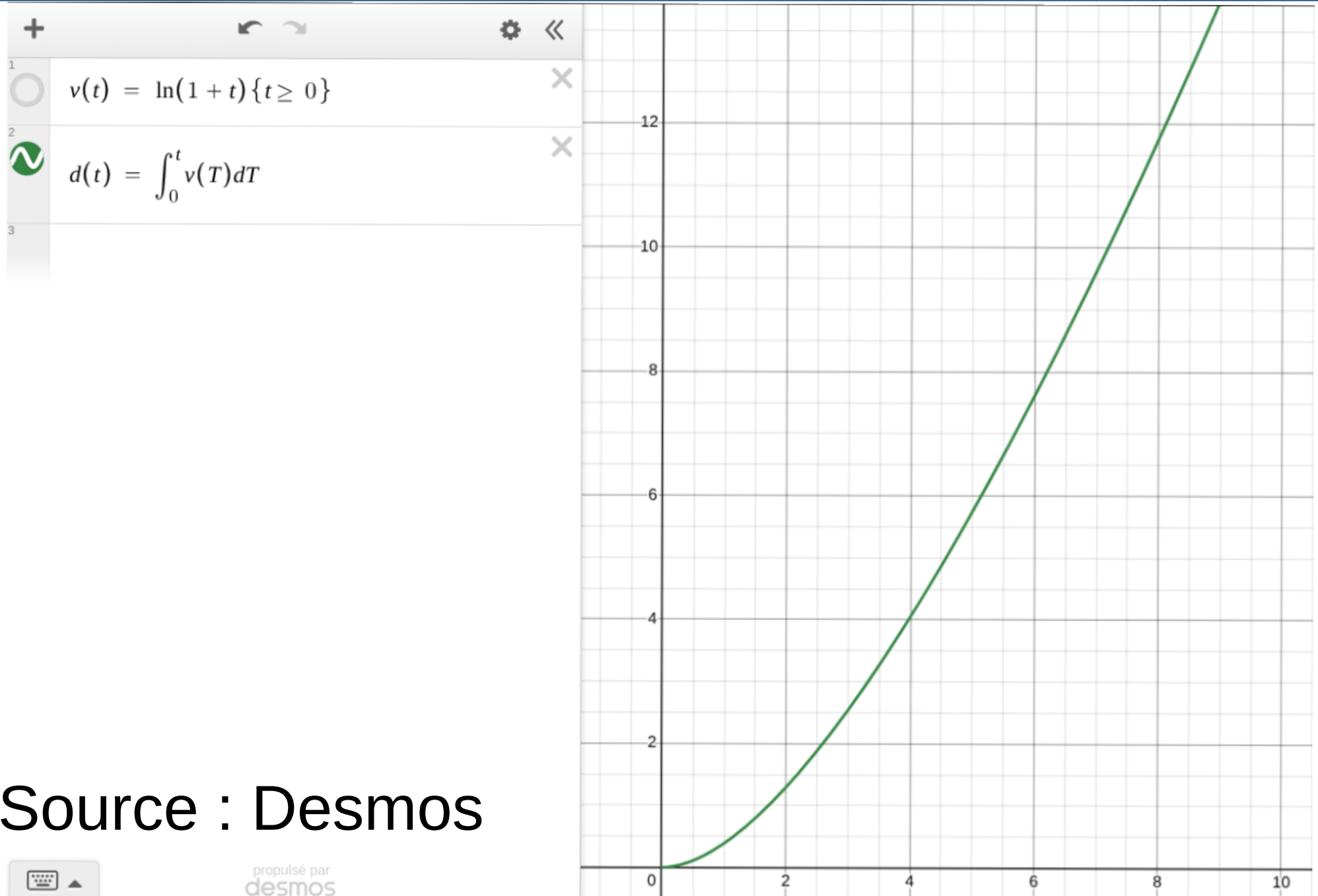
Procédure de primitivation de fonctions rationnelles et logarithmiques par le biais de l'algorithme de Risch

Introduction



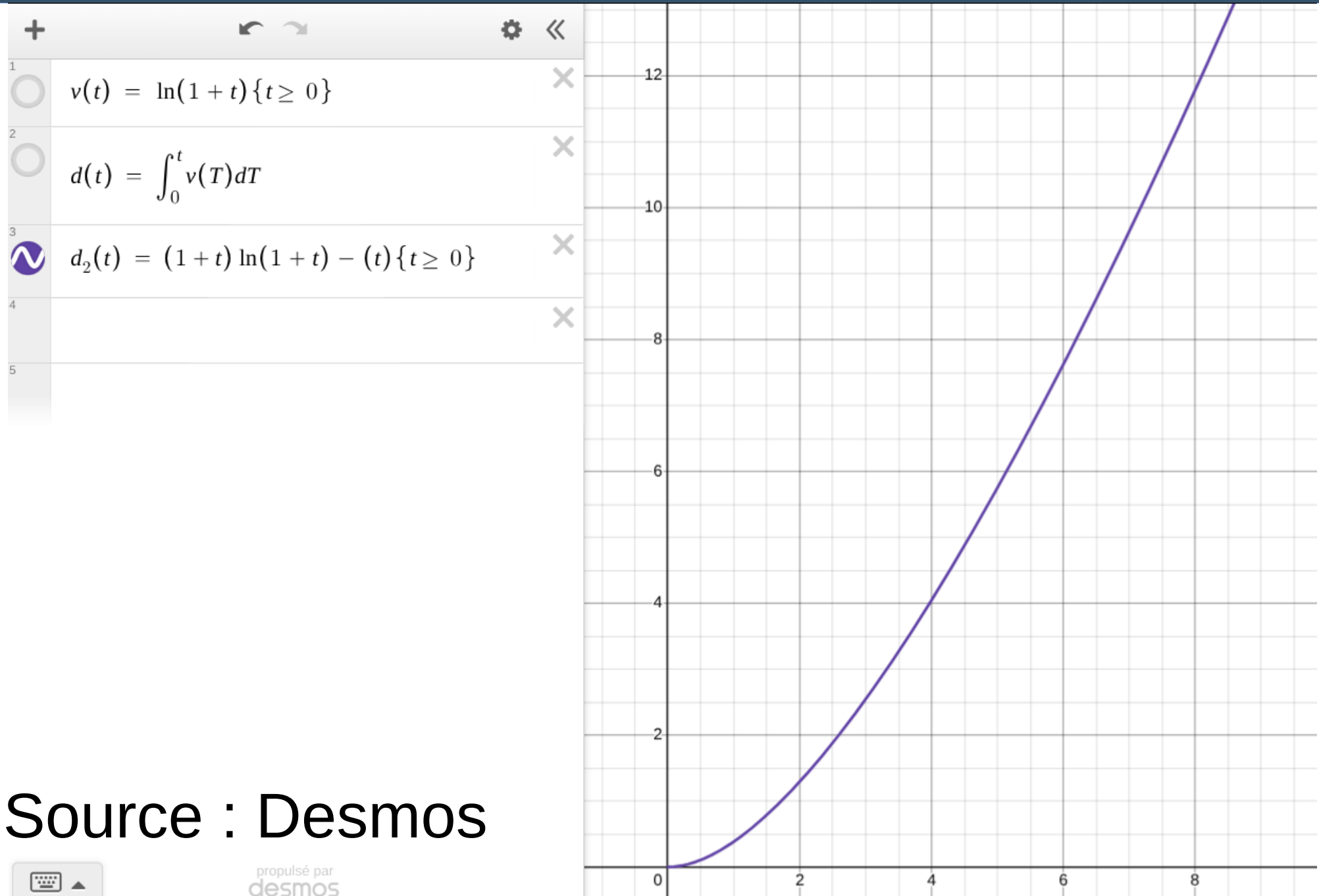
Source : Desmos

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

$$\operatorname{li}(x) = \int_0^x \frac{dt}{\ln(t)}.$$

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

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;;

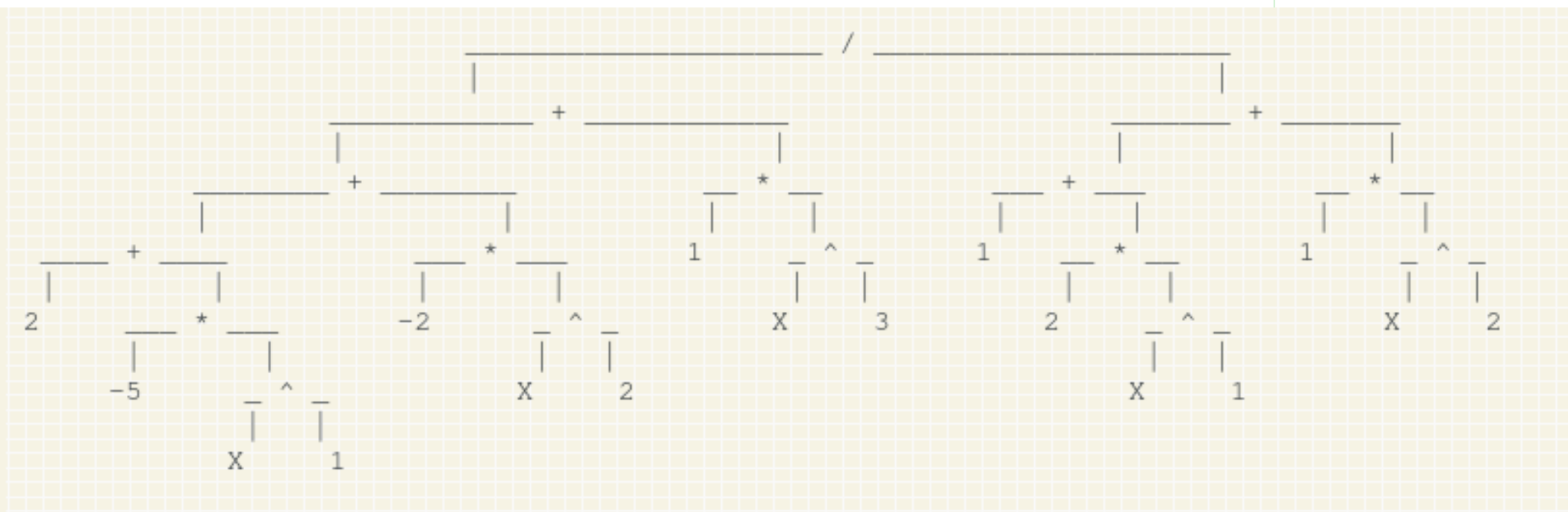
(* type contenant les opérations élémentaires *)
type op_elem = Plus | Moins | Fois | Divise | Puissance | Compose;;

(* 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};;
```

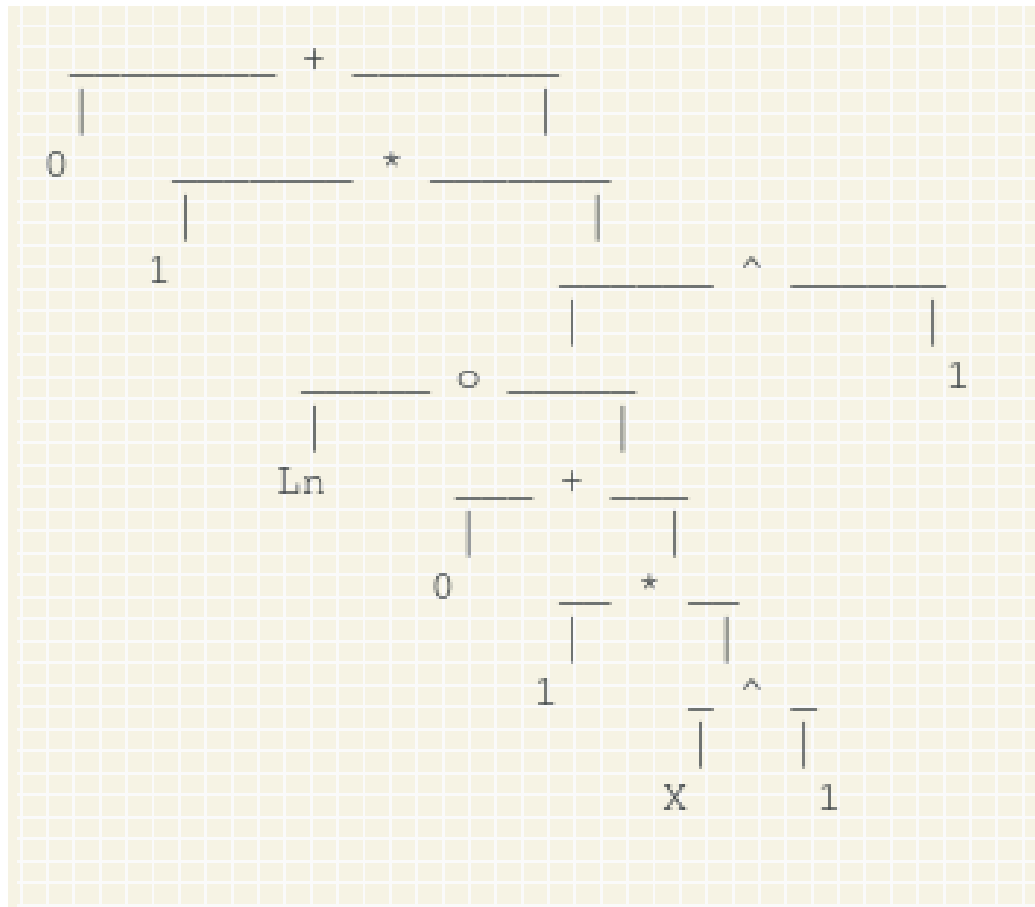

Les fonctions

```
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-Rosenthal :

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 $v_0, v_1, \dots, v_n \in F$ et $c_1, \dots, c_n \in \text{Con}(F)$ 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

```
1: procedure RISCH( $f, ext :: lst$ )                                ▷ avec  $f \in \mathbb{K}[X][ext :: lst]$  et  $ext$  la dernière extension
2:
3:    $f \leftarrow normalise(f, ext)$ 
4:
5:   if  $type(ext) = rationnel$  then
6:     return  $integration\_rationnel(f)$ 
7:
8:   else if  $type(ext) = logarithmique$  then
9:     return  $integration\_logarithmique(f, lst)$                 ▷ appel à Risch récursif dans ce cas
10:
11:   else
12:     return  $Non\_implémenté$ 
13:   end if
14:
15: end procedure
```

L'algorithme de Risch

Algorithm 2 Méthode d'Hermite

```
1: procedure HERMITE( $F(\theta)$ )
2:    $P \leftarrow \text{partie\_polynomial}(F)$ 
3:    $G \leftarrow \text{partie\_fraction}(F)$ 
4:    $FSC \leftarrow \text{factorisation\_sans\_carré}(G)$ 
5:    $DSC \leftarrow \text{décomposition\_sans\_carré}(G, FSC)$ 
6:    $FI, FL \leftarrow 0, 0$ 
7:
8:   for  $H \in DSC$  do
9:      $A, Q \leftarrow H = \frac{A}{Q^i}$ 
10:    if  $i = 1$  then
11:       $FL \leftarrow FL + H$ 
12:
13:    else
14:       $i \leftarrow i - 1$ 
15:       $S, T \leftarrow \text{euclide\_étendu}(Q, A)$  ▷  $S \cdot Q + T \cdot Q' = A$  possible car  $Q \wedge Q' = 1$ 
16:       $FI \leftarrow FI - \frac{T}{i \cdot Q^i}$  ▷ Intégration par partie
17:       $DSC \leftarrow \frac{i \cdot S + T'}{i \cdot Q^i} :: DSC$ 
18:    end if
19:  end for
20:  return  $(P, FI, FL)$ 
21: end procedure
```

L'algorithme de Risch

Algorithm 3 Méthode de Rothstein Trager

```
1: procedure ROTHSTEIN-TRAGER( $F(\theta) = \frac{A(\theta)}{B(\theta)}$ )    ▷  $B$  unitaire et sans carré et  $\deg(A) < \deg(B)$ 
2:
3:    $R(Z), \text{réussite} \leftarrow \text{partie\_primitive}(\text{résultante}(A(\theta) - Z \cdot B(\theta)', B(\theta)))$ 
4:    $[R_1; \dots; R_n] \leftarrow \text{factorisation}(R(Z))$                                 ▷ Si possible
5:
6:   if  $\text{réussite}$  then
7:      $L \leftarrow 0$ 
8:     for  $i = 1$  to  $n$  do
9:        $[\alpha_1; \dots; \alpha_{\deg(R_i)}] \leftarrow \text{racines}(R_i)$ 
10:      for  $j = 1$  to  $\deg(R_i)$  do
11:         $L \leftarrow L + \alpha_j \cdot \ln(\text{PGCD}(A(\theta) - \alpha_j \cdot B(\theta)', B(\theta)))$ 
12:      end for
13:    end for
14:    return  $L$ 
15:  else
16:    return Primitive non élémentaire
17:  end if
18:
19: end procedure
```

L'algorithme de Risch

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 $\mathbb{K}[\theta]$

$$\sum_{i=0}^n p_i \theta^i = \left(\sum_{i=0}^{n+1} q_i \theta^i \right)' + \sum_{j=0}^m c_j \frac{v_j'}{v_j} \iff \begin{cases} 0 = q'_{n+1} \\ p_n = (n+1)q_{n+1}\theta' + q'_n \\ \vdots \\ p_1 = 2q_2\theta' + q'_1 \\ p_0 = q_1\theta' + q'_0 + \sum_{j=0}^m c_j \frac{v_j'}{v_j} \end{cases}$$

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

L'algorithme de Risch

Intégration de polynômes logarithmiques récursivement

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

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

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

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

L'algorithme de Risch

Intégration de polynômes logarithmiques
récursivement

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

L'algorithme de Risch

Intégration de polynômes logarithmiques
récursivement

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

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

$$\text{Si } \left\{ \begin{array}{l} 1 + x \neq x + 1 \\ (1 + x)^2 \neq 1 + 2x + x^2 \\ x - x \neq 0 \\ F(x) = \frac{1+x}{1} \neq 1 + x = P(x) \end{array} \right.$$

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

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_constant x y))
| Arg2 (f1, Plus, f2) when egal_ast f1 f2 -> Arg2 (Arg0 (C (Q {a = 2; b = 1})), Foiss, 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, Foiss, f4)
| Arg2 (f3, Moins, Arg0 (C (Q {a = 0; b = 1}))) -> f3
| Arg2 (Arg0 (C x), Moins, Arg0 (C y)) -> Arg0 (C (minus_constant x y))
| Arg2 (f3, Moins, Arg0 (C x)) -> Arg2 (f3, Plus, Arg0 (C (neg_constant 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})), Foiss , f4) -> ast_null
| Arg2 (f3 ,Foiss , Arg0 (C (Q {a = 0; b = 1}))) -> ast_null
| Arg2 (Arg0 (C x) ,Foiss , Arg0 (C y)) -> Arg0 (C (mult_constant x y))
| Arg2 (Arg0 (C (Q {a = 1; b = 1})), Foiss , f4) -> f4
| Arg2 (f3 ,Foiss , Arg0 (C (Q {a = 1; b = 1}))) -> f3
| Arg2 (P (_, [|Arg0 (C (Q {a = 1; b = 1})))|], Foiss , f4) -> f4
| Arg2 (f3 ,Foiss , P (_, [|Arg0 (C (Q {a = 1; b = 1})))|]) -> f3
| Arg2 (f1, Foiss, 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 :

```
and is_zero_ast ast_arbre =
  (* test de manière imprécise si une expression est nul, risque 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

 $\ln(x)$

$$\frac{\frac{\frac{0}{x-1} + \frac{-1}{x-1}}{x-1} + \frac{\frac{0}{x-1} + \frac{1}{x-1}}{x-1}}{x-1} = \frac{\frac{0}{x-1} + \frac{1}{x-1}}{x-1} = \frac{1}{x-1}$$

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

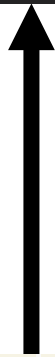
[Done] exited with code=0 in 312.923 seconds
```

Résultats

Avec cinq pôles maximum

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

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



- 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

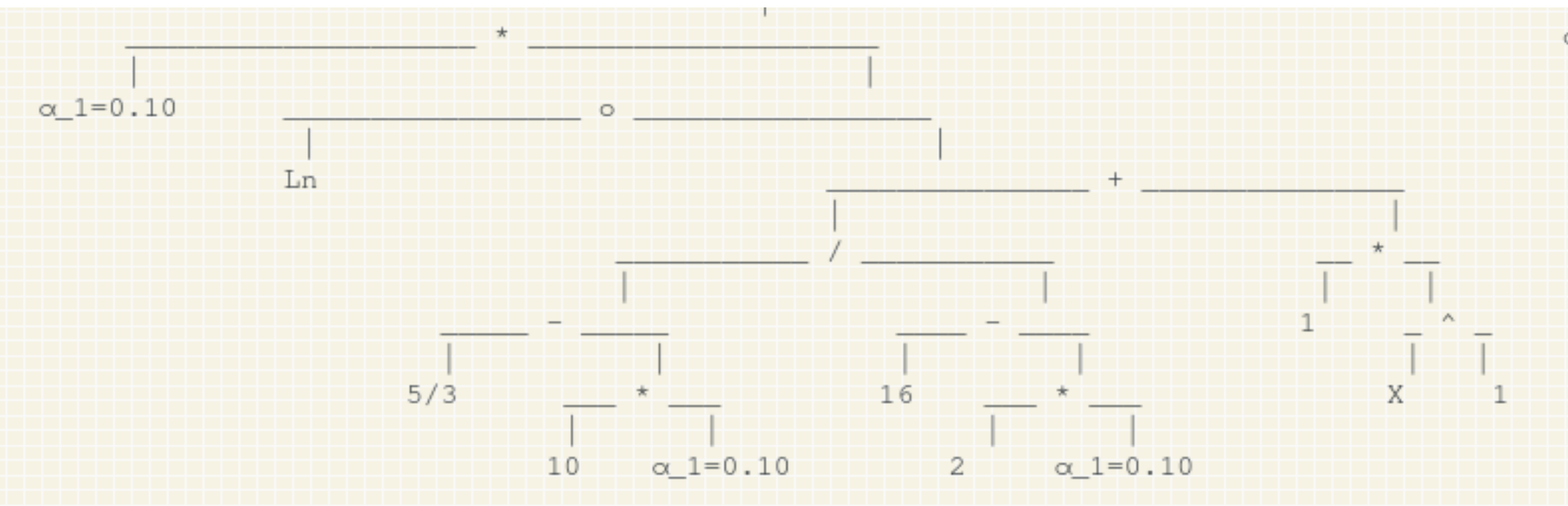
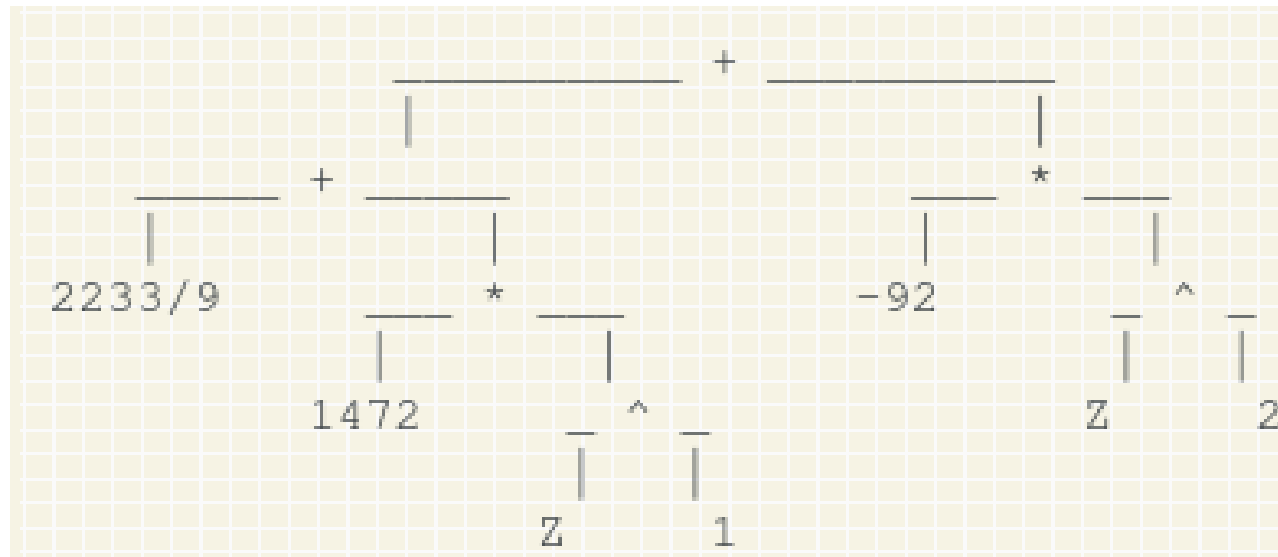
Résultats

Et un échec est un Int_overflow

Une réussite non vérifiable est :

$$\begin{array}{r} 5 \\ - + (16) * ((X) ^ (1)) \\ 3 \\ \hline 2 + (10) * ((X) ^ (1)) + (1) * ((X) ^ (2)) \end{array}$$

Résultats



Résultats

Algorithme de Risch

$$1/(\ln(x))$$

Pas de primitive elementaire



$$\int \frac{1}{\ln(x)} dx$$

 NATURAL LANGUAGE

 MATH INPUT

Indefinite integral

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

Résultats

Algorithme de Risch

$\ln(\ln(x))$

Pas de primitive elementaire



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



NATURAL LANGUAGE



$\int_{\Sigma}^{\pi} \partial$ MATH INPUT

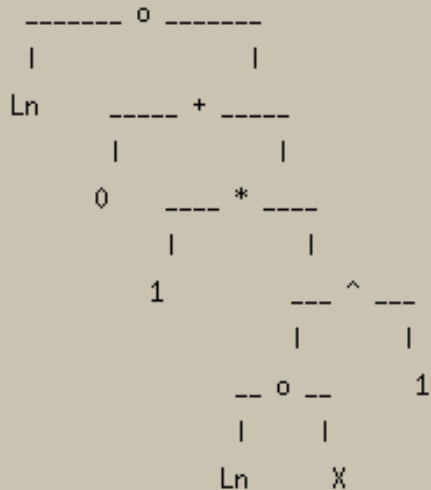
Indefinite integral

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

Résultats

Algorithme de Risch

$$1/(x*(\ln(x)))$$



$$\int \frac{1}{x \ln(x)} dx$$

NATURAL LANGUAGE

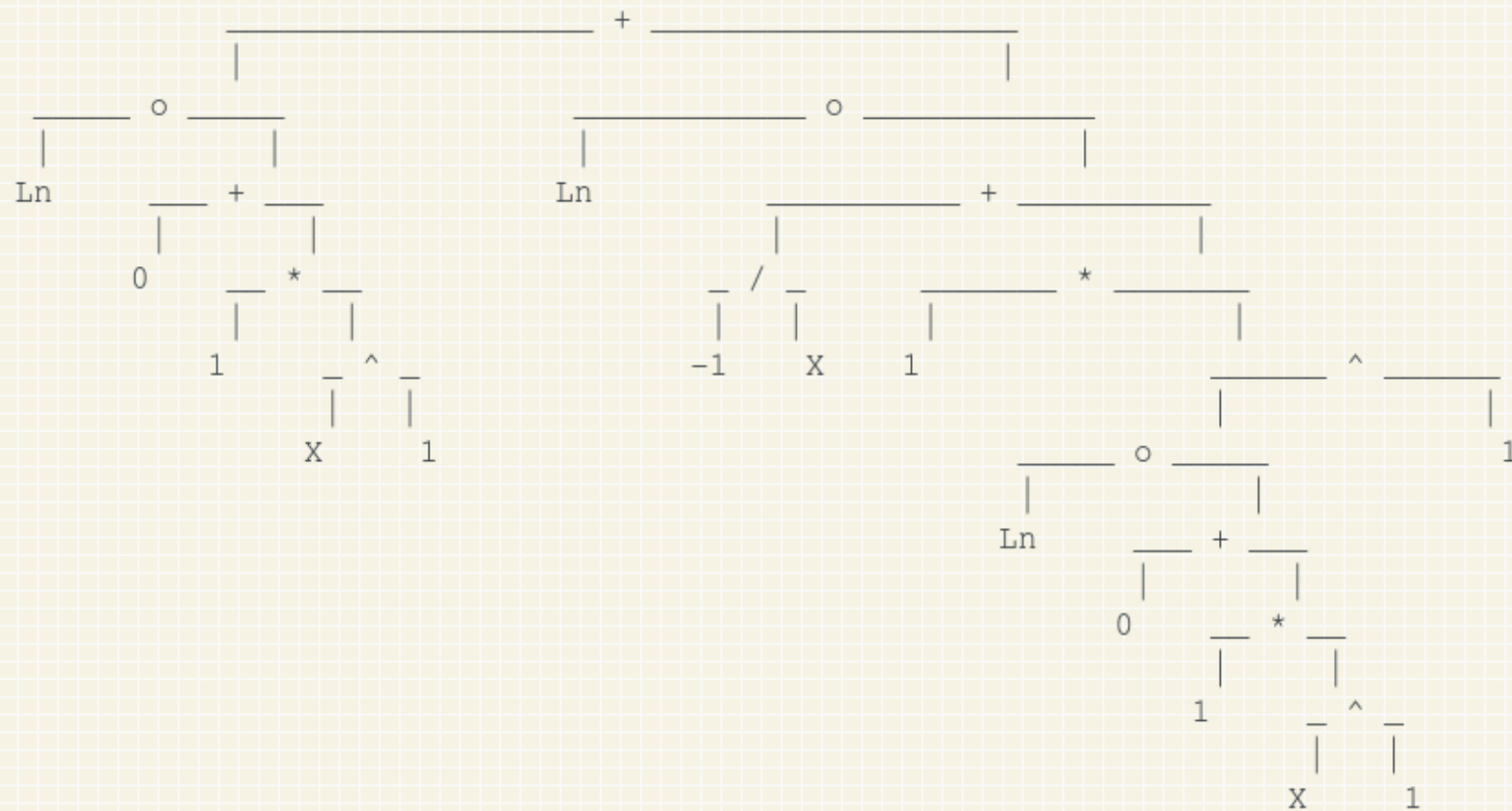
MATH INPUT

Indefinite integral

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

Résultats

$\text{Ln} \circ (0 + (1) * ((X) ^ (1))) + \text{Ln} \circ (---- + (1) * ((\text{Ln} \circ (0 + (1) * ((X) ^ (1)))) ^ (1)))$
 (X)



Résultats



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

 NATURAL LANGUAGE

 MATH INPUT

Indefinite integral

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

(assuming a complex-valued logarithm)

Résultats

$$1 + (1) * ((\text{Ln} \circ (0 + (1) * ((X)^{(1)})))^{(1)})$$

$$1 + ((2) * (X)) * ((\text{Ln} \circ (0 + (1) * ((X)^{(1)})))^{(1)}) + ((X)^{(2)}) * ((\text{Ln} \circ (0 + (1) * ((X)^{(1)})))^{(2)})$$

$$\begin{aligned} & \left(\frac{((2) * (X))}{((X)^{(2)})} - \left(\frac{((2) * (X))^{(2)}}{((X)^{(4)})} \right) + \frac{1}{0 + (1) * ((X)^{(1)})} \right) \\ & - \left(\frac{((2) * (X))^{(2)}}{((X)^{(4)})} \right) + (1) * ((\text{Ln} \circ (0 + (1) * ((X)^{(1)})))^{(1)}) \end{aligned}$$

$$\begin{aligned} & \left(\frac{((2) * (X))}{((X)^{(2)})} - \left(\frac{((2) * (X))^{(2)}}{((X)^{(4)})} \right) + (1) * ((\text{Ln} \circ (0 + (1) * ((X)^{(1)})))^{(1)}) \right) \\ & - \left(\frac{((2) * (X))^{(2)}}{((X)^{(4)})} \right) + (1) * ((\text{Ln} \circ (0 + (1) * ((X)^{(1)})))^{(1)}) \end{aligned}$$

Résultats



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

 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

```

1  (* ----- Entête ----- *)
2
3  (* #require bigdecimal;; *)
4  #use "topfind";;
5  #require "graphics";;
6
7  (* seuil auquel on considère qu'un nombre est nul*)
8  let seuil_zero = 1e-10 ;;
9
10 (* seuil max *)
11 let seuil_max_int = max_int/100;;
12
13 exception Int_overflow;;
14
15
16 (* pour des statistiques simplification constante 0 - simplification ast 1 - int max 2 *)
17 let appel_tab = [|0;0;0|];;
18 Random.self_init ();;
19
20
21
22
23
24 (* ----- Déclaration des types ----- *)
25
26
27
28 (* type constante exacte ancien type constante complexe = { re : float; im : float};; *)
29 (* type rationnel et extensions algebriques *)
30 type constante = Q of {a : int; b : int;} | E of {nom : string; approx : float};;
31
32
33 (* type des fonctions élémentaires de base *)
34 type f_elem = Exp | Ln | X | Z | C of constante;;
35
36 (* type contenant les opérations élémentaires *)
37 type op_elem = Plus | Moins | Fois | Divise | Puissance | Compose;;
38
39 (* type contenant la structure des fonctions : a symbolic tree *)
40 type ast_elem =
41   | Arg0 of f_elem
42   | Arg2 of (ast_elem * op_elem * ast_elem)
43   | Abstrait of fonction_abstraite
44   | P of (ast_elem * (ast_elem array))

```

```

45 | F of (ast_elem * (ast_elem array) * (ast_elem array))
46
47 (* type fonction abstraite nom; fonction; deriv n ieme; n le nombre de dérivation depuis fonction *)
48 and fonction_abstraite = {nom: string; fonction: ast_elem; d_fonction: ast_elem; etat_derive: int};;
49
50
51 (* implémentation pratique apt = a praticopratique tree *)
52 type apt = Fonction of (constante -> constante) | Node of (apt * ((constante -> constante) -> (constante -> constante) ->
constante -> constante) * apt);;
53
54
55 (* type extension de corps différentiel, Exp Ln X ou algebric *)
56 type corps_diff = Xe | Ext of (f_elem * fonction_abstraite * corps_diff);;
57
58
59 (* type pour echec renvoie vide ou branche non implementer *)
60 type option = Res of ast_elem | Null | Notimplementederror;;
61
62
63 let file = open_out "output.txt";;
64
65 (*voir verbatim pour fichier *)
66
67
68
69
70
71 (* ----- Déclaration des Constantes et Ast fondamentaux ----- *)
72
73
74
75 let c_zero = Q {a = 0; b = 1};;
76 let c_un = Q {a = 1; b = 1};;
77 let c_minus_un = Q {a = -1; b = 1};;
78 let c_const nom den = Q {a = nom; b = den};;
79 let c_ext ext a = E {nom = ext; approx = a};;
80
81
82 let ast_null = Arg0 (C (Q {a = 0; b = 1}));;
83 let ast_un = Arg0 (C (Q {a = 1; b = 1}));;
84 let ast_minus_un = Arg0 (C (Q {a = -1; b = 1}));;
85
86
87 let ast_const a b = assert (b <> 0); Arg0 (C (c_const a b));;

```

```
88
89
90 let ast_x = Arg0 X;;
91 let ast_Z = Arg0 Z;;
92 let ast_ln = Arg0 Ln;;
93 let ast_exp = Arg0 Exp;;
94
95
96 let ast_plus ast_1 ast_2 = Arg2 (ast_1, Plus, ast_2);;
97 let ast_moins ast_1 ast_2 = Arg2 (ast_1, Moins, ast_2);;
98 let ast_fois ast_1 ast_2 = Arg2 (ast_1, Fois, ast_2);;
99 let ast_divise ast_1 ast_2 = Arg2 (ast_1, Divise, ast_2);;
100 let ast_puissance ast_1 ast_2 = Arg2 (ast_1, Puissance, ast_2);;
101 let ast_compose ast_1 ast_2 = Arg2 (ast_1, Compose, ast_2);;
102
103
104
105
106
107 (* ----- Déclaration des fonctions ----- *)
108
109
110
111 (* rajouter les nombres négatifs *)
112 type token = Pl | Mo | Pu | Di | Fo | Num of int | Log | Expo | PaG | PaD | XX ;;
113
114
115 let tok_of_string s = match s with
116 | "+" -> [Pl]
117 | "-" -> [Mo]
118 | "/" -> [Di]
119 | "^" -> [Pu]
120 | "*" -> [Fo]
121 | "(" -> [PaG]
122 | ")" -> [PaD]
123 | "ln" -> [Log]
124 | "exp" -> [Expo]
125 | "x" -> PaG::XX::[PaD]
126 | _ -> ( match int_of_string_opt s with
127 | Some i -> PaG::(Num i)::[PaD]
128 | None -> failwith "tok_of_string : caractère non reconnu"
129 )
130 ;;
131
```

```

132
133 let analyse_lexicale (texte:string) =
134   let re = Str.regexp "\\+\\|-\\/\\/\\^\\|\\*\\|\\(\\|)\\|\\n\\|exp\\|x\\|[0-9]+" in (* regexp *)
135   let rec tokenize texte i = (* renvoie la liste de tokens pour texte, pris à partir de l'indice i *)
136     if i = String.length texte then
137       []
138     else
139       try
140         let j = Str.search_forward re texte i in
141         let str = Str.matched_string texte in
142         if str = " " || str = "\n" then
143           tokenize texte (j + String.length str)
144         else
145           (tok_of_string str) @ (tokenize texte (j + String.length str))
146       with
147         Not_found -> []
148   in
149   tokenize texte 0
150 ;;
151
152 let rec extract_paranthese lst i =
153   match lst with
154   | x::llst -> (
155     match x with
156     | PaD -> if i-1 = 0 then [],llst else (let lint,lext = extract_paranthese llst (i-1) in x::lint,lext)
157     | PaG -> (let lint,lext = extract_paranthese llst (i+1) in x::lint,lext)
158     | _ -> (let lint,lext = extract_paranthese llst i in x::lint,lext)
159   )
160   | [] -> failwith "extract_paranthese : mal parenthéser"
161 ;;
162
163
164 let rec parse lst = match lst with
165   | PaG::llst -> (
166     let lint,lext = extract_paranthese llst 1 in
167     match lext with
168     | [] -> parse lint
169     | Pl::ls -> Arg2 (parse lint, Plus, parse ls)
170     | Pu::ls -> Arg2 (parse lint, Puissance, parse ls)
171     | Mo::ls -> Arg2 (parse lint, Moins, parse ls)
172     | Fo::ls -> Arg2 (parse lint, Fois, parse ls)
173     | Di::ls -> Arg2 (parse lint, Divise, parse ls)
174     | _ -> failwith "parse : mauvais syntaxe"
175   )

```



```

176 | [Num i] -> ast_const i 1
177
178 | [XX] -> Arg0 X
179
180 | Log::llst -> Arg2 (Arg0 Ln, Compose, parse llst)
181
182 | Expo::llst -> Arg2 (Arg0 Exp, Compose, parse llst)
183
184 | _ -> failwith "parse : fail"
185 ;;
186
187
188
189
190
191 (* ----- Fonctions utilitaires pour l'affichage ----- *)
192
193
194
195 let print_debug n = Printf.fprintf file " endroit a regarder indice %d \n" n
196 ;;
197
198
199 let get_constante_string (c:constante) = match c with
200 | Q rationnel -> (Printf.sprintf "%d" rationnel.a) ^ (if rationnel.b = 1 then "" else "/" ^ (Printf.sprintf "%d"
rationnel.b))
201 | E extension -> (Printf.sprintf "%s" extension.nom) ^ "=" ^ (Printf.sprintf "%.2f" extension.approx)
202 ;;
203
204
205 let rec repete n c = if n <= 0 then "" else c ^ (repete (n-1) c)
206 ;;
207
208
209 (* cree un bloc correspondant à une valeur ainsi que les deux blocs des sous-arbres *)
210 let fusionne v (l1, b1) (l2, b2) =
211   let nb_lignes = max (Array.length b1) (Array.length b2) in
212   let block = Array.make (nb_lignes + 2) "" in
213   let entete = repete ((l1+1) / 2) " " ^ repete ((l1+l2+2) / 4) "_" ^ v ^ repete ((l1+l2+2) / 4) "_" ^ repete ((l2+1) / 2) " " in
214   let largeur = String.length entete in
215   block.(0) <- entete;
216   block.(1) <- repete ((l1+1) / 2) " " ^ "|" ^ repete (largeur - (l1+1) / 2 - (l2+1) / 2 - 2) " " ^ "|" ^ repete ((l2+1) / 2) " ";

```

```

217   assert (String.length block.(0) = String.length block.(1));
218   for i = 0 to nb_lignes - 1 do
219     if (i < Array.length b1 && i < Array.length b2) then
220       block.(i+2) <- b1.(i) ^ repete (largeur - l1 - l2) " " ^ b2.(i)
221     else if i < Array.length b1 then
222       block.(i+2) <- b1.(i) ^ repete (largeur - l1) " "
223     else if i < Array.length b2 then
224       block.(i+2) <- repete (largeur - l2) " " ^ b2.(i)
225   done;
226   (largeur, block)
227 ;;
228
229
230 let rec block_of_ast_arbre ast_arbre =
231
232   let ast_of_poly_array_t poly indet =
233     (* transforme un ast P en ça version ast *)
234     let f = ref poly.(0) in
235     for i = 1 to Array.length poly - 1 do
236       f := Arg2 (!f, Plus, Arg2(poly.(i), Fois, Arg2 (indet, Puissance, ast_const i 1)))
237     done;
238     !f
239   in
240
241   match ast_arbre with
242   | Arg0 x ->
243     (match x with
244     | C x ->
245       (match x with
246       | Q rationnel -> let s = get_constante_string x in (String.length s, [|s|])
247       | E extension -> let s = (Printf.sprintf "%s" extension.nom) ^ "=" ^ (Printf.sprintf "%.2f" extension.approx) ^
248         in (String.length s, [|s|])
249       )
250     | X -> (3, [|" X "|])
251     | Z -> (3, [|" Z "|])
252     | Ln -> (3, [|" Ln"|])
253     | Exp -> (3, [|"Exp"|])
254     )
255   | Arg2 (f1, oper, f2) ->
256     let str = ( match oper with
257     | Plus -> " + "
258     | Moins -> " - "
259     | Fois -> " * "

```

```

260 | Divise -> " / "
261 | Puissance -> " ^ "
262 | Compose -> " o "
263 ) in fusionne str (block_of_ast_arbre f1) (block_of_ast_arbre f2)
264
265 | Abstrait f1 -> (String.length f1.nom, [|f1.nom|])
266
267 | P (x,p) -> block_of_ast_arbre (ast_of_poly_array_t p x)
268
269 | 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))
270 ;;
271
272
273 let print_ast_arbre ast_arbre =
274   let (largeur, block) = block_of_ast_arbre ast_arbre in
275   for i = 0 to Array.length block - 1 do
276     (* print_endline block.(i) *)
277     Printf.fprintf file "%s\n" block.(i)
278   done
279 ;;
280
281
282 let print_ast_arbre_graphics ast_arbre : unit =
283   let (largeur, block) = block_of_ast_arbre ast_arbre in
284   let x = Graphics.current_x () in
285   for i = 0 to Array.length block - 1 do
286     Graphics.moveto x (Graphics.current_y () - 20);
287     Graphics.draw_string (Printf.sprintf "%s" block.(i));
288   done;
289   ()
290 ;;
291
292
293 let rec block_of_ast_arbre_lineaire ast_arbre =
294
295   let ast_of_poly_array_t poly indet =
296     (* transforme un ast P en ça version ast *)
297     let f = ref poly.(0) in
298     for i = 1 to Array.length poly - 1 do
299       f := Arg2 (!f, Plus, Arg2(poly.(i), Fois, Arg2 (indet, Puissance, ast_const i 1)))
300     done;
301     !f
302   in

```

```

303
304 match ast_arbre with
305 | Arg0 x ->
306   (match x with
307   | C x ->
308     (match x with
309     | Q rationnel -> get_constante_string x
310     | E extension -> extension.nom ^ "=" ^ (string_of_float extension.approx)
311     )
312   | X -> "X"
313   | Z -> "Z"
314   | Ln -> "Ln"
315   | Exp -> "Exp"
316   )
317
318 | Arg2 (f1, oper, f2) ->
319   let str = ( match oper with
320   | Plus -> " + "
321   | Moins -> " - "
322   | Fois -> " * "
323   | Divise -> " / "
324   | Puissance -> " ^ "
325   | Compose -> " o "
326   ) in "(" ^ (block_of_ast_arbre_lineaire f1) ^ ")" ^ str ^ "(" ^ (block_of_ast_arbre_lineaire f2) ^ ")"
327
328 | Abstrait f1 -> f1.nom
329
330 | P (x,p) -> block_of_ast_arbre_lineaire (ast_of_poly_array_t p x)
331
332 | F (x,a,b) -> (
333   let s1 = (block_of_ast_arbre_lineaire (ast_of_poly_array_t a x)) ^ "\n" in
334   let s2 = "\n" ^ (block_of_ast_arbre_lineaire (ast_of_poly_array_t b x)) in
335   s1 ^ (String.make (max (String.length s1) (String.length s2)) '_') ^ s2
336 )
337
338 ;;
339
340
341 let print_ast_arbre_lineaire ast_arbre = Printf.fprintf file "%s\n" (block_of_ast_arbre_lineaire ast_arbre)
342 ;;
343
344 let print_ast ast =
345
346

```

```

347   let ast_of_poly_array_t poly indet =
348     (* transforme un ast P en ça version ast *)
349     let f = ref poly.(0) in
350     for i = 1 to Array.length poly - 1 do
351       f := Arg2 (!f, Plus, Arg2(poly.(i), Fois, Arg2 (indet, Puissance, ast_const i 1)))
352     done;
353     !f
354   in
355
356   let rec explore_haut f =
357     match f with
358     | Arg0 x ->
359       (match x with
360        | C x -> (
361            match x with
362            | Q rationnel -> (if rationnel.b = 1 then (0,1) else (-1,2))
363            | E extension -> (0,1)
364          )
365        | _ -> (0,1)
366       )
367     | Arg2 (f1, oper, f2) ->
368       let (l1,h1) = explore_haut f1 in
369       let (l2,h2) = explore_haut f2 in
370       (
371         match oper with
372         | Divise -> (l2-h2,h1-l1+1)
373         | _ -> (min l1 l2, max h1 h2)
374       )
375     | Abstrait f1 -> (0,1)
376     | P (x,p) -> explore_haut (ast_of_poly_array_t p x)
377     | F (x,a,b) -> (
378       explore_haut (Arg2 ((ast_of_poly_array_t a x),Divise,(ast_of_poly_array_t b x)))
379     )
380   in
381
382   let (l,h) = explore_haut ast in
383
384   let array_print = Array.make (h-l) " " in
385
386   let rise_up_to n i =

```

```

391   let m = String.length array_print.(i) in
392   if m < n then array_print.(i) <- array_print.(i) ^ (String.make (n-m-1) ' ')
393 in
394
395 let rec aux_ast_arbre f haut =
396
397   let ind = haut - 1 in (* indice correspondant dans le tableau *)
398
399   match f with
400   | Arg0 x ->
401     (match x with
402     | C x -> (
403       match x with
404       | Q rationnel -> (
405         if rationnel.b = 1
406         then array_print.(ind) <- array_print.(ind) ^ (Printf.sprintf "%d" rationnel.a)
407         else
408           (
409             rise_up_to (String.length array_print.(ind)) (ind+1);
410             rise_up_to (String.length array_print.(ind)) (ind-1);
411             let n = (max (String.length (Printf.sprintf "%d" rationnel.a)) (String.length (Printf.sprintf "%d"
412               array_print.(ind+1) <- array_print.(ind+1) ^ (Printf.sprintf "%d" rationnel.a) ^ (String.make (n - (
413               String.length (Printf.sprintf "%d" rationnel.a))) ' ');
414               array_print.(ind) <- array_print.(ind) ^ (String.make n '-');
415               array_print.(ind-1) <- array_print.(ind-1) ^ (Printf.sprintf "%d" rationnel.b) ^ (String.make (n - (
416               String.length (Printf.sprintf "%d" rationnel.b))) ' ');
417             )
418           )
419       | E extension -> array_print.(ind) <- array_print.(ind) ^ (Printf.sprintf "%s" extension.nom) ^ "=" ^ (
420         Printf.sprintf "%.2f" extension.approx);
421       | X -> array_print.(ind) <- array_print.(ind) ^ "X";
422       | Z -> array_print.(ind) <- array_print.(ind) ^ "Z";
423       | Ln -> array_print.(ind) <- array_print.(ind) ^ "Ln";
424       | Exp -> array_print.(ind) <- array_print.(ind) ^ "Exp";
425     )
426
427   | Arg2 (f1, oper, f2) -> (
428     match oper with
429     | Plus -> (
430       aux_ast_arbre f1 haut;

```

```

431         array_print.(ind) <- array_print.(ind) ^ " + ";
432         aux_ast_arbre f2 haut;
433     )
434
435 | Divise -> (
436     let d1,m1 = explore_haut f1 in
437     let d2,m2 = explore_haut f2 in
438
439     let n = String.length array_print.(ind) in
440
441     for i = ind-m2 to ind-d1+1 do
442         if i <> ind then rise_up_to (n + 1) i;
443     done;
444     aux_ast_arbre f1 (haut-d1+1);
445     aux_ast_arbre f2 (haut-m2);
446
447     array_print.(ind) <- array_print.(ind) ^ (String.make (max (String.length array_print.(ind-d1+1)) (String.
448         length array_print.(ind-m2)) - n + 2) '-');
449 )
450
451 | Fois | Puissance -> (
452     let d1,m1 = explore_haut f1 in
453     let d2,m2 = explore_haut f2 in
454     let d,m = (min d1 d2,max m1 m2) in
455
456     let n = ref ((String.length array_print.(ind)) + 1) in
457     for i = ind+d to ind+m-1 do
458         rise_up_to !n i;
459         array_print.(i) <- array_print.(i) ^ "("
460     done;
461
462     aux_ast_arbre f1 haut;
463
464     n := (String.length array_print.(ind));
465     for i = ind+d to ind+m-1 do
466         rise_up_to !n i;
467         array_print.(i) <- array_print.(i) ^ ")"
468     done;
469
470     array_print.(ind) <- array_print.(ind) ^ (
471         match oper with
472         | Fois -> " * "
473         | Puissance -> " ^ "
474         | _ -> failwith "print_ast : n'arrive pas"

```

```

474         );
475
476         for i = ind+d to ind+m-1 do
477             rise_up_to (!n+4) i;
478             array_print.(i) <- array_print.(i) ^ "("
479         done;
480
481         aux_ast_arbre f2 haut;
482
483         n := (String.length array_print.(ind));
484         for i = ind+d to ind+m-1 do
485             rise_up_to !n i;
486             array_print.(i) <- array_print.(i) ^ ")"
487         done;
488     )
489
490 | Moins | Compose -> (
491     let d,m = explore_haut f2 in
492     aux_ast_arbre f1 haut;
493     array_print.(ind) <- array_print.(ind) ^ (
494         match oper with
495         | Moins -> " - "
496         | Compose -> " o "
497         | _ -> failwith "print_ast : n'arrive pas"
498     );
499     let n = String.length array_print.(ind) in
500     for i = ind+d to ind+m-1 do
501         rise_up_to n i;
502         array_print.(i) <- array_print.(i) ^ "("
503     done;
504     aux_ast_arbre f2 haut;
505     let n = String.length array_print.(ind)+1 in
506     for i = ind+d to ind+m-1 do
507         rise_up_to n i;
508         array_print.(i) <- array_print.(i) ^ ")"
509     done;
510 )
511 )
512
513 | Abstrait f1 -> array_print.(ind) <- array_print.(ind) ^ f1.nom;
514
515 | P (x,p) -> aux_ast_arbre (ast_of_poly_array_t p x) haut;
516
517 | F (x,a,b) -> aux_ast_arbre (Arg2 ((ast_of_poly_array_t a x),Divise,(ast_of_poly_array_t b x))) haut;

```



```

518   in
519
520   aux_ast_arbre ast 0;
521   Printf.fprintf file "\n";
522   for i = h-1-1 downto 0 do
523     Printf.fprintf file "%s\n" array_print.(i)
524   done;
525   Printf.fprintf file "\n";
526 ;;
527
528
529
530
531
532 (* ----- Dérivé ----- *)
533
534
535
536 let rec derive ast_arbre =
537   (* Dérivé *)
538   match ast_arbre with
539
540   | Arg0 f -> (match f with
541     | C c -> ast_null
542     | X -> ast_un
543     | Z -> ast_un
544     | Exp -> Arg0 Exp
545     | Ln -> Arg2 (ast_un, Divise, Arg0 X)
546   )
547
548   | Arg2 (f1, oper, f2) -> (match oper with
549     | Compose -> Arg2 (derive f2, Fois, Arg2 (derive f1, Compose, f2))
550     | Puissance -> Arg2 (Arg2 (Arg2 (Arg2 (Arg0 Ln, Compose, f1), Fois, Arg2 (f1, Puissance, f2)), Fois, derive f2),
551       Plus
552       , Arg2 (f2 , Fois, Arg2(derive f1 , Fois, Arg2(f1, Puissance, Arg2 (f2 , Moins, ast_un))))))
553     | Plus -> Arg2 (derive f1, Plus, derive f2)
554     | Moins -> Arg2 (derive f1, Moins, derive f2)
555     | Fois -> Arg2 (Arg2 (derive f1, Fois, f2), Plus, Arg2 (f1, Fois, derive f2))
556     | Divise -> Arg2 (Arg2 (Arg2 (derive f1, Fois, f2), Moins, Arg2 (f1, Fois, derive f2)), Divise, Arg2 (f2, Fois, f2))
557   )
558   )
559
560   | Abstrait f -> Abstrait {nom = f.nom; fonction = f.fonction ; d_fonction = derive f.d_fonction; etat_derive = f.
561     etat_derive + 1}

```

```

559 | P (x,p) -> let p2 = Array.make (Array.length p) ast_null in
560 |
561 | (
562 |   for i = 0 to Array.length p - 2 do
563 |     p2.(i) <- Arg2 (derive p.(i), Plus, ast_fois (ast_fois p.(i+1) (ast_const (i+1) 1)) (derive x));
564 |   done;
565 |   if Array.length p - 1 >= 0
566 |   then p2.(Array.length p - 1) <- derive p.(Array.length p - 1)
567 | );
568 | P (x, p2)
569 |
570 | F (x,a,b) ->
571 | (
572 |   let a1 = minus_poly (mult_poly (derive (P (x,a))) (P (x,b))) (mult_poly (derive (P (x,b))) (P (x,a))) in
573 |   let b1 = mult_poly (P (x,b)) (P (x,b)) in
574 |   F (x,poly_array a1,poly_array b1)
575 | )
576 |
577 |
578 | and derive_n ast n =
579 |   let f = ref ast in
580 |   for i = 1 to n do
581 |     f := derive !f;
582 |   done;
583 |   !f
584 |
585 |
586 | and ast_abs str f n =
587 |   (* Crée un ast abstrait qui nécessite la dérivé de f donc la fonction dérive avant*)
588 |   let fd = ref f in
589 |   for i = 0 to n-1 do
590 |     fd := derive !fd
591 |   done;
592 |   {nom = str; fonction = f ; d_fonction = !fd; etat_derive = n}
593 |
594 |
595 |
596 |
597 |
598 | (* ----- Fonctions d'operations sur le type constante ----- *)
599 |
600 |
601 |
602 | and c_q_to_e (c:constante) = match c with

```

```

603 | Q q -> (float_of_int q.a) /. (float_of_int q.b)
604 | E e -> e.approx
605
606
607 and add_constante (c1:constante) (c2:constante) = match (c1,c2) with
608   (* addition *)
609   | Q q1,Q q2 -> (
610     if (abs q1.a > seuil_max_int || abs q1.b > seuil_max_int) then raise Int_overflow;
611     if (abs q2.a > seuil_max_int || abs q2.b > seuil_max_int) then raise Int_overflow;
612     simplifie_constante (Q {a = q1.a * q2.b + q2.a * q1.b; b = q1.b * q2.b})
613   )
614
615   | E e1,E e2 -> E {nom = e1.nom ^ "+" ^ e2.nom; approx = e1.approx +. e2.approx}
616   | 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) }
617
618
619 and neg_constante (c1:constante) = match c1 with
620   (* negation *)
621   | 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}
622   | E e -> E {nom = "-" ^ e.nom; approx = -.e.approx}
623
624
625 and minus_constante (c1:constante) (c2:constante) = (* c1 - c2 *)
626   (* soustraction *)
627   simplifie_constante (add_constante c1 (neg_constante c2))
628
629
630 and mult_constante (c1:constante) (c2:constante) = match (c1,c2) with
631   (* multiplication *)
632   | Q q1,Q q2 -> (
633     if (abs q1.a > seuil_max_int || abs q1.b > seuil_max_int) then raise Int_overflow;
634     if (abs q2.a > seuil_max_int || abs q2.b > seuil_max_int) then raise Int_overflow;
635     simplifie_constante (Q {a = q1.a * q2.a; b = q1.b * q2.b})
636   )
637
638   | E e1,E e2 -> E {nom = e1.nom ^ ")*(" ^ e2.nom; approx = e1.approx *. e2.approx}
639   | Q q,E e | E e,Q q -> (
640     if (abs q.a > seuil_max_int || abs q.b > seuil_max_int) then raise Int_overflow;
641     E {nom = e.nom ^ ")*(" ^ (string_of_int q.a) ^ "/" ^ (string_of_int q.b); approx = e.approx *. c_q_to_e (Q q) }
642   )
643
644
645 and inv_constante (c1:constante) = match c1 with

```

```

646 (* fonction inverse *)
647 | 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. a
b; b = q.a}
648 | E e -> assert (e.approx <> 0.); E {nom = "1/(" ^ e.nom; approx = 1./e.approx}
649
650
651 and div_constante (c1:constante) (c2:constante) = (* c1 / c2 *)
652 (* division *)
653 simplifie_constante (mult_constante c1 (inv_constante c2))
654
655
656 and pow_constante_ent (c1:constante) (n:int) = match c1 with
657 (* puissance entiere *)
658 | Q q ->
659 (
660   if (abs q.a > seuil_max_int || abs q.b > seuil_max_int) then raise Int_overflow;
661   let c,d = ref 1,ref 1 in
662   for i = 1 to n do c := !c * q.a; d := !d * q.b done;
663   if n >= 0 then Q {a = !c; b = !d}
664   else Q {a = !d; b = !c}
665 )
666 | E e -> E {nom = e.nom ^ "^" ^ (string_of_int n); approx = e.approx ** (float_of_int n)}
667
668
669 and exp_cste (c1:constante) = match c1 with
670 (* exponentielle *)
671 | Q q -> E {nom = "exp("^(string_of_int q.a) ^ "/" ^ (string_of_int q.b)^")"; approx = exp (c_q_to_e (Q q))}
672 | E e -> E {nom = "exp("e.nom^")"; approx = exp e.approx}
673
674
675 and log_cste (c1:constante) = match c1 with
676 (* logarithme *)
677 | Q q -> E {nom = "ln("^(string_of_int q.a) ^ "/" ^ (string_of_int q.b)^")"; approx = log (c_q_to_e (Q q))}
678 | E e -> E {nom = "ln("e.nom^")"; approx = log e.approx}
679
680
681 and pow_constante (c1:constante) (c2:constante) = match (c1,c2) with
682 (* puissance *)
683 | Q q1,Q q2 ->
684 (
685   if (abs q1.a > seuil_max_int || abs q1.b > seuil_max_int) then raise Int_overflow;
686   if (abs q2.a > seuil_max_int || abs q2.b > seuil_max_int) then raise Int_overflow;
687   let c,d = match (simplifie_constante c2) with | Q q -> q.a,q.b | _ -> failwith "" in
688   if d = 1 then (

```

```

689         pow_constante_ent c1 c
690     ) else (
691         E {nom = "p ration"; approx = c_q_to_e (Q q1) ** c_q_to_e (Q q2)}
692     )
693 )
694 | E e1,E e2 -> E {nom = e1.nom ^ "(" ^ e2.nom; approx = e1.approx ** e2.approx}
695 | 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}
696 | 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) }

697
698
699 and abs_constante (c1:constante) = match c1 with
700 | 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}
701 | E e -> E {nom = "|" ^ e.nom ^ "|"; approx = abs_float e.approx}
702
703
704 and pgcd_entier a b =
705     let c = ref (abs a) in
706     let d = ref (abs b) in
707     while !d <> 0 do
708         let r = !c mod !d in
709         c := !d;
710         d := r;
711     done;
712     !c
713
714
715 and signum a = if a = 0 then 0 else if a > 0 then 1 else -1
716
717
718 and simplifie_constante (c1:constante) = match c1 with
719 | Q q ->
720     (
721         if (abs q.a > seuil_max_int || abs q.b > seuil_max_int) then raise Int_overflow;
722         appel_tab.(0) <- appel_tab.(0) + 1;
723         if max q.a q.b > appel_tab.(2) then appel_tab.(2) <- max q.a q.b;
724         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
725     )
726 | E e -> E {nom = "s(" ^ (String.sub e.nom 0 3) ^ ")"; approx = e.approx}
727
728
729

```

```

730
731
732 (* ----- Fonctions de calcul de fonction ----- *)
733
734
735
736 and ast_of_poly_array poly indet =
737   (* transforme un ast P en ça version ast *)
738   let f = ref poly.(0) in
739   for i = 1 to Array.length poly - 1 do
740     f := Arg2 (!f, Plus, Arg2(poly.(i), Fois, Arg2 (indet, Puissance, ast_const i 1)))
741   done;
742   !f
743
744
745 and ast_of_frac_array a b indet =
746   (* transforme un ast F en ça version ast *)
747   let ast_a = ast_of_poly_array a indet in
748   let ast_b = ast_of_poly_array b indet in
749   Arg2 (ast_a, Divise, ast_b)
750
751
752 and replace_Abstrait f =
753   (* remplace les parties abstraites *)
754   match f with
755
756   | Arg0 felem -> f
757
758   | Arg2 (f1, oper, f2) -> Arg2 (replace_Abstrait f1, oper, replace_Abstrait f2)
759
760   | Abstrait f -> f.d_fonction
761
762   | P (x,p) ->
763     (
764       let p1 = Array.make (Array.length p) p.(0) in
765       for i = 0 to Array.length p - 1 do
766         p1.(i) <- replace_Abstrait p.(i)
767       done;
768       P (replace_Abstrait x, p1)
769     )
770
771   | F (x,a,b) ->
772     (
773       let pa = Array.make (Array.length a) a.(0) in

```

```

774   let pb = Array.make (Array.length b) b.(0) in
775   for i = 0 to Array.length a - 1 do
776     pa.(i) <- replace_Abstrait a.(i)
777   done;
778   for i = 0 to Array.length b - 1 do
779     pb.(i) <- replace_Abstrait b.(i)
780   done;
781   F (replace_Abstrait x, pa, pb)
782 )
783
784
785 and ast_arg_of_ast_arbre f =
786   (* simplifie f en arg0 et arg2 pour la transformation en apt *)
787   let fa = replace_Abstrait f in
788
789   match fa with
790
791   | Arg0 felem -> fa
792
793   | Arg2 (f1, oper, f2) -> Arg2 (ast_arg_of_ast_arbre f1, oper, ast_arg_of_ast_arbre f2)
794
795   | Abstrait f1 -> ast_arg_of_ast_arbre (derive_n f1.fonction f1.etat_derive)
796
797   | P (x,p) ->
798     (
799       let p1 = Array.make (Array.length p) p.(0) in
800       for i = 0 to Array.length p - 1 do
801         p1.(i) <- ast_arg_of_ast_arbre p.(i)
802       done;
803       ast_of_poly_array p1 (ast_arg_of_ast_arbre x)
804     )
805
806   | F (x,a,b) ->
807     (
808       let pa = Array.make (Array.length a) a.(0) in
809       let pb = Array.make (Array.length b) b.(0) in
810       for i = 0 to Array.length a - 1 do
811         pa.(i) <- ast_arg_of_ast_arbre a.(i)
812       done;
813       for i = 0 to Array.length b - 1 do
814         pb.(i) <- ast_arg_of_ast_arbre b.(i)
815       done;
816       ast_of_frac_array a b (ast_arg_of_ast_arbre x)
817     )

```

```

818
819
820 and apt_of_ast ast_arbre =
821   let ast = ast_arg_of_ast_arbre ast_arbre in
822   match ast with
823   | Arg0 f_elem ->
824     ( match f_elem with
825     | X -> Fonction (fun x -> x)
826     | Z -> Fonction (fun x -> x)
827     | C c -> Fonction (fun x -> c)
828     | Exp -> Fonction (fun x -> exp_cste x)
829     | Ln -> Fonction (fun x -> log_cste x)
830     )
831
832   | Arg2 (f1, oper, f2) ->
833     Node (apt_of_ast f1 ,
834           (match oper with
835           | Plus -> (fun f g -> (fun x -> add_constant (f x) (g x)))
836           | Moins -> (fun f g -> (fun x -> minus_constant (f x) (g x)))
837           | Fois -> (fun f g -> (fun x -> mult_constant (f x) (g x)))
838           | Divise -> (fun f g -> (fun x -> div_constant (f x) (g x)))
839           | Puissance -> (fun f g -> (fun x -> pow_constant (f x) (g x)))
840           | Compose -> (fun f g -> (fun x -> f (g x))))
841           , apt_of_ast f2 )
842
843   | P (x, poly) -> apt_of_ast (ast_of_poly_array poly x)
844
845   | Abstrait f -> apt_of_ast f.d_fonction
846
847   | _ -> failwith "N'arrive pas sauf bug avant le match"
848
849
850 and build_ast_arbre =
851   (* Construit une fonction ocaml permettant d'évaluer la fonction *)
852   match apt_arbre with
853   | Fonction f -> f
854   | Node (f1, fc, f2) -> (fun x -> fc (build f1) (build f2) x )
855
856
857 and eval_ast ast_arbre z = build (apt_of_ast ast_arbre) z
858
859
860 and appartient_f_elem ast_arbre elem = match ast_arbre with
861

```



```

862 | Arg0 felem -> (
863   match felem,elem with
864   | C a, C b -> true
865   | X,X | Z,Z | Ln,Ln | Exp,Exp -> true
866   | _ -> false
867 )
868
869 | Arg2 (f1, oper, f2) -> (appartient_f_elem f1 elem) || (appartient_f_elem f2 elem)
870
871 | Abstrait f1 -> appartient_f_elem f1.d_fonction elem
872
873 | P (x,p) -> (
874   let res = ref (appartient_f_elem x elem) in
875   for i = 0 to Array.length p - 1 do
876     res := !res || (appartient_f_elem p.(i) elem)
877   done;
878   !res
879 )
880
881 | F (x,a,b) -> (
882   let res = ref (appartient_f_elem x elem) in
883   for i = 0 to Array.length a - 1 do
884     res := !res || (appartient_f_elem a.(i) elem)
885   done;
886   for i = 0 to Array.length b - 1 do
887     res := !res || (appartient_f_elem b.(i) elem)
888   done;
889   !res
890 )
891
892
893 and is_zero_ast ast_arbre =
894   (* test de manière imprécise si une expression est nul, risque d'échec massif du à l'implémentation *)
895   let fonction = build (apt_of_ast ast_arbre) in
896   let compte = ref 0 in
897   let test = ref true in
898   while !compte < int_of_float (10.**3.) && !test do
899     compte := !compte + 1;
900     let z1 = E {nom = "z1" ; approx = Random.float (707.48)} in
901     let f1 = fonction z1 in
902
903     (*
904       OCaml's floating-point numbers follow the IEEE 754 standard, using double precision (64 bits) numbers.
905       binary64 Double precision 1.80*10^308 max

```

```

906     donc le float max en évaluation pour ne pas obtenir nan avec exp et inférieur à 707.48
907 *)
908
909 let valeur = match abs_constante f1 with | Q q -> c_q_to_e (Q q) | E e -> e.approx in
910
911 if valeur > seuil_zero (*|| abs_constante f2 > seuil_zero || abs_constante f3 > seuil_zero || abs_constante f4 >
seuil_zero*)
912 then (
913     (* (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); *)
914     test := false
915 )
916 done;
917 !test && (not (appartient_f_elem ast_arbre X && appartient_f_elem ast_arbre Z))
918
919
920 and is_const_ast ast_arbre =
921     (* test de manière imprécise si une expression est nul, risque d'échec massif du à l'implémentation *)
922     let fonction = build (apt_of_ast ast_arbre) in
923     let compte = ref 0 in
924     let test = ref true in
925     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
926
927 while !compte < int_of_float (10.**3.) && !test do
928     compte := !compte + 1;
929     let z1 = E {nom = "z1" ; approx = Random.float (707.48)} in
930     let f1 = fonction z1 in
931     let valeur = match abs_constante f1 with | Q q -> c_q_to_e (Q q) | E e -> e.approx in
932
933     if abs_float (valeur -. v) > 10.*seuil_zero (*|| abs_constante f2 > seuil_zero || abs_constante f3 > seuil_zero ||
abs_constante f4 > seuil_zero*)
934     then test := false
935 done;
936 (!test, fonction (Q {a = 1; b = 1}))
937
938
939
940
941
942 (* ----- Vérifie une égalité entre deux ast ----- *)
943
944
945

```

```

946 and egal_ast1 ast_1 ast_2 = match (ast_1,ast_2) with
947   | Arg0 f_elm1, Arg0 f_elm2 -> f_elm1 = f_elm2
948   | Arg2 (f1, oper1, f2), Arg2 (f3, oper2, f4) -> (egal_ast1 f1 f3) && (oper1 = oper2) && (egal_ast1 f2 f4)
949   | Abstrait f1, Abstrait f2 -> egal_ast1 f1.d_fonction f2.d_fonction
950   | _,_ -> false
951
952
953 and egal_ast2 ast_1 ast_2 = is_zero_ast (Arg2 (ast_1, Moins, ast_2))
954
955
956 and egal_ast ast_1 ast_2 = egal_ast1 ast_1 ast_2 || egal_ast2 ast_1 ast_2
957
958
959
960
961
962 (* ----- Début de simplification / formatage ----- *)
963
964
965 and simplifie_ast ast_arbre = appel_tab.(1) <- appel_tab.(1) + 1; match ast_arbre with
966   (* tente de simplifier quelques élément d'un ast *)
967   | Arg0 f -> (match f with
968     | X -> ast_arbre
969     | Z -> ast_arbre
970     | C c -> Arg0 (C (simplifie_constante c))
971     | Exp -> ast_arbre
972     | Ln -> ast_arbre
973   );
974
975   | Arg2 (f1, oper, f2) ->
976     let g1 = (if is_zero_ast f1 then ast_null else simplifie_ast f1) in
977     let g2 = (if is_zero_ast f2 then ast_null else simplifie_ast f2) in
978     let ast_simp = Arg2(g1, oper, g2) in
979     (match ast_simp with
980       (* Simplifie la compositon *)
981       | Arg2 (Arg0 (C x), Compose, f4) -> Arg0 (C x)
982       | Arg2 (f3, Compose, Arg0 (C x)) -> Arg0 (C (build (apt_of_ast f3) x))
983       (* | Arg2 (f3, Compose, Arg0 X) -> f3 *)
984       | Arg2 (Arg0 X, Compose, f4) -> f4
985       | Arg2 (Arg2 (Arg0 (C (Q {a = 1; b = 1})), Divise, Arg0 X), Compose, f4) -> Arg2(ast_un, Divise, f4)
986       | Arg2 (P (Arg0 X, [|Arg0 (C (Q {a = 0; b = 1})); Arg0 (C (Q {a = 1; b = 1}));|]), Compose, f4) -> f4
987       (* Simplifie la puissance *)
988       | Arg2 (Arg0 (C x), Puissance, Arg0 (C y)) -> Arg0 (C (pow_constante x y))
989       | Arg2 (f3, Puissance, Arg0 (C (Q {a = 0; b = 1}))) -> ast_un

```

```

990 | Arg2 (f3, Puissance, Arg0 (C (Q {a = 1; b = 1}))) -> f3
991 | Arg2 (Arg0 (C (Q {a = 1; b = 1})), Puissance, f4) -> ast_un
992 | Arg2 (Arg2 (f3, Puissance, f4), Puissance, f5) -> Arg2 (f3, Puissance, Arg2 (f4, Fois, f5))
993 | Arg2 (Arg2 (Arg0 Exp, Compose, f4), Puissance, f5) -> Arg2 (Arg0 Exp, Compose, Arg2 (f4, Fois, f5))
994 | Arg2 (P (x,a), Puissance, Arg0 (C (Q {a = n; b = 1}))) -> puissance_poly_ent (P (x,a)) n
995 | 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)))
996 (* Simplifie l'addition *)
997 | Arg2 (Arg0 (C (Q {a = 0; b = 1})), Plus, f4) -> f4
998 | Arg2 (f3, Plus, Arg0 (C (Q {a = 0; b = 1}))) -> f3
999 | Arg2 (Arg0 (C x), Plus, Arg0 (C y)) -> Arg0 (C (add_constant x y))
1000 | Arg2 (f1, Plus, f2) when egal_ast f1 f2 -> Arg2 (Arg0 (C (Q {a = 2; b = 1})), Fois, f1)
1001 | Arg2 (P (x,a), Plus, P (y,b)) when egal_ast x y -> add_poly (P (x,a)) (P (y,b))
1002 | 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))
1003 (* Simplifie la soustraction *)
1004 | Arg2 (Arg0 (C (Q {a = 0; b = 1})), Moins, f4) -> Arg2 (ast_minus_un, Fois, f4)
1005 | Arg2 (f3, Moins, Arg0 (C (Q {a = 0; b = 1}))) -> f3
1006 | Arg2 (Arg0 (C x), Moins, Arg0 (C y)) -> Arg0 (C (minus_constant x y))
1007 | Arg2 (f3, Moins, Arg0 (C x)) -> Arg2 (f3, Plus, Arg0 (C (neg_constant x)))
1008 | Arg2 (f1, Moins, f2) when egal_ast f1 f2 -> ast_null
1009 | Arg2 (P (x,a), Moins, P (y,b)) when egal_ast x y -> minus_poly (P (x,a)) (P (y,b))
1010 | 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))
1011 (* Simplifie la multiplication *)
1012 | Arg2 (Arg0 (C (Q {a = 0; b = 1})), Fois, f4) -> ast_null
1013 | Arg2 (f3, Fois, Arg0 (C (Q {a = 0; b = 1}))) -> ast_null
1014 | Arg2 (Arg0 (C x), Fois, Arg0 (C y)) -> Arg0 (C (mult_constant x y))
1015 | Arg2 (Arg0 (C (Q {a = 1; b = 1})), Fois, f4) -> f4
1016 | Arg2 (f3, Fois, Arg0 (C (Q {a = 1; b = 1}))) -> f3
1017 | Arg2 (P (_, [|Arg0 (C (Q {a = 1; b = 1}))] ), Fois, f4) -> f4
1018 | Arg2 (f3, Fois, P (_, [|Arg0 (C (Q {a = 1; b = 1}))] )) -> f3
1019 | Arg2 (f1, Fois, f2) when egal_ast f1 f2 -> Arg2 (f1, Puissance, Arg0 (C (Q {a = 2; b = 1})))
1020 | Arg2 (P (x,a), Fois, P (y,b)) when egal_ast x y -> mult_poly (P (x,a)) (P (y,b))
1021 | 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))
1022 | 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))
1023 (* Simplifie la division *)
1024 | Arg2 (f3, Divise, Arg0 (C (Q {a = 0; b = 1}))) -> (* print_ast_arbre f3; f3 *) failwith "Div par 0 dans simplifie"
1025 | Arg2 (Arg0 (C (Q {a = 0; b = 1})), Divise, f4) -> ast_null
1026 | Arg2 (Arg0 (C x), Divise, Arg0 (C y)) -> Arg0 (C (div_constant x y))
1027 | Arg2 (f3, Divise, Arg0 (C (Q {a = 1; b = 1}))) -> f3
1028 | Arg2 (f1, Divise, f2) when egal_ast f1 f2 -> ast_un
1029 | Arg2 (f1, Divise, Arg2 (Arg0 (C (Q {a = 1; b = 1})), Divise, f2)) -> simplifie_ast (Arg2 (f1, Fois, f2))

```

```

1030 | Arg2 (P (x,a), Divise, P (y,b)) when egal_ast x y -> simplifie_ast (polys_to_frac (P (x,a)) (P (y,b)))
1031 | 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)) ↗

1032 (* Simplifié au max *)
1033 | _ -> ast_simp
1034 )
1035
1036 | P (x,a) -> (
1037   (* if Array.length a = 0 then ast_null
1038   else if Array.length a = 1 then simplifie_ast a.(0)
1039   else *) ajuste_poly (P (x,a))
1040 )
1041
1042 | F (x,a,b) -> ajuste_frac ast_arbre
1043
1044 | _ -> ast_arbre
1045
1046
1047
1048
1049
1050 (* ----- Matrices et determinant cramer et det ----- *)
1051
1052
1053
1054 and det_c mat =
1055   (* calcul d'un déterminant scalaire *)
1056   let n = Array.length mat in
1057   assert(n = Array.length mat.(0));
1058   if n = 1
1059   then mat.(0).(0)
1060   else
1061     (
1062       let mat2 = Array.make_matrix (n-1) (n-1) c_zero in
1063       let res = ref c_zero in
1064       for i = 0 to n-1 do
1065         for j = 1 to n-1 do
1066           for k = 0 to n-1 do
1067             if k < i
1068             then mat2.(j-1).(k) <- mat.(j).(k)
1069             else if k > i
1070             then mat2.(j-1).(k-1) <- mat.(j).(k)
1071             else ()
1072       done;

```

```

1073     done;
1074     if not (is_zero_ast (Arg0 (C mat.(0).(i)))) then (
1075         res := add_constante !res (mult_constante (if i mod 2 = 0 then c_un else c_minus_un) (mult_constante (det_c
1076             mat2) mat.(0).(i)));
1077     )
1078     done;
1079     !res
1080 )
1081
1082 and det_ast mat =
1083     (* calcul d'un déterminant scalaire *)
1084     let n = Array.length mat in
1085     assert(n = Array.length mat.(0));
1086     if n = 1
1087     then mat.(0).(0)
1088     else
1089     (
1090         let mat2 = Array.make_matrix (n-1) (n-1) ast_null in
1091         let res = ref ast_null in
1092         for i = 0 to n-1 do
1093             for j = 1 to n-1 do
1094                 for k = 0 to n-1 do
1095                     if k < i
1096                     then mat2.(j-1).(k) <- mat.(j).(k)
1097                     else if k > i
1098                     then mat2.(j-1).(k-1) <- mat.(j).(k)
1099                     else ()
1100                 done;
1101             done;
1102
1103             if not (is_zero_ast mat.(0).(i)) then (
1104                 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,
1105                     Fois, mat.(0).(i)))));
1106             )
1107             done;
1108             simplifie_ast !res
1109         )
1110
1111 and systeme_cramer famille_colonne colonne =
1112     (* Résout un système de Cramer de type constante *)
1113     let n1 = Array.length colonne in
1114     let n2 = Array.length famille_colonne in

```

```

1115  assert (n2 > 0);
1116  let n3 = Array.length famille_colonne.(n2-1) in
1117  assert (n1 = n2 && n2 = n3);
1118
1119  let colonne_res = Array.make n1 c_zero in
1120  let det_p = det_c famille_colonne in
1121
1122  for k = 0 to n1 -1 do
1123    let mat_k = Array.make_matrix n1 n1 c_zero in
1124
1125    for i = 0 to n1 - 1 do
1126      for j = 0 to n1 - 1 do
1127        if j <> k
1128        then mat_k.(i).(j) <- famille_colonne.(i).(j)
1129        else mat_k.(i).(j) <- colonne.(i)
1130      done;
1131    done;
1132    (* Array.iter (fun tab -> Array.iter (fun x -> Printf.fprintf file "(%f,%f);" x.re x.im) tab; rintf.fprintf file
1133      "\n") mat_k; *)
1134    colonne_res.(k) <- div_constante (det_c mat_k) det_p;
1135  done;
1136  colonne_res
1137
1138
1139  and systeme_cramer_log famille_colonne colonne =
1140    (* Résout un systeme de Cramer de type constante *)
1141    let n1 = Array.length colonne in
1142    let n2 = Array.length famille_colonne in
1143    assert (n2 > 0);
1144    let n3 = Array.length famille_colonne.(n2-1) in
1145    assert (n1 = n2 && n2 = n3);
1146
1147    let colonne_res = Array.make n1 ast_null in
1148    let det_p = det_ast famille_colonne in
1149
1150    for k = 0 to n1 -1 do
1151      let mat_k = Array.make_matrix n1 n1 ast_null in
1152
1153      for i = 0 to n1 - 1 do
1154        for j = 0 to n1 - 1 do
1155          if j <> k
1156          then mat_k.(i).(j) <- famille_colonne.(i).(j)
1157          else mat_k.(i).(j) <- colonne.(i)

```

```

1158     done;
1159     done;
1160     (* Array.iter (fun tab -> Array.iter (fun x -> Printf.fprintf file "(%f,%f);" x.re x.im) tab; rintf.fprintf file
1161        "\n") mat_k; *)
1162     colonne_res.(k) <- Arg2((det_ast mat_k),Divise, det_p);
1163     done;
1164     colonne_res
1165
1166
1167
1168
1169     (* ----- Fonctions d'operation sur polynome ----- *)
1170
1171
1172
1173     and poly_copie poly =
1174       (* copie un polynome pour eviter les effets de bords *)
1175       match poly with
1176       | P (x,p) -> (
1177         let n = Array.length p in
1178         let pc = Array.make n p.(0) in
1179         for i = 0 to n-1 do
1180           pc.(i) <- p.(i)
1181         done;
1182         P (x, pc)
1183       )
1184       | _ -> failwith "poly_copie : pas un polynome"
1185
1186
1187     and poly_indet poly =
1188       (* renvoie l'indeterminer du polynome *)
1189       match poly with
1190       | P (x,p) -> x
1191       | _ -> failwith "poly_indet : pas un polynome"
1192
1193
1194     and poly_array poly =
1195       (* renvoie l'array du polynome *)
1196       match poly with
1197       | P (x,p) -> p
1198       | _ -> failwith "poly_array : pas un polynome"
1199
1200

```



```

1201 and deg_poly (poly:ast_elem) =
1202   (* renvoie le degré d'un polynome *)
1203   match poly with
1204   | P (x,p) ->
1205     (
1206       let n = ref (Array.length p) in
1207
1208       while !n > 0 && is_zero_ast p.(!n-1) do
1209         n := !n - 1;
1210       done;
1211
1212       if !n <= 0 then min_int else !n-1
1213     )
1214
1215   | _ -> failwith "deg_poly : pas un polynome"
1216
1217
1218 and ajuste_poly (poly:ast_elem) =
1219   (* remet le polynôme à une taille correspondant à son degré *)
1220   match poly with
1221   | P (x,p) -> (
1222     let degree = deg_poly poly in
1223
1224     (* print_debug degree; *)
1225
1226     if degree < 0
1227     then P (x, [|ast_null|])
1228     else
1229       (
1230         let new_poly = P (x, Array.make (degree+1) ast_null) in
1231         match new_poly with
1232         | P (y,np) -> (
1233           for i = 0 to degree do
1234             np.(i) <- simplifie_ast p.(i);
1235           done;
1236           P (y,np))
1237         | _ -> failwith "ajuste_poly : impossible case"
1238       )
1239     )
1240
1241   | _ -> failwith "ajuste_poly : pas un polynome"
1242
1243
1244 and poly_unitaire poly =

```

```

1245 (* renvoie le polynome unitaire associé et son ancien coeff dominant *)
1246 match poly with
1247 | P (x,p) -> (
1248   let n = Array.length p - 1 in
1249   let pu = Array.make (n+1) p.(0) in
1250   for i = 0 to n do
1251     pu.(i) <- simplifie_ast (ast_divise p.(i) p.(n));
1252   done;
1253   (P (x,pu),p.(n))
1254 )
1255 | _ -> failwith "poly unitaire : pas un polynome"
1256
1257
1258 and add_poly (poly1:ast_elem) (poly2:ast_elem) =
1259 (* addition *)
1260 match (poly1,poly2) with
1261 | P (x,p1),P (y,p2) when egal_ast x y -> (
1262   let n = max (deg_poly poly1) (deg_poly poly2) in
1263
1264   if n < 0 then
1265     P (x, [|ast_null|])
1266
1267   else
1268     (
1269       let res_poly_array = Array.make (n+1) ast_null in
1270       for i = 0 to (deg_poly poly1) do
1271         res_poly_array.(i) <- p1.(i)
1272       done;
1273       for i = 0 to (deg_poly poly2) do
1274         res_poly_array.(i) <- Arg2 (res_poly_array.(i), Plus, p2.(i))
1275       done;
1276       ajuste_poly (P (x, res_poly_array))
1277     )
1278 )
1279
1280 | _,_ -> failwith "add_poly : pas des polynômes ou polynome d'indet diff"
1281
1282
1283 and neg_poly (poly:ast_elem) =
1284 (* négation *)
1285 match poly with
1286 | P (x,p) -> (
1287   let n = deg_poly poly in
1288

```

```

1289     if n >= 0
1290     then
1291         (
1292             let res_poly_array = Array.make (n+1) p.(0) in
1293             for i = 0 to n do
1294                 res_poly_array.(i) <- Arg2 (ast_minus_un, Fois, p.(i))
1295             done;
1296             (P (x, res_poly_array))
1297         )
1298     else poly
1299 )
1300
1301 | _ -> failwith "neg_poly : pas un polynôme"
1302
1303
1304 and minus_poly (poly1:ast_elem) (poly2:ast_elem) =
1305     (* soustraction *)
1306     match (poly1,poly2) with
1307     | P (x,p1),P (y,p2) when egal_ast x y -> (
1308         let n = max (deg_poly poly1) (deg_poly poly2) in
1309
1310         if n < 0 then
1311             P (x, [|ast_null|])
1312
1313         else
1314             (
1315                 let res_poly_array = Array.make (n+1) ast_null in
1316
1317                 for i = 0 to (deg_poly poly1) do
1318                     res_poly_array.(i) <- p1.(i)
1319                 done;
1320                 for i = 0 to (deg_poly poly2) do
1321                     res_poly_array.(i) <- Arg2 (res_poly_array.(i), Moins, p2.(i))
1322                 done;
1323
1324                 ajuste_poly (P (x, res_poly_array))
1325             )
1326         )
1327
1328 | _,_ -> failwith "minus_poly : pas des polynômes ou polynome d'indet diff"
1329
1330
1331 and mult_poly (poly1:ast_elem) (poly2:ast_elem) =
1332     (* multiplication *)

```

```

1333 match (poly1,poly2) with
1334 | P (x,p1),P (y,p2) when egal_ast x y ->
1335 (
1336   let n = (deg_poly poly1) + (deg_poly poly2) in
1337
1338   if n < 0 then
1339     P (x, [|ast_null|])
1340
1341   else
1342     (
1343       let res_poly_array = Array.make (n+1) ast_null in
1344       for i = 0 to (deg_poly poly1) do
1345         for j = 0 to (deg_poly poly2) do
1346           res_poly_array.(i + j) <- Arg2 (res_poly_array.(i + j), Plus, Arg2 (p1.(i) ,Fois ,p2.(j) ))
1347         done;
1348       done;
1349       ajuste_poly (P (x, res_poly_array))
1350     )
1351 )
1352
1353 | P (x,p1), ast when not (egal_ast x ast)->
1354 (
1355   let n = deg_poly poly1 in
1356
1357   if n < 0 then
1358     P (x, [|ast_null|])
1359
1360   else
1361     (
1362       let res_poly_array = Array.make (n+1) ast_null in
1363       for i = 0 to n do
1364         res_poly_array.(i) <- Arg2 (p1.(i), Fois, ast);
1365       done;
1366       ajuste_poly (P (x, res_poly_array))
1367     )
1368 )
1369
1370 | ast,P (x, p2) when not (egal_ast x ast)->
1371 (
1372   let n = deg_poly poly2 in
1373
1374   if n < 0 then
1375     P (x, [|ast_null|])
1376

```

```

1377     else
1378     (
1379         let res_poly_array = Array.make (n+1) ast_null in
1380         for i = 0 to n do
1381             res_poly_array.(i) <- Arg2 (p2.(i), Fois, ast)
1382         done;
1383         ajuste_poly (P (x, res_poly_array))
1384     )
1385 )
1386
1387 | _,_ -> failwith "mult_poly : pas des polynômes ou polynome d'indet diff"
1388
1389
1390 and mult_poly_scal (poly:ast_elem) (lambda:constante) =
1391     (* multiplication *)
1392     match poly with
1393     | P (x,p) ->
1394     (
1395         let pi = Array.make (Array.length p) p.(0) in
1396         for i = 0 to Array.length p - 1 do
1397             pi.(i) <- Arg2 (Arg0 (C lambda), Fois, p.(i));
1398         done;
1399         ajuste_poly (P (x,pi))
1400     )
1401
1402 | _ -> failwith "mult_poly_scal : pas de polynome"
1403
1404
1405 and puissance_poly_ent (poly:ast_elem) (n:int) =
1406     (* puissance non optimisé / naive *)
1407     match poly with
1408     | P (x,p) ->
1409     (
1410         let pres = ref (P (x, [|ast_un|])) in
1411         for i = 0 to n-1 do
1412             pres := mult_poly !pres poly
1413         done;
1414         ajuste_poly (!pres)
1415     )
1416 | _ -> failwith "puissance_poly_ent : pas un polynome"
1417
1418
1419 and div_euclid_poly (poly1:ast_elem) (poly2:ast_elem) =
1420     (* division euclidienne *)

```

```

1421 let poly1_a = (ajuste_poly poly1) in
1422 let poly2_a = (ajuste_poly poly2) in
1423
1424 match (poly1_a,poly2_a) with
1425
1426 | P (x,p1),P (y,p2) when egal_ast x y ->
1427 (
1428   let d1 = deg_poly poly1_a in
1429   let d2 = deg_poly poly2_a in
1430
1431   if d1 < d2 || d2 < 0
1432   then (P (x,[|ast_null|]) , poly1_a)
1433
1434   else if d2 = 0
1435   then (ajuste_poly (mult_poly poly1_a (Arg2 (ast_un, Divise, p2.(0)))) , P (x,[|ast_null|]))
1436
1437   else
1438     (
1439       let quotient_poly_array = Array.make (d1 - d2 + 1) ast_null in
1440       let reste_poly_array = ref (poly_array (poly_copie poly1_a)) in
1441
1442       for i = (d1 - d2) downto 0 do
1443         if i + d2 < Array.length !reste_poly_array
1444         then begin
1445
1446           let quotient_poly_array_t = Array.make (d1 - d2 + 1) ast_null in
1447           quotient_poly_array.(i) <- simplifie_ast (Arg2 (!reste_poly_array.(i + d2), Divise, simplifie_ast p2.(d2)));
1448           quotient_poly_array_t.(i) <- simplifie_ast (Arg2 (!reste_poly_array.(i + d2), Divise, simplifie_ast p2.(d2)));
1449
1450           let poly_soustrait = ajuste_poly (mult_poly poly2_a (P (x,quotient_poly_array_t))) in
1451           reste_poly_array := poly_array (ajuste_poly (minus_poly (P (x,!reste_poly_array)) poly_soustrait));
1452
1453         end;
1454       done;
1455
1456       (ajuste_poly (P (x,quotient_poly_array)) ,ajuste_poly (P (x,!reste_poly_array)))
1457     )
1458
1459 | _,_ -> failwith "div_euclid_poly : pas des polynômes ou polynome d'indet diff"
1460
1461
1462 and modulo_poly (poly1:ast_elem) (poly2:ast_elem) =
1463 (* modulo sur les polynomes poly1 mod poly2*)

```

```

1464   match (div_euclid_poly poly1 poly2) with
1465   | quotient,reste -> reste
1466
1467
1468 and pgcd_poly (poly1:ast_elem) (poly2:ast_elem) =
1469   (* pgcd / algorithme d'euclide *)
1470   match (poly1,poly2) with
1471   | P (x,p1),P (y,p2) -> (
1472
1473     if not (egal_ast x y) then failwith "pgcd_poly : indet differente";
1474
1475     let poly_r = ref (ajuste_poly (modulo_poly poly1 poly2)) in
1476     let poly_p = ref poly1 in
1477     let poly_q = ref poly2 in
1478
1479     while not (is_zero_ast !poly_r) do
1480
1481       poly_p := !poly_q;
1482       poly_q := !poly_r;
1483       poly_r := ajuste_poly (modulo_poly !poly_p !poly_q);
1484
1485     done;
1486
1487     match poly_unitaire (ajuste_poly !poly_q) with | a,b -> a )
1488
1489   | _,_ -> failwith "pgcd_poly : pas un couple de polynômes"
1490
1491
1492 and pgcd_poly_algebrique (poly1:ast_elem) (poly2:ast_elem) =
1493   (* pgcd / algorithme d'euclide *)
1494   match (poly1,poly2) with
1495   | P (x,p1),P (y,p2) -> (
1496
1497     if not (egal_ast x y) then failwith "pgcd_poly : indet differente";
1498
1499     let poly_p = ref poly1 in
1500     let poly_q = ref poly2 in
1501
1502     while not (deg_poly !poly_q < 1) do
1503       let poly_r = modulo_poly !poly_p !poly_q in
1504       poly_p := !poly_q;
1505       poly_q := poly_r;
1506     done;
1507

```

```

1508     match poly_unitaire (ajuste_poly !poly_p) with | a,b -> a )
1509
1510 | _,_ -> failwith "pgcd_poly : pas un couple de polynômes"
1511
1512
1513 and poly_bezout (poly1:ast_elem) (poly2:ast_elem) =
1514   (* renvoie s et t deux polys tels que poly1 * s + t * poly2 = poly1 ^ poly2 *)
1515   match (poly1,poly2) with
1516   | P (x,p1),P (y,p2) when egal_ast x y -> (
1517
1518     let c = ref poly1 in
1519     let c1 = ref (P (x,[|ast_un|])) in
1520     let c2 = ref (P (x,[|ast_null|])) in
1521
1522     let d = ref poly2 in
1523     let d1 = ref (P (x,[|ast_null|])) in
1524     let d2 = ref (P (x,[|ast_un|])) in
1525
1526     while not (is_zero_ast !d) do
1527       let quotient,reste = div_euclid_poly !c !d in
1528
1529       let q = ajuste_poly quotient in
1530
1531       let r1 = ajuste_poly (minus_poly !c1 (mult_poly q !d1)) in
1532       let r2 = ajuste_poly (minus_poly !c2 (mult_poly q !d2)) in
1533
1534       c := !d;
1535       c1 := !d1;
1536       c2 := !d2;
1537
1538       d := reste;
1539       d1 := r1;
1540       d2 := r2;
1541     done;
1542
1543     let uc = match !c with
1544       | P (x,tab) -> ( tab.(Array.length tab - 1)
1545         )
1546       | _ -> failwith "poly_bezout : pas un poly bug"
1547     in
1548
1549     let s = mult_poly !c1 (simplifie_ast (ast_divise ast_un uc)) in
1550     let t = mult_poly !c2 (simplifie_ast (ast_divise ast_un uc)) in
1551

```



```

1552     (ajuste_poly s,ajuste_poly t)
1553   )
1554
1555   | _,_ -> failwith "poly_bezout : pas un couple de polynômes"
1556
1557
1558   and poly_extend_euclidean (poly1:ast_elem) (poly2:ast_elem) (poly3:ast_elem) =
1559     (* page 13 transcendantal integration bronstein suppose poly1 ^ poly2 = 1 *)
1560     match (poly1,poly2,poly3) with
1561     | P (x,p1),P (y,p2),P (z,p3) when egal_ast x y && egal_ast y z -> (
1562       let s,t = poly_bezout poly1 poly2 in
1563       if (not (egal_ast s ast_null)) && deg_poly (mult_poly poly3 s) >= deg_poly poly2 then (
1564         let q,r = div_euclid_poly (mult_poly poly3 s) poly2 in
1565         (ajuste_poly r,ajuste_poly (add_poly (mult_poly poly3 t) (mult_poly q poly1)))
1566       ) else (
1567         (s,t)
1568       )
1569     )
1570   | _,_,_ -> failwith "poly_extend_euclidean : entré non valide"
1571
1572
1573   and frac_indet frac =
1574     (* renvoie l'indeterminer de la fraction rationnelle *)
1575     match frac with
1576     | F (x,a,b) -> x
1577     | _ -> failwith "frac_indet : pas une fraction rationnelle"
1578
1579
1580   and frac_nom frac =
1581     (* renvoie l'array du nominateur de la fraction rationnelle *)
1582     match frac with
1583     | F (x,a,b) -> a
1584     | _ -> failwith "frac_nom : pas une fraction rationnelle"
1585
1586
1587   and frac_denom frac =
1588     (* renvoie l'array du dénominateur de la fraction rationnelle *)
1589     match frac with
1590     | F (x,a,b) -> b
1591     | _ -> failwith "frac_denom : pas une fraction rationnelle"
1592
1593
1594   and deg_frac (frac:ast_elem) =
1595     (* renvoie le degré d'un polynome *)

```

```

1596   match frac with
1597   | F (x,a,b) ->
1598       (
1599           let pa = P (x, a) in
1600           let pb = P (x, b) in
1601           deg_poly pa - deg_poly pb
1602       )
1603
1604   | _ -> failwith "deg_frac : pas une fraction rationnelle "
1605
1606
1607 and ajuste_frac (frac:ast_elem) =
1608   (* remet la fraction rationnelle à une taille correspondant à son degré *)
1609   match frac with
1610   | F (x,a,b) ->
1611       (
1612           assert (Array.length b > 0);
1613
1614           let pa = ajuste_poly (P (x, a)) in
1615           let pb = ajuste_poly (P (x, b)) in
1616           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
1617           let pa2,pra = div_euclid_poly pa pgcd in
1618           let pb2,prb = div_euclid_poly pb pgcd in
1619
1620           (F (x,poly_array (ajuste_poly pa2), poly_array (ajuste_poly pb2)))
1621       )
1622
1623   | _ -> failwith "ajuste_frac : pas une fraction rationnelle"
1624
1625
1626 and add_frac (frac1:ast_elem) (frac2:ast_elem) =
1627   (* addition *)
1628   match (frac1,frac2) with
1629   | F (x,pa1,pb1),F (y,pa2,pb2) when egal_ast x y ->
1630       (
1631           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) -> a
1632           b | _ -> failwith "erreur" in
1633           let denom = match mult_poly (P (x,pb1)) (P (x,pb2)) with | P (a,b) -> b | _ -> failwith "erreur" in
1634           ajuste_frac (F (x, nom, denom))
1635       )
1636
1637   | _,_ -> failwith "add_frac : pas des fractions rationnelles ou frac d'indet diff"
1638

```

```

1639 and neg_frac (frac:ast_elem) =
1640   (* négation *)
1641   match frac with
1642   | F (x,a,b) ->
1643     (
1644       let a1 = match neg_poly (P (x,a)) with | P (i,k) -> k | _ -> failwith "erreur" in
1645       F (x, a1, b)
1646     )
1647   | _ -> failwith "neg_frac : pas une fraction rationnelle"
1648
1649
1650
1651 and minus_frac (frac1:ast_elem) (frac2:ast_elem) =
1652   (* soustraction *)
1653   match (frac1,frac2) with
1654   | F (x,pa1,pb1),F (y,pa2,pb2) when egal_ast x y ->
1655     (
1656       add_frac frac1 (neg_frac frac2)
1657     )
1658   | _,_ -> failwith "minus_frac : pas des fractions rationnelles ou frac d'indet diff"
1659
1660
1661
1662 and mult_frac (frac1:ast_elem) (frac2:ast_elem) =
1663   (* multiplication *)
1664   match (frac1,frac2) with
1665   | F (x,pa1,pb1),F (y,pa2,pb2) when egal_ast x y ->
1666     (
1667       let nom = match mult_poly (P (x,pa1)) (P (x,pa2)) with | P (a,b) -> b | _ -> failwith "erreur" in
1668       let denom = match mult_poly (P (x,pb1)) (P (x,pb2)) with | P (a,b) -> b | _ -> failwith "erreur" in
1669       F (x, nom, denom)
1670     )
1671   | _,_ -> failwith "mult_frac : pas des fractions rationnelles ou frac d'indet diff"
1672
1673
1674
1675 and puissance_frac_ent (frac:ast_elem) (n:int) =
1676   (* puissance non optimisé / naive *)
1677   match frac with
1678   | F (x,a,b) ->
1679     (
1680       let pres = ref (F (x, [|ast_un|], [|ast_un|])) in
1681       for i = 0 to n-1 do
1682         pres := mult_frac !pres frac

```

```

1683     done;
1684     ajuste_frac (!pres)
1685   )
1686 | _ -> failwith "puissance_frac_ent : pas une fraction"
1687
1688
1689 and mult_frac_scal (frac:ast_elem) (lambda:constante) =
1690   (* multiplication *)
1691   match frac with
1692   | F (x,pa,pb) ->
1693     (
1694       let nom = match (mult_poly_scal (P (x,pa)) lambda) with | P (y,p) -> p | _ -> failwith "bug" in
1695       F (x, nom, pb)
1696     )
1697
1698 | _ -> failwith "mult_frac_scal : pas de fraction rationnelle"
1699
1700
1701 and inv_frac (frac:ast_elem) =
1702   (match frac with | F (x,a,b) -> assert (not (is_zero_ast frac)); F (x, b, a) | _ -> failwith "inv_frac : pas une
1703     fraction rationnel")
1704
1705 and divide_frac (frac1:ast_elem) (frac2:ast_elem) =
1706   (* division *)
1707   match (frac1,frac2) with
1708   | F (x,pa1,pb1),F (y,pa2,pb2) when egal_ast x y -> mult_frac frac1 (inv_frac frac2)
1709   | _,_ -> failwith "divide_frac : pas des fractions rationnelles ou frac d'indet diff"
1710
1711
1712 and compose_frac (frac1:ast_elem) (frac2:ast_elem) =
1713   (* composition de fraction *)
1714   match (frac1,frac2) with
1715   | F (x,pa1,pb1),F (y,pa2,pb2) when egal_ast x y -> (
1716     let ft1 = ref (F (x, [|pa1.(0)|], [|ast_un|])) in (* partie haute *)
1717     let ft2 = ref (F (x, [|pb1.(0)|], [|ast_un|])) in (* partie basse *)
1718
1719     for i = 1 to Array.length pa1 - 1 do
1720       ft1 := add_frac !ft1 (mult_frac (F (x, [|pa1.(i)|], [|ast_un|])) (puissance_frac_ent frac2 i))
1721     done;
1722
1723     for i = 1 to Array.length pb1 - 1 do
1724       ft2 := add_frac !ft2 (mult_frac (F (x, [|pb1.(i)|], [|ast_un|])) (puissance_frac_ent frac2 i))
1725     done;

```

```

1726
1727     divide_frac !ft1 !ft2
1728 )
1729 | _,_ -> failwith "compose_frac : pas des fractions rationnelles ou frac d'indet diff"
1730
1731
1732 and polys_to_frac (poly1:ast_elem) (poly2:ast_elem) =
1733 (* fait une fraction rationnelle à partir de deux polynomes *)
1734 match (poly1,poly2) with
1735 | P (x,p1),P (y,p2) when egal_ast x y ->
1736 (
1737     assert (not (is_zero_ast poly2));
1738     F (x, p1, p2)
1739 )
1740
1741 | _,_ -> failwith "polys_to_frac : pas des polynomes ou frac d'indet diff"
1742 ;;
1743
1744
1745 let decomp_en_polynome_sans_carre polynome =
1746 (* algorithme de yun pour la décomposition *)
1747
1748 let derive_theta poly = match poly with
1749 | P (x,a) -> (
1750     let n = deg_poly poly in
1751     if n <= 0 then P (x,[|ast_null|]) else (
1752         let b = Array.make n ast_null in
1753         for i = 0 to n-1 do
1754             b.(i) <- simplifie_ast (Arg2 (ast_const (i+1) 1, Foix, a.(i+1)))
1755         done;
1756         P (x,b)
1757     )
1758 )
1759 | _ -> failwith "decomp en poly sans carré"
1760 in
1761
1762 let poly,u = poly_unitaire polynome in
1763 (* version unitaire du polynome *)
1764 let i = ref 1 in
1765 let factorisation = ref [] in
1766 let b = ref (ajuste_poly (derive_theta poly)) in
1767 let c = ref (ajuste_poly (pgcd_poly poly !b)) in
1768 let w = ref (ajuste_poly poly) in
1769

```

```

1770   if not (egal_ast !c (P (poly_indet !c, [|ast_un|])))
1771   then (
1772
1773       let w1,w2 = div_euclid_poly poly !c in
1774       w := w1; assert (is_zero_ast w2);
1775       let y1,y2 = div_euclid_poly !b !c in
1776       let y = ref (ajuste_poly y1) in assert (is_zero_ast y2);
1777       let z = ref (ajuste_poly (minus_poly !y (derive_theta !w))) in
1778
1779       while (not (is_zero_ast !z)) do
1780
1781           let g = pgcd_poly !w !z in
1782           factorisation := (g,!i)::(!factorisation);
1783           i := !i + 1;
1784           let w3,w4 = div_euclid_poly !w g in
1785           w := ajuste_poly w3; assert (is_zero_ast w4);
1786           let y3,y4 = div_euclid_poly !z g in
1787           y := ajuste_poly y3; assert (is_zero_ast y4);
1788           z := ajuste_poly (minus_poly !y (derive_theta !w));
1789       done;
1790   )
1791   ;
1792
1793   factorisation := (!w,!i)::(!factorisation);
1794
1795   let facto_array = (Array.of_list !factorisation) in
1796   Array.sort (fun (p1,j1) (p2,j2) -> j1-j2 ) facto_array;
1797   Array.map (fun (x,j) -> (simplifie_ast x,j)) facto_array
1798   ;;
1799
1800
1801   let calcul_des_fractions_partielles fraction decomp_en_polynome_sans_carre =
1802   (* nb la fraction d'entré vérifier deg A < def B*)
1803   let get_0 (a,b) = a in
1804   let get_1 (a,b) = b in
1805
1806   let const_of_ast ast = match ast with
1807   | Arg0 f -> ( match f with
1808   | C c -> c
1809   | _ -> failwith "calcul_des_fractions_partielles : pas une cste"
1810   )
1811   | _ -> failwith "calcul_des_fractions_partielles : pas un ast"
1812   in
1813

```

```

1814 let coeff poly k = match poly with
1815   | P (x,tab) -> tab.(k)
1816   | _ -> failwith "calcul_des_fractions_partielles : pas un polynome"
1817 in
1818
1819 let poly_x_k indet k =
1820   let p = Array.make (k+1) ast_null in
1821   p.(k) <- ast_un;
1822   P (indet,p)
1823 in
1824
1825 let fa = frac_nom fraction in
1826 let fb = frac_denom fraction in
1827 let fx = frac_indet fraction in
1828
1829 let n = Array.length decomp_en_polynome_sans_carre in
1830 let m = ref 0 in
1831
1832 for i = 0 to n-1 do
1833   m := !m + (deg_poly (get_0 decomp_en_polynome_sans_carre.(i))) * (get_1 decomp_en_polynome_sans_carre.(i))
1834 done;
1835
1836 let mat_partielles = Array.make_matrix !m !m c_zero in
1837 let array_poly = Array.make !m c_zero in
1838
1839 let list_decomp = ref [] in
1840
1841 for i = 0 to n-1 do
1842   if deg_poly (get_0 decomp_en_polynome_sans_carre.(i)) > 0 then
1843     (
1844       for j = 1 to get_1 decomp_en_polynome_sans_carre.(i) do
1845         let q,r = div_euclid_poly (P (fx,fb)) (puissance_poly_ent (get_0 decomp_en_polynome_sans_carre.(i)) j) in
1846         assert (is_zero_ast r);
1847         list_decomp := (q,deg_poly (get_0 decomp_en_polynome_sans_carre.(i)))::(!list_decomp)
1848       done;
1849     )
1850 done;
1851
1852 let tab_decomp = Array.of_list (List.rev !list_decomp) in
1853 let repere = ref 0 in
1854
1855 for i = 0 to Array.length tab_decomp - 1 do
1856   for j = get_1 tab_decomp.(i) - 1 downto 0 do
1857     for k = 0 to deg_poly (get_0 tab_decomp.(i)) do

```

```

1858     mat_partielles.(k+j).(!repere) <- const_of_ast (coeff (get_0 tab_decomp.(i)) k)
1859   done;
1860   repere := !repere + 1;
1861   done;
1862 done;
1863
1864 for i = 0 to deg_poly (P (fx,fa)) do
1865   array_poly.(i) <- const_of_ast (coeff (P (fx,fa)) i)
1866 done;
1867
1868 let mat_res = systeme_cramer mat_partielles array_poly in
1869 let liste_res = ref [] in
1870 repere := 0;
1871
1872 for i = 0 to Array.length tab_decomp - 1 do
1873   let p = ref (P (fx,[|ast_null|])) in
1874   for j = get_1 tab_decomp.(i) - 1 downto 0 do
1875     p := add_poly !p (mult_poly_scal (poly_x_k fx j) mat_res.(!repere));
1876     repere := !repere + 1
1877   done;
1878   liste_res := (!p)::!liste_res
1879 done;
1880
1881 let array_res = Array.of_list (List.rev !liste_res) in
1882
1883 let mat_decomp = Array.make_matrix n !m (P (fx, [|ast_null|])) in
1884 repere := 0;
1885
1886 for i = 0 to n-1 do
1887   if deg_poly (get_0 decomp_en_polynome_sans_carre.(i)) <> 0 then (
1888     for j = 0 to ((get_1 decomp_en_polynome_sans_carre.(i))-1) do
1889       mat_decomp.(i).(j) <- array_res.(!repere); (* mettre un marker*)
1890       repere := !repere + 1;
1891     done;
1892   )
1893 done;
1894
1895 mat_decomp
1896 ;;
1897
1898
1899 let calcul_des_fractions_partielles_log fraction decomp_en_polynome_sans_carre =
1900   (* nb la fraction d'entré vérifier deg A < def B*)
1901   let get_0 (a,b) = a in

```



```

1902   let get_1 (a,b) = b in
1903
1904   let coeff poly k = match poly with
1905     | P (x,tab) -> tab.(k)
1906     | _ -> failwith "calcul_des_fractions_partielles : pas un polynome"
1907   in
1908
1909   let poly_x_k indet k =
1910     let p = Array.make (k+1) ast_null in
1911     p.(k) <- ast_un;
1912     P (indet,p)
1913   in
1914
1915   let fa = frac_nom fraction in
1916   let fb = frac_denom fraction in
1917   let fx = frac_indet fraction in
1918
1919   let n = Array.length decomp_en_polynome_sans_carre in
1920   let m = ref 0 in
1921
1922   for i = 0 to n-1 do
1923     m := !m + (deg_poly (get_0 decomp_en_polynome_sans_carre.(i))) * (get_1 decomp_en_polynome_sans_carre.(i))
1924   done;
1925
1926   let mat_partielles = Array.make_matrix !m !m ast_null in
1927   let array_poly = Array.make !m ast_null in
1928
1929   let list_decomp = ref [] in
1930
1931   for i = 0 to n-1 do
1932     if deg_poly (get_0 decomp_en_polynome_sans_carre.(i)) > 0 then
1933       (
1934         for j = 1 to get_1 decomp_en_polynome_sans_carre.(i) do
1935           let q,r = div_euclid_poly (P (fx,fb)) (puissance_poly_ent (get_0 decomp_en_polynome_sans_carre.(i)) j) in
1936           assert (is_zero_ast r);
1937           list_decomp := (q,deg_poly (get_0 decomp_en_polynome_sans_carre.(i)))::(!list_decomp)
1938         done;
1939       )
1940   done;
1941
1942   let tab_decomp = Array.of_list (List.rev !list_decomp) in
1943   let repere = ref 0 in
1944
1945   for i = 0 to Array.length tab_decomp - 1 do

```

```

1946   for j = get_1 tab_decomp.(i) - 1 downto 0 do
1947     for k = 0 to deg_poly (get_0 tab_decomp.(i)) do
1948       mat_partielles.(k+j).(!repere) <- (coeff (get_0 tab_decomp.(i)) k)
1949     done;
1950     repere := !repere + 1;
1951   done;
1952 done;
1953
1954 for i = 0 to deg_poly (P (fx,fa)) do
1955   array_poly.(i) <- (coeff (P (fx,fa)) i)
1956 done;
1957
1958 let mat_res = systeme_cramer_log mat_partielles array_poly in
1959 let liste_res = ref [] in
1960 repere := 0;
1961
1962 for i = 0 to Array.length tab_decomp - 1 do
1963   let p = ref (P (fx, [|ast_null|])) in
1964   for j = get_1 tab_decomp.(i) - 1 downto 0 do
1965     p := add_poly !p (mult_poly (poly_x_k fx j) mat_res.(!repere));
1966     repere := !repere + 1
1967   done;
1968   liste_res := (!p)::!liste_res
1969 done;
1970
1971 let array_res = Array.of_list (List.rev !liste_res) in
1972 let mat_decomp = Array.make_matrix n !m (P (fx, [|ast_null|])) in
1973 repere := 0;
1974
1975 for i = 0 to n-1 do
1976   if deg_poly (get_0 decomp_en_polynome_sans_carre.(i)) <> 0 then (
1977     for j = 0 to ((get_1 decomp_en_polynome_sans_carre.(i))-1) do
1978       mat_decomp.(i).(j) <- array_res.(!repere); (* mettre un marker*)
1979       repere := !repere + 1;
1980     done;
1981   )
1982 done;
1983
1984 mat_decomp
1985 ;;
1986
1987
1988 let rec frac_reconnait (fonction:ast_elem) (indet:ast_elem) =
1989   if egal_ast fonction indet then F (indet, [|ast_null;ast_un|], [|ast_un|]) (* voir Z = X quand reconnait faire

```

```

1990   attention *)
1991   else ( match fonction with
1992         | Arg0 f -> F (indet, [|fonction|], [|ast_un|])
1993
1994         | Arg2 (f1, oper, f2) ->
1995           ( let frac1,frac2 = ajuste_frac (frac_reconnait f1 indet),ajuste_frac (frac_reconnait f2 indet) in
1996             match oper with
1997             | Plus -> add_frac frac1 frac2
1998             | Moins -> minus_frac frac1 frac2
1999             | Fois -> mult_frac frac1 frac2
2000             | Divise -> divise_frac frac1 frac2
2001             | Puissance -> (match frac2 with
2002                           | F (x, [|Arg0 (C (Q {a = 1; b = 1})|)], [|Arg0 (C (Q {a = 1; b = 1})|)]) -> puissance_frac_ent frac1 n
2003                           | _ -> F (indet, [|Arg2 (f1, Puissance, f2)|], [|ast_un|])
2004                           )
2005             | Compose -> compose_frac frac1 frac2
2006           )
2007
2008         | Abstrait f -> frac_reconnait f.d_fonction indet
2009
2010         | P (indet_x,poly_a) ->
2011           ( if egal_ast indet_x indet then F (indet,poly_a, [|ast_un|])
2012             else frac_reconnait (ast_of_poly_array poly_a indet_x) indet
2013           )
2014
2015         | F (indet_x,poly_a,poly_b) ->
2016           ( if egal_ast indet_x indet then F (indet,poly_a,poly_b)
2017             else frac_reconnait (ast_of_frac_array poly_a poly_b indet_x) indet
2018           )
2019       ;;
2020
2021
2022   let poly_reconnait (fonction:ast_elem) (indet:ast_elem) =
2023     match ajuste_frac (frac_reconnait fonction indet) with
2024     | F (x,poly,[|Arg0 (C (Q {a = 1; b = 1})|)]) -> P (x, poly)
2025     | _ -> failwith "poly_reconnait : n'arrive pas"
2026   ;;
2027
2028
2029   let resultant (poly1:ast_elem) (poly2:ast_elem) =
2030     (* résultante *)
2031     match (poly1,poly2) with
2032     | P (x,p1),P (y,p2) when egal_ast x y ->

```

```

2033 (
2034   let d1 = deg_poly poly1 in
2035   let d2 = deg_poly poly2 in
2036
2037   let mat = Array.make_matrix (d1+d2) (d1+d2) ast_null in
2038
2039   for i = 0 to d2 - 1 do
2040     for j = 0 to d1 do
2041       mat.(i).(i+j) <- p1.(d1-j)
2042     done;
2043   done;
2044
2045   for i = 0 to d1 - 1 do
2046     for j = 0 to d2 do
2047       mat.(i+d2).(i+j) <- p2.(d2-j)
2048     done;
2049   done;
2050
2051   det_ast mat
2052 )
2053
2054 | _,_ -> failwith "polys_to_frac : pas des polynomes ou frac d'indet diff"
2055 ;;
2056
2057
2058 let poly_factorisation poly =
2059
2060   match poly with
2061   | P (x,p) ->
2062     (
2063       decomp_en_polynome_sans_carre poly
2064     )
2065   | _ -> failwith "poly_factorisation : pas un poly"
2066 ;;
2067
2068
2069 let rec subsitue_abs ast alpha = match ast with
2070   | Abstrait a -> Arg0 (C alpha)
2071   | Arg2 (f1, oper, f2) -> Arg2 (subsitue_abs f1 alpha, oper, subsitue_abs f2 alpha)
2072   | P (x,a) -> P (x,Array.map (fun x -> subsitue_abs x alpha) a)
2073   | _ -> ast
2074 ;;
2075
2076

```

```

2077 let partie_primitive poly = match poly with
2078 | P (x,a) -> (
2079   if deg_poly poly < 0 then Res (P (x, [|ast_const 0 1|]))
2080   else if deg_poly poly = 0 then Res (P (x, [|ast_const 1 1|]))
2081   else (
2082     let v = ref ast_un in
2083     for i = deg_poly poly downto 0 do
2084       if not (is_zero_ast a.(i)) then v := a.(i)
2085     done;
2086     let b = poly_array (simplifie_ast (mult_poly poly (Arg2 (ast_un, Divise, !v)))) in
2087     let res = ref true in
2088     for i = 0 to deg_poly poly do
2089       let bc,v = is_const_ast b.(i) in
2090       if not bc then res := false
2091       else b.(i) <- Arg0 (C v)
2092     done;
2093     if !res then Res (P (x,b)) else Null
2094   )
2095 )
2096 | _ -> failwith "partie_primitive : pas un poly"
2097 ;;
2098
2099
2100
2101
2102
2103 (* ----- Integration ----- *)
2104
2105
2106 (* On détermine les extensions nécessaire à la primitivation et
2107 on modifie la fonction en remplaçant les fonctions par leurs extensions *)
2108 let rec determiner_extension_aux fonction = match fonction with
2109
2110 | Arg2 (f1, oper, f2) -> ( match f1,oper with
2111   | (Arg0 Ln,Compose) -> (1,fonction)::(determiner_extension_aux f2)
2112   | (Arg0 Exp,Compose) -> (2,fonction)::(determiner_extension_aux f2)
2113   | _ -> (determiner_extension_aux f1) @ (determiner_extension_aux f2)
2114 )
2115
2116 | Abstrait f -> determiner_extension_aux f.fonction
2117
2118 | P (x,a) -> (
2119   let res = ref (determiner_extension_aux x) in
2120   for i = 0 to Array.length a - 1 do

```

```

2121     res := !res @ (determiner_extension_aux a.(i))
2122   done;
2123   !res
2124 )
2125
2126 | F (x,a,b) -> (
2127   let res = ref (determiner_extension_aux x) in
2128   for i = 0 to Array.length a - 1 do
2129     res := !res @ (determiner_extension_aux a.(i))
2130   done;
2131   for i = 0 to Array.length b - 1 do
2132     res := !res @ (determiner_extension_aux b.(i))
2133   done;
2134   !res
2135 )
2136
2137 | _ -> []
2138 ;;
2139
2140
2141 let determiner_extension fonction =
2142   let extension = ref (Ext (X,{nom = "X";fonction = Arg0 X;d_fonction = Arg0 X;etat_derive = 0},Xe)) in
2143   let lst_temp = ref [] in
2144
2145   List.iter
2146     (
2147       fun (x,y) ->
2148         if not (List.exists (fun (a,b) -> a = x && egal_ast y b) !lst_temp)
2149         then (lst_temp := (x,y)::!lst_temp;
2150             extension := (Ext (if x = 1 then (Ln,{nom = "Ln";fonction = y;d_fonction = y;etat_derive = 0},!extension)
2151                               else (Exp,{nom = "Exp";fonction = y;d_fonction = y;etat_derive = 0},!extension))))
2152     )
2153   (List.sort (fun (a,b) (c,d) -> compare a c) (List.rev (determiner_extension_aux fonction)));
2154   !extension
2155 ;;
2156
2157
2158 let rec appartient_ext ext elem =
2159   (* vérifie si un élément est inclus dans une extension *)
2160   match ext with
2161   | Xe -> false
2162   | Ext (f_elem,fonction_abstraite,extt) -> if egal_ast elem (Abstrait fonction_abstraite) then true else appartient_ext extt elem

```

```

2163 ;;
2164
2165
2166 let rec inclus_ext ext1 ext2 =
2167   (* vérifie si l'extension deux est inclus dans la premiere *)
2168   match ext2 with
2169   | Xe -> true
2170   | Ext (f_elem,fonction_abstraite,extt2) -> (appartient_ext ext1 (Abstrait fonction_abstraite)) && (inclus_ext ext1
extt2)
2171 ;;
2172
2173
2174 let rec print_extension extension = match extension with
2175   | Xe -> ()
2176
2177   | Ext (f_elem,f,ext_suiv) -> (print_ast (Abstrait f); print_ast f.fonction ; print_extension ext_suiv)
2178 ;;
2179
2180
2181 (* On a  $f(\theta_n) = a(\theta_b)/b(\theta_n)$  on normalise et on rend f unitaire par rapport à un *)
2182 let rec normalise fonction theta =
2183
2184   if egal_ast fonction theta.fonction then fonction
2185
2186   else (
2187     match fonction with
2188
2189     | Arg2 (f1, oper, f2) -> Arg2 (normalise f1 theta, oper, normalise f2 theta)
2190
2191     | Abstrait f -> normalise f.d_fonction theta
2192
2193     | P (x,a) -> (
2194       let indet = normalise x theta in
2195       let aa = Array.make (Array.length a) ast_null in
2196       for i = 0 to Array.length a - 1 do
2197         aa.(i) <- normalise a.(i) theta
2198       done;
2199       P (indet,aa)
2200     )
2201
2202     | F (x,a,b) -> (
2203       let indet = normalise x theta in
2204       let aa = Array.make (Array.length a) ast_null in
2205       for i = 0 to Array.length a - 1 do

```

```

2206     aa.(i) <- normalise a.(i) theta
2207     done;
2208     let bb = Array.make (Array.length b) ast_null in
2209     for i = 0 to Array.length b - 1 do
2210         bb.(i) <- normalise b.(i) theta
2211     done;
2212     F (indet,aa,bb)
2213 )
2214
2215 | _ -> fonction
2216 )
2217 ;;
2218
2219
2220
2221 (* prend en parametre une fraction rationnel / un polynome et renvoie la partie rationnel intégré,
2222 la partie polynomiale non intégrés et la partie logarithmique non intégré *)
2223 let hermite_method fonction =
2224     (* voir p503 computer algebra *)
2225
2226     let get_0 (a,b) = a in
2227     (* let get_1 (a,b) = b in *)
2228
2229     match fonction with
2230     | F (Arg0 X,a,b) ->
2231     (
2232         let part_poly,part_frac_nom = div_euclid_poly (P (Arg0 X, a)) (P (Arg0 X, b)) in
2233         let part_frac_den,u = poly_unitaire (P (Arg0 X, b)) in
2234         let frac_red = polys_to_frac (mult_poly part_frac_nom (P (Arg0 X,[|Arg2(ast_un,Divise,u)|]))) part_frac_den in
2235         if (not (is_zero_ast frac_red)) then (
2236             let decomp_sans_carre = decomp_en_polynome_sans_carre part_frac_den in (* tableau de polynome *)
2237             let frac_partielles = calcul_des_fractions_partielles frac_red decomp_sans_carre in (* matrice carré avec nb *)
2238
2239             let part_rationnelle = ref (F (Arg0 X, [|ast_null|], [|ast_un|])) in
2240             let part_int = ref (F (Arg0 X, [|ast_null|], [|ast_un|])) in
2241
2242             let nb_carre = Array.length decomp_sans_carre in
2243
2244             for i = 0 to nb_carre - 1 do
2245
2246                 part_int := ajuste_frac (add_frac !part_int (polys_to_frac frac_partielles.(i).(0) (get_0 decomp_sans_carre.(i) 0
2247 )))
2248
2249                 for j = 2 to i+1 do

```



```

2249
2250   let n = ref j in
2251   while !n > 1 do
2252     (* 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 *)
2253     let (ps,pt) =
2254       match poly_extend_euclidean (get_0 decomp_sans_carre.(i)) (ajuste_poly (derive (get_0 decomp_sans_carre.(i)
2255         i)))) frac_partielles.(i).(!n-1) with
2256       | p1,p2 ->
2257
2258         let pp3,u3 = poly_unitaire frac_partielles.(i).(!n-1) in
2259
2260         (ajuste_poly (mult_poly p1 (P (Arg0 X, [|u3|])),ajuste_poly (mult_poly p2 (P (Arg0 X, [|u3|]))))
2261 in (* résout une équation pour trouver les polynomes ...*)
2262
2263     n := !n - 1;
2264
2265     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
2266
2267     part_rationnelle := ajuste_frac (minus_frac !part_rationnelle ft);
2268
2269     frac_partielles.(i).(!n-1) <- add_poly ps (mult_poly_scal (derive pt) (Q {a = 1; b = !n})); (* r[i,n] <- s + t'/n *)
2270
2271   done;
2272
2273   part_int := ajuste_frac (add_frac !part_int (polys_to_frac frac_partielles.(i).(0) (get_0 decomp_sans_carre.(i) 0)));
2274   assert (deg_frac !part_int < 0);
2275   done;
2276 done;
2277
2278   (ajuste_frac !part_rationnelle,ajuste_poly part_poly,ajuste_frac !part_int)
2279 )
2280 else
2281   (F (Arg0 X,[|ast_null|],[|ast_un|]),ajuste_poly part_poly, F (Arg0 X,[|ast_null|],[|ast_un|]))
2282 )
2283
2284 | P (Arg0 X,p) -> (F (Arg0 X, [|ast_null|], [|ast_un|]),fonction,F (Arg0 X, [|ast_null|], [|ast_un|]))
2285 | _ -> failwith "Fonction ne correspondant pas à hermite"
2286 ;;
2287
2288

```

```

2289 let rosthein_trager_method fonction =
2290   (* p507 intégration de la partie log de la fraction rationnel monic et square free *)
2291   let get_0 (a,b) = a in
2292   let get_c ast = match ast with | Arg0 (C c) -> c | _ -> E {nom = "erreur"; approx = 0.} in
2293
2294   match fonction with
2295   | F (x,a,b) when deg_frac fonction < 0 ->
2296     (
2297       let pa = P (x,a) in
2298       let pb = P (x,b) in
2299       let pdb = ajuste_poly (derive pb) in
2300
2301       let res_z = ajuste_poly (poly_reconnait (resultant (minus_poly pa (mult_poly (P (x,[|ast_Z|])) pdb)) pb) ast_Z) in
2302
2303       print_ast_arbre res_z;
2304
2305       let tab_facteur = poly_factorisation res_z in
2306       let part_log = ref ast_null in
2307       let id_alpha = ref 0 in
2308
2309       for i = 0 to Array.length tab_facteur - 1 do
2310         let p,x = match get_0 tab_facteur.(i) with | P (x,p) -> (p,x) | _ -> ([|],ast_null) in
2311         if deg_poly (get_0 tab_facteur.(i)) = 1
2312         then (
2313           let c = mult_constante (div_constante (get_c p.(0)) (get_c p.(1))) (c_const (-1) 1) in (* c = -c0 ou c0 racine*)
2314           let v = pgcd_poly (minus_poly pa (mult_poly_scal pdb c)) pb in
2315           part_log := Arg2 (!part_log, Plus, Arg2 (Arg0 (C c), Foies, Arg2 (Arg0 Ln, Compose, v)))
2316         )
2317         else (
2318           let c = {nom = "α_"; fonction = Arg0 Exp; d_fonction = Arg0 Ln; etat_derive = 0} in
2319           let v = pgcd_poly_algebrique (minus_poly pa (mult_poly pdb (Abstrait c))) pb in
2320           print_ast_arbre (pgcd_poly (minus_poly pa (mult_poly_scal pdb (c_const (1) 1))) pb);
2321
2322           for i = 0 to deg_poly (get_0 tab_facteur.(i)) - 1 do
2323             id_alpha := !id_alpha + 1;
2324             part_log := Arg2 (!part_log, Plus,
2325               Arg2 (Arg0 (C (E {nom = "α_"^(string_of_int !id_alpha); approx = 0.1})), Foies,
2326               Arg2 (Arg0 Ln, Compose, subsitue_abs v (E {nom = "α_"^(string_of_int !id_alpha); approx = 0.1})))));
2327
2328         done;
2329       )
2330

```

```

2331     done;
2332     !part_log
2333   )
2334   | _ -> failwith "rosthein_trager_method : pas une fraction rationnelle ou degré incohérent"
2335 ;;
2336
2337
2338 let integration_polynome fonction =
2339   (* intègre un polynome d'indéterminé X a coefficient dans Q *)
2340   match fonction with
2341   | P (Arg0 X, p1) ->
2342     ( let primitive = Array.make (Array.length p1 + 1) (ast_null) in
2343       for i = 1 to (Array.length p1) do
2344         match p1.(i-1) with
2345         | Arg0 (C c) -> primitive.(i) <- simplifie_ast (Arg2 (p1.(i-1), Divise, ast_const i 1))
2346         | _ -> failwith "pas un polynome à coeff dans C"
2347       done;
2348       ajuste_poly (P (Arg0 X,primitive))
2349     )
2350
2351   | _ -> failwith "integration_polynome : pas un polynome"
2352 ;;
2353
2354
2355 let integration_rationnel fonction =
2356   (* intègre les fonctions d'indéterminé x à coefficient dans C *)
2357   match fonction with
2358   | F (Arg0 X,a,b) -> (
2359
2360     let d = ajuste_frac fonction in
2361
2362     if deg_frac d >= 0 then (
2363       let f,g,h = hermite_method d in
2364
2365       let partie_rationnel = f in
2366       let partie_poly = ajuste_poly (integration_polynome g) in
2367       let partie_log = if is_zero_ast h then ast_null else (rosthein_trager_method h) in
2368
2369       (Arg2 (partie_rationnel, Plus, Arg2 (partie_poly, Plus, partie_log)))
2370     ) else (
2371       if is_zero_ast d then ast_null else (rosthein_trager_method d)
2372     )
2373   )
2374

```

```

2375 | P (x,p) -> ajuste_poly (integration_polynome fonction)
2376
2377 | _ -> (
2378   failwith "integration rationnel pas une fraction rationnel"
2379 )
2380 ;;
2381
2382
2383
2384
2385
2386 (* primitive si possible la fonction sinon renvoie Null *)
2387 let rec risch fonction extensions =
2388   (* On détermine les extensions nécessaire à la primitivation et
2389    on modifie la fonction en remplaçant les fonctions par leurs extensions theta_i *)
2390
2391   (* Recupère la dernière extension *)
2392   let type_ext = ref X in
2393   let theta_n = ref ({nom = "BL"; fonction = Arg0 X; d_fonction = Arg0 X; etat_derive = 0}) in
2394   let corps_diff = ref Xe in
2395
2396   (
2397     match extensions with
2398     | Xe -> ()
2399     | Ext (t_ext, theta, reste_corps) -> begin type_ext := t_ext; theta_n := theta; corps_diff := reste_corps end
2400   );
2401
2402   if ( match !type_ext with
2403       | X | Ln -> false
2404       | _ -> true )
2405   then Notimplementederror
2406
2407   else (
2408     (* On a f(theta_n) = a(theta_b)/b(theta_n) on normalise et on rend f unitaire par rapport à theta_n *)
2409     let frac_f = match ajuste_frac (frac_reconnait (normalise fonction !theta_n) !theta_n.fonction) with
2410         | F (indet,a,[|Arg0 C (Q {a = 1; b = 1})|]) -> P (indet, a)
2411         | x -> x
2412     in
2413
2414     (* print_ast frac_f; *)
2415
2416     match !type_ext with
2417     | X -> Res (integration_rationnel frac_f)
2418     | Ln -> log_case frac_f !corps_diff

```

```

2419 | Exp -> exp_case frac_f !corps_diff
2420 | _ -> Null (* erreur inutile déjà prévenu *)
2421 )
2422
2423
2424 and log_case fonction extensions =
2425
2426   let hermite_method_log f1 =
2427     (* voir p503 computer algebra *)
2428
2429     let get_0 (a,b) = a in
2430     (* let get_1 (a,b) = b in *)
2431
2432     match f1 with
2433     | F (x,a,b) ->
2434       (
2435
2436         assert (
2437           match x with
2438           | Arg2 (Arg0 Ln, Compose, _) -> true
2439           | _ -> false
2440         );
2441
2442         let part_poly_z,part_frac_nom_z = div_euclid_poly (P (Arg0 Z, a)) (P (Arg0 Z, b)) in
2443         let part_frac_den,u = poly_unitaire (P (Arg0 Z, b)) in
2444         let frac_red = polys_to_frac (mult_poly part_frac_nom_z (P (Arg0 Z,[|Arg2(ast_un,Divise,u)|]))) part_frac_den in
2445
2446         if (not (is_zero_ast frac_red)) then (
2447
2448           let decomp_sans_carre_z = decomp_en_polynome_sans_carre part_frac_den in (* tableau de polynome *)
2449           let frac_partielles_z = calcul_des_fractions_partielles_log frac_red decomp_sans_carre_z in (* matrice carré avec nb *)
2450
2451           (*transformer en ln les Z*)
2452           let part_poly = match part_poly_z with | P (Arg0 Z,a) -> P (x,a) | _ -> failwith "hermite_log_poly" in
2453           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 *)
2454           let frac_partielles = Array.make_matrix (Array.length frac_partielles_z) (Array.length frac_partielles_z.(0)) ast_null in (* matrice carré avec nb *)
2455           for i = 0 to (Array.length frac_partielles_z) - 1 do
2456             for j = 0 to Array.length frac_partielles_z.(0) - 1 do
2457               frac_partielles.(i).(j) <- match frac_partielles_z.(i).(j) with | P (Arg0 Z,a) -> P (x,a) | _ -> failwith "hermite_log"
2458           done;

```

```

2459     done;
2460
2461     let part_rationnelle = ref (F (x, [|ast_null|], [|ast_un|])) in
2462     let part_int = ref (F (x, [|ast_null|], [|ast_un|])) in
2463
2464     let nb_carre = Array.length decomp_sans_carre in
2465
2466     for i = 0 to nb_carre - 1 do
2467
2468         part_int := ajuste_frac (add_frac !part_int (polys_to_frac frac_partielles.(i).(0) (get_0 decomp_sans_carre.(i)))));
2469
2470         for j = 2 to i+1 do
2471
2472             let n = ref j in
2473             while !n > 1 do
2474
2475                 let (ps,pt) =
2476                     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
2477                     | p1,p2 ->
2478                         let pp3,u3 = poly_unitaire frac_partielles.(i).(!n-1) in
2479                         (ajuste_poly (mult_poly p1 (P (Arg0 X, [|u3|])),ajuste_poly (mult_poly p2 (P (Arg0 X, [|u3|]))))
2480 in (* résout une équation pour trouver les polynomes ...*)
2481
2482                 n := !n - 1;
2483
2484                 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
2485
2486                 part_rationnelle := ajuste_frac (minus_frac !part_rationnelle ft);
2487
2488                 frac_partielles.(i).(!n-1) <- add_poly ps (mult_poly_scal (derive pt) (Q {a = 1; b = !n})); (* r[i,n] <- s + t'/n *)
2489
2490             done;
2491
2492             part_int := ajuste_frac (add_frac !part_int (polys_to_frac frac_partielles.(i).(0) (get_0 decomp_sans_carre.(i)))));
2493
2494             assert (deg_frac !part_int < 0);
2495         done;
2496     done;
2497

```

```

2498         (ajuste_frac !part_rationnelle,ajuste_poly part_poly,ajuste_frac !part_int)
2499     ) else (
2500         (ast_null,P (x,poly_array part_poly_z),ast_null)
2501     )
2502 )
2503
2504 | P (x,p) -> (F (x, [|ast_null|], [|ast_un|]),f1,F (x, [|ast_null|], [|ast_un|]))
2505
2506 | _ -> failwith "Fonction ne correspondant pas à hermite_log"
2507 in
2508
2509 let integration_polynome_log f1 =
2510     match f1 with
2511     | P (x, p1) -> (
2512         assert (
2513             match x with
2514             | Arg2 (Arg0 Ln, Compose, _) -> true
2515             | _ -> false
2516         );
2517
2518         let l = deg_poly f1 in
2519         let dtheta = derive x in
2520         let ppoly = Array.make (l+2) ast_null in
2521
2522         let valide = ref true in
2523
2524         for i = l downto 0 do
2525
2526             ppoly.(i) <- (match risch (simplifie_ast (Arg2 (p1.(i)),Moins,Arg2(ast_const (i+1) 1,Fois,Arg2(ppoly.(i+1),Fois,
2527                 dtheta)))) extensions with
2528                 | Res f -> (* print_ast f; *) if (not (inclus_ext extensions (determiner_extension f)) && i <> 0) then valide := false; f
2529                 | x -> valide := false ; ast_null
2530             )
2531         done;
2532
2533         if !valide then Res (P (x, ppoly)) else Null
2534     )
2535 | _ -> failwith "integration_polynome_log"
2536 in
2537
2538 let rosthein_trager_method_log f1 =
2539     (* p507 intégration de la partie log de la fraction rationnel monic et square free *)

```

```

2540 let get_0 (a,b) = a in
2541 let get_c ast = match ast with | Arg0 (C c) -> c | _ -> E {nom = "erreur"; approx = 0.} in
2542
2543 match f1 with
2544 | F (x,a,b) when deg_frac f1 < 0 ->
2545 (
2546   assert (
2547     match x with
2548     | Arg2 (Arg0 Ln, Compose, _) -> true
2549     | _ -> false
2550   );
2551
2552   let pa = P (x,a) in
2553   let pb = P (x,b) in
2554   let pdb = ajuste_poly (derive pb) in
2555
2556   let res_z = ajuste_poly (poly_reconnait (resultant (minus_poly pa (mult_poly (P (x,[|ast_Z|])) pdb)) pb) ast_Z) in
2557
2558   print_ast_arbre res_z;
2559
2560   (*
2561     vérifier si la partie primitive de res_z est dans Q[x] p25 bronstein
2562     sinon n'est pas élémentaire
2563   *)
2564   match partie_primitive res_z with
2565   | Null | Notimplementederror -> Null
2566   | Res poly_z -> (
2567     let tab_facteur = poly_factorisation poly_z in
2568
2569     let part_log = ref ast_null in
2570     let id_alpha = ref 0 in
2571
2572     for i = 0 to Array.length tab_facteur - 1 do
2573       let p,x = match get_0 tab_facteur.(i) with | P (x,p) -> (p,x) | _ -> ([|],ast_null) in
2574       if deg_poly (get_0 tab_facteur.(i)) = 1
2575       then (
2576         let c = mult_constant (div_constant (get_c p.(0)) (get_c p.(1))) (c_const (-1) 1) in (* c = -c0 ou c0 racine*)
2577         let v = pgcd_poly (minus_poly pa (mult_poly_scal pdb c)) pb in
2578         part_log := Arg2 (!part_log, Plus, Arg2 (Arg0 (C c), Fois, Arg2 (Arg0 Ln, Compose, v)))
2579       )
2580     else (
2581       let c = {nom = "α_"; fonction = Arg0 Exp; d_fonction = Arg0 Ln; etat_derive = 0} in

```



```

2582         let v = pgcd_poly_algebrique (minus_poly pa (mult_poly pdb (Abstrait c))) pb in
2583
2584         print_ast_arbre (pgcd_poly (minus_poly pa (mult_poly_scal pdb (c_const (1) 1))) pb);
2585
2586         (* Lazard-Rioboo-Trager algorithm remove pgcd issue *)
2587         for i = 0 to deg_poly (get_0 tab_facteur.(i)) - 1 do
2588             id_alpha := !id_alpha + 1;
2589             part_log := Arg2 (!part_log, Plus,
2590                 Arg2 (Arg0 (C (E {nom = "α_"^(string_of_int !id_alpha); approx = 0.1})), Fois,
2591                     Arg2 (Arg0 Ln, Compose, subsitue_abs v (E {nom = "α_"^(string_of_int !id_alpha); approx = 0.1})))));
2592         done;
2593     )
2594
2595     done;
2596     Res !part_log
2597 )
2598 )
2599 | _ -> failwith "rosthein_trager_method : pas une fraction rationnelle ou degré incohérent"
2600 in
2601
2602 let r,p,l = hermite_method_log fonction in
2603
2604 let pi = if not (is_zero_ast p) then integration_polynome_log p else Res ast_null in
2605
2606 let pl = if not (is_zero_ast l) then rosthein_trager_method_log l else Res ast_null in
2607
2608 match pl,pi with
2609 | Res pll,Res pii -> Res (simplifie_ast (Arg2 (simplifie_ast pii, Plus, Arg2 (simplifie_ast r, Plus, simplifie_ast pll
2610 )))
2611 | Null,Null | Null,_ | _,Null -> Null
2612 | NotImplementederror,NotImplementederror | _,NotImplementederror | NotImplementederror,_ -> NotImplementederror
2613
2614 and exp_case fonction extension_list =
2615     (* pas implémenter peut être dans un futur proche *)
2616     NotImplementederror
2617 ;;
2618
2619
2620
2621
2622
2623 let ast_of_string str =

```

```

2624 (* input un string , lit caractère par caractère utilise les () pour chaque expression ex : ((X) + [1.01] / ((exp) o
      ((X) ^ [2/1])) *)
2625 simplifie_ast (parse (analyse_lexicale str))
2626 ;;
2627
2628
2629 let jolie_affichage () =
2630   Graphics.set_color (Graphics.rgb 203 195 177);
2631   Graphics.fill_rect 0 0 800 800;
2632   Graphics.set_color (Graphics.rgb 60 64 63);
2633   Graphics.fill_rect 0 0 50 800;
2634   Graphics.fill_rect 0 750 800 50;
2635   Graphics.fill_rect 0 0 800 50;
2636   Graphics.fill_rect 750 0 50 800;
2637   Graphics.fill_rect 0 690 800 25;
2638   Graphics.set_color Graphics.black;
2639 ;;
2640
2641
2642 let interface () =
2643   Graphics.open_graph " 800x800";
2644   Graphics.set_window_title "Algorithme de Risch";
2645   Graphics.moveto 75 725;
2646   jolie_affichage ();
2647   (* Graphics.set_text_size 10; ne fonctionne pas *)
2648
2649   let input_str = ref "" in
2650   let input_char = ref (Graphics.read_key ()) in
2651   let x = Graphics.current_x () in
2652
2653   while !input_char <> '!' do
2654     input_str := !input_str ^ (String.make 1 !input_char);
2655     Graphics.clear_graph ();
2656     jolie_affichage ();
2657     Graphics.draw_string !input_str;
2658     Graphics.moveto x (Graphics.current_y ());
2659     input_char := (Graphics.read_key ());
2660   done;
2661
2662   let fonction = ast_of_string !input_str in
2663
2664   print_ast_arbre fonction;
2665
2666   Graphics.moveto x (Graphics.current_y () - 50);

```

```

2667
2668   (match risch fonction (determiner_extension fonction) with
2669   | Res f -> (print_ast f; print_ast_arbre f; print_ast_arbre_graphics f)
2670   | _ -> (Printf.fprintf file "Pas de primitive elementaire\n\n\n"; Graphics.draw_string "Pas de primitive elementaire" ↵
2671   ));
2672
2673   Graphics.read_key ()
2674 ;;
2675
2676
2677
2678
2679
2680   (* ----- Test et assertions ----- *)
2681
2682
2683
2684   let mass_test nb =
2685
2686     let echec = ref 0 in
2687     let reussite = ref 0 in
2688
2689     let print_test ast =
2690
2691       print_ast ast;
2692
2693       match risch ast (determiner_extension ast) with
2694       | Res f -> (
2695         print_ast_arbre f;
2696         if egal_ast ((frac_reconnait (derive f) (Arg0 X))) ast
2697         then (reussite := !reussite + 1; Printf.fprintf file "\n\n Réussite total ! \n\n ")
2698         else Printf.fprintf file "\n\n Echec \n\n"
2699       )
2700       | _ -> Printf.fprintf file "\n\n Echec \n\n"
2701
2702     in
2703
2704     let randint a b = (Random.int (b+1-a)) + a in
2705
2706     let retest = Array.make nb ast_null in
2707
2708     for i = 0 to nb-1 do
2709

```

```

2710   let n = randint 2 6 in
2711   let m = randint 2 5 in
2712   let pa = Array.make n ast_null in
2713   let pb = Array.make m ast_null in
2714   for j = 0 to n - 1 do
2715     pa.(j) <- simplifie_ast (ast_const (randint 1 20) (randint 1 7));
2716   done;
2717   for j = 0 to m - 2 do
2718     pb.(j) <- simplifie_ast (ast_const (randint 1 10) (randint 1 1));
2719   done;
2720   pb.(m-1) <- (ast_const 1 1);
2721
2722   assert (not (is_zero_ast (P (Arg0 X, pa))));
2723   assert (not (is_zero_ast (P (Arg0 X, pb))));
2724
2725   let frac = F (Arg0 X, pa, pb) in
2726
2727   retest.(i) <- frac;
2728
2729   print_ast_arbre frac;
2730
2731   try print_test frac with
2732     Int_overflow -> echec := !echec + 1; retest.(i) <- ast_null
2733
2734 done;
2735
2736 print_int nb;
2737 print_newline ();
2738 print_int !echec;
2739 print_newline ();
2740 print_int !reussite;
2741 print_newline ();
2742
2743 for i = 0 to nb-1 do
2744   print_debug i;
2745   if not (is_zero_ast retest.(i)) then print_test retest.(i);
2746   print_debug i;
2747 done;
2748 ;;
2749
2750
2751 let print_stat () =
2752   print_debug appel_tab.(0);
2753   print_debug appel_tab.(1);

```

```
2754     print_debug appel_tab.(2)
2755 ;;
2756
2757
2758
2759
2760
2761  (*
2762     mass_test 1000;;
2763     Printf.fprintf file "\n\n\n\n";;
2764  *)
2765
2766
2767
2768  interface ();;
2769
2770
2771
2772  print_stat ();;
2773
2774
2775
2776
2777
2778  (* ----- FIN DU FICHIER ----- *)
2779
2780
2781  close_out file;;
2782
```