L'INSTITUT NATIONAL DES POSTES ET TELECOMMUNICATIONS

Traitement des problèmes basés sur le traitement des expressions arithmétiques

Préparation à l'ACM 2009

BOUZKRAOUI Hicham
YAZIDI Abdellah
ZARHRI Badr

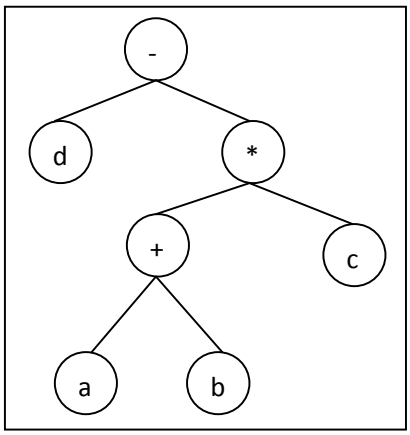
03 Novembre 2009

Il s'agit de la mise en place d'un algorithme général de traitement des expressions arithmétiques, ainsi que son application aux différents problèmes posés lors des concours de programmation.

Sommaire

1. Introduction (théorique et peut être sautée) :

Les expressions arithmétiques contiennent généralement un nombre parenthèses imbriqués, ce qui nous pousse vers les traitements récursifs. Mais puisqu'on ne traite qu'une ligne à la fois, on n'a besoin ni de programmation dynamique ni rien du tout. De la simple récursivité fait l'affaire. Et parmi les traitements récursifs, il y a la représentation par arbres. Mais un arbre pour s'exécuter nécessite une pile. Voici à quoi va ressembler un arbre binaire représentant l'expression « d-(a+b)*c » :



On peut donc calculer l'expression en deux étapes : d'abord, on construit l'arbre, et ensuite on utilise une pile pour effectuer un parcours en profondeur. Mais c'est trop encombrant en code, et par conséquent en temps et en erreurs.

L'idée est d'utiliser la représentation sous format d'arbre sans utiliser l'arbre.
L'astuce le plus utilisé lorsqu'il s'agit de traiter des problèmes s'agissant des arbres binaires, est d'essayer de repérer les sous arbres gauche et droite.

Si on applique ce raisonnement dans notre exemple, quels sont ces sous arbres ?

Ce sont tout simplement les termes de l'expression, c.à.d. « d » et « (a+b)*c » sont les sous arbres de la racine. Et de même, les sous arbres du sous arbre droit sont « (a+b) » et « c », et ainsi de suite.

Maintenant qu'on possède cette simple et efficace remarque, on n'a plus besoin de construire l'arbre. Et même si on essaie de le construire, on va être obligés d'utiliser cette astuce. **Alors pourquoi construire un arbre si l'ordre de sa construction est le même que celui de son utilisation?** Il suffit de n'en construire que ce dont on a besoin à chaque moment.

Quand à la pile pour exécuter un parcours en profondeur : la pile ainsi que le code pour gérer le parcours peut être réalisé par une fonction récursive de moins de trois lignes.

C'était une brève introduction théorique à ce qui suit. Dans le reste de ce document on va être plus pratiques en essayant de résoudre des problèmes de difficulté croissante à chaque fois afin d'aboutir au milieu du document à un algorithme vous permettant de traiter les expressions les plus générales. Ensuite vous allez être confrontés à des problèmes que vous n'osiez même pas lire auparavant et qui sont facilement solvables en utilisant le même algorithme.
Bonne lecture …

2. Construire son algorithme :

1) L'opérateur '+' :

Tout d'abord essayons de trouver un algorithme qui calcule le résultat d'une expression arithmétique contenant seulement l'opérateur '+' et des nombres ?

A ce niveau vous devriez laisser ce document et chercher à trouver plusieurs manières de résoudre ce problème.

Voici ce qui vous est passé par la tête :

- On peut parcourir l'expression (qui est stockée dans une chaine de caractères), et chaque fois qu'on trouve le symbole '+' augmente une variable qui contient le résultat par le nombre qui le précède.
  Une implémentation rapide en java de cette idée est l'utilisation de la méthode « split() », de la classe « String », ou bien l'utilisation d'un « StringTokenizer » pour séparer les différents termes, et après on les parcours pour les sommer.
- Mais essayons de penser récursivement, car comme on a dit dans l'introduction, les expressions possèdent une caractéristique de laquelle il faut profiter, c'est qu'ils n'ont pas un grand degré d'imbrication, car le pire des cas consiste en des lignes de 100 caractères ne contenant que des parenthèses imbriqués ce qui aboutit à une récursivité linéaire (c.à.d. un seul appel récursif est effectué à chaque étape, contrairement à une récursivité arborescente qui contient plusieurs appels dans la même étape) donc un temps d'exécution raisonnable, et au maximum, 50 niveaux d'imbrication, donc la pile de la machine peut facilement le gérer.
  On peut donc voir l'expression « a+b+c+d+e » comme étant « a+(b+c+d+e) », et le même traitement appliqué à l'expression initiale sera appliqué à la sous expression entre parenthèses.

On va prendre la deuxième méthode car elle est facilement généralisable. Une implémentation en java de cette solution pourra ressembler à ceci :

```
long somme(String expr)

{

If(expr.equals(""))

return 0;

int i=expr.indexOf('+') ;

if(i<0)

return Long.parseLong(expr);
```

```
return Long.parseLong. Long.parseLong
(expr.subString(0,i))+somme(expr.subString(i+1));

}
```

On parcourt l'expression jusqu'à trouver le premier caractère '+', et on traite séparément les deux expressions à gauche et à droite de ce caractère (c.à.d. les sous arbres gauche et droite) : celle à gauche nécessite seulement un parsing puisqu'elle ne contient nécessairement aucun signe '+', donc c'est un nombre, et celle à droite est aussi une expression. On remarque aussi deux conditions d'arrêt : La première servira, si jamais l'utilisateur a fourni une chaine vide (pour des raisons d'optimisation on peut mettre ce teste dans la fonction « main », mais on n'optimisera pas grand-chose), le deuxième teste sera beaucoup plus utilisé, c'est la vraie condition d'arrêt de notre fonction.

Une autre remarque que vous avez nécessairement fait, est l'utilisation de « long » alors qu'on peut utiliser « int ». La réponse est que dans un concours de programmation on doit s'attendre à tout !

C'était la première partie. Elle est un peu facile car elle primordiale pour la compréhension du reste du document qui va devenir de plus en plus moins évident. J'espère que vous avez tout compris, sinon je dirai que vous n'avez pas lu l'introduction.

2) L’opérateur ‘-‘ :

L’étape suivante est d’essayer de calculer le résultat d’une expression qui ne contient que le signe ‘-‘ et des nombres. Au premier abord, cet algorithme peut vous sembler similaire au précédent, mais il faut faire attention à un petit détail qu’on va voir. Essayons les deux approches pour voir :

En ce moment aussi vous devez être entrain de chercher à implémenter les deux approches. Cela vous aidera à découvrir vous mêmes ce qui va venir.

- L’idée de parcourir la chaine de caractères de gauche à droite fonctionne si on n’oubli pas de tronquer le premier terme pour le mettre dans la variable résultat avant de commencer le traitement pour le reste de la chaine.
- Mais essayons l’approche récursive qui nous intéresse le plus. L’expression « a-b-c-d » n’est pas égale à « a-(b-c-d) », alors c’est quoi la solution ? On doit remédier à ce changement de signe qui va apparaitre chaque fois. C’est faisable si on utilise une récursivité croisée. Mais dans un concours on doit rester aussi simple et claire que possible. La solution est très simple il suffit d’écrire l’expression précédente sous la forme : « (a-b-c)-d ». Donc il suffit de chercher le dernier symbole ‘-‘ et non le premier et l’algorithme précédent est fonctionnel.

Voici une implémentation d’une méthode qui calcule le résultat d’une telle expression :

```
long calcul(String expr)
   {
       if(expr.equals(""))
           return 0;
       int i=expr.lastIndexOf('-');
       if(i<0)
           return Long.parseLong(expr);
       return calcul(expr.substring(0, i))-
Long.parseLong(expr.substring(i+1));
   }
```

Il suffit de remplacer l’appel à la méthode indexOf(), par lastIndexOf(), et le tour est joué. Bravo, vous y êtes.

Il faut remarquer que grâce au fait de la ressemblance des traitements, vous pourrez faire du copier-coller, ce qui vous fera gagner un peu de temps.

En ce moment vous devez vous demander à quoi sert cette approche récursive alors qu'on peut encore tout faire avec les boucles. Mais rassurez-vous, car à partir de maintenant la première approche ne peut plus rien contre la complexité des expressions à venir.

3) Combiner ‘+’ et ‘-‘ :

Avant de commencer la lecture de ce qui suit, vous devez avoir observé qu’une expression ne contenant que des signes de multiplication est traitée exactement comme celle du signe ‘+’, car « a*b*c*d » est égale à « a*(b*c*d) ». Et qu’une expression ne contenant que des signes de division est traitée de la même manière que celle ne contenant que des signes ‘-‘, car « a/b/c/d » est égale à « (a/b/c/)d ». Si vous avez déjà fait ces deux remarques, alors vous pouvez continuer la lecture, sinon vous devez relire depuis le traitement des expressions contenant le signe ‘+’.

Maintenant nous allons nous attaquer aux expressions qui contiennent plus d’un symbole, ce qui nécessite la gestion de la priorité entre les différents opérateurs. Mais avant de commencer vous devez vous familiariser avec un aspect important de la récursivité que vous allez observer à travers l’exemple suivant :

On désire tracer un sablier en utilisant uniquement des espaces et des étoiles. Pour faciliter l’implémentation, on va supposer qu’on a déjà une chaine de caractères qui contient 5 espaces concaténés avec 10 étoiles, et qu’on désire écrire une fonction qui prend en argument un entier n inférieur à 10 et qui dessine un sablier de largeur n. Si vous choisissez l’approche la moins coûteuse, vous allez aboutir au code suivant :

```
1 : static final String CTE= "     **********";
2 :    void sablier(int n)
3 :    {
4 :        if(n>0)
5 :        {
6 :            System.out.println(CTE.substring((n+1)/2, n+5));
7 :            sablier(n-2);
8 :            System.out.println(CTE.substring((n+1)/2, n+5));
9 :        }
10 :    }
```

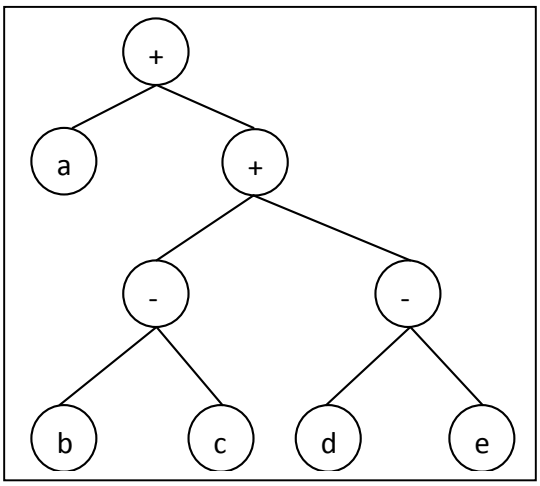
Si on effectue un appel à la méthode sablier(), avec un argument de 9 par exemple, on aura la sortie suivante :

```
*********
 *******
  *****
```

```
   ***
    *
    *
   ***
  *****
 *******
*********
```

Ce qui montre que le deuxième appel à println() (ligne 8), n'est effectué qu'après que le premier appel à println() (ligne 6), soit effectué pour tous les appels récursifs. En d'autres mots, aucun appel à la ligne 8 n'est effectué avant un appel à la ligne 6.

On va donc essayer de profiter de cette remarque pour gérer la priorité entre les différents opérateurs. Voici une explication dans une expression contenant les signes '+' et '-' en même temps : « a+b-c+d-e ».



On remarque que les signes '+' sont en haut de l'arbre, c'est-à-dire qu'on ne commence à traiter les signes '-' qu'après avoir traité tous les signes '+'. Et c'est exactement le comportement qu'on a observé dans l'exemple du sablier.

Essayons maintenant d'implémenter un algorithme qui calcule le résultat d'une telle expression :

```
long plusMoins(String expr)
   {
       if(expr.equals(""))
           return 0;
```

```
        //traitement pour tous les '+' qu'il y a d'abord

        int i=expr.indexOf('+');

        if(i>0)

            return plusMoins(expr.substring(0,
i))+plusMoins(expr.substring(i+1));



        //ensuite, on traite tous les '-'

        i=expr.lastIndexOf('-');

        if(i>0)

            return plusMoins(expr.substring(0, i))-
plusMoins(expr.substring(i+1));



        //si on arrive ici c'est que l'expression ne contient aucun signe + ni
-

        return Long.parseLong(expr);

    }
```

On n'a fait que répéter le traitement pour les deux expressions précédentes dans la même expression. Il y a aussi la remarque qu'on a commencé par le signe '+', pourquoi ne pas commencer par le '-' ? car comme on a dit plus haut : les signes les moins prioritaires doivent se situer en haut de l'arbre. J'espère que c'est beaucoup plus clair maintenant, et que vous trouvez le traitement de telles expressions assez facile maintenant. Nous n'allons pas voir l'introduction des signes de multiplication et de division, car je suppose que vous pouvez le faire par vous-mêmes, il vous faut juste vous rappeler de l'ordre inverse de la priorité des opérateurs qui est « +-*/ ». Par la suite on voir l'introduction des parenthèses sur les expressions.

4) Traitement des parenthèses :

Avant de se lancer dans des solutions longues et complexes, vous devez savoir que pour traiter les parenthèses dans une expression, il suffit de connaître la réponse à la fameuse question : comment traiter l'expression « (a+b-d-d+e) » ?

Bien sûr que vous connaissez la réponse (ce qui vous manquait c'était la question), il faut enlever les parenthèses des extrémités. (Ceci affirme la théorie qui dit qu'il suffit de poser la bonne question).

Une autre question pertinente : Comment traiter une expression de la forme « a-(b+c)-d » ? Il faut considérer (a+b) comme un seul terme dans l'expression.

Si vous connaissez la réponse à ces deux questions, alors on peut avancer et essayer d'implémenter un algorithme qui traite les expressions contenant à la fois les signes '+' et '-' et aussi les parenthèses, à condition que vous connaissiez une manière pour trouver la parenthèse fermante d'une parenthèse ouvrante. Il y a plusieurs manières de le faire, donc on vous propose celle qu'on a utilisée pour qu'il n'y ait pas d'ambiguïté. La fonction suivante retourne l'indice de la parenthèse fermante d'une parenthèse ouvrante fournie en paramètre :

```
int fermante(String expr, int ouvrante)
  {
      int k=1;
      do
      {
          ouvrante++;
          if(expr.charAt(ouvrante)==')')
              k--;
          if(expr.charAt(ouvrante)=='(')
              k++;
      } while(k>0);
      return ouvrante;
  }
```

Si la première parenthèse qu'on trouve est fermante, alors c'est celle qu'on cherche. Si elle est ouvrante, alors la prochaine fermante lui revient et on doit en trouver une autre. Donc on doit

enregistrer le nombre de parenthèses qu'on doit fermer avant de trouver celle qui nous intéresse. Ce nombre est enregistré dans la variable k.

Voici enfin le code qui permet de traiter de telles expressions :

```
long traiter(String expr)
   {
       //ce teste servira aussi pour traiter les '-' du signe
       if(expr.equals(""))
           return 0;


       int i=0;


       //la condition d'arrêt pour les parenthèses (expr) -> expr
       if(expr.charAt(i)=='(')
       {
       int k=1;
           do{
           i++;
           if(expr.charAt(i)=='(')
               k++;
           if(expr.charAt(i)==')')
               k--;
          }while(k>0);
       if(i==expr.length()-1)
           return traiter(expr.substring(1,expr.length()-1));
       }
```

```
        //cette boucle traite le signe '+'
        //il suffit de la copier, coller, et modifier pour traiter les autres
signes '-', '*', '/'
        i=0;
        while(i<expr.length())
        {
            if(expr.charAt(i)=='+')
                return traiter(expr.substring(0,
i))+traiter(expr.substring(i+1));

            //on saute les parentheses
            if(expr.charAt(i)=='(')
            {
                int k=1;
                do{
                i++;
                if(expr.charAt(i)=='(')
                    k++;
                if(expr.charAt(i)==')')
                    k--;
                }while(k>0);
            }
            i++;
        }

        //cette boucle traite le signe '-'
        //c'est la même que pour '+', apart qu'il faut parcourir l'expression
depuis la fin
```

```
        i=expr.length()-1;
        while(i>0)
        {
            if(expr.charAt(i)=='-')
                return traiter(expr.substring(0, i))-
traiter(expr.substring(i+1));

            //on saute les parentheses
            if(expr.charAt(i)==')')
            {
                int k=1;
                do{
                i--;
                if(expr.charAt(i)==')')
                    k++;
                if(expr.charAt(i)=='(')
                    k--;
                }while(k>0);
            }
            i--;
        }

        //et le condition d'arrêt
        return Long.parseLong(expr);
    }
```

Ce code peut vous sembler long, complexe, et susceptible de produire des erreurs, mais pas du tout, car on n'écrit que le bout pour traiter le signe '+'. Pendant cette phase, il faut un maximum de concentration. Après, il s'agit seulement de copier et coller du code sûr et fonctionnel. Sauf si

on veut regrouper le code qui se répète dans le corps d’une fonction, ce qui nécessite une petite remarque : pour les signes ‘-‘,et ‘/’, on est obligés de commencer par la fin de la chaine, mais pour le ‘+’, et le ‘*’, on peut commencer par là où on veut.

5) Le code final :

Donc pourquoi ne pas commencer par la fin pour tous les symboles, et de cette manière, on n'aura qu'écrire notre propre fonction : « lastIndexOf() » qui se charge de sauter les parenthèses. Voici le même code que précédemment avec cette nouvelle fonction :

```
int lastIndexOf(String expr, char c)
   {
       int i=expr.length()-1;
       while(i>0)
       {
           if(expr.charAt(i)==c)
               return i;

           //on saute les parentheses
           if(expr.charAt(i)==')')
           {
               int k=1;
               do{
               i--;
               if(expr.charAt(i)==')')
                   k++;
               if(expr.charAt(i)=='(')
                   k--;
               }while(k>0);
           }
           i--;
       }
       return -1;
   }
```

La fonction de traitement devient très compacte et lisible :

```
double traiter2(String expr)
   {
        if(expr.equals(""))
            return 0;

        int i=lastIndexOf(expr, '+');
        if(i>0)
            return
traiter2(expr.substring(0,i))+traiter2(expr.substring(i+1));

        i=lastIndexOf(expr, '-');
        if(i>0)
            return traiter2(expr.substring(0,i))-
traiter2(expr.substring(i+1));


        i=lastIndexOf(expr, '*');
        if(i>0)
            return
traiter2(expr.substring(0,i))*traiter2(expr.substring(i+1));

        i=lastIndexOf(expr, '/');
        if(i>0)
            return
traiter2(expr.substring(0,i))/traiter2(expr.substring(i+1));

        //si on arrive ici, c'est que soit l'expression est contenue dans des
parenthèses,
        //soit c'est un nombre
```

```
        if(expr.startsWith("("))
            return traiter(expr.substring(1, expr.length()-1));

        return Double.parseDouble(expr);
    }
```

Il reste encore une dernière remarque à faire, c'est qu'on peut décider si une expression est mise entre parenthèses si on ne rencontre aucun des 4 opérateurs durant notre parcours.

Le dernier code contient le traitement des 4 opérateurs au cas où vous voudriez faire du copier-coller dans un concours où la documentation est autorisée.

Si vous êtes arrivés à lire cette ligne, c'est que vous avez appris une nouvelle technique et non un algorithme. Il faut apprendre à adapter ce raisonnement aux différents problèmes et situations que vous allez rencontrer, c'est le but de la partie suivante du document.

3. Exemples d’utilisation :

Dans cette partie, on va essayer d’appliquer ce raisonnement pour différents problèmes qui, à première vue, chacun semble nécessiter un traitement différent. On va voir à quel point la méthode d’implémentation est décisive dans la faisabilité ou pas de la solution, et on va aboutir à la conclusion qu’**il faut optimiser autant qu’on peut lors du choix de l’implémentation**.

a) Problème de dérivation :

Ce problème a été proposé lors du concours régional du sud-est d'Europe en Octobre 2000, en Romanie.

Il consiste en la dérivation des fonctions dont les expressions contenant les 4 opérateurs, une variable « x », et la fonction « ln », arbitrairement imbriqués comme par exemple la formule :
« (2*ln(x+1.7*ln(x*x))-x*x)/((-7)+3.2*x*x)+(x+3*x)*x »

Pour commencer, on sait que les règles de dérivation sont comme suit :
(a + b)' = a' + b'
(a - b)' = a' - b'
(a * b)' = (a' * b + a * b')
(a / b)' = (a' * b - a * b') / b^2
ln(a)' = (a')/(a)

Il ne reste donc qu'utiliser ces règles au lieu de ceux qu'on connait pour les 4 opérateurs.

Voici à quoi ressemble notre solution :

```
String traiter(String expr)

   {

       if(expr.equals(""))

           return "";


       int i=lastIndexOf(expr, '+');

       if(i>0)

           return
traiter(expr.substring(0,i))+"+"+traiter(expr.substring(i+1));


       i=lastIndexOf(expr, '-');

       if(i>0)

           return traiter(expr.substring(0,i))+"-
"+traiter(expr.substring(i+1));
```

```
        i=lastIndexOf(expr, '*');

        if(i>0)

            return
"("+traiter(expr.substring(0,i))+"*"+expr.substring(i+1)+"+"+expr.substring(0
,i)+"*"+traiter(expr.substring(i+1))+")";




        i=lastIndexOf(expr, '/');

        if(i>0)

            return "("+traiter(expr.substring(0,i))+"*"+expr.substring(i+1)+"-
"+expr.substring(0,i)+"*"+traiter(expr.substring(i+1))+")/"+expr.substring(i+
1)+"^2";




        //si on arrive ici, c'est que soit l'expression est contenue dans des
parenthèses,

        //soit c'est un nombre, soit c'est x, soit c'est ln()


        //ici, on peut ajouter autant de fonctions qu'on veut

        if(expr.startsWith("ln"))

            return traiter(expr.substring(2))+"/"+expr.substring(2);


        if(expr.equals("x"))

            return "1";


if(expr.startsWith("("))

            return "("+traiter(expr.substring(1, expr.length()-1))+")";


        return "0";

    }
```

Comme on peut voir, on a l’impression de ne faire que du copier-coller, mais ce n’est pas tout à fait le cas, cas la plus grande partie du travail a été faite avant le codage.

b) Résolution d'une équation :

Ce problème a été posé dans le concours local de l'université d'ULM en 1997/98.

Il consiste en la résolution d'une équation de premier degré de la forme :
« (42-6*7)*x+9=3*x+1+2+3 »

A premier abord, ce problème peut sembler difficile, mais comme d'habitude il suffit de connaître deux astuces :

Le premier, est qu'on doit qu'on doit diviser l'expression en deux au niveau du signe '=', et de calculer chaque partie à part, ensuite il suffit de déduire la valeur de x à partir de ces deux résultats. Cette astuce est assez triviale, mais le problème est plutôt celui de traiter ces parties. Pour résoudre ce problème, on va essayer de répondre à une question, qui est aussi moins évidente que sa réponse : Que va retourner la fonction qui traitera ces parties?

La réponse à cette question constitue la deuxième astuce qui manquait pour entamer l'implémentation de la solution : Elle retournera deux nombres, Qui représentent les coefficients d'un polynôme de 1er degré.

Afin d'être aussi modulaire qu'on peut, on va donc tout d'abord définir la classe que va retourner notre méthode :

class Polynome //ax+b

```
{
    double a,b;

        public Polynome(double a, double b) {
            this.a = a;
            this.b = b;
        }
    Polynome add(Polynome p)
    {
        a+=p.a;
        b+=p.b;
        return this;
    }
    Polynome mul(Polynome p)
    {
        a=p.a*b+a*p.b;
        b*=p.b;
        return this;
    }
    Polynome sub(Polynome p)
    {
        a-=p.a;
        b-=p.b;
        return this;
    }
```

```
}
```

La rédaction de ce bout de code ne demande pas beaucoup de concentration, ni de déflection. Vous avez intérêt à vous habituer à ce type de programmation car avec la programmation procédurale on ne peut pas aller bien loin en seulement 5 heures.

Passons maintenant à notre fameuse méthode « traiter() », que vous avez sûrement déjà deviné :

Polynome traiter(String expr)

```
   {
        if(expr.equals(""))
            return new Polynome(0, 0);

        int i=lastIndexOf(expr, '+');
        if(i>0)
            return
traiter(expr.substring(0,i)).add(traiter(expr.substring(i+1)));

        i=lastIndexOf(expr, '-');
        if(i>0)
            return
traiter(expr.substring(0,i)).sub(traiter(expr.substring(i+1)));


        i=lastIndexOf(expr, '*');
        if(i>0)
            return
traiter(expr.substring(0,i)).mul(traiter(expr.substring(i+1)));

        if(expr.startsWith("("))
            return traiter(expr.substring(1, expr.length()-1));

        if(expr.equals("x"))
            return new Polynome(1, 0);

        return new Polynome(0, Double.parseDouble(expr));
   }
```

Si on divise l'expression en deux parties comme on a dit, et on les stocke dans les chaines « exp1 » et « exp2 », alors le polynôme :

p=traiter(exp1).sub(traiter(exp2)) est la forme « a*x+b=0 » de l'équation initiale.