TP4 : Calculatrice

Grammaire

Le système est composé d'une calculatrice capable de prendre en compte les opérateurs binaires classiques (plus, moins, multiplier, diviser), les opérateurs unaires plus et moins. Dans le cadre de ce TP, nous ne considérerons que les constantes. (Exemple : $2+3*-4$) Voici la grammaire d'une expression arithmétique :

```
E        →  E '+' E_MUL
          |  E '-' E_MUL
          |  E_MUL
E_MUL    →  E_MUL '*' E_UNARY
          |  E_MUL '/' E_UNARY
          |  E_UNARY
E_UNARY  →  '+' E_UNARY
          |  '-' E_UNARY
          |  E_CST
E_CST    →  '(' E ')'
          |  constant
```

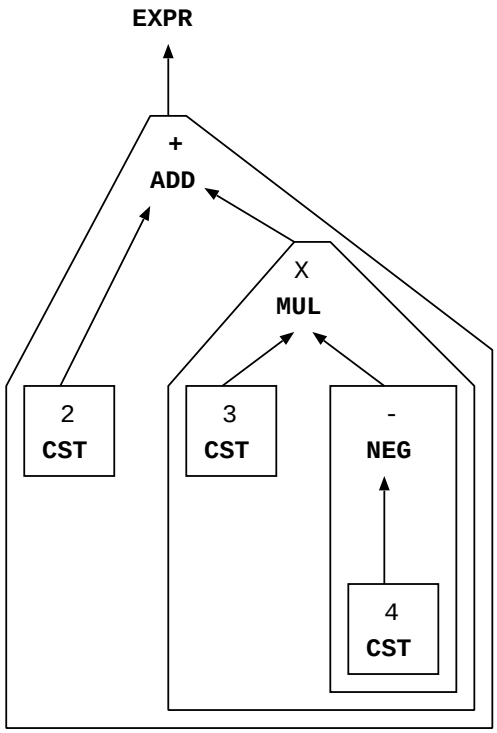
Ecrivez une classe PARSER avec une unique méthode publique ayant le profil suivant : `public static EXPR parse_on(String txt)`. Les autres méthodes seront aussi `static`, mais `private`. Dans cette classe, un ensemble de méthodes (une par règle) doit permettre de décripter caractère par caractère votre expression pour produire l'arbre syntaxique correspondant.

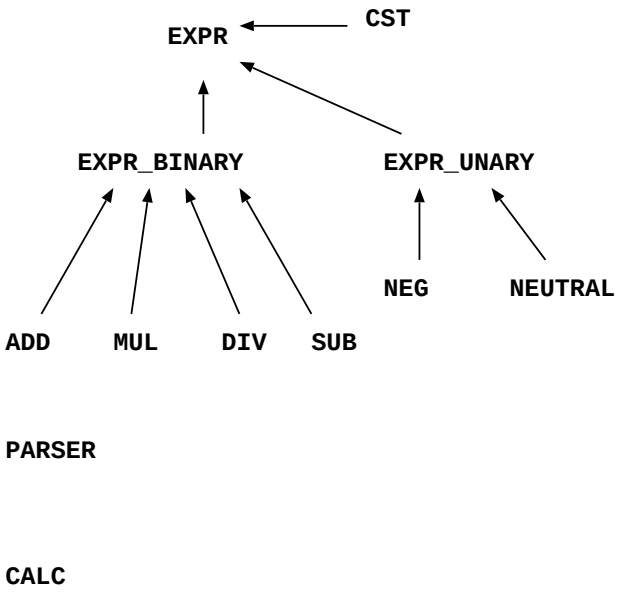
Remarque : Attention à la récurssivité gauche ! Par exemple, si vous prenez la grammaire tel quel, la régle `E` commence par `E` (récurssivité sans fin...). En analysant un peu mieux les régles, vous pouvez transformer la premier régle de la manière suivante : `E → E '+' E_MUL`
devient

`E → E_MUL { '+' E_MUL }`

Les { } indiquent une répétition de 0 à n fois.

Arbres et diagramme d'héritage





Définition des classes

```
class CST extends EXPR {
  int value;

  CST(int v) { ... }

  int eval()
  { return value; }
}
```

```
class ADD extends EXPR_BINARY {

  ADD(EXPR l,EXPR r) { ... }

  int eval()
  { return left.eval() + right.eval(); }
}
```

```
abstract class EXPR_BINARY extends EXPR {
  protected EXPR left;
  protected EXPR right;
}
```

```
abstract class EXPR {
  abstract int eval();
}
```

```
class CALC {
  static void main(String[] args)
  { EXPR e;
    e = PARSER.parse_on(args[0]);
    System.out.println("Resultat : "+e.eval());
  }
}
```

```
class PARSER {
  private static String src;
  private static int idx;

  private static char last_char;
  private static int last_cst;

  private static boolean read_char(char c)
  {
    if ((idx < src.length()) && (src.charAt(idx) == c)) {
      idx++;
      last_char = c;
      return true;
    }
    return false;
```

```
  }

  private static boolean read_cst()
  { // Indication : '4'-'0' == 4
    // Indication : 234 == (((2*10)+3)*10)+4
    ...
  }

  private static EXPR read_e()
  { EXPR result,right;
    char op;
    result = read_e_mul();
    if (result != null) {
      while ((read_char('+') || (read_char('-')) {
        op = last_char;
        right = read_e_mul();
        if (right == null) error();
        if (op == '+')
          result = new ADD(result,right);
        else
          result = new SUB(result,right);
      }
    }
    return result;
  }

  ... // Indication: Une fonction par régle de grammaire

  private static void error()
  { int j;
    System.out.println(src);
    for (j=0;j<idx;j++) System.out.print(' ');
    System.out.println('I');
    System.exit(1);
  }

  static EXPR parse_on(String txt)
  { EXPR e;
    src = txt;
    idx = 0;
    e = read_e();
    if ((e == null) || (idx < src.length())) error();
    return e;
  }
}
```