Université de Sherbrooke
Faculté de génie
Département de génie informatique

Rapport

Éléments de compilation

GIF340

Par:
Langevin, Clovis - Lanc0902
Gratton, Francis - Graf2102
Gasse, Bryan - Gasb3002

Présenté à:
L’équipe professorale

Remis le 23 juillet 2025

Table des matières

Table des figures

1 Analyseur lexical

1.1 Unité lexical nb

Expression régulière : $(0|1|2|3|4|5|6|7|8|9)^+$

Automate :
chiffre = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

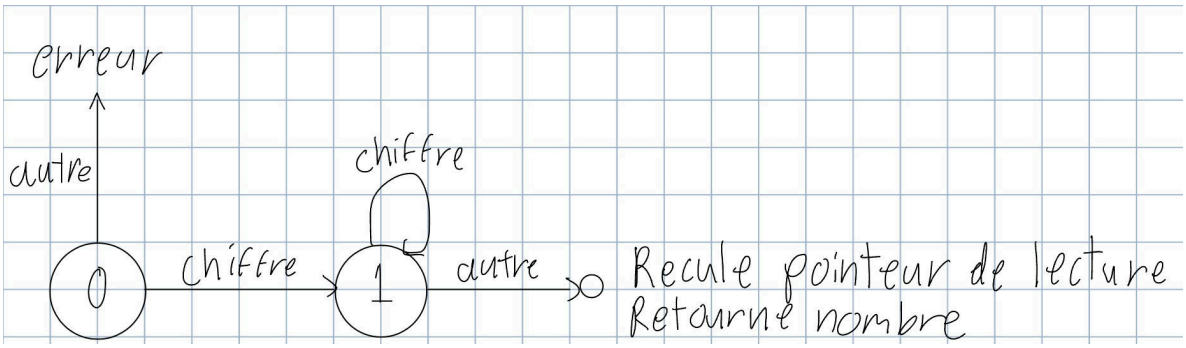


Fig. 1. – Chiffre

1.2 Unité lexical variable

Maj = (A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z)
min = (a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)

Expression régulière : $(\text{Maj})(\text{Maj}|\text{min})^*(\_((\text{Maj}|\text{min})^+))^*$

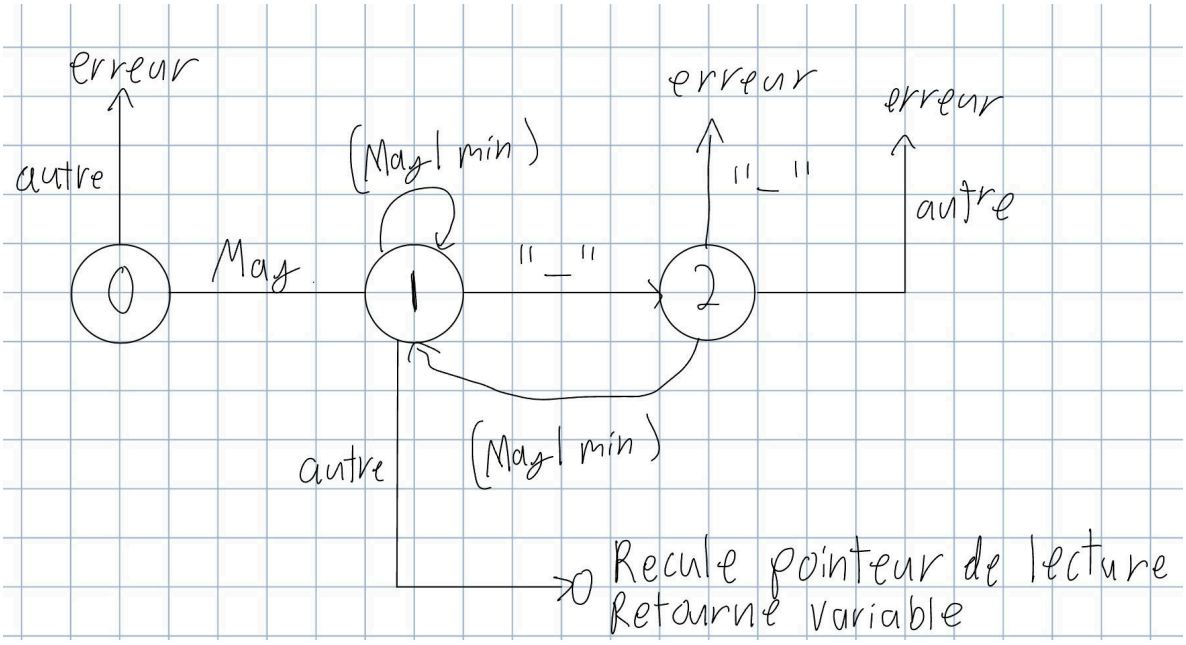


Fig. 2. – Automate Variable

1.3 Unité lexical délimiteur

Expression régulière : [(|)]

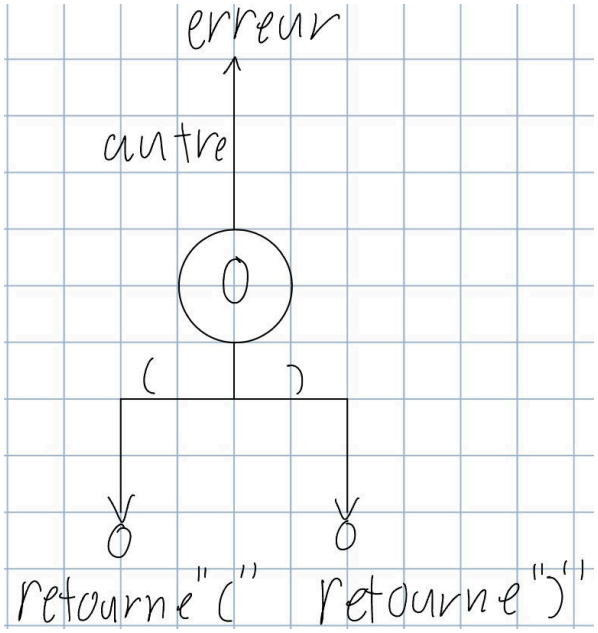


Fig. 3. – Automate Delimiteur

1.4 Unité lexical opérateur

Expression régulière : (+| − | ∗ |/)

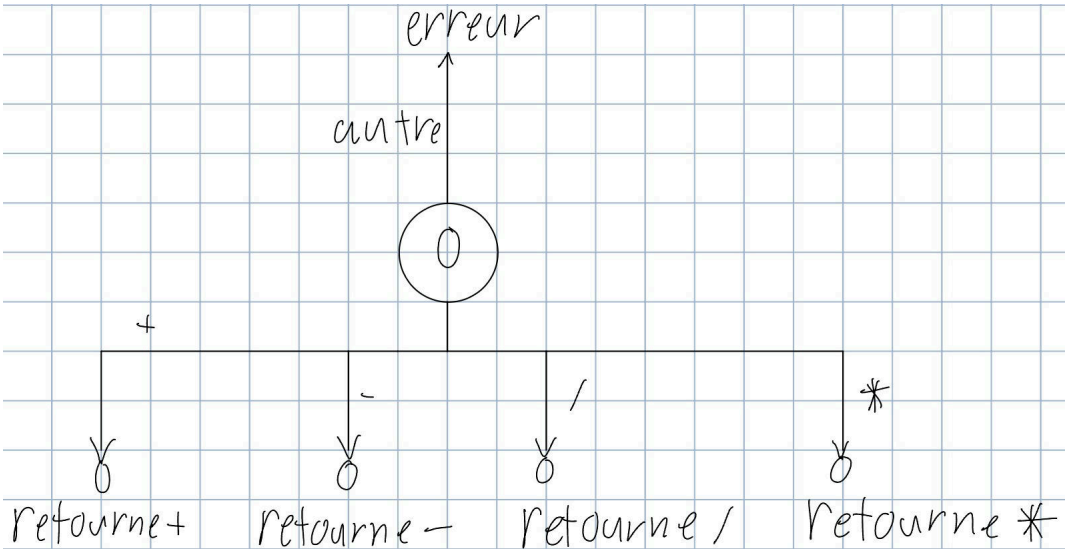


Fig. 4. – Automate Operateur

2 Structure de données d'arbres syntaxiques abstraits

2.1 ElemAST

Cette classe est la base de FeuilleAST et NoeudAST. Son principale intérêt est de permettre d'avoir une structure commune où les éléments peuvent s'interchanger entre eux. En effet, ElemAST l'utilisation d'un Terminal en faisant sa lecture ou son évaluation.

2.2 FeuilleST

Cette classe représente la fin d'une branche d'un arbre. C'est-à-dire qu'aucun ElemAST peut être attacher à cette classe pour agrandir l'arbre.

2.3 NoeudAST

Cette classe représente la division par deux d'une branche de l'arbre. Elle est simuler à partir de deux ElemAST en attribut, appelé respectivement « right » et « left ».

2.4 AnalLex

Cette classe permet d'analyser une expression pour la séparer en opérateur et en opérande. Pour avoir une expression valide, on vérifie si elle contient des erreurs de syntaxes. Nous analysons si l'expression à des variables qui commence par une minuscule ou qu'il y a deux tiret bas. Si oui, le programme imprimer en console l'erreur observer et sort du programme.

```
87    public void ErreurLex(String s) {
88      if (s == null || s.isEmpty())
89        return;
90
91      Character c1 =  s.charAt(0);
92      Character c2 = null;
93      boolean error = false;
94
95      if (Character.isLetter(c1) && Character.isLowerCase(c1)) {
96        System.out.println("Erreur le premier charactere ne peut pas etre miniscule: \n" + s);
97        System.exit(1);
98      }
99
100     for (int i = 0; i < s.length(); i++) {
101       c1 = s.charAt(i);
102       if (c2 != null) {
103         if (c1 == '_' && c2 == '_') {
104           System.out.println("Erreur vous avez pas le droit d'avoir une variable avec 2 __ de suite: \n" + s);
105           System.exit(1);
106         }
107       }
108       c2 =  c1;
109     }
110     if (s.charAt(s.length() - 1) == '_'){
111       System.out.println("Erreur vous avez pas le droit d'avoir un _ a la fin du nom de variable: \n" + s);
112       System.exit(1);
113     }
114
115   }
```

Fig. 5. – Gestion d'erreur d'AnalLex

2.5 DescenteRecursive

Cette classe permet de construire un arbre syntaxique abstrait à partir d'une expression arithmétique. La construction se fait de manière récursive en analysant premièrement les expressions (+,-), puis deuxièmement les termes (*, /) et finalement les facteurs (5, variable, (,)). L'expression est analyser pour vérifier s'il n'y a pas d'expressions et ou des termes suivant le premier. De plus, une vérification est fait pour observer si chaque parenthèse ouvrante n'est pas fermé par une parenthèse fermante. Ces situations mentionnés donne une erreur mentionné à la console et sort du programme.

```
39    // Methode pour chaque symbole non-terminal de la grammaire retenue
40    private Terminal partieE() {
41      Terminal currentTerminal = readNext(lexical); // T
42      ElemAST right = partieT(currentTerminal);
43
44      Terminal op = readNext(lexical); // try to read + or -
45      ErreurSynt(currentTerminal, op);
46
47      while (op != null && (op.chaine.equals("+") || op.chaine.equals("-"))) {
48        Terminal t2 = readNext(lexical); // next value
49        ElemAST left = partieT(t2);
50
51        NoeudAST node = new NoeudAST(op.chaine);
52        node.elemASTLeft = left;
53        node.elemASTRight = right;
54        right = node;
55
56        op = readNext(lexical);
57      }
58
59      // op is not + or -, bring back pointer
60      lexical.pushBack(op);
61
62      racine = right;
63      return op;
64    }
```

Fig. 6. – Analyse d'expressions de DescenteRecursive

```
65
66
67    private ElemAST partieT(Terminal first) {
68      ElemAST right = partieF(first);
69
70      Terminal op = readNext(lexical); // read possible * or /
71      ErreurSynt(first, op);
72
73      while (op != null && (op.chaine.equals("*") || op.chaine.equals("/"))) {
74        Terminal next = readNext(lexical); // next operand
75        ElemAST left = partieF(next);
76
77        NoeudAST node = new NoeudAST(op.chaine);
78        node.elemASTLeft = left;
79        node.elemASTRight = right;
80        right = node;
81
82        op = readNext(lexical); // read another * or /
83      }
84
85      // op is not * or /, go back to partieE
86      lexical.pushBack(op);
87
88      return right;
89    }
90
```

Fig. 7. – Analyse de termes de DescenteRecursive

```
92      private ElemAST partieF(Terminal current) {
93        if (current.chaine.equals(")")) {
94          partieE(); // parse inside the parentheses
95          Terminal closing = readNext(lexical); // consume ')'
96          if (!closing.chaine.equals("(")) {
97            ErreurSynt("Il manque une fermeture de parenthese");
98          }
99          return racine;
100       } else if (Character.isLetterOrDigit(current.chaine.charAt(0))) {
101         return new FeuilleAST(current.chaine); // id
102       }
103
104       return null;
105     }
106
```

Fig. 8. – Analyse de facteurs de DescenteRecursive

```
115   /** ErreurSynt() envoie un message d'erreur syntaxique
116    */
117    public void ErreurSynt(String s)
118    {
119        System.out.println("Erreur : " + s);
120        System.exit(1);
121    }
122
123    public void ErreurSynt(Terminal current, Terminal nextOne)
124    {
125      if( current.chaine.isEmpty() || nextOne.chaine.isEmpty() )
126        return;
127
128      char c = current.chaine.charAt(0);
129      char c2 = nextOne.chaine.charAt(0);
130
131      if (OPERATORS_NO_P.contains(c) && OPERATORS_NO_P.contains(c2)) {
132        System.out.println("Erreur de syntaxe, plusieurs operateur on ete mis un apres l'autre: " + current.chaine);
133        System.exit(1);
134      }
135    }
```

Fig. 9. – Gestion d'erreur de DescenteRecursive

3 Analyseur syntaxique par la méthode descendante

3.1 Fonctionnement d'un analyseur descendant

Pour utiliser un analyseur, une grammaire est nécessaire. Celle-ci est définie par les éléments suivants : Vt (terminaux), Vn (non-terminaux), S (symbole de départ), P (ensemble de règles de production) et p (phrase à analyser).

L'analyseur fonctionne de la manière suivante : à partir du symbole initial SS, les règles de production P sont appliquées pour tenter de générer la phrase p. Si aucune séquence d'applications des règles ne permet d'obtenir p, cela signifie que la phrase ne respecte pas la grammaire.

3.2 Fonctionnement d'un analyseur LL

Le fonctionnement d'un analyseur *LL* est similaire à celui d'un analyseur descendant, mais avec certaines restrictions. Premièrement, la lecture de la phrase *p* se fait toujours de gauche à droite. Deuxièmement, les règles de production *P* sont systématiquement appliquées sur le symbole le plus à gauche, jusqu'à l'obtention de la phrase attendue.

3.3 Analyseur *LL* le plus utilisé

L'analyseur *LL(1)* est un bon compromis entre simplicité d'implémentation et possibilité de grammaires applicable.

3.4 Grammaire utilisé pour *LL(1)*

La grammaire *LL(1)* est une grammaire qui est applicable par le parser *LL(1)* et donc elle sera utilisé pour l'analyseur *LL(1)*.

3.5 Syntaxe d'expression arithmétique de la grammaire utilisé

$$V_t = \text{id}, +, -, *, /, (,)$$

$$V_n = E, T, F$$

$$S = E$$

$$P = \{$$

$$E \rightarrow T[+E|-E]$$

$$T \rightarrow F[*\,T|/T]$$

$$F \rightarrow (E)$$

$$F \rightarrow \text{id}$$

$$\}$$

3.6 Exemple d'application du LL(1) avec la grammaire créer au 3.5

Ex.1

$p = 1 + 2 * 3$

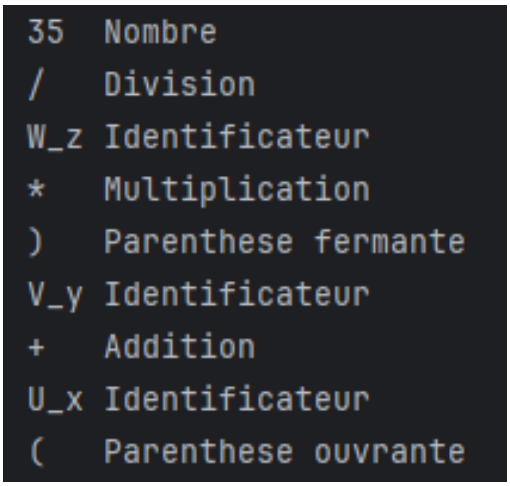
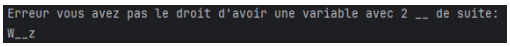
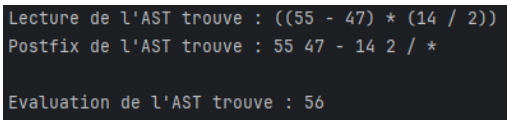
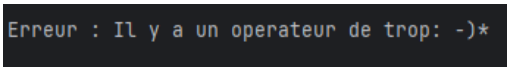
$$
\begin{aligned}
&E\\
&T + E\\
&F + E\\
&\mathrm{id} + E\\
&\mathrm{id} + T\\
&\mathrm{id} + F * T\\
&\mathrm{id} + \mathrm{id} * T\\
&\mathrm{id} + \mathrm{id} * F\\
&\mathrm{id} + \mathrm{id} * \mathrm{id}
\end{aligned}
$$

Ex.2

$p = 3 - 4 + 1/5$

$$
\begin{aligned}
&E\\
&T - E\\
&F - E\\
&\mathrm{id} - E\\
&\mathrm{id} - T + E\\
&\mathrm{id} - F + E\\
&\mathrm{id} - \mathrm{id} + E\\
&\mathrm{id} - \mathrm{id} + T\\
&\mathrm{id} - \mathrm{id} + F / T\\
&\mathrm{id} - \mathrm{id} + \mathrm{id} / T\\
&\mathrm{id} - \mathrm{id} + \mathrm{id} / F\\
&\mathrm{id} - \mathrm{id} + \mathrm{id} / \mathrm{id}
\end{aligned}
$$

4 Test de validation

test	Résultats attendus	Résultats obtenu
Expression arithmétique sans erreur lexicale (U_x+V_y)*W_z/35	35 Nombre / Division W_z Identificateur * Multiplication) Parenthèse fermante V_y Identificateur + Addition U_x Identificateur (Parenthèse ouvrante	
Expression arithmétique avec erreur lexicale (U_x+V_y)*W__z/35	Avoir un message d'erreur qui dit le type d'erreur	
Expression arithmétique (55-47)*14/2	Lecture AST: ((55-47)*(14/2)) Postfix: 55 47 - 14 2 / * Évaluation: 56	
Expression arithmétique avec erreur (U_x-)*W_z/35	Erreur il y a trop d'opérateur un après l'autre	

5 Annexe

5.1 ElemAST

```
public abstract class ElemAST {  13 usages  2 inheritors  Franky55 *

  /** Evaluation d'AST
   */
  public abstract int EvalAST();  9 usages  2 implementations  Franky55


  /** Lecture d'AST
   */
  public abstract String LectAST();  3 usages  2 implementations  Franky55

  public abstract String LectAST(String prefix);  3 usages  2 implementations  Franky55

  public abstract String ASTPostfix();  3 usages  2 implementations  Franky55

/** ErreurEvalAST() envoie un message d'erreur lors de la construction d'AST
 */
  public void ErreurEvalAST(String s) {   no usages  Franky55
    System.out.println("Erreur element: " + s);
  }

}
```

Fig. 10. – ElemAST

5.2 FeuilleAST

```java
public class FeuilleAST extends ElemAST {  1 usage  Franky55 *

  // Attribut(s)
  private String expression;  5 usages

/**Constructeur pour l'initialisation d'attribut(s)
 */
  public FeuilleAST(String _expression) { expression = _expression; }

  /** Evaluation de feuille d'AST
   * May throw error
   */
  public int EvalAST( ) {  9 usages  Franky55
      return Integer.parseInt(expression);
  }

 /** Lecture de chaine de caracteres correspondant a la feuille d'AST
  */
  public String LectAST( ) {  3 usages  Franky55
      return expression;
  }

    public String ASTPostfix(){  3 usages  Franky55
        return expression;
    }

    public String LectAST(String prefix) {  3 usages  Franky55
        return prefix + expression + "\n";
    }
}
```

Fig. 11. – FeuilleAST

5.3 NoeudAST

```java
public class NoeudAST extends ElemAST {  4 usages  Franky55 *
  // Attributs
  private String expression;  5 usages
  public ElemAST elemASTLeft;  12 usages
  public ElemAST elemASTRight;  12 usages

  /** Constructeur pour l'initialisation d'attributs
   */
  public NoeudAST(String _expression) {  2 usages  Franky55
    expression = _expression;
    elemASTRight = null;
    elemASTLeft = null;
  }

  /** Evaluation de noeud d'AST
   */
  public int EvalAST( ) {  9 usages  Franky55

      return switch (expression) {
          case "+" -> elemASTLeft.EvalAST() + elemASTRight.EvalAST();
          case "-" -> elemASTLeft.EvalAST() - elemASTRight.EvalAST();
          case "*" -> elemASTLeft.EvalAST() * elemASTRight.EvalAST();
          case "/" -> elemASTLeft.EvalAST() / elemASTRight.EvalAST();
          default -> 0;
      };
  }

  public String LectAST( ) {  3 usages  Franky55
      String left = (elemASTLeft != null) ? elemASTLeft.LectAST() : "null";
      String right = (elemASTRight != null) ? elemASTRight.LectAST() : "null";
      return "(" + left + " " + expression + " " + right + ")";
  }

    public String ASTPostfix(){  3 usages  Franky55
        String left = (elemASTLeft != null) ? elemASTLeft.ASTPostfix() : "null";
        String right = (elemASTRight != null) ? elemASTRight.ASTPostfix() : "null";
        return left + " " + right + " " + expression;
    }

    public String LectAST(String prefix) {  3 usages  Franky55
        return prefix + expression + "\n" + elemASTLeft.LectAST( prefix: prefix + "\t") + elemASTRight.LectAST( prefix: prefix + "\t");
    }
```

Fig. 12. – NoeudAST