UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et génie informatique

RAPPORT

Éléments de compilation
APP5

Présenté à
Ahmed Khoumsi

Présenté par
Pascal-Emmanuel Lachance – lacp3102
Jacty Milena Saenz Rosales – saej3101

Sherbrooke – 18 juillet 2022

TABLE DES MATIÈRES

1. Analyseur lexical 2

1.1 Expressions régulières 2
1.2 Automates 2

2. Structures de données d'arbres syntaxiques abstraits 4

3. Analyseur syntaxique par la méthode descendante 5

3.1 Analyseur descendant et analyseur LL 5
3.2 Grammaire 6
3.3 Exemples de dérivation 6
3.4 Classe Java 7

4. Plan de tests 9

4.1 Analyse lexicale 9
4.2 Analyse syntaxique 10

LISTE DES FIGURES

Figure 1. Automate pour les opérateurs 2

Figure 2. Automate pour les identifiants 3

Figure 3. Automate pour les littéraux 3

Figure 4. Automate complet de l'analyseur lexical 3

Figure 5. Diagramme de classe de l'arbre syntaxique abstrait 4

Figure 6. Diagramme de classe complet de l'analyseur syntaxique et lexical 8

1. ANALYSEUR LEXICAL

1.1 EXPRESSIONS RÉGULIÈRES

Les unités lexicales sont définies par les expressions régulières suivantes :

Unité lexicale	Expression régulière
Littéraux	[0-9]+
Opérateurs	(++?)|[-/*]
Identifiants	[A-Z][A-Za-z|_A-Za-z]*

Les expressions régulières ont été divisées en trois catégories : des identifiants, des littéraux (nombres) et des opérateurs.

1.2 AUTOMATES

Les unités lexicales peuvent aussi être représentées par les automates de la Figure 1 pour les opérateurs, de la Figure 2 pour les identifiants et de la Figure 3 pour les littéraux. La Figure 4 représente les trois automates mises ensembles avec les cas d'erreur.

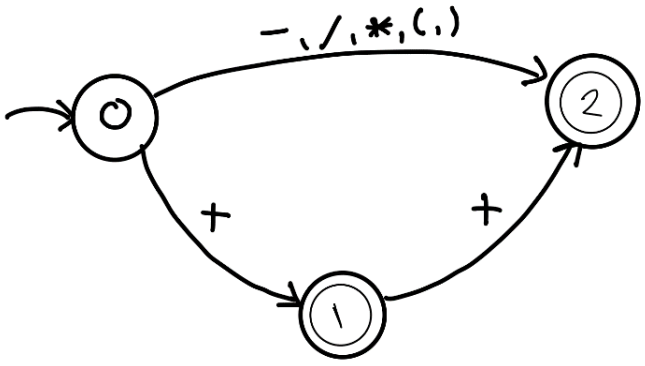

Figure 1. Automate pour les opérateurs

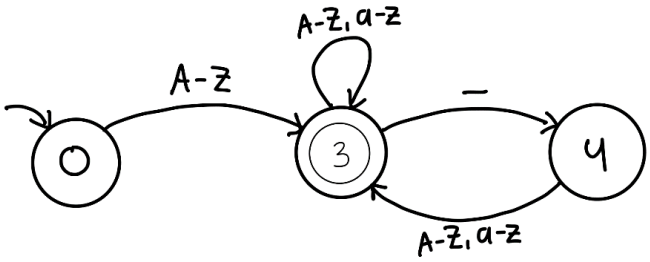

Figure 2. Automate pour les identifiants

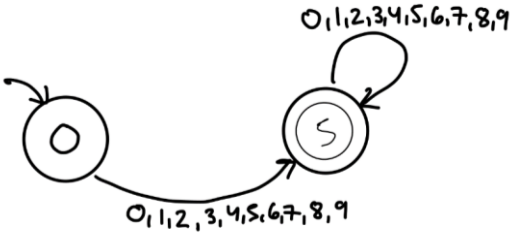

Figure 3. Automate pour les littéraux

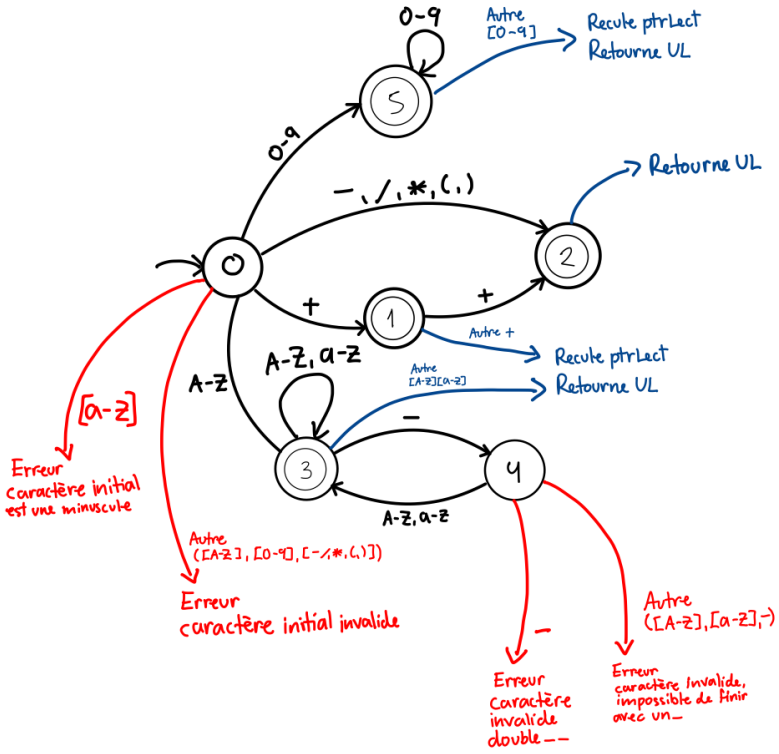

Figure 4. Automate complet de l'analyseur lexical

2. STRUCTURES DE DONNÉES D'ARBRES SYNTAXIQUES ABSTRAITS

Le diagramme de classe de la Figure 5 représente la structure de l'arbre syntaxique abstrait (AST) qui est utilisé lors de l'analyse syntaxique et lexicale. Un arbre est composé de nœuds et de feuilles. Dans le contexte de la problématique, les feuilles de l'AST contiennent un opérande soit un identifiant ou un littéral. Aussi, les nœuds de l'AST ont des opérations ce qui inclut la multiplication, l'addition, la division et la soustraction. Dans notre cas, il y a aussi l'addition postfixe (++).

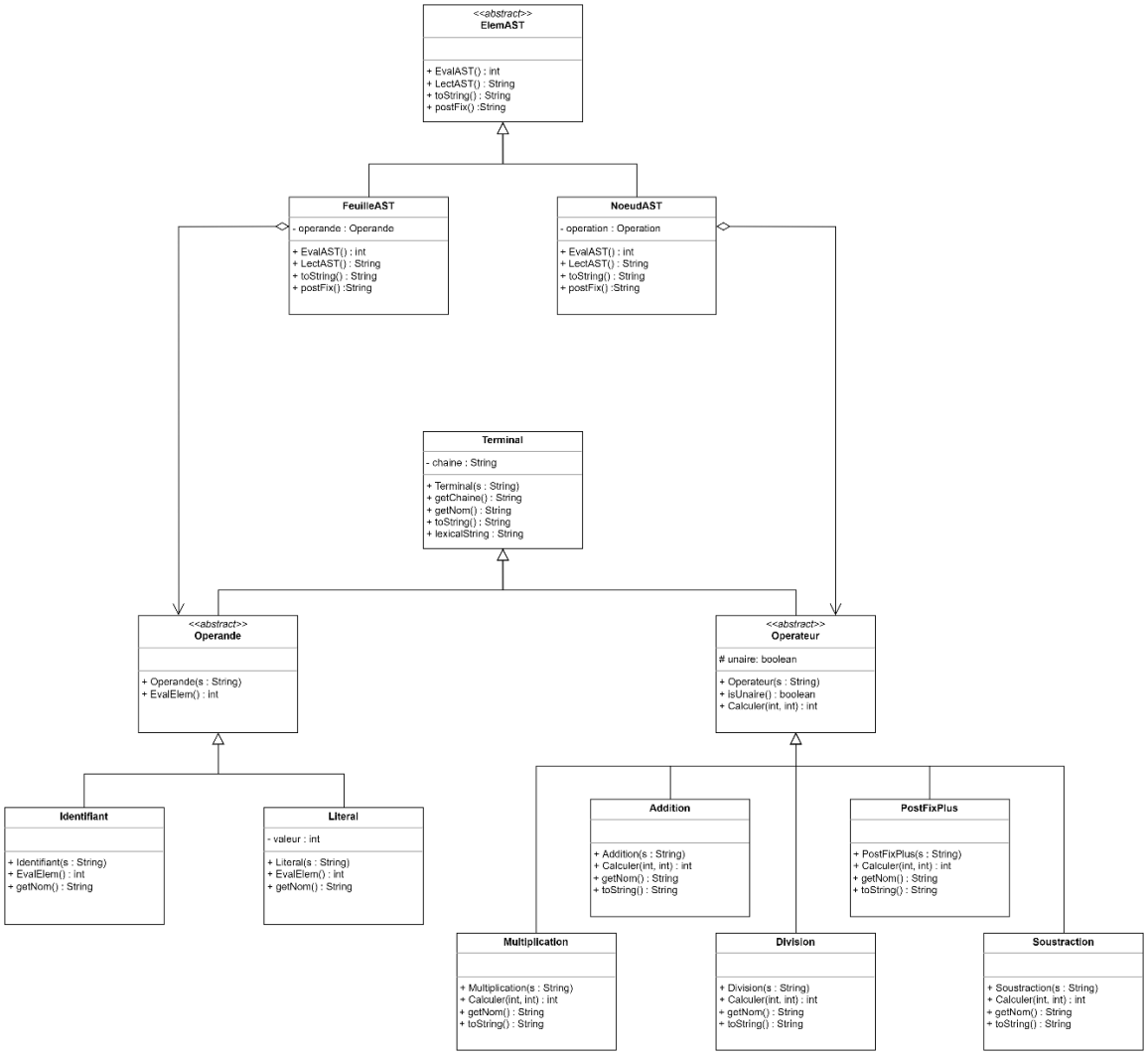

Figure 5. Diagramme de classe de l'arbre syntaxique abstrait

3. ANALYSEUR SYNTAXIQUE PAR LA MÉTHODE DESCENDANTE

3.1 ANALYSEUR DESCENDANT ET ANALYSEUR LL

Un analyseur descendant permet de construire un arbre syntaxique selon une grammaire, partant d'un premier symbole et appliquant des règles grammaticales pour arriver aux symboles terminaux. On part donc de la racine de l'arbre syntaxique, en descendant vers les feuilles. Il est important de mettre des contraintes sur la grammaire, pour s'assurer que les bons chemins de dérivation sont pris pour se rendre aux terminaux; on souhaite qu'un seul chemin existe pour créer l'arbre à partir d'une phrase syntaxiquement correcte.

Un analyseur LL est une forme d'analyseur descendant, qui cherche à prioriser l'évaluation de chaque terme jusqu'au bout avant de passer au suivant. La phrase est lue de gauche à droite, et la séquence de dérivation se fait également à gauche, jusqu'à l'atteinte d'une unité lexicale. Les parsers LL peuvent suivre plusieurs approches, selon le nombre *k* de prochains terminaux connus. Nous avons utilisé l'approche LL(1), un bon compromis, qui est conscient d'une unité lexicale à l'avance, qui s'applique particulièrement aux règles de notre grammaire régulière.

Un concept important pour la réalisation d'un algorithme LL(1) sont ceux du *premier* et du *suivant*, qui représentent respectivement la liste des symboles terminaux pouvant débuter une chaîne de dérivation, ou en suivant une.

3.2 GRAMMAIRE

Notre grammaire peut être définie des deux façons synonymes qui suivent :

$$x = \{+, -\}\ et\ y = \{*, /\}$$

$$E \to T[xE]$$

$$T \to F[yT]$$

$$F \to U[++]$$

$$U \to operande \mid (E)$$

$$E \underset{1}{\to} T$$

$$E \underset{2}{\to} T\ x\ E$$

$$T \underset{3}{\to} F$$

$$T \underset{4}{\to} F\ y\ T$$

$$F \underset{5}{\to} U$$

$$F \underset{6}{\to} U++$$

$$U \underset{7}{\to} operande$$

$$U \underset{8}{\to} (E)$$

3.3 EXEMPLES DE DÉRIVATION

En partant de la phrase suivante : 2 + 4

$$E \underset{2}{\to} T + E \underset{3}{\to} F + E \underset{5}{\to} U + E \underset{7}{\to} 2 + E \underset{1}{\to} 2 + T \underset{3}{\to} 2 + F \underset{5}{\to} 2 + U \underset{7}{\to} 2 + 4$$

En partant de la phrase suivante: (U_x + V_y) * W_z / 35++

$$E \underset{1}{\to} T \underset{4}{\to} F * T \underset{5}{\to} U * T \underset{8}{\to} (E) * T \underset{2}{\to} (T + E) * T \underset{3,5,7}{\longrightarrow} (U_x + E) * T \underset{1,3,5,7}{\longrightarrow} (U_x + V_y) * T \underset{4}{\to}$$

$$(U_x + V_y) * F/T \underset{5,7}{\to} (U_x + V\_y) * W_z/T \underset{3,5}{\to} (U_x + V_y) * W_z/U \underset{6}{\to}$$

$$(U_x + V_y) * W_z/U++ \underset{8}{\to} (U_x + V_y) * W_z/35++$$

3.4 CLASSE JAVA

La grammaire définie dans la section 3.2 est représentée dans la classe Descente récursive par les méthodes parseE(), parseT(), parseF(), parseU(). Cette classe est présentée dans le diagramme de la Figure 6. Le pseudo-code des méthodes parseE() et parseT() est le suivant. La même logique s'applique pour les autres méthodes.

```
parseE()
    Si terminalCourant est un opérande ou un '(' alors
        Appel de parseT() dans terme
        Si terminalCourant est + ou - alors
            Création du nœud n avec terme comme enfant gauche
            Lecture du prochain terminal dans terminalCourant
            Appel de parseE()
            Retour de parseE() dans l'enfant droit du nœud n
            Retourner n
        Retourner terme
    Sinon
        Lancer une exception pour erreur de syntaxe
```

```
parseT()
    Appel de parseF() dans facteur
    Si terminalCourant est une multiplication ou une division alors
        Création du nœud n avec facteur comme enfant gauche
        Lecture du prochain terminal dans terminalCourant
        Appel de parseT()
        Retour de parseT() dans l'enfant droit du nœud n
        Retourner n
    Retourner facteur
```

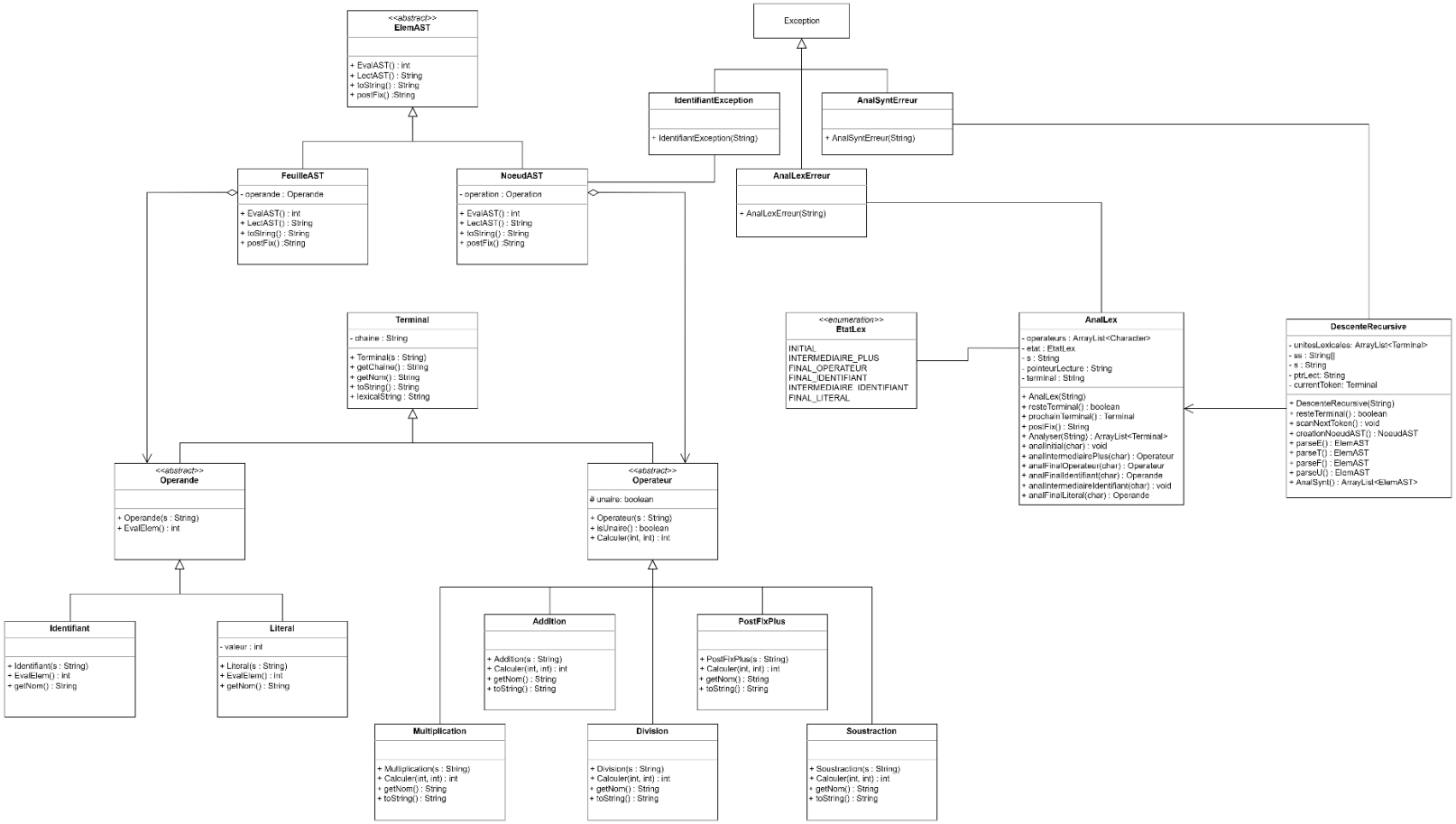

Figure 6. Diagramme de classe complet de l'analyseur syntaxique et lexical

4. PLAN DE TESTS

4.1 ANALYSE LEXICALE

Les tests d'analyse lexicales doivent tester plusieurs éléments : les erreurs lexicales et la récupération d'identifiants, de littéraux et d'opérateurs.

#	Phrase d'entrée	Résultat attendu (Terminaux récupérés; explication de l'erreur)
1	aA	Erreur (un littéral doit commencer par une lettre majuscule)
2	A__a	Erreur (un identifiant ne peut pas contenir deux `_` adjacents)
3	A_	Erreur (un identifiant ne peut pas finir par un `_`)
4	A$	Récupération de l'identifiant « A », puis erreur car `$` est un caractère illégal
5	Aa_aA_a	Récupération de l'identifiant « Aa_aA_a »
6	+	Récupération de l'opérateur « + »
7	++	Récupération de l'opérateur « ++ »
8	**	Récupération de deux opérateurs « * »
9	-12	Récupération de l'opérateur « - » et du littéral « 12 » contenant la valeur numérique *12*
10	000000000000	Récupération du littéral « 000000000000 » contenant la valeur numérique *0*
11	3.14	Récupération du littéral « 3 », contenant la valeur numérique *3*, puis erreur, car `.` est un caractère illégal

4.2 ANALYSE SYNTAXIQUE

Les tests d'analyse syntaxique doivent également tester plusieurs éléments, dans deux catégories importantes : la construction d'AST, et la détection d'erreur.

#	Terminaux d'entrée	Résultat attendu (arbres sous forme parenthésées + évaluation; explication de l'erreur)
1	`1`	(1) : 1
2	`1` `1`	Erreur (un opérant ne peut pas être suivi directement par un autre opérant)
3.1	`+`	
3.2	`1` `+`	Erreur (un opérateur binaire doit être accompagné de deux opérants)
3.3	`1` `+` `*` `2`	
4	`1` `++` `2`	Erreur (une expression ne peut pas être suivi d'une autre expression sans opérateur)
5	`1` `++` `+` `2`	((1++)+2) : 4
6	`(` `1`	Erreur (une parenthèse doit être refermée)
7	`2` `+` `4` `*` `2`	(2 + (4 * 2)) : 10
8	`(` `2` `+` `4` `)` `*` `2`	((2 + 4) * 2) : 12