UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie informatique

RAPPORT APP5

Éléments de compilation
GIF340

Présenté à
Équipe de formateurs de la session S4

Présenté par
Raphael Bouchard – bour0703
Alexis Guérard – guea0902
Gabriel Lamontagne – lamg0502

Sherbrooke – 8 avril 2024

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

1. ANALYSEUR LEXICAL

1.1 UNITÉS LEXICALES

L'analyseur lexical conçu utilise 3 unités lexicales. En effet, nos lexèmes sont séparés en 3 types, soit les nombres, les opérateurs et les variables.

1.1.1 EXPRESSIONS RÉGULIÈRES

Voici les expressions régulières pour chaque unité lexicale.

Tableau 1 : Expression régulière pour les unités lexicales

Unité lexicale	Expression régulière
Nombres	(0-9)+
Opérateurs	[+|-|/|*|(|)]
Variables	(A-Z)(A-Z|a-z|_A-Z|_a-z)*

1.1.2 AUTOMATE

Les 3 unités lexicales présentées ci-haut peuvent être représentées par des automates. La figure 1 est l'automate pour les nombres, la figure 2 est celle pour les opérateurs, la figure 3 est pour les variables et la figure 4 représente les 3 automates ensemble avec la gestion des erreurs et des retours.

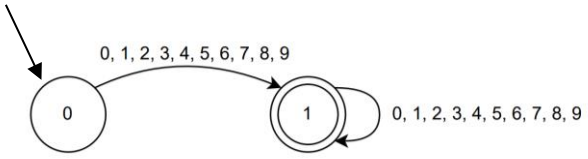

Figure 1 : Automate pour les nombres

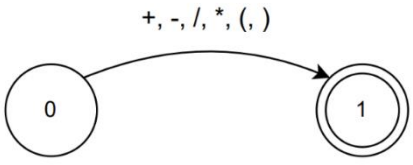

Figure 2 : Automate pour les opérateurs

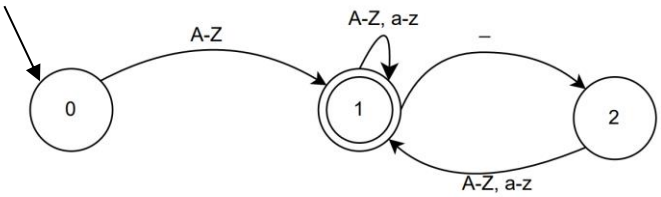

Figure 3 : Automate pour les variables

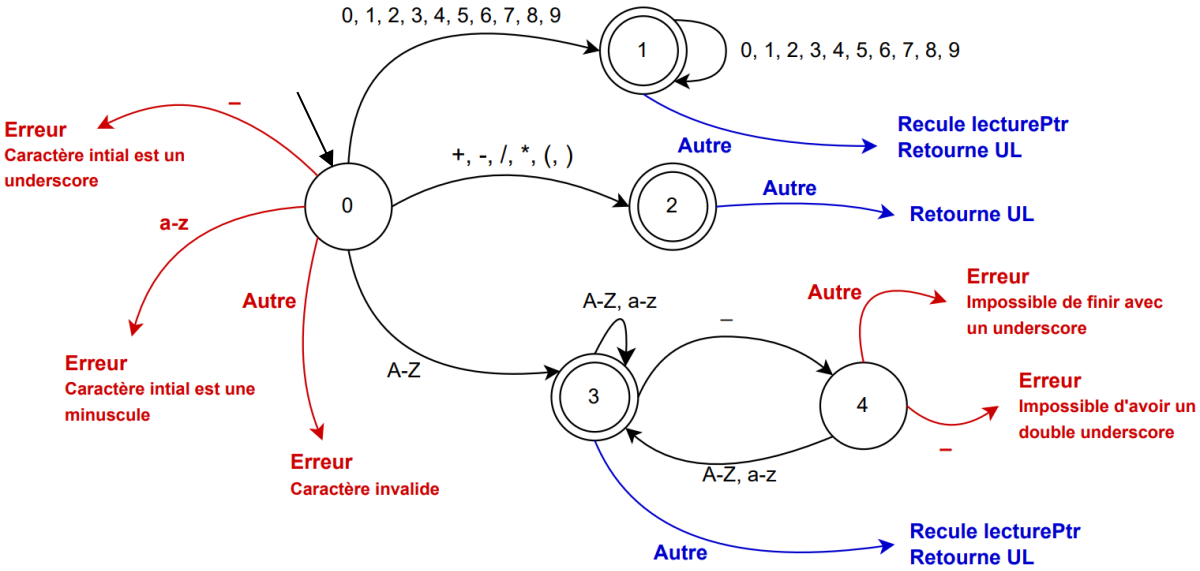

Figure 4 : Automate complet de l'analyseur lexical

1.2 CLASSE DE L'ANALYSE LEXICALE

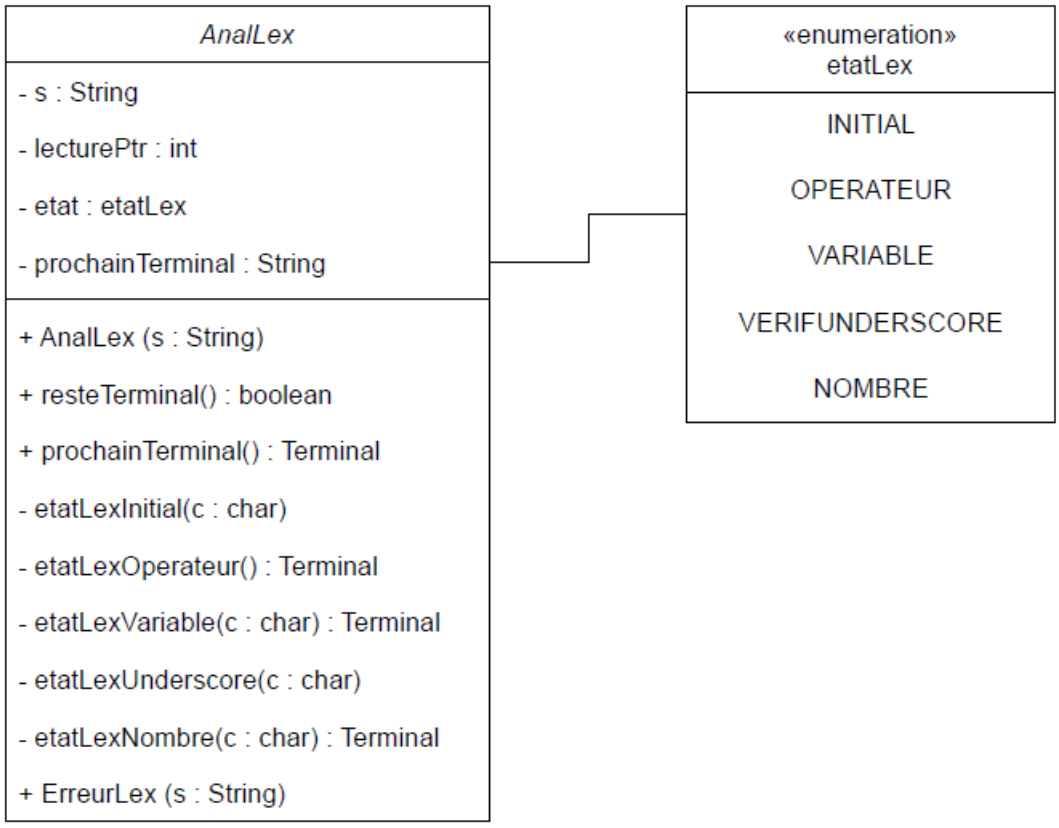

Figure 5 : Diagramme de classe de l'analyse lexicale

La fonction prochainTerminal() nous sert, à l'aide d'un switch case, de faire différentes étapes en fonction d'où nous sommes dans le terminal, donc le mot ou l'opérateur. Chaque cas va appeler une fonction parmi les cinq fonctions etatLex.

La fonction etatLexInitial vérifie le premier caractère. Si c'est une lettre, on vérifie si c'est une majuscule. Si oui, on change l'état à VARIABLE sinon on retourne une erreur. On vérifie sinon si c'est un nombre ou un opérateur et on change l'état pour celui des deux qui est valide. Dans le cas contraire, on retourne une erreur qui mentionne que le caractère n'est pas valide. La fonction etatLexOperateur change le type du terminal pour correspondre au bon opérateur. La fonction etatLexVariable vérifie si le caractère est un « _ ». Si c'est le cas, l'état change pour VERIFUNDERSCORE, sinon si c'est le dernier caractère on retourne le terminal. « etatLexUnderscore » sert à vérifier que l'underscore ne soit pas double et qu'il ne soit pas à la fin du mot. Si c'est le cas, on retourne une erreur sinon on retourne à l'état VARIABLE.

2. ARBRE SYNTAXIQUE ABSTRAIT

Le diagramme de classe de la Figure 6 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 alors que les nœuds ont des opérations.

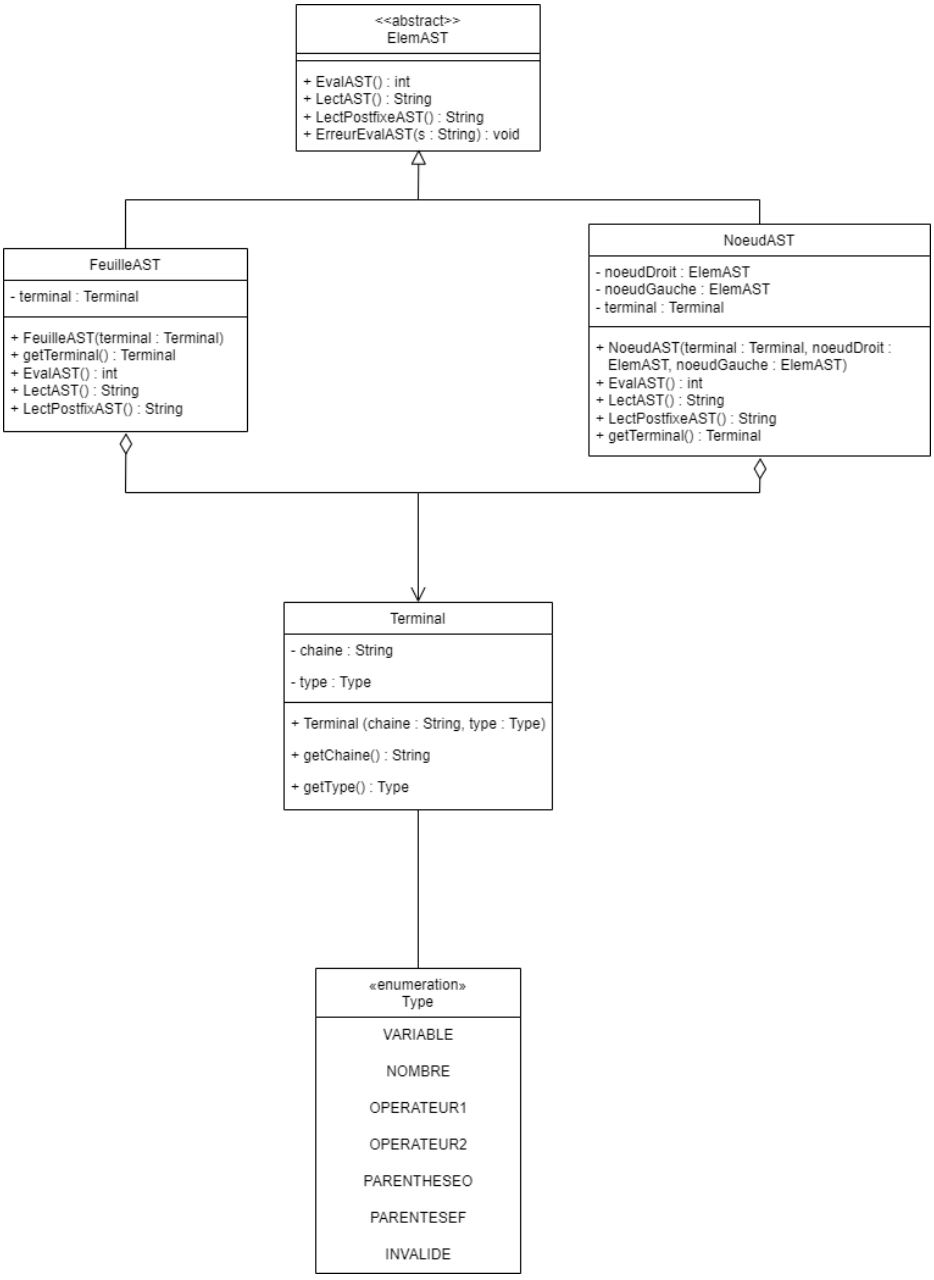

Figure 6 : Arbre syntaxique des classes AST

Classe FeuilleAST :
La fonction EvalAST vérifie si le terminal est bien un nombre et le retourne si oui. Si c'est une variable on retourne une erreur, sinon on retourne 0. LectAST et LectPostfixeAST servent tous deux à retourner la chaine du terminal.

Classe NoeudAST :
La fonction EvalAST exécute l'action de l'opérateur avec les feuilles qu'on lui donne préalablement avec le constructeur. LectAST et LectPostfixeAST servent tous deux à retourner la chaine du terminal.

3. ANALYSEUR SYNTAXIQUE

3.1 ANALYSEUR DESCENDANT

Un analyseur descendant est un outil clé dans l'analyse syntaxique d'un langage formel. Son fonctionnement repose sur la décomposition progressive d'une expression selon les règles de production d'une grammaire donnée. En utilisant la récursivité, l'analyseur sélectionne les règles appropriées pour chaque étape de l'analyse, descendant ainsi dans l'arbre syntaxique jusqu'aux éléments lexicaux de l'expression. Tout en vérifiant la conformité de l'expression à la grammaire, l'analyseur construit simultanément l'arbre syntaxique correspondant, ce qui permet une analyse structurée et précise de la syntaxe de l'expression.

3.2 ANALYSEUR LL

Un analyseur LL est une variante d'analyseur descendant qui privilégie l'évaluation complète de chaque terme avant de passer au suivant. La lecture de la phrase se fait de gauche à droite, et la dérivation s'effectue également de gauche à droite jusqu'à atteindre une unité lexicale. Les analyseurs LL peuvent adopter diverses approches selon le nombre k de prochains terminaux connus.

La catégorie d'analyseurs syntaxiques LL la plus utilisée en pratique est généralement celle des analyseurs LL(1). Cela est dû au fait qu'elle anticipe une unité lexicale adaptée aux règles de notre grammaire régulière.

Pour mettre en œuvre un algorithme LL(1), il est essentiel de comprendre les concepts de "premier" et "suivant". Le "premier" représente la liste des symboles terminaux pouvant commencer une dérivation, tandis que le "suivant" représente ceux qui peuvent suivre une dérivation.

3.3 TYPE DE GRAMMAIRE

La grammaire peut être définie de façon suivante:

$$a = \{+, -\}\ et\ b = \{*, /\}$$

$$E \rightarrow T[aE]$$

$$T \rightarrow F[bT]$$

$$F \rightarrow (E) \mid operande$$

$$E \rightarrow_1 T$$

$$E \rightarrow_2 T\ a\ E$$

$$T \rightarrow_3 F$$

$$T \rightarrow_4 T\ b\ F$$

$$F \rightarrow_5 operande$$

$$F \rightarrow_6 (E)$$

3.4 EXEMPLE DE DÉRIVATION

Exemple avec 5 + 7 :

$$E \rightarrow_2 T + E \rightarrow_3 F + E \rightarrow_5 5 + E \rightarrow_2 5 + T \rightarrow_3 5 + F \rightarrow_5 5 + 7$$

Exemple avec 4 * (A_b + 1) / 3 – 1

$$E \rightarrow_2 T + E \rightarrow_4 F * T + E \rightarrow_5 4 * T + E \rightarrow_4 4 * T/F + E \rightarrow_3 4 * F/F + E \rightarrow_6$$

$$4 * (E)/F + E \rightarrow_2 4 * (T + E)/F + E \rightarrow_3 4 * (F + E)/F + E \rightarrow_6$$

$$4 * (A\_b + E)/F + E \rightarrow_1 4 * (A_b + T)/F + E \rightarrow_3 4 * (A\_b + F)/F - E \rightarrow_5$$

$$4 * (A\_b + 1)/F - E \rightarrow_5 4 * (A\_b + 1)/3 - E \rightarrow_1 4 * (A\_b + 1)/3 - T \rightarrow_3$$

$$4 * (A\_b + 1)/3 - F \rightarrow_5 4 * (A\_b + 1)/3 - 1$$

3.5 CLASSE JAVA

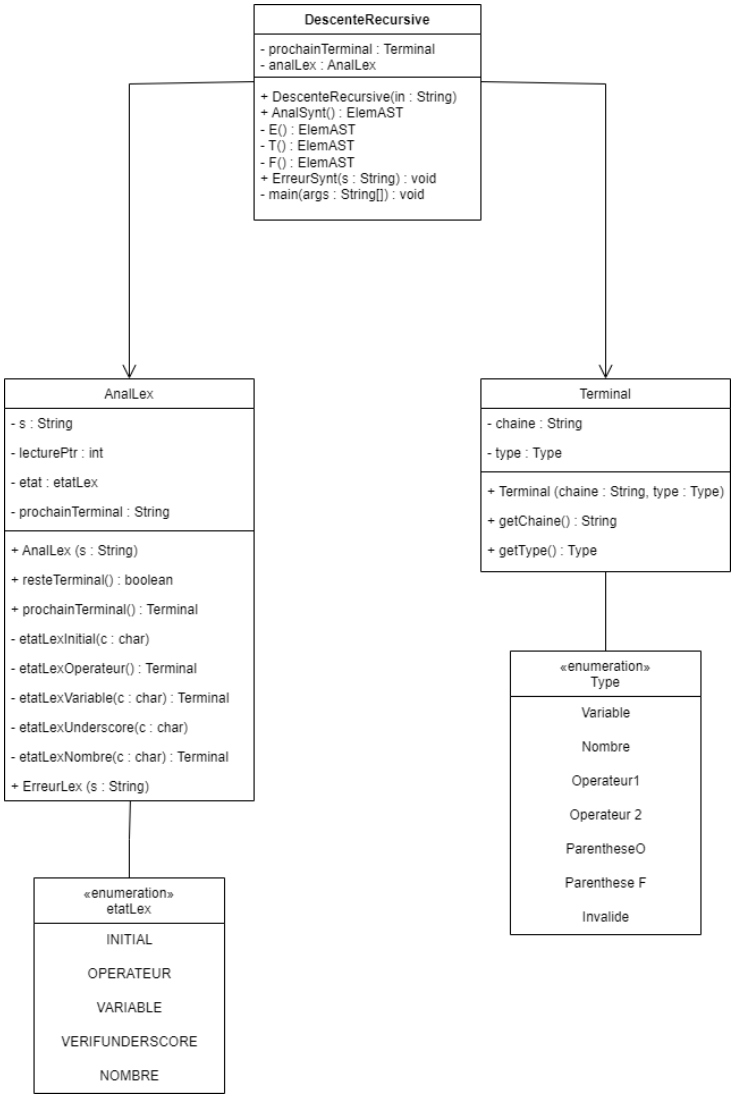

Figure 7 : Diagramme de classe de Descente Récursive

La fonction AnalSynt incrémente un pointeur et avance au prochain terminal pour la prochaine itération, puis retourne la fonction E(). Les fonctions E(), T() et F() vont construire notre arbre AST de manière récursive puisque chacune des trois fonctions appelle la suivante. En premier lieu, les fonctions E() et T() vont générer les différents nœuds de l'arbre, donc les opérateurs de notre AST. La fonction F(), elle, va plutôt créer les feuilles de l'arbre, donc elle écrit les nombres et les variables. Lors de la construction de l'arbre AST, ces trois fonctions vérifient aussi la syntaxe de l'équation afin de s'assurer qu'il ne manque pas de parenthèses, qu'il n'y a pas deux opérateurs

qui se suivent ou encore qu'on ne termine pas avec un opérateur. Dans tous ces cas, on retourne une erreur qui indique de quel genre d'erreur il s'agit et où elle se trouve dans l'équation initiale.

4. PLAN DE TESTS

4.1 ANALYSEUR LEXICAL

Pour avoir un plan de test complet pour l'analyse lexicale, il faut choisir des tests qui recouvrent tous les cas possibles.

Tableau 2 : Plan de tests de l'analyseur lexical

Objectif ciblé		**Validé**	
Condition à proscrire	**Aucun**		
Test	**Action**	**Résultats attendus**	
Variable valide seulement de lettre	Écrire dans le fichier ExpArith.txt « Aa ».	Récupération de la variable Aa.	[1]
Variable valide avec des _	Écrire dans le fichier ExpArith.txt « Aa_dB_T ».	Récupération de la valeur Aa_dB_T.	[1]
Opérateur valide	Écrire dans le fichier ExpArith.txt « + ».	Récupération de l'opérateur +	[1]
Opérateurs valides	Écrire dans le fichier ExpArith.txt « // ».	Récupération de deux opérateurs /.	[1]
Nombre valide	Écrire dans le fichier ExpArith.txt « 123456 ».	Récupération du nombre 123456.	[1]
Expression complète	Écrire dans le fichier ExpArith.txt « (Ah_at + 123/790*30) ».	Séparation de l'expression en une parenthèse ouverte (, une variable Ah_at, un opérateur +, un nombre 123, un opérateur /, un nombre 790, un opérateur *, un nombre 30 et une parenthèse fermée).	[1]
Erreur minuscule	Écrire dans le fichier ExpArith.txt « aA ».	Renvoie une erreur lexicale au niveau du caractère 1 (Une variable ne peut commencer par une minuscule).	[1]
Erreur double _	Écrire dans le fichier ExpArith.txt « A__ert ».	Renvoie une erreur lexicale au niveau du caractère 3 (Une variable ne peut pas avoir un double underscore).	[1]
Erreur fin _	Écrire dans le fichier ExpArith.txt « Aa_ ».	Renvoie une erreur lexicale au niveau du caractère 3 (Une variable ne peut pas finir par un _).	[1]
Erreur début _	Écrire dans le fichier ExpArith.txt « _Aa ».	Renvoie une erreur lexicale au niveau du caractère 3 (Une variable ne peut pas commencer par un _).	[1]

Caractère invalide	Écrire dans le fichier ExpArith.txt « Aa! ».	Renvoie une erreur lexicale au niveau du caractère 3 (Caractère invalide).	[1]

4.2 CONSTRUCTION D'AST

Tableau 3 : Plan de tests construction d'AST

Objectif ciblé		**Validé**	
Condition à proscrire	**Aucun**		
Test	**Action**	**Résultats attendus**	
Construction d'équation	Écrire dans le fichier ExpArith.txt « (U_x - V_y) * W_z / 35 »	AST construit : (35 / W_z) * (V_y - U_x)	[1]
	Écrire dans le fichier ExpArith.txt « (55-47) * 14 / 2 »	AST construit : (2 / 14) * (47 - 55)	[1]
	Écrire dans le fichier ExpArith.txt « 55-47 * 14 / 2 »	AST construit : ((2 / 14) * 47) - 55	[1]

4.3 ANALYSEUR SYNTAXIQUE

Tableau 3 : Plan de tests l'analyseur syntaxique

Objectif ciblé		**Validé**	
Condition à proscrire	**Aucun**		
Test	**Action**	**Résultats attendus**	
Avoir uniquement un terminal numérique.	Écrire dans le fichier ExpArith.txt « 1 ».	Affichage du résultat 1 dans la console.	[1]
Avoir uniquement un opérande comme terminal.	Écrire dans le fichier ExpArith.txt « + »	Erreur de syntaxe (Niveau F) OPERATEUR2	[1]
Avoir un terminal de type OPERATEUR1 dans l'équation.	Écrire dans le fichier ExpArith.txt « 2 * 2 ».	Affichage du résultat 4 dans la console.	[1]
Avoir un terminal de type OPERATEUR1 dans l'équation.	Écrire dans le fichier ExpArith.txt « 2 ** 2 ».	Erreur de syntaxe (Niveau F) OPERATEUR1	[1]
Avoir un terminal de type OPERATEUR1 dans l'équation.	Écrire dans le fichier ExpArith.txt « 2 * 2 * ».	Erreur de syntaxe (Niveau F) OPERATEUR1	[1]
Avoir un terminal de type OPERATEUR2 dans l'équation.	Écrire dans le fichier ExpArith.txt « 2 + 4 ».	Affichage du résultat 6 dans la console.	[1]

Avoir un terminal de type OPERATEUR2 dans l'équation.	Écrire dans le fichier ExpArith.txt « 2 ++ 4 ».	Erreur de syntaxe (Niveau F) OPERATEUR2	[1]
Avoir un terminal de type OPERATEUR2 dans l'équation.	Écrire dans le fichier ExpArith.txt « 2 + 2 + ».	Erreur de syntaxe (Niveau F) OPERATEUR2	[1]
Avoir un terminal de type VARIABLE dans l'équation.	Écrire dans le fichier ExpArith.txt « A + 2 ».	Erreur de syntaxe : Évaluation impossible car variable	[1]
Avoir un terminal de type NOMBRE dans l'équation.	Écrire dans le fichier ExpArith.txt « 5 ».	Affichage du résultat 5 dans la console.	[1]
Avoir un terminal de type PARENTHESEO dans l'équation.	Écrire dans le fichier ExpArith.txt « (5 + 3) ».	Affichage du résultat 8 dans la console.	[1]
Avoir un terminal de type PARENTHESEO dans l'équation.	Écrire dans le fichier ExpArith.txt « (5 + 3 ».	Erreur de syntaxe (Niveau F), parenthèse manquante	[1]
Avoir un terminal de type PARENTHESEF dans l'équation.	Écrire dans le fichier ExpArith.txt « (5 + 2) ».	Affichage du résultat 7 dans la console.	[1]
Avoir un terminal de chaque type valide dans l'équation.	Écrire dans le fichier ExpArith.txt « (2 + 3) * 2 - 2 / 24 ».	Affichage du résultat 2 dans la console.	[1]