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

RAPPORT DE L'APP 5

Éléments de compilation
GIF340

Présenté à
Ahmed Khoumsi

Présenté par
Équipe numéro 2
Matthieu Daoust – DAOM2504
William Pépin – PEPW3101
Gabriel Vachon – VACG3501

Sherbrooke – 5 avril 2023

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

1. ANALYSE LEXICALE

L'analyse lexicale permet de détecter et d'identifier les unités lexicales présentes dans un code source. En effectuant une analyse caractère par caractère, le programme permet de fournir une série d'unité lexicale à l'analyseur syntaxique. Elle permet aussi de trouver des erreurs lexicales, soit des erreurs dans l'écriture des identificateurs, des mots clés, des opérateurs, des nombres, etc. Dans le cadre de la problématique, l'analyseur lexical doit identifier les opérateurs suivants : **+ - */,** les unités de contrôle suivant : **(** , **)**, les nombres naturels et les variables définies dans le guide [1]. Les sections suivantes présentent les expressions régulières ainsi que les automates requis afin de détecter les unités lexicales du contexte défini, en finissant par la construction d'un automate complet qui définit la logique globale de l'analyseur lexical.

1.1 OPÉRATEURS

Définition de l'alphabet, du langage et des expressions régulières :

$$\Sigma = \{+, -, *, /\} \tag{1}$$

$$L_\Sigma = \{(+, -, *, /)\}$$

$$ER_+ = + \qquad ER_- = - \qquad ER_* = * \qquad ER_/ = /$$

Automate :

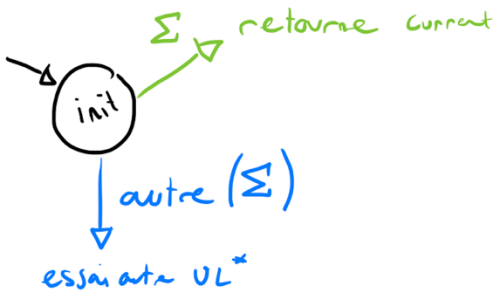

Figure 1 : Automate à état fini de la détection des opérateurs

*La fonction « essai autre UL » n'est pas une vraie fonction de l'automate, cette flèche sera définie à la fin de la section, lors de la création de l'automate complet.

En somme, l'alphabet des opérateurs constitue des opérateurs directement et l'automate détecte l'opérateur et le retourne. Dans un autre cas, elle vérifie si ce n'est pas une autre unité lexicale.

1.2 PARENTHÈSES

Définition de l'alphabet, du langage et des expressions régulières :

$$\Sigma = \{(,)\} \tag{2}$$
$$L_\Sigma = \{(,)\}$$
$$ER_(= (\qquad ER_) =)$$

Automate :

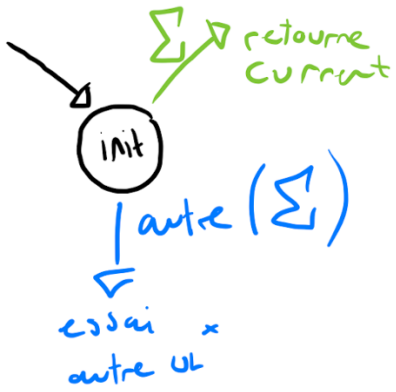

Figure 2 : Automate a état fini de la détection des parenthèses

*La fonction « essai autre UL » n'est pas une vraie fonction de l'automate, cette flèche sera définie à la fin de la section, lors de la création de l'automate complet.

En somme, l'alphabet des parenthèses constitue des parenthèses ouvrantes et fermantes. L'automate détecte alors les parenthèses individuellement. Dans un autre cas, elle vérifie si ce n'est pas une autre unité lexicale.

1.3 NOMBRE

Définition de l'alphabet, du langage et des expressions régulières :

$$\Sigma = \{0,1,2,3,4,5,6,7,8,9\} \tag{3}$$
$$L_\Sigma = \mathbb{N}$$
$$ER = (0|1|2|3|4|5|6|7|8|9)^+$$

Automate :

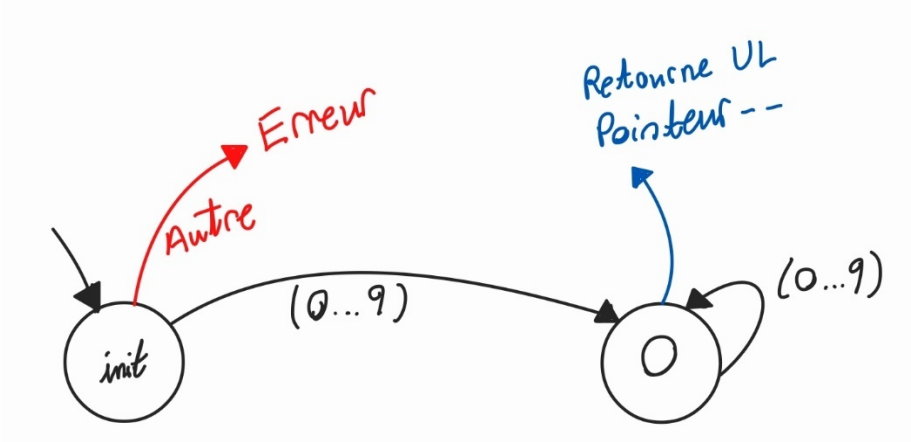

Figure 3 : Automate à état fini pour un opérande à chiffres

L'automate à état fini ci-dessus permet d'identifier un opérande composé de chiffres. À l'état initial, l'automate s'attend à recevoir un chiffre de l'ensemble mentionné plus haut. S'il n'en reçoit pas ou il reçoit un caractère non représenté dans l'ensemble, une erreur est retournée. Il arrive ensuite à l'état final, ici la chaîne de l'opérande a au moins un caractère. Si un caractère qui n'est pas représenté dans l'ensemble est lu, l'unité lexicale est retournée et on recule le pointeur.

1.4 VARIABLE

Définition de l'alphabet, du langage et des expressions régulières :

$$\Sigma = \{A, B, C, D, \dots, Z, a, b, c, d, \dots, z\} \quad (4)$$

$$\Sigma_{maj,min} = \{A, B, C, D, \dots, Z, a, b, c, d, \dots, z\}$$

$$\Sigma_{maj} = \{A, B, C, D, \dots, Z\}$$

$$\Sigma_{min} = \{a, b, c, d, \dots, z\}$$

$$ER = \Sigma_{maj}[\_? (\Sigma_{maj,min})]^*$$

Automate :

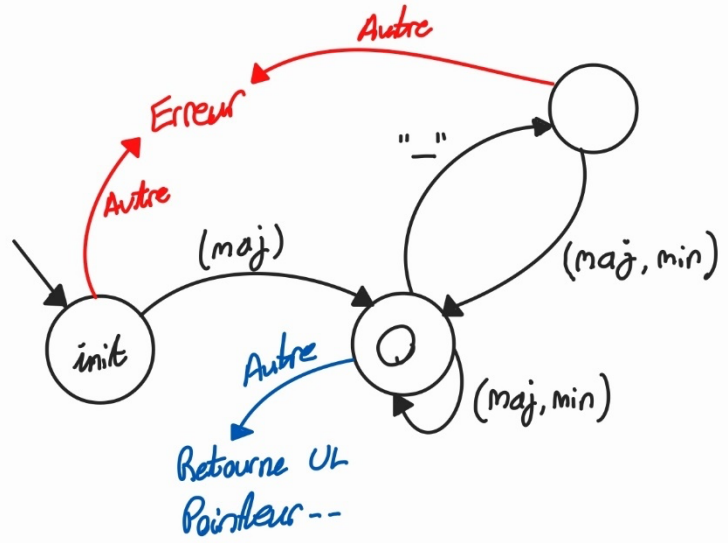

Figure 4 : Automate à état fini pour un opérande contenant des caractères maj, min, « _ »

L'automate ci-dessus permet d'identifier une chaine de caractères contenant des majuscules, minuscules ou tirets bas. À l'état initial, il est impératif d'avoir une lettre majuscule, si un caractère autre est identifié, une erreur est lancée [le premier caractère ne peut commencer par un tiret bas]. Après le premier caractère, une majuscule, une minuscule ou un tiret bas peut être ajouté. Il n'y a pas de limite de caractères majuscules ou minuscules dans la chaine, cependant, il ne peut y avoir plusieurs tirets bas consécutifs et la chaine ne peut finir par ce caractère. Pour cette raison, si l'état est « tiret_bas », une erreur est lancée si le caractère n'est pas une lettre. À l'état final, si un autre caractère autre qu'une lettre ou un tiret bas est entré, on retourne l'unité lexicale et on recule le pointeur.

1.5 CONSTRUCTION DE L'AUTOMATE COMPLET

La réalisation suivante constitue l'automate complet de l'analyseur lexicale, elle incorpore les automates de chacune des sections dans un seul automate, sans ambiguïté. Les ensembles sont définies avec les noms suivants :

- Op = ensemble des opérateurs ($\Sigma_{\text{op}} = \{+, -, *, /\}$);
- Co = ensembles des parenthèses ($\Sigma_{\text{co}} = \{(,)\}$);
- No = ensemble des nombres ($\Sigma_{\text{no}} = \{0,1,2,3,4,5,6,7,8,9\}$);
- Maj = ensemble des majuscules (Σ_{maj}) = {A,B,C,D,…,Z};
- Min = ensemble des minuscules (Σ_{min}) = {a,b,c,d,…z};
- X ensemble des variables ($\Sigma_{\text{X}} = \Sigma_{maj} \cup \Sigma_{min} \cup \{\_\}$).

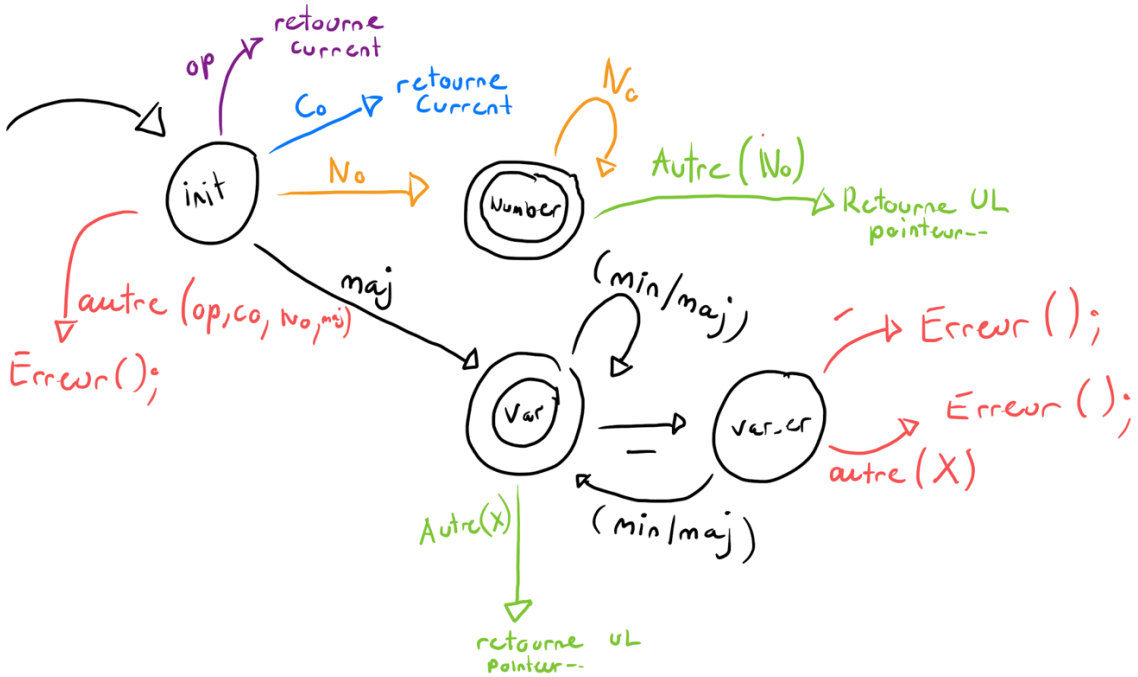

Figure 5 : Automate complet de l'analyseur lexicale

Les états sont les suivants : *init, number, var et var_er.* L'état « *init* » est le point d'entrée de l'automate, elle permet de filtrer l'alphabet de l'analyseur lexical en incorporant une vérification sur les opérateurs, les parenthèses, les nombres et les variables, si le caractère lu n'est pas dans l'alphabet de ses 4 types d'unités lexicales, une erreur est retournée. Si c'est un opérateur ou une parenthèse, l'état retourne directement l'unité lexicale. Si c'est un nombre, l'automate passe à l'état « *number* » et effectue la logique de vérification des nombres. Si c'est une Majuscule, l'automate passe à l'état « *var* » et effectue la logique de vérification des variables. Cette logique retourne une erreur si la variable termine par « _ » ou contient deux « _ » à la suite de l'autre.

2. ANALYSE SYNTAXIQUE

L'analyseur syntaxique permet d'effectuer l'analyse des unités lexicales afin de trouver des erreurs de syntaxes. Elle permet de définir le comportement du compilateur de sorte à accepter [ou refuser] des expressions arithmétiques en fonction d'une grammaire. Une fois le regroupement des unités lexicales en structures grammaticales, l'analyseur syntaxique peut générer un arbre syntaxique abstrait qui représente la structure hiérarchique du flot d'unités lexicales. [2] Ainsi, pour réaliser un analyseur syntaxique, il est nécessaire de définir une

grammaire rigoureuse qui permet la vérification efficace des flots d'unités lexicales. L'algorithme utilisé pour la réalisation de cet analyseur syntaxique est l'algorithme LL [1].

2.1 RÉALISATION DE LA GRAMMAIRE DE BASE

Afin de réaliser une grammaire de base pour un analyseur syntaxique, celle-ci doit être de type « non-contextuelle ». C.-à-d. qu'elle doit respecter avoir deux caractéristiques suivantes : un seul symbole doit être présent sur la partie gauche de la règle et la partie droite de la règle ne doit pas nécessairement débuter par un symbole terminal. Le chapitre 3 des notes de cours offre une grammaire non contextuelle afin de modéliser des expressions arithmétiques [2] :

- Vt = ensemble des symboles terminaux;
- Vn = ensemble des symboles non-terminaux;
- S = symbole initiale;
- P = ensemble fini des règles de production;
- Id = opérande.

Tableau 1 : Définition d'expression arithmétique par une grammaire [2]

Vt = {id, nombre, +, -, *, /, (,)}

Vn = {E, T, F}

S = E

P {

Symbole		Dérivation	Numéro de la règle
E	⇨	T	1
E	⇨	T+T	2
E	⇨	T-T	3
T	⇨	F	4
T	⇨	F*F	5
T	⇨	F/F	6
F	⇨	(E)	7
F	⇨	id	8
F	⇨	nombre	9

}

Toutefois, cette grammaire ne respecte pas les contraintes d'une grammaire LL (1). Les sections suivantes présentent la démarche afin de transformer la grammaire suivante en grammaire qui respecte les contraintes définies par une grammaire LL (1).

2.2 TRANSFORMATION DE LA GRAMMAIRE POUR UN ALGORITHME LL(1)

2.2.1 DÉFINITION DES CONTRAINTES D'UNE GRAMMAIRE DITE LL(1)

Afin d'obtenir une grammaire LL (1), il est nécessaire de respecter les trois contraintes suivantes (assumant une grammaire non contextuelle) :

1- Pour chaque règle ayant la même partie gauche, l'ensemble premier de chaque règle ne doit pas avoir d'élément(s) en commun entre les ensembles [2] ;

$$\begin{gathered} A \rightarrow \alpha 1 \\ A \rightarrow \alpha 2 \\ PREMIER(\alpha 1) \cap PREMIER(\alpha 2) = \emptyset \end{gathered} \tag{5}$$

2- Pour chaque règle capable de devenir un ensemble vide, l'ensemble premier de cette règle ne doit pas avoir d'élément(s) en commun avec l'ensemble suivant de cette même règle [2].

$$\begin{gathered} A \rightarrow \alpha 1 \mid \epsilon \\ PREMIER(\alpha 1) \cap SUIVANT(\alpha 1) = \emptyset \end{gathered} \tag{6}$$

Autrement dit, la grammaire LL (1) ne doit pas être récursive à gauche, et ne doit pas être ambiguë. Les sections suivantes vérifient les contraintes ci-dessus afin de modifier la grammaire pour qu'elle respecte les contraintes d'une grammaire LL (1).

2.2.2 MODIFICATION DE LA GRAMMAIRE POUR RESPECTER LA PREMIÈRE CONTRAINTE

La première contrainte n'est pas respectée ici. La raison est la suivante : l'ensemble des premiers des différentes dérivations de E partagent tous les premiers de T, c'est la même chose pour les dérivations de T où les premiers de T qui contient l'ensemble des premiers de F. cependant les ensembles des premiers des règles de dérivation de F sont tous des ensembles différents, ce qui respecte la première règle.

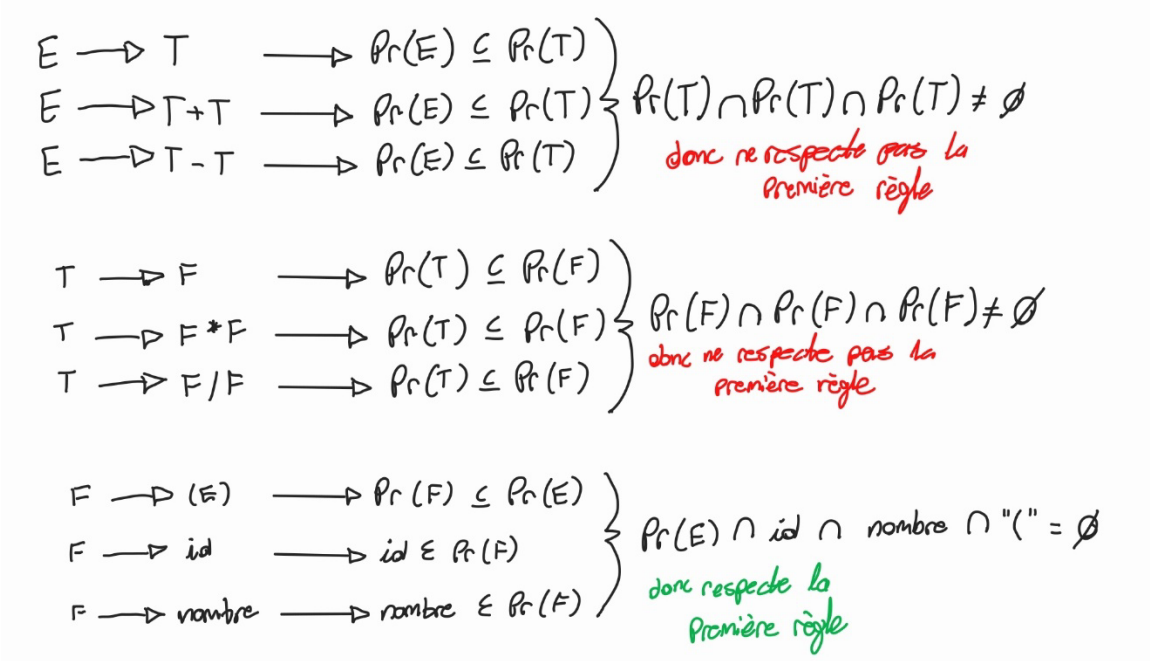

Figure 6 : Preuve mathématique de la première règle

La grammaire peut être modifiée ainsi pour convenir à la première règle :

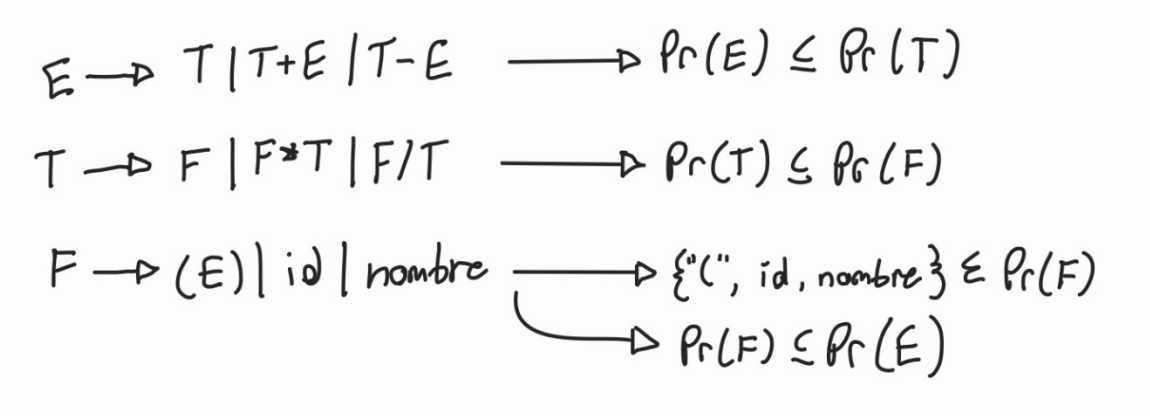

Figure 7 : Démarches de la modification de la grammaire pour la première règle

La grammaire respectant la première règle est indiquée au tableau suivant.

Tableau 2 : Définition de la grammaire LL(1)

Vt = {id, nombre, +, -, *, /, (,)}

Vn = {E, T, F}

S = E

P {

Symbole		Dérivation	Numéro de la règle
E	⇨	T | T+E | T-E	1
T	⇨	F | F*T | F/T	2
F	⇨	(E) | id | nombre	3

}

2.2.3 MODIFICATION DE LA GRAMMAIRE POUR RESPECTER LA DEUXIÈME CONTRAINTE

La deuxième contrainte est respectée dans ce cas-ci, car il n'y a aucune règle qui contient d'ensemble vide. Puisqu'il n'est pas possible d'obtenir d'ensemble vide dans aucune des règles, la deuxième règle est respectée par défaut.

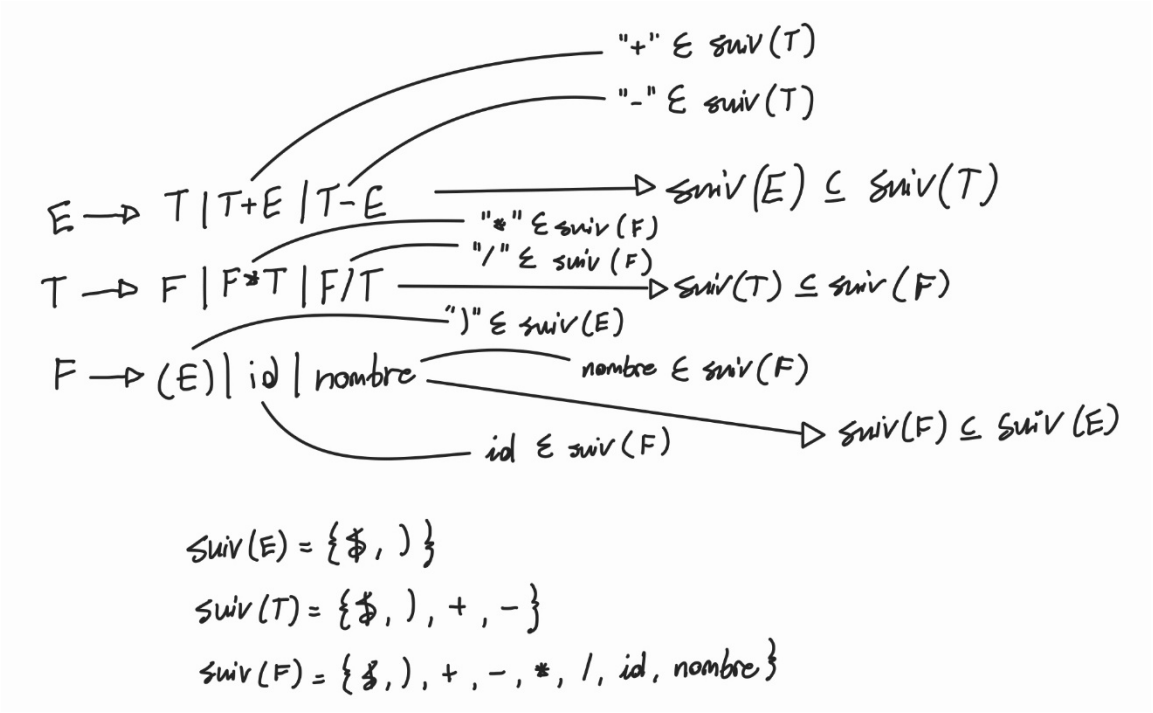

Figure 8 : Démarches pour le respect de la deuxième règle de la grammaire

La grammaire résultante est donc celle définie au **Tableau 2**.

2.3 GRAMMAIRE TRANSFORMÉ

En somme, en modifiant la grammaire de base, il est possible d'arriver à la grammaire suivante (même que le **Tableau 2**) :

Tableau 3 : Définition de la grammaire LL(1)

Vt = {id, nombre, +, -, *, /, (,)}

Vn = {E, T, F}

S = E

P {

Symbole		Dérivation	Numéro de la règle
E	⇨	T | T+E | T-E	1
T	⇨	F | F*T | F/T	2
F	⇨	(E) | id | nombre	3

}

Cette grammaire est valide pour un analyseur syntaxique utilisant une descente récursive comme algorithme.

3. RÉSULTATS DES TESTS DE L'IMPLÉMENTATION

Le présent tableau offre une synthèse des résultats des tests unitaires de l'implémentation du compilateur. Les méthodes sont décrites dans les classes de tests (*AnalLexTest* et *AnalSyntTest*) du projet.

Tableau 4 : Résultats des tests de l'implémentation de l'analyseur lexical

ID	Test	Description	Résultats attendu	OK
1	*prochainTerminalWhenParentheses()*	Vérifie la détection de parenthèses.	Tableau d'unités lexicales contenant [(,)].	OK
2	*prochainTerminalWhenVariableStartsWithLowercase()*	Vérifie l'erreur de variable commençant par une minuscule.	*AnalLexException*	OK
3	*prochainTerminalWhenVariableEndsWithUnderScore()*	Vérifie l'erreur de variable terminant par un tiret bas.	*AnalLexException*	OK
4	*prochainTerminalWhenNumberIsOk()*	Vérifie un cas général d'un nombre.	Tableau contenant les unités lexicales.	OK
5	*prochainTerminalWhenVariableHasTwoUnderScores()*	Vérifie l'erreur de variable ayant deux tirets bas de suite.	*AnalLexException*	OK
6	*prochainTerminalWhenVariableisOk()*	Vérifie un cas général d'une variable.	Tableau contenant les variables.	OK
7	*prochainTerminalWhenCharNotInAlphabet()*	Vérifie l'erreur d'un caractère qui n'est pas dans l'alphabet du compilateur.	*AnalLexException*	OK
8	*prochainTerminalWhenOperators()*	Vérifie la détection des 4 opérateurs (+, -, *, /).	Tableau contenant les 4 opérateurs.	OK
9	*prochainTerminalGeneralCase()*	Vérifie un cas général couvrant les unités lexicales. ((X_a + Y_b) * Z_c / 59) - 4	Tableau contenant l'expression	OK

Tableau 5 : Résultats des tests de l'implémentation de la méthode de descente récursive

ID	Test	Description	Résultats attendu	OK
1	*AnalSyntTestSyntaxeOperateurFini()*	Test d'erreur de syntaxe lorsqu'elle termine avec un opérateur	*AnalSyntException*	OK
2	*AnalSyntTestLectureAST1()*	Test de lecture du cas suivant : (X_a + Y_b) * Z_c / 2	*((X_a+Y_b)*(Z_c/2))*	OK
3	*AnalSyntTestLectureAST2()*	Test de lecture du cas suivant : (X_a - Y_b / Z_c + 59) + 4	((X_a+((Y_b/Z_c)+59))-4)	OK
4	*AnalSyntTestLectureAST3()*	Test de lecture du cas suivant : 4 * 4 * 4 + 12 / 12	((4*(4*4))+(12/12))	OK
5	*AnalSyntTestSyntaxeParentheseFermanteTrop()*	Test d'erreur de syntaxe lorsqu'elle a une parenthèse fermante de trop	AnalSyntException	OK
6	*AnalSyntTestSyntaxeParentheseOuvranteTrop()*	Test d'erreur de syntaxe lorsqu'elle a une parenthèse ouvrante de trop	*AnalSyntException*	OK
7	*AnalSyntTestPostFixCase1()*	Test de lecture *postfix* du cas suivant : (X_a + Y_b) * Z_c / 2	X_aY_bZ_c*2/	OK
8	*AnalSyntTestPostFixCase2()*	Test de lecture *postfix* du cas suivant : (X_a - Y_b / Z_c + 59) + 4	X_AY_bZ_c/-59+4+	OK
9	*AnalSyntTestPostFixCase3()*	Test de lecture *postfix* du cas suivant : 4 * 4 * 4 + 12 / 12	44*4*1212/+"	OK
10	*AnalSyntTestEvalAST1()*	Test d'évaluation du cas suivant : (X_a + Y_b) * Z_c / 2	19	OK
11	*AnalSyntTestEvalAST2()*	Test d'évaluation du cas suivant : (X_a + Y_b / Z_c + 59) - 4	69	OK
12	*AnalSyntTestEvalAST3()*	Test d'évaluation du cas suivant : 4 * 4 * 4 + 12 / 12	65	OK
13	*AnalSyntTestSyntaxeManqueVariable()*	Tests d'évaluation du cas suivant : (X_a +) * Z_c / 2	AnalSyntException	OK

4. RÉFÉRENCES

[1] A. Khoumsi, Développement d'un analyseur syntaxique d'expressions arithmétiques, Sherbrooke: Faculté de génie, Université de Sherbrooke, 2023.

[2] A. Khoumsi, GIF 340 : Éléments de compilation, Sherbrooke: Faculté de génie, Université de Sherbrooke, 2005.