Rapport de Projet de Compilation

Encadré par : Devan Sohier

Institut en Sciences et Techniques des Yvelines (ISTY)

Thème : Analyseur et Évaluateur d'Expressions Mathématiques (Calculatrice)

Étudiants : Amayes DJERMOUNE, Yao HIPPOLYTE

Classe : IATIC 4

March 13, 2024

Sommaire

Amayes DJERMOUNE, Yao HIPPOLYTE

March 13, 2024

Contents

1 Introduction

L'introduction des interpréteurs dans notre quotidien a profondément transformé la manière dont nous interagissons avec la programmation, l'informatique, et même au-delà. Ces outils informatiques, capables de comprendre et d'exécuter des instructions directement à partir du code source, offrent une flexibilité et une accessibilité qui ont des implications significatives. Que ce soit dans le développement logiciel, l'éducation en informatique, l'automatisation de tâches, ou encore l'analyse de données, les interpréteurs sont devenus des compagnons indispensables, facilitant la vie quotidienne de nombreuses façons. Dans cette exploration, nous examinerons l'importance des interpréteurs, leurs applications variées, et comment ils ont contribué à façonner notre expérience numérique au jour le jour.

Pour ne pas s'éloigner du contexte, le but de ce projet est d'implémenter une sorte de calculatrice qui agira comme un mini-interpréteur d'expressions mathématiques , voir certaines expressions prenant en charge la notation scientifique. Et cette implémentation est pratiquement basée sur deux parties essentielles qui seront abordés lors de ce rapport, à savoir : l'Analyse Lexicale et l'Analyse Syntaxique.

2 Analyse Lexicale

Dans cette partie, nous avons abordé la première étape d'un compilateur, à savoir l'analyse lexicale qui fait appel à un automate fini déterministe construit à partir d'une expression régulière pour reconnaître les mots du langage. En retour, il génère des jetons (tokens) que l'analyse syntaxique va utiliser. Dans le cours de compilation avec M. Pablo, nous avions appris l'utilisation de Flex qui crée un automate fini déterministe à partir d'une expression régulière. Cette fois-ci, nous avons effectué toutes les étapes de construction de l'automate à la main, et une fois l'automate construit, nous l'avons implémenté.

2.1 Approche

Pour construire l'automate, nous avons suivi les méthodes enseignées en cours. Tout d'abord, nous avons défini une expression régulière à partir d'un alphabet comme suit :

- Alphabet Sigma: {[0-9], ., e, +, -, *, (,)}

Dans la suite, pour faciliter les manipulations, nous avons défini $d = [0 - 9]$ pour représenter les chiffres de 0 à 9 et $op = \{+, *, (,)\}$ pour les opérations arithmétiques.

- Expression régulière : $(d + |d + .d + (|e(+|-)d+)|op)$.

2.2 Construction de l'Automate

À partir de cette expression, nous avons construit l'automate pas à pas en appliquant les opérations d'union, de concaténation et de fermeture itérative. L'automate non déterministe obtenu est présent sur l'image ci-dessous.

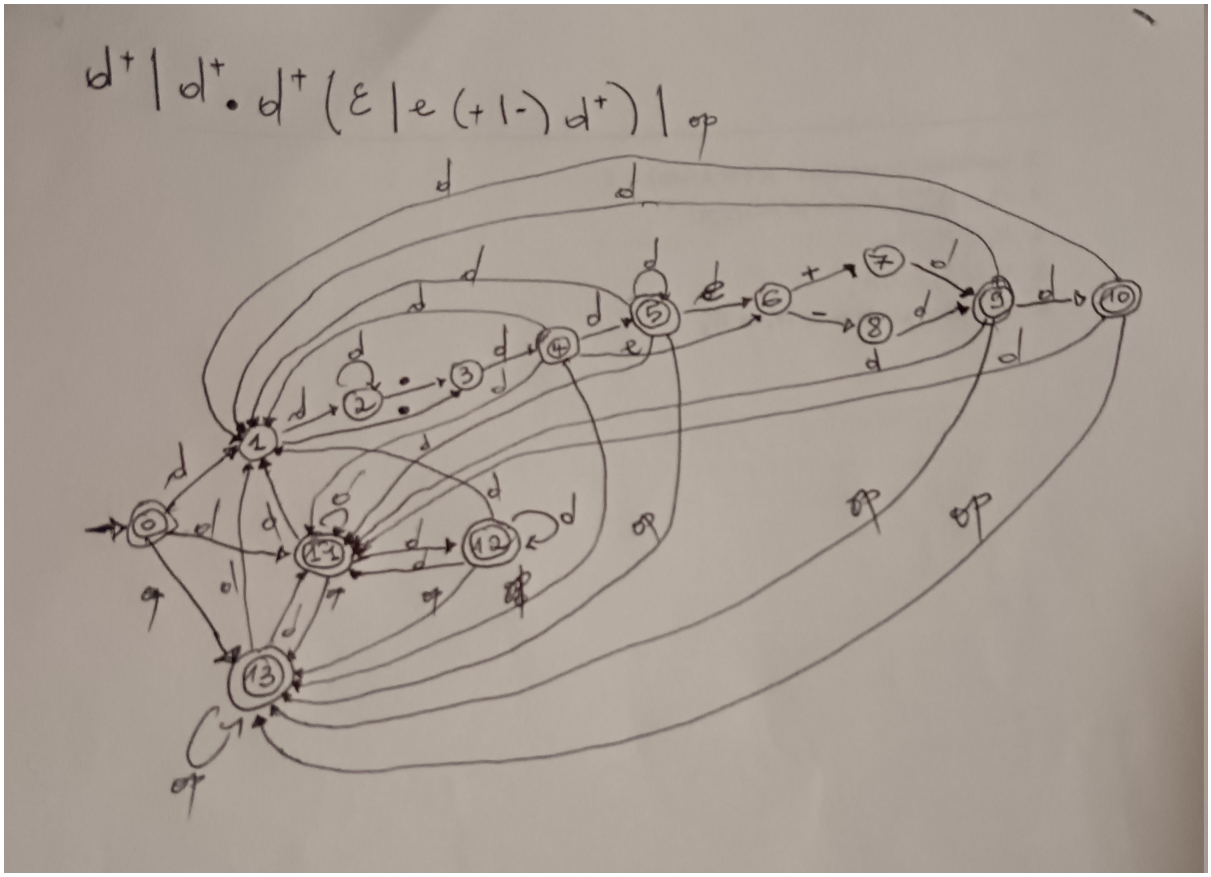

Figure 1: L'automate non déterministe construit.

2.3 La Déterministation

En appliquant l'algorithme de déterminisation, nous avions pu déterminiser notre automate présenté sur la figure ci-dessous. Cependant, il est clair que nous aurions encore pu apporter des améliorations à ce dernier en réduisant le nombre d'états de l'automate en fusionnant certains états.

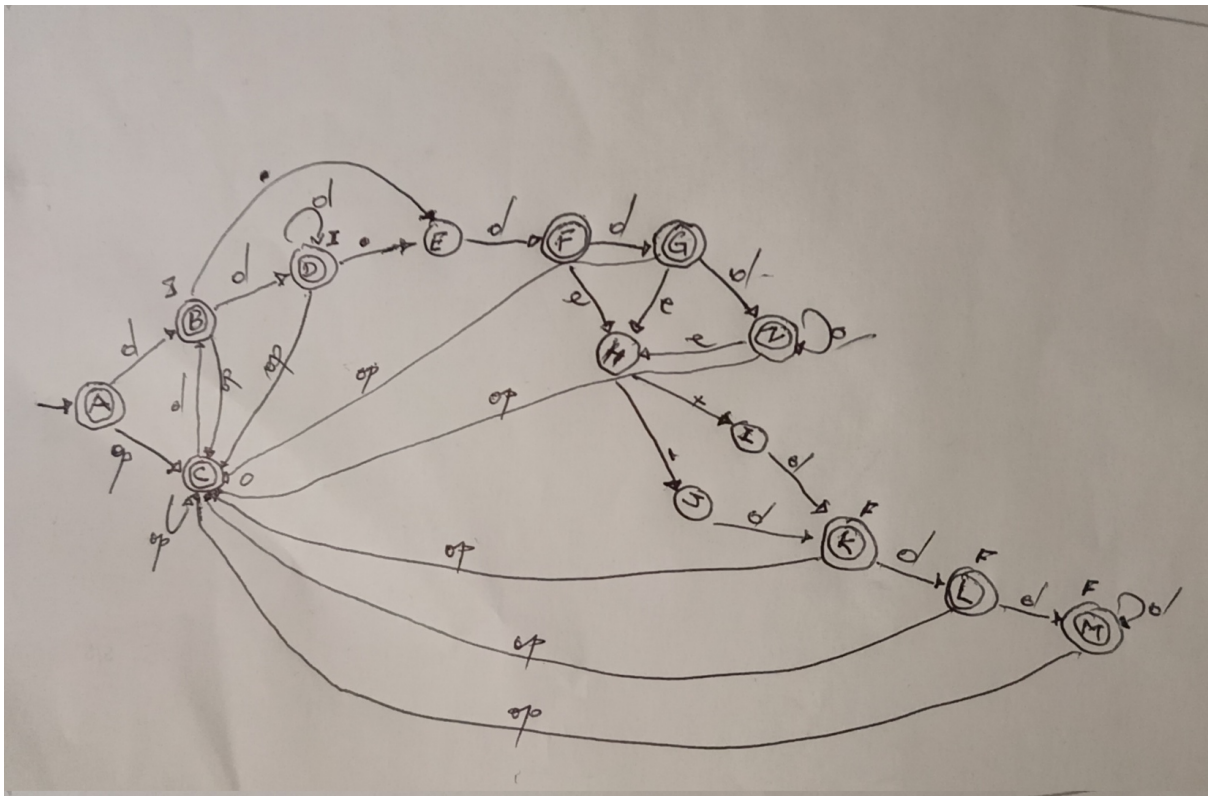

Figure 2: L'automate déterministe construit.

2.4 La Minimisation

L'algorithme de minimisation nous a permis de réduire considérablement le nombre d'états de notre automate, ainsi nous avons obtenu un automate ci-dessous plus simple et surtout plus facile à implémenter.

2.5 Implémentation

Pour implémenter notre automate, nous avions opté pour les structures de données suivantes :

- **Transition :** qui définit les éléments d'une transition, c'est-à-dire l'état courant, le symbole lu et l'état final.
- **AFD (Automate Fini Déterministe) :** qui représente l'automate fini déterministe, comprenant un tableau d'alphabet, un tableau des états, un état initial, un tableau d'états finaux et un tableau de toutes les transitions possibles.

Nous avions ensuite défini deux fonctions principales qui sont :

- **do_transition :** qui prend en paramètre l'état courant, un pointeur sur l'ensemble des transitions et le symbole. Cette fonction nous permet d'effectuer, comme son nom l'indique, une transition sur un symbole et renvoie l'état suivant si la transition est possible. Au cas contraire, elle renvoie '/0'. Maintenant, voyons comment cette fonction effectue réellement la transition. Tout d'abord, comme en C, il n'existe pas de méthode qui renvoie la taille d'un tableau, nous avons donc défini au préalable la taille du tableau de transitions. Ainsi, nous avons défini un compteur pour parcourir les transitions et pour chaque transition, nous vérifions si son début correspond à l'état reçu en argument de la fonction do_transition. Si c'est le cas, nous vérifions ensuite si le symbole de cette

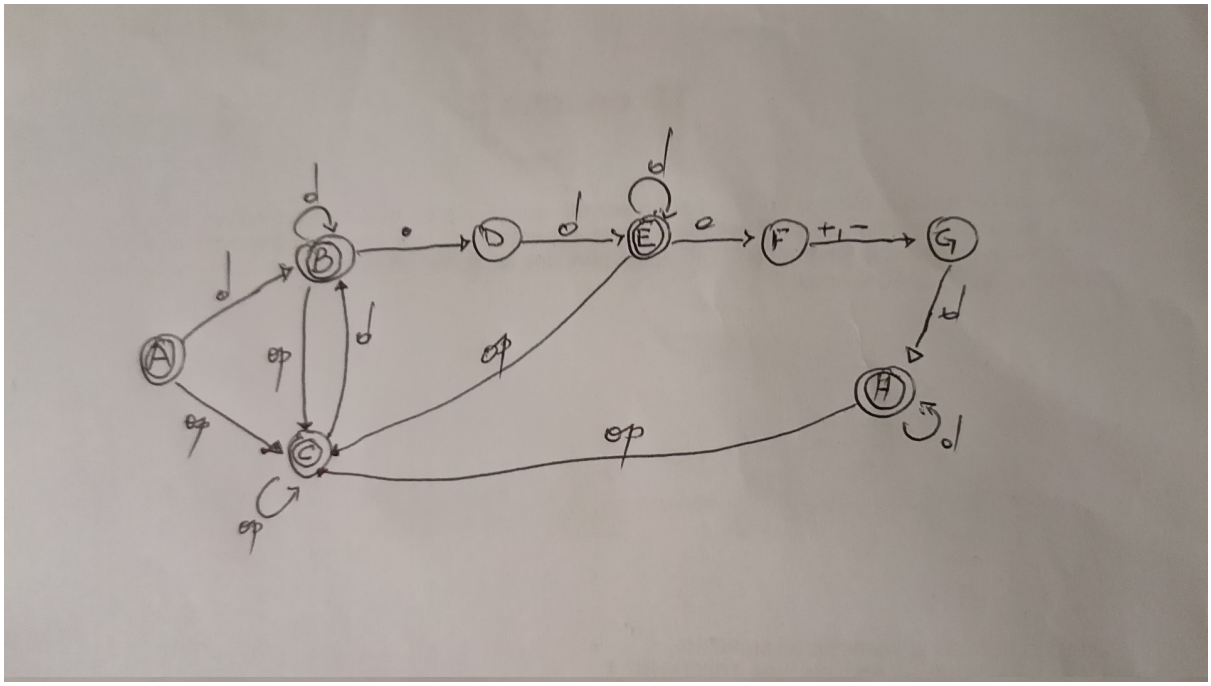

Figure 3: L'automate minimal construit.

transition correspond au symbole reçu en argument. Si les deux conditions sont vérifiées, nous mettons à jour l'état suivant avec l'état final de la transition et nous arrêtons la boucle pour renvoyer cet état.

- **lexical_analysis :** qui prend en paramètre l'AFD et le mot saisi par l'utilisateur, sans oublier qu'au préalable nous effectuons un traitement pour supprimer les espaces. Ainsi, cette fonction est chargée de vérifier si ce mot respecte les syntaxes de notre calculatrice représentée par notre automate fini déterministe (Bien sûr, le bon parenthésage n'est pas géré à ce niveau). Pour cela, nous faisons une boucle pour parcourir le mot et pour chaque caractère, nous faisons appel à la fonction do_transition. En fonction du retour de cette fonction, nous continuons l'exécution. En effet, si la fonction do_transition renvoie une valeur nulle, c'est-à-dire '/0', nous arrêtons en même temps l'exécution et nous signalons que le mot n'est pas accepté par l'automate. Au contraire, nous continuons en prenant le caractère suivant et l'état courant devient l'état renvoyé par la méthode do_transition. À la fin, cette fonction retourne le mot de l'utilisateur sous forme de tokens si le dernier état de l'automate après avoir lu le mot est un état accepteur, et c'est ce tokens qui est passé en argument à la partie analyse syntaxique.

Ces deux fonctions sont les principales pour la partie analyse lexicale. Bien sûr, pour le bon fonctionnement de ces deux fonctions, nous avions implémenté d'autres fonctions secondaires telles que **verify_digit**, **verify_op** et la méthode **contains**, qui sont bien commentées dans le code source.

2.6 Exemple

Afin de comprendre notre implémentation, prenons l'entrée suivante : '3 * 5 + 2'.

1. **Première étape** : Un traitement supprime les espaces, donc à la sortie, nous avons : '35+2'.
2. **Deuxième étape** : Cette entrée ainsi que l'automate sont passés à la fonction **lexical_analysis**.
3. **Troisième étape** : La fonction **lexical_analysis** boucle sur tous les symboles de l'entrée en commençant par '3' à partir de l'état initial de l'automate, c'est-à-dire 'A', en appelant la fonction **do_transition** à chaque fois. L'automate accepte le symbole '3' à partir de l'état 'A' et passe à l'état 'B'. De l'état 'B', il lit " et passe à l'état 'C'. De 'C', il lit '5' et passe à l'état 'B'. De 'B', il lit '+' et passe à l'état 'C'. Enfin, de 'C', il lit '2' et passe à l'état 'B'. Comme 'B' est un état accepteur, il accepte donc l'entrée et la passe sous forme de jetons à l'analyse syntaxique.

3 Analyse Syntaxique

Dans cette partie, on va se concentrer sur la partie syntaxique de notre calculatrice. La partie syntaxique étant responsable de l'interprétation de l'évaluation des expressions et des opérations, une approche intuitive consiste à utiliser une grammaire et éventuellement l'évaluer et vérifier si les expressions saisies par l'utilisateur sont validés ou pas. Dans le cas ou c'est validé, on effectue le traitement nécessaire, dans le cas contraire on renvoie le message d'erreur.

3.1 Grammaire et Ambiguité

En annexe, une grammaire ainsi qu'une table SLR a été donnée dans le but de l'utiliser dans ce projet. Le soucis c'est qu'avant d'implémenter une solution, on constate que cette dernière est ambigue, c'est à dire que la grammaire proposée peut etre exécutée de plusieurs manières différentes (Ce qui n'est pas agréable à voir et risque de causer des conflits plus tard).

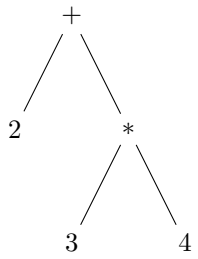

Figure 4: Arbre syntaxique 1

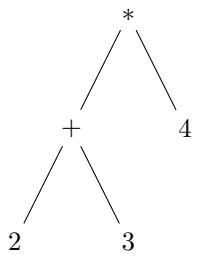

Figure 5: Arbre syntaxique 2

En prenant l'expression 2+3*4 comme exemple, on remarque que nous avons deux arbres de syntaxe. Et donc, dans un premier temps, il faudra lever l'ambiguité de notre grammaire afin d'éviter les mauvaises surprises. Ainsi, tout en prenant en compte des améliorations mentionnés ci-dessus (opération de soustraction '-' et de division '/') , on va se retrouver avec cette grammaire :

0. $S \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow E - T$
3. $E \rightarrow T$
4. $T \rightarrow T * F$
5. $T \rightarrow T/F$
6. $T \rightarrow F$
7. $F \rightarrow (E)$
8. $F \rightarrow \text{id}$

Où :

- Alphabet = (S,E,T,F, id, + , - , *, /).
- id = Nombre (Qui est un symbole terminal) .
- S,E,T,F = Ensemble des symboles non Terminaux .

3.2 Approche LR et Table SLR

Le but principal de l'Approche LR est de concevoir un Automate à Pile à partir d'une Table SLR à construire en construisant leurs composantes canoniques qui est en effet l'ensemble des items et bien évidemment leurs clotures afin de déterminer les états de notre automate, qui nous sera utile pour la gestion de la pile et le flow.

États	+	−	∗	/	(	)	Nombre	$	S	E	T	F
0					s4		s5			1	2	3
1	s6	s7						acc				
2	r3	r3	s8	s9		r3		r3				
3	r6	r6	r6	r6		r6		r6				
4					s4		s5			10	2	3
5	r8	r8	r8	r8		r8		r8				
6					s4		s5				11	3
7					s4		s5				12	3
8					s4		s5					13
9					s4		s5					14
10	s6	s7				s15						
11	r1	r1	s8	s9		r1		r1				
12	r2	r2	s8	s9		r2		r2				
13	r4	r4	r4	r4		r4		r4				
14	r5	r5	r5	r5		r5		r5				
15	r7	r7	r7	r7		r7		r7				

Une fois l'ensemble de ses taches effectués, on passera à l'aspect technique de la chose, qui est quels structures de données manipuler afin de réaliser quelque chose de consistant. (Vous trouverez en Annexe, la construction de l'ensemble des items et suivants pour la construction de notre Table SLR).

3.3 Implémentation

- **Analyseur Syntaxique :**
  - On va utiliser une structure `AnalyseurSyntaxique` pour suivre les jetons d'une expression.
  - Le but de notre fonction `tokenizer` est de séparer une chaîne de caractères en jetons.
  - Et bien sur on utilise un état et une position pour suivre la progression de l'analyse.
- **Arbre d'Expression :**
  - `ArbreExpression` nous sera utile pour représenter les nœuds de l'arbre de l'expression à analyser.
  - Chaque nœud peut être une opération (`+`, `-`, `*`, `/`) ou un nombre.
  - L'arbre est construit pendant l'analyse syntaxique.
- **Table SLR :**
  - On définit nos états et les actions possibles avec des macros (`ACTION_SHIFT`, `ACTION_REDUCE`, `ACTION_ACCEPT`, `GOTO_STATE`, `ERROR_ACTION`).
- **Fonctions Auxiliaires d'Analyse Syntaxique :**
  - `analyserExpressionAdditive`, `analyserExpressionMultiplicative`, `analyserExpressionAtomique`, et `analyserNombre` analysent et construisent différentes parties de l'arbre d'expression selon la nature de l'expression analysée.
- **Fonctions d'Obtention d'Action et de Nouvel État :**
  - `On` retourne l'action à effectuer (décalage, réduction ou acceptation) en fonction de l'état et du jeton.
  - `obtenirNouvelEtat` on renvoie le nouvel état après une action spécifique.
- **Boucle Principale :**
  - La boucle principale analyse en continu des expressions.

- Et à chaque charactère de l'expression, l'analyseur syntaxique va l'interpréter pour obtenir un arbre d'expression.
- Met à jour l'état en fonction de l'action (ça reviens à empiler l'état en question), puis calcule la valeur de l'expression, en fonction de notre table SLR.

- **Logique d'Action (SHIFT, REDUCE, ACCEPT) :**
  - Les actions SHIFT décalent la position de l'analyseur et mettent à jour l'état (ça reviens à empiler l'élément) .
  - Les actions REDUCE effectuent des réductions et construisent l'arbre d'expression (Dépiler l'élément).
  - L'action ACCEPT termine la boucle principale.

- **Libération de Mémoire :**
  - La mémoire allouée pour l'arbre d'expression est libérée à la fin de chaque itération de la boucle principale.

En gros, on aura un pseudo-code qui va ressembler à ça :

Algorithm 1: Analyse Syntaxique LR(1)

```
Data: Expression à analyser
Result: Arbre d'expression résultant
Initialiser la pile avec l'état initial;
Initialiser l'analyseur syntaxique avec l'expression;
Initialiser l'arbre d'expression;
while la pile n'est pas vide et il reste des jetons dans l'expression do
    ÉtatCourant ← Sommet de la pile;
    JetonCourant ← Prochain jeton dans l'expression;
    Action ← obtenirAction(ÉtatCourant, JetonCourant);
    if Action est une action de décalage (shift) then
        Effectuer l'action de décalage;
        Mettre à jour la pile et l'analyseur syntaxique;
    else if Action est une action de réduction (reduce) then
        Effectuer l'action de réduction;
        Construire l'arbre d'expression réduit;
        Mettre à jour la pile;
    else if Action est une action d'acceptation then
        Terminer l'analyse syntaxique avec succès;
if la pile est vide et tous les jetons ont été consommés then
    Retourner l'arbre d'expression résultant;
else
    Signaler une erreur syntaxique;
```

3.4 Exemple

Afin de mieux comprendre notre implémentation, soit l'expression mathématique à analyser : "3 + 5 * 2"**Initialisation**

- La pile est initialisée avec l'état initial (qui est l'état 0).
- L'analyseur syntaxique est initialisé avec l'expression "3 + 5 * 2".
- Un arbre d'expression vide est à priori initialisé.

- **Boucle Principale**

- Lorsque je rentre dans la boucle, la pile va contenir l'état initial et l'analyseur syntaxique pointera vers le début de l'expression.

- **Obtention de l'Action**
  - On utilisera la fonction `obtenirAction` pour déterminer l'action à effectuer. Supposons que l'action soit "Décalage" vers l'état 3 pour le jeton '+'.

- **Décalage (Shift)**
  - Si l'action à effectuer est le "décalage", effectue l'action de décalage en ajoutant l'état 3 à la pile et en déplaçant la position de l'analyseur syntaxique au jeton suivant.

- **Boucle Principale (itération suivante)**
  - On continue avec la pile contenant l'état 3 et l'analyseur syntaxique pointant vers le jeton '+'.

- **Obtention de l'Action**
  - Notre solution aura l'embarras du choix de ce qu'est la prochaine action. à effectuer est une "Réduction" selon la règle par exemple "E → E + T".

- **Réduction (Reduce)**
  - L'algorithme applique la réduction en créant un nœud d'arbre pour l'opération '+' et en mettant à jour la pile en conséquence.

- **Boucle Principale (itération suivante)**
  - La boucle continue avec la pile et l'analyseur syntaxique mis à jour.

- **Étapes suivantes**
  - Le processus se répète en décalant, réduisant et mettant à jour la pile jusqu'à ce que l'expression soit analysée complètement.

- **Scénario Complet**
  - **Étape 1 :** Décalage de "3" (état 3).
  - **Étape 2 :** Réduction de "3" à une expression (par exemple, E → T) et mise à jour de la pile.
  - **Étape 3 :** Décalage de "+" (état 6).
  - **Étape 4 :** Décalage de "5" (état 3).
  - **Étape 5 :** Réduction de "5" à une expression (par exemple, E → T) et mise à jour de la pile.
  - **Étape 6 :** Réduction de l'expression "T + T" à une expression (par exemple, E → E + T) et mise à jour de la pile.
  - **Étape 7 :** Décalage de "*" (état 8).
  - **Étape 8 :** Décalage de "2" (état 3).
  - **Étape 9 :** Réduction de "2" à une expression (par exemple, E → T) et mise à jour de la pile.
  - **Étape 10 :** Réduction de l'expression "T * T" à une expression (par exemple, E → E * T) et mise à jour de la pile.
  - **Étape 11 :** Réduction de l'expression "E + E" à une expression (par exemple, E → E + E) et mise à jour de la pile.
  - **Étape 12 :** Réduction finale de l'expression complète à une expression (par exemple, E → E + E) et mise à jour de la pile.

- **Fin**
  - À la fin de ce processus, l'arbre d'expression doit représenter correctement la structure de l'expression "3 + 5 * 2", et l'évaluation de cet arbre donnerait le résultat attendu.

4 Optimisations

4.1 Limitations de SLR et Analyse IR

Bien que la solution utilisant la Table offre un certain prestige, arrivant au point de frimer, mais cette dernière offre pas mal d'inconvénients qui risquent d'etre couteuse en terme de complexité et de performances. En effet :

- **Taille de la table d'analyse :** Les tables d'analyse SLR peuvent devenir assez surchargées, en particulier pour des grammaires complexes. Cela peut entraîner une utilisation couteuse de la mémoire, ce qui est en soi inconvénient, surtout pour des applications limitées en termes de ressources.
- **Maintenance de la table :** Des modifications dans la grammaire aura comme conséquence directe la nécessité d'effectuer ajustements significatifs dans la table, ce qui peut entraîner des erreurs difficiles à repérer.
- **Flexibilité :** Une table SLR est spécifique à une grammaire particulière, on ne peut pas avoir la meme table SLR pour deux grammaires distinctes. Si la grammaire évolue ou si on veut prendre en considération certains optimisations (à savoir les opérations de soustraction , voir la division)sont nécessaires, la modification de la table risque d'etre couteuse. Ainsi, Une implémentation récursive descendante peut être plus souple et permettre des modifications plus facilement.
- **Performance :** Bien que les analyseurs SLR soient généralement efficaces, dans certains cas, la structure de la table peut entraîner des performances sous-optimales, notamment lorsqu'il y a beaucoup de conflits. Ceci affectera à coup sur la rapidité de l'analyse syntaxique.

  Ceci nous pousse à proposer une autre solution, basée sur La Méthode Récursive Descendante.

4.2 Implémentation

- **Analyseur Syntaxique :**
  - - L'analyseur syntaxique est initialisé avec une expression mathématique sous la forme d'une chaîne de caractères. Cette chaîne est ensuite transformée en un tableau de jetons (tokens) à l'aide de la fonction `tokenizer`.
- **Fonctions d'Analyse des Expressions :**
  - - On commence par la fonction `analyserExpression`, qui fait appel à `analyserExpressionAdditive`. Chaque fonction dans la hiérarchie appelle la fonction inférieure pour traiter des éléments de priorité plus basse (principe de la récursivité).
- **Opérations Additives et Multiplicatives :**
  - - Les opérations d'addition, de soustraction, de multiplication et de division sont gérées dans les fonctions spécifiques `analyserExpressionAdditive` et `analyserExpressionMultiplicative`. Ces fonctions utilisent une approche itérative pour détecter et traiter les opérateurs de manière séquentielle, tout en construisant un arbre d'expression.
- **Expressions Atomiques :**
  - - Les expressions atomiques sont soit des nombres, soit des expressions entre parenthèses. La fonction `analyserExpressionAtomique` prend en charge ces deux cas. Si je détecte une parenthèse ouvrante, on appelle récursivement `analyserExpressionAdditive` pour traiter l'expression entre parenthèses.
- **Analyse des Nombres :**
  - - La fonction `analyserNombre` identifie et lit les chiffres consécutifs dans la séquence de caractères, traitant également la partie décimale et la notation scientifique présente. Elle crée ensuite un nœud d'arbre d'expression de type 'n' représentant le nombre.
- **Construction de l'Arbre d'Expression :**
  - - L'arbre d'expression est construit au fur et à mesure. Les fonctions utilisent la fonction `creerArbreExpression` pour créer des nœuds d'arbre d'expression représentant les opérations et les nombres.

- **Libération de la Mémoire :**
  - - Une fois l'analyse complète réalisée, la fonction `libererArbreExpression` est utilisée pour libérer la mémoire occupée par l'arbre d'expression.
- **Gestion des Erreurs :**
  - - Des mécanismes de gestion des erreurs sont inclus, notamment des vérifications pour s'assurer que les expressions entre parenthèses sont correctes et des messages d'erreur appropriés en cas d'entrée invalide.

4.3 Exemple

- **Expression mathématique :** "3 + 5 * 2"
- **Initialisation :**
  - L'expression est passée à la fonction `analyserExpression("3 + 5 * 2")`.
  - L'analyseur est initialisé avec l'expression et les jetons sont obtenus à l'aide de la fonction `tokenizer`.
- **Analyse de l'Expression Additive :**
  - La fonction `analyserExpressionAdditive` est appelée, démarrant l'analyse additive.
  - La fonction `analyserExpressionMultiplicative` est appelée pour la partie "3", créant un nœud d'expression avec la valeur 3.
  - La position dans les jetons est maintenant sur le signe "+".
- **Opération Additive :**
  - Le `while loop` dans `analyserExpressionAdditive` détecte le signe "+".
  - Le type est défini comme "+".
  - La position dans les jetons est incrémentée.
  - La fonction `analyserExpressionMultiplicative` est appelée pour la partie "5 * 2", créant un nœud d'expression avec la valeur 5, l'opérateur "*", et un autre nœud avec la valeur 2.
  - Un nouveau nœud d'expression additive est créé avec les valeurs obtenues et le type "+".
- **Calcul et Affichage du Résultat :**
  - L'arbre d'expression est passé à la fonction `calculer`.
  - Le résultat du calcul est obtenu récursivement selon la structure de l'arbre.
  - Le résultat final est affiché, et dans ce cas, il devrait être "13".

5 Conclusion

En conclusion, les interpréteurs ont émergé comme des outils essentiels dans notre quotidien, apportant une révolution dans la manière dont nous abordons la programmation et l'informatique. Leur flexibilité, facilité d'utilisation et capacité à exécuter du code source directement ont considérablement simplifié de nombreuses facettes de nos vies numériques.

6 Références Bibliographiques

- Andrew W. Appel, *Modern Compiler Implementation in ML*. Cambridge University Press, 1998. Disponible à : https://www.cs.princeton.edu/ appel/modern/ml/
- Alfred Aho, Jeffrey Ullman, *Principles of Compiler Design*. Manuel classique sur les compilateurs pour les langages de programmation informatique.
- Robert Nystrom, *Crafting Interpreters*. Disponible à : https://craftinginterpreters.com/
- Stanford Online, *Automata Theory I*. Cours en ligne sur la théorie des automates. Disponible à : https://online.stanford.edu/courses/automata-theory-i

7 Annexe - SLR

I	Kernel
I(0,E)	{S → E.; E → E.+ T; E → E.-T}
I(0, T)	{E → T.; T → T.*F; T → T./ F}
I(0, F)	{T → F.}
I(0, ())	{F → (.E)}
I(0, id)	{F → id.}
I(1,+)	{E → E +.T}
I(1, -)	{E → E -.T}
I(2,*)	{T →T *.F}
I(2,/)	{T → T /.F}
I(4,E)	{F → (E.); E → E.+T;E → E.-T}
I(4,T)	{E → T.; T → T.* F; T → T./F}
I(4, F)	{T → F.}
I(4, ())	{F→ (.E)}
I(4, id)	{F→id.}
I(6,T)	{E → E + T.; T → T.*F; T →T./F}
I(6,F)	{T → F.}
I(6,())	{F→ (.E)}
I(6, id)	{F→id.}
I(7, T)	{E → E - T.;T → T.* F; T → T./ F}
I(7, ())	{F → (.E)}
I(7,1d)	{F → id.}
I(8, F)	{T →T*F.}
I(8,())	{F → (.E)}
I(8, id)	{F →id.}
I(9, F)	{T → T / F.}
I(9,())	{F→ (.E)}
I(9,id)	{F→id.}
I(10,))	{F→ (E).}
I(10, +)	{E→ E +.T}
I(10, -)	{E → E -.T}
I(11,*)	{T →T*.F}
I(11,/)	{T→T/F.}
I(12,*)	{T →T*.F}
I(12,/)	{T→T/F.}

Table 1: Table des Items SLR

État	Clôture
0	{S → .E; E → .E + T; E → .E - T; E → .T; T →.T*F; T →.T/F; T → .F; F → .(E); F →.id}
1	{S → E.; E → E.+T; E → E.-T}
2	{E → T.; T → T.*F; T → T./ F}
3	{T→F.}
4	{F → (.E); E → .E + T; E → .E - T; E → .T; T → .T*F; T → .T / F; T → .F; F → .(E); F → .id}
5	{F→id.}
6	{E→ E +.T; T →.T*F;T →.T/F; T → .F;F →.(E);F→.id}
7	{E → E -.T; T → .T *F; T → .T / F; T →.F;F →·(E);F →.id}
8	{T →T *.F; F →.(E);F →.id}
9	{T →T /.F; F → .(E); F →.id}
10	{F → (E.); E → E.+T;E → E.-T}
11	{F→ (E.);E → E.+T; E → E.-T}
12	{E → T.; T → T.*F; T → T./ F}
13	{T→ F.}
14	{T →T*.F}
15	{T→T/F.}

Table 2: Table de Clôture SLR