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

Rapport d'APP

Éléments de compilation
GIF340

Présenté à
Marouane Adnane

Présenté par
Benjamin Chausse – CHAB1704
Shawn Couture – COUS1912

Sherbrooke – 23 juillet 2025

Table des matières

Table des figures

Liste des tableaux

1 Conception de l'analyseur lexical

1.1 Définir les unités lexicales à l'aide d'expressions régulières et d'automates

1.1.1 Littéraux

Les littéraux sont des unités lexicales comme des nombres, des valeurs booléennes, etc. C'est une valeur fixe directement écrite dans le code source d'un programme. Or, la tâche demandée nécessite uniquement la détection de nombres entiers sans exposants. L'alphabet des littéraux contient alors uniquement les chiffres de 0 à 9.

1.1.1.1 Expression régulière

$$\sum_{\text{literal}} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \tag{1.1}$$

$$\text{Expression} = \sum_{\text{literal}}^{+} \tag{1.2}$$

L'expression régulière suivante présente celle d'en haut d'une façon plus régulière :

$$\text{Expression} = (0|1|2|3|4|5|6|7|8|9)^{+} \tag{1.3}$$

L'automate de la figure A-4 est une partie de l'automate complet de l'analyseur lexical. Il fonctionne uniquement pour les littéraux numériques entiers.

1.1.2 Opérateurs

Dans le contexte du langage de la problématique, les opérateurs sont des unités lexicales permettant d'exécuter une évaluation arithmétique entre un nombre à gauche de l'unité et un autre à droite. Les deux nombres étant des unités de type littéral. Les opérateurs reconnus sont l'addition, la soustraction, la multiplication et la division. Aucun opérateur n'est composé de plusieurs caractères.

1.1.2.1 Expression régulière

$$\sum_{\text{opérateurs}} = \{+, -, *, /\} \tag{1.4}$$

$$\text{Expression} = \sum_{\text{opérateurs}} \tag{1.5}$$

De façon plus lisible, cette expression peut être présentée comme suit. Notez que les apostrophes sont là uniquement pour indiquer qu'il s'agit de caractères littéraux et non de symboles d'expression régulière.

$$\text{Expression} = +|-|*|/ \tag{1.6}$$

L'automate de la figure A-5 est une partie de l'automate complet de l'analyseur lexical. Il fonctionne uniquement pour les opérateurs à symbole unique.

1.1.3 Identificateurs

Dans le contexte du langage de la problématique, deux types d'opérandes sont possibles : les littéraux ou les chaînes de caractères. Ces dernières sont formellement appelées "identificateurs". Ils sont valides uniquement s'ils commencent par une lettre majuscule. Ils peuvent contenir des lettres minuscules et majuscules ainsi que des underscores. Cependant, ils ne peuvent pas se terminer par un underscore et il ne peut pas y avoir deux underscores consécutifs. Ils ont une longueur minimale d'un caractère.

1.1.3.1 Expression régulière

Par souci de simplicité, l'alphabet n'est pas mentionné. Les expressions suivantes présentent l'expression régulière pour la détection d'identificateurs valides :

$$\text{Maj} = (A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z) \tag{1.7}$$

$$\text{Min} = (a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z) \tag{1.8}$$

$$\text{Lettres} = \text{Maj}|\text{Min} \tag{1.9}$$

Il serait plus simple de les écrire en *Regex* comme suit, mais ce n'est pas demandé :

$$\text{Maj} = [\text{A-Z}] \tag{1.10}$$

$$\text{Min} = [\text{a-z}] \tag{1.11}$$

$$\text{Lettres} = [\text{A-Za-z}] = [\text{A-z}] \tag{1.12}$$

L'expression est donc la suivante :

$$\text{Expression} = \text{Maj}(\_?\text{Lettres}^{+})* \tag{1.13}$$

Cela signifie : une lettre majuscule suivie de zéro ou plusieurs occurrences d'un underscore optionnel suivi d'une ou plusieurs lettres.

L'automate de la figure A-3 est une partie de l'automate complet de l'analyseur lexical. Il fonctionne uniquement pour les identificateurs.

1.1.4 Délimiteurs

Par nature, un délimiteur est une unité lexicale qui délimite des portions de programme. Dans le contexte du langage de la problématique, les délimiteurs sont uniquement les parenthèses standards. L'automate et l'expression régulière seront donc identiques à ceux des opérateurs. L'alphabet change pour ne contenir que les parenthèses.

1.1.4.1 Expression régulière

$$\sum_{\text{délimiteurs}} = \{(,)\} \tag{1.14}$$

$$\text{Expression} = \sum^{\text{délimiteurs}} \tag{1.15}$$

L'expression régulière suivante présente celle d'en haut d'une façon plus régulière :

$$\text{Expression} = \text{'('|')'} \tag{1.16}$$

L'automate de la figure A-2 est une partie de l'automate complet de l'analyseur lexical. Il fonctionne uniquement pour les délimiteurs.

1.2 Classes Java qui implémentent l'analyse lexicale

Pour effectuer l'analyse lexicale, des classes déjà existantes du projet de départ ont été utilisées. `AnaLex` effectue l'analyse lexicale à partir d'une expression donnée et implémente les automates définis plus haut. `Terminal` a été utilisée comme classe représentant une unité lexicale et contient donc son type et son lexème. `AnaLex` construit une liste d'unités lexicales, qui sont en fait des objets `Terminal`. Ensuite, une classe statique d'alphabet a été ajoutée au projet, permettant d'utiliser des fonctions qui vérifient si une lettre appartient à certains alphabets comme celui d'un opérateur, du début d'un identificateur, du corps d'un identificateur, etc. Cette dernière est utilisée pour éviter de coder directement les chaînes de caractères dans `AnaLex`. Afin d'éviter d'alourdir inutilement ce rapport, le code est omis puisqu'il est déjà fourni dans la remise.

2 Structures de données d'arbres syntaxiques abstraits

Un arbre syntaxique abstrait, d'un point de vue programmation, n'est qu'un arbre binaire dont les différents membres stockent des informations sur eux- mêmes. Pour plus de détails sur l'implémentation, le code Java est fourni en annexe du rapport, mais n'est pas transcrit textuellement pour garder le rapport concis.

2.1 ElemAST (abstraite)

Tout élément d'un arbre fait partie de cette classe. Cela permet d'agglomérer et d'imposer, si nécessaire, certaines fonctions telles que `toString`, `asPostFix`, etc. De plus, un nœud ne sait pas si ses branches contiendront d'autres nœuds ou des feuilles. Cette classe permet de faire abstraction de cette distinction.

2.2 NoeudAST

Un nœud contient les informations suivantes :

— `type` : (toujours un opérateur dans notre cas, mais pourrait être un appel à une fonction dans une grammaire différente)

— `lexeme` : permet d'évaluer l'expression. Omettre un lexème comme `*` rendrait impossible de connaître l'opération désirée lors de l'évaluation.

— `left` : contient le terme gauche du nœud (feuille ou un autre nœud)

— `right` : contient le terme droit du nœud (comme `left`)

2.3 FeuilleAST

Une feuille contient les informations suivantes :

— `type` : litéral ou identificateur dans la grammaire actuelle, mais pourrait contenir autre chose comme booléen, flottant, etc.

— `lexeme` : tout comme dans le nœud, il serait impossible d'évaluer l'arbre par la suite s'il n'était pas présent.

3 Conception de l'analyseur syntaxique par la méthode descendante

3.1 Principes de fonctionnement

3.1.1 Analyseur descendant

Un analyseur syntaxique descendant construit l'*AST* à partir d'un symbole de départ. Il dérive l'entrée de gauche à droite ou de droite à gauche en appliquant des règles de grammaire. Ces règles permettent de choisir le bon prochain symbole ou la bonne combinaison de symboles. Une méthode peut implémenter des *lookahead tokens* pour déterminer les symboles suivants et choisir la règle de production adéquate. L'analyseur essaie plusieurs règles de production jusqu'à ce que la chaîne d'entrée soit analysée ou qu'une erreur soit détectée.

3.2 Analyseur LL

Le premier L de la méthode LL indique que l'entrée est lue de gauche à droite. Le second L signifie que les dérivations sont effectuées à gauche. Dans l'énoncé de la problématique, on indique d'utiliser LL(1). Le nombre entre parenthèses indique la quantité de *lookahead tokens* disponibles. Dans ce cas, un seul. Cela signifie que les fonctions d'analyse ne peuvent lire que le caractère actuel et un seul suivant.

La méthode LL est prédictive. Elle anticipe la règle à utiliser et utilise des appels récursifs aux règles de production.

L'analyseur commence à gauche de l'expression, avec le symbole de départ. Il regarde le caractère suivant, et s'il peut répondre à une règle avec ce caractère, il crée un nœud d'arbre avec deux symboles ; sinon, il passe au suivant. Plus loin dans ce document, la grammaire utilisée démontre ce principe.

3.3 Catégorie LL la plus utilisée en pratique

La méthode la plus utilisée en pratique est `LL(1)`. Cela s'explique par sa simplicité d'implémentation, souvent via une descente récursive comme celle de l'APP. Elle est suffisante pour de nombreux langages de programmation à la syntaxe robuste.

Des méthodes beaucoup plus complexes existent pour traiter des syntaxes ambiguës ou des grammaires complexes.

3.4 Type de grammaires applicables

Les grammaires pour la méthode `LL(1)` ne doivent pas contenir d'ambiguïtés. Étant donné que la lecture est de gauche à droite, il ne doit pas y avoir de récursivité à gauche, seulement à droite. De plus, les ensembles de premiers et de suivants doivent être disjoints. Les grammaires dites "LL(1)" sont utilisées pour cette méthode. Les règles de production doivent aussi être associatives à droite, car il n'est pas possible de vérifier l'ensemble de la règle avec un seul *lookahead token* dans le cas d'une associativité à gauche.

3.5 Définition formelle de la syntaxe d'expressions arithmétiques

Cette section présente la grammaire utilisée pour résoudre des équations arithmétiques sans nombres à virgule, sans nombres négatifs, avec parenthèses. La priorité des opérations stipule que la division et la multiplication ont la même priorité, supérieure à celle de l'addition et la soustraction.

Comme la méthode doit construire à gauche, les règles de production récursives sont réécrites pour une récursivité à droite. La construction descendante nécessite une définition allant du moins prioritaire au plus prioritaire.

$$E \rightarrow \text{T} \mid [\ +\text{E} \mid \text{-E}\ ] \tag{3.1}$$

$$T \rightarrow \text{P} \mid [\ *\text{T} \mid /\text{T}\ ] \tag{3.2}$$

$$P \rightarrow (\ \text{E}\ ) \tag{3.3}$$

$$P \rightarrow \text{délimiteur} \mid \text{littéral} \tag{3.4}$$

3.6 Exemples de séquences de dérivations

L'exemple suivant montre une séquence de dérivations d'une expression arithmétique à l'aide de la grammaire définie ci-dessus :

$$1 + 2 * (3/5) - 6 \tag{3.5}$$

1. E
2. $T + E$
3. $P + E$
4. $l + E$
5. $l + T$
6. $l + P * T$
7. $l + l * T$
8. $l + l * P$
9. $l + l * (E)$
10. $l + l * (T)$
11. $l + l * (P/T)$
12. $l + l * (l/T)$
13. $l + l * (l/P)$
14. $l + l * (l/l)$
15. $l + l * (l/lE$
16. $l + l * (l/lT - E$
17. $l + l * (l/lP - E$
18. $l + l * (l/l) - E$
19. $l + l * (l/l) - T$
20. $l + l * (l/l) - P$
21. $l + l * (l/l) - l$

Lorsqu'il n'est plus possible d'appliquer les règles de production de la grammaire, et qu'il reste des unités lexicales, l'analyseur recommence à partir du caractère actuel. C'est pour cela qu'un symbole E apparaît à la position de la parenthèse fermante. La fonction P s'est assurée qu'une parenthèse fermante suivait l'appel de E, puis l'analyse est remontée et a repris à partir de ce point.

4 Validation et tests

Tableau 4-1 – Tests d'analyse lexicale

Objectif Ciblé			Test des nouvelles opérations
# Test	**Entrée**	**Résultat Attendu**	**Résultat Obtenu**
1	`(U_x + V_y ) * W_z / 35`	Les unités sont présentées sur une ligne contenant le type et le contenu du lexème. Le résultat attendu est : — Délimiteur : `(` — Identificateur : `U_x` — Opérateur : `+` — Identificateur : `V_y` — Délimiteur : `)` — Opérateur : `*` — Identificateur : `W_z` — Opérateur : `/` — Littéral : `35`	Le programme répond à ce critère avec une présentation légèrement différente. Toutefois, les données sont présentes et dans le bon ordre.
2	`(U_x + V_y ) * W__z / 35`	Même unités que le premier test. Toutefois, une erreur est donnée, indiquant que des underscores consécutifs ont été trouvés. Les unités affichées s'arrêtent à l'opérateur de multiplication.	Le programme a donné une erreur et a retourné les unités lexicales jusqu'à l'opérateur de multiplication.
3	`&`	Une erreur de caractère invalide est attendue.	Le programme identifie le caractère invalide.
4	`aA_z`	Une erreur est attendue indiquant qu'un identificateur ne peut pas commencer par une lettre minuscule.	Une erreur d'identificateur est donnée indiquant que le nom ne peut pas commencer par une lettre minuscule.
5	`Aa_`	Une erreur est attendue, indiquant qu'un identificateur ne peut pas se terminer par un underscore.	Une erreur de fin d'identificateur a été donnée, indiquant qu'il ne peut pas se terminer par un underscore.

Tableau 4-2 – Tests d'analyse syntaxique

Objectif Ciblé			Test des nouvelles opérations
# Test	**Entrée**	**Résultat Attendu**	**Résultat Obtenu**
6	`+ 2 - 3`	Une expression ne peut pas commencer par un opérateur. Une erreur devrait survenir.	Voir l'erreur C.1
7	`2 - 3 *`	Un opérateur est utilisé sans opérande à droite. Une erreur devrait survenir.	Voir l'erreur C.2

Continué à la prochaine page

Tableau 4-2 – Continué de la page précédente...

# Test	**Entrée**	**Résultat Attendu**	**Résultat Obtenu**
8	`1 +/ 2 - 3`	Deux opérateurs consécutifs sont invalides. Une erreur devrait survenir.	Voir l'erreur C.3
9	`(1 + 2 - 3`	Une parenthèse ouverte n'a pas été fermée. Une erreur devrait survenir.	Voir l'erreur C.4
10	`1 + 2) - 3`	Une parenthèse fermante apparaît alors qu'aucune n'a été ouverte. Une erreur devrait survenir.	Voir l'erreur C.5
11	`1 + 2 - () / 3`	Les parenthèses doivent contenir une expression valide. Une erreur devrait survenir.	Voir l'erreur C.6

Des tests syntaxiques avec une syntaxe valide n'ont pas été effectués puisque tous les tests de construction d'AST les emploient (pré-requis pour ce type de test).

Tableau 4-3 – Tests de construction d'arbres syntaxiques abstraits

Objectif Ciblé			Test des nouvelles opérations
# Test	**Entrée**	**Résultat Attendu**	**Résultat Obtenu**
12	`( U_x - V_y ) * W_z / 35`	Sans les parenthèses, l'opérateur de soustraction serait au sommet de l'arbre (voir figure B-2). Ce test montre que la priorité des opérations est respectée et que les parenthèses modifient bien la structure pour obtenir ce qui est observé à la figure B-1.	Représentation textuelle du JSON illustré à la figure B-1. Elle confirme nos attentes.
13	`1+2-3`	Comme la grammaire est associative à droite pour l'addition et la soustraction, l'opération au sommet de l'arbre doit être la soustraction, placée à droite.	Représentation textuelle du JSON illustré à la figure B-3. Elle confirme nos attentes.
14	`1*2/3`	La grammaire étant associative à droite pour la multiplication et la division, l'opération au sommet de l'arbre doit être la division, située à droite.	Représentation textuelle du JSON illustré à la figure B-3. Elle confirme nos attentes.

A Machines à état finie

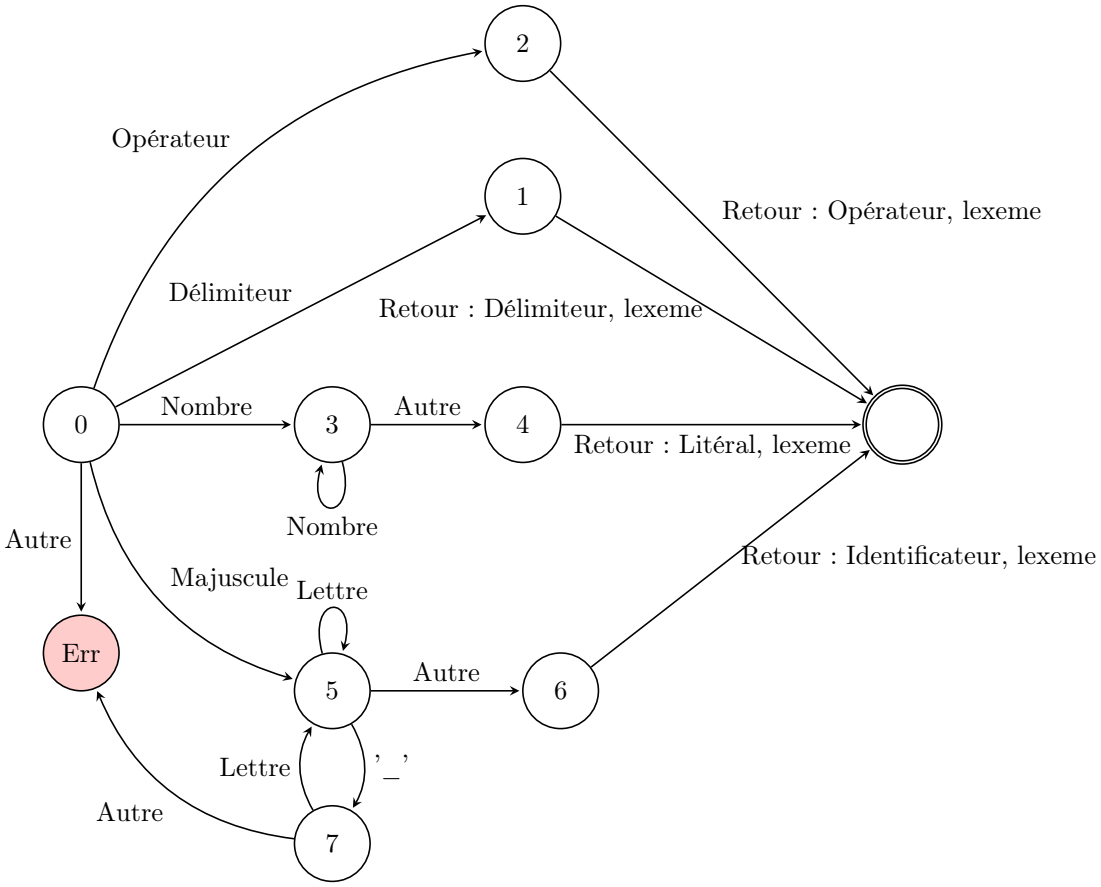

FIGURE A-1 – Mise en unités lexicales

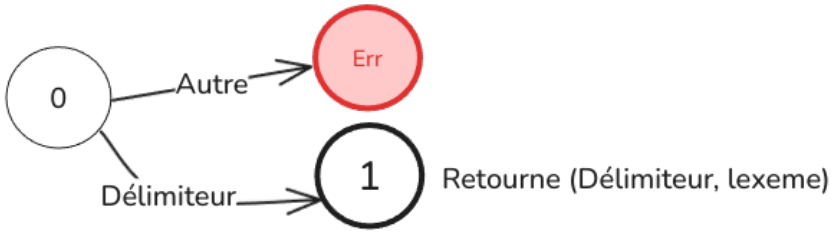

FIGURE A-2 – MEF de détection d'un délimiteur

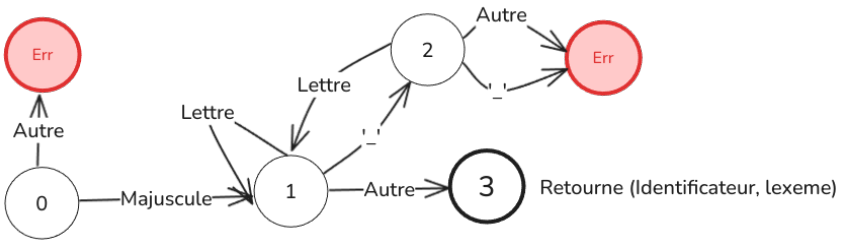

FIGURE A-3 – MEF de détection d'un identificateur

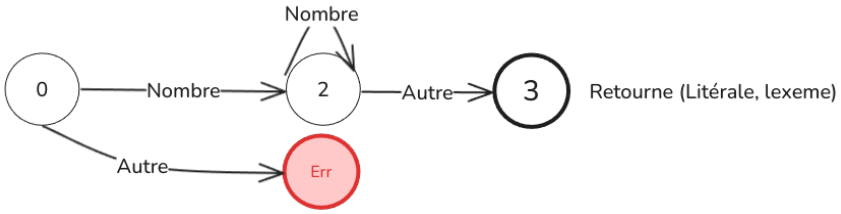

FIGURE A-4 – MEF de détection d'un litéral

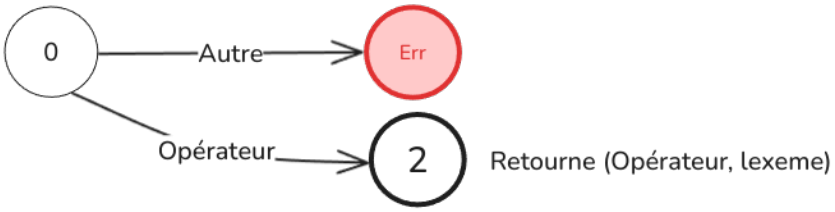

FIGURE A-5 – MEF de détection d'un opérateur

B Arbres syntaxiques

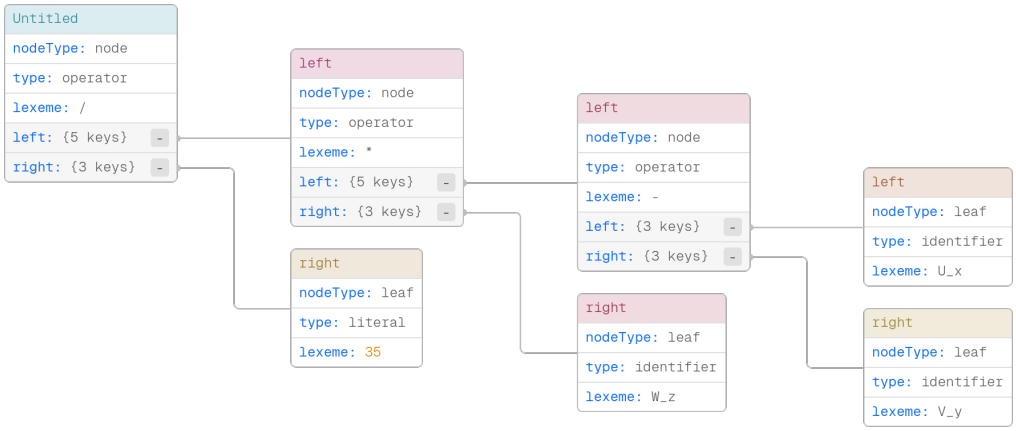

FIGURE B-1 – Représentation graphique du json généré par le test #12

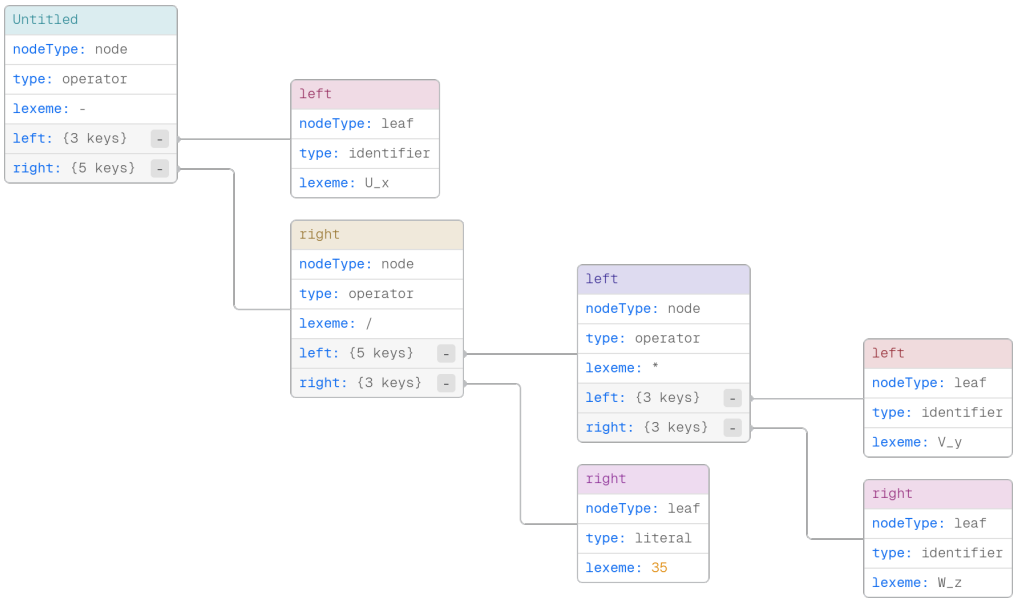

FIGURE B-2 – Représentation graphique du json généré si le test #12 n'avait pas de parenthèses

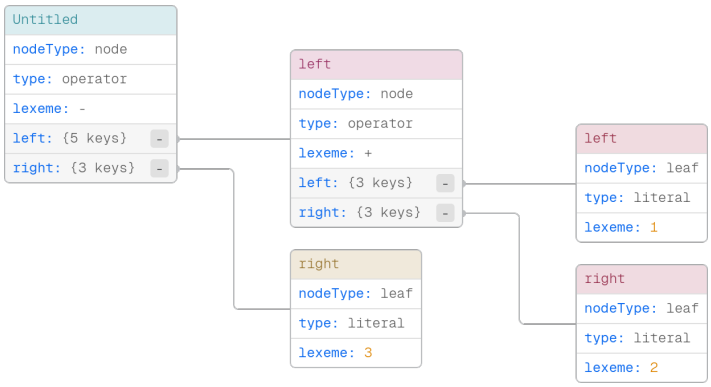

FIGURE B-3 – Représentation graphique du json généré par le test #13

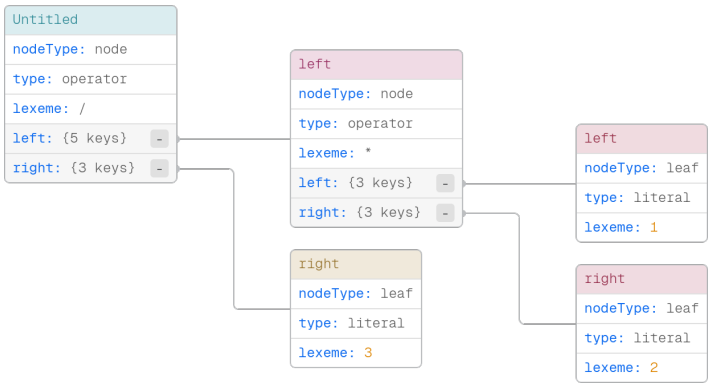

FIGURE B-4 – Représentation graphique du json généré par le test #14

C Messages d'erreurs

Erreur C.1 Message d'erreur issu du test #6

```
app6.SyntaxException: Syntax Error at position 0: Unexpected token
  Expected: literal, identifier, or '('
  Found: +
  Context: +2-3
```

Erreur C.2 Message d'erreur issu du test #7

```
app6.SyntaxException: Syntax Error at position 3: Expected operand after operator
  Expected: number or expression
  Found: end of input
  Context: *
```

Erreur C.3 Message d'erreur issu du test #8

```
app6.SyntaxException: Syntax Error at position 2: Unexpected token
  Expected: literal, identifier, or '('
  Found: /
  Context: /2-3
```

Erreur C.4 Message d'erreur issu du test #9

```
app6.SyntaxException: Syntax Error at position 6: Expected closing parenthesis
  Expected: ')'
  Found: end of input
```

Erreur C.5 Message d'erreur issu du test #10

```
app6.SyntaxException: Syntax Error at position 3: Failed to parse complete
expression - unexpected tokens remain
  Found: )
  Context: ) - 3
```

Erreur C.6 Message d'erreur issu du test #11

```
app6.SyntaxException: Syntax Error at position 5: Expected opening parenthesis
  Expected: '('
  Found: )
  Context: )/3
```