Projet de Conception et de Langage (PCL)

YASAR Hazar, PLISSON Clément, LIENARD Nathan,
GEORGEON Gautier

January 21, 2025

Contents

1 Gestion de projet

1.1 Grammaire

- Clément : Transformation de la grammaire de base en une grammaire LL(1) (4h)
- Nathan : Travail sur l'élaboration de la grammaire finale (gérer la factorisation, l'ambiguïté ou encore la récursivité) + configuration de Gradle pour le projet (4h et 1h)
- Hazar : Travail sur la transformation de la grammaire de base en une grammaire LL(1) (4h)
- Gautier : Aide lors des réflexions sur la grammaire (1h)

1.2 Lexeur

- Nathan : Travail sur la structure de données utilisée pour représenter les tokens en ajoutant les lignes en erreur et messages différenciés en fonction de l'erreur + tests unitaires (3h chacun)
- Gautier : Réalisation de l'automate puis création du lexeur à partir de celui-ci (10h)
- Clément : Création d'un premier lexeur pour commencer à réflechir à l'approche utilisée (3h)

1.3 Parseur

- Hazar : Réalisation du parseur afin d'effectuer l'analyse sémantique (10h)
- Clément : Reprise du code d'Hazar pour y intégrer les fonctions sémantiques qui créent l'AST (6h)
- Nathan : Tests unitaires sur les deux parseurs, adaptation des fichiers tests en fonction de l'évolution des parseurs, construction de différents scénarios de tests pour les règles de la grammaire et ajustements dans le code en fonction des résultats (4h)
- Gautier : Reprise telle quelle de la méthode itérative d'analyse descendante pour faire un parseur itératif afin de tester le parseur récursif (8h)

1.4 AST (Abstract Syntax Tree)

- Clément : Création de la structure de l'AST, décomposée en plusieurs classes selon la grammaire de base. Et implémentation des méthodes de simplification et de rotation à gauche, principalement pour les expressions arithmétiques, et de la methode de visualisation de l'AST (4h)
- Hazar : Suppressions des branches inutiles afin d'avoir un AST contenant seulement les informations necéssaires (3h)
- Gautier : Création d'un arbre non simplifié à partir du parseur itératif (2h)

2 Grammaire

2.1 Grammaire de base

```
<file> := NEWLINE? <def>* <stmt>+ EOF

<def> := def <ident> ( <ident>*, )  :  <suite>

<suite> := <simple_stmt> NEWLINE
         | NEWLINE BEGIN <stmt>+ END

<simple_stmt> := return <expr>
               | <ident> = <expr>
               | <expr> [ <expr> ] = <expr>
               | print ( <expr> )
               | <expr>

<stmt> := <simple_stmt> NEWLINE
        | if <expr> : <suite>
        | if <expr> : <suite> else : <suite>
        | for <ident> in <expr> : <suite>

<expr> := <const>
        | <ident>
        | <expr> [ <expr> ]
        | - <expr>
        | not <expr>
        | <expr> <binop> <expr>
        | <ident> ( <expr>*, )
        | [ <expr>*, ]
        | ( <expr> )

<binop> := + | - | * | // | % | <= | >= | > | < | != |
    == | and | or

<const> := <integer> | <string> | True | False | None
```

2.2 Introduction des priorités des opérateurs

```
<expr> := <or_expr>

<or_expr> := <and_expr>+or
```

```
<and_expr> := <not_expr>+and
<not_expr> := not? <comp_expr>
<comp_expr> := <add_expr> (<binop_comp> <add_expr>)?
<add_expr> := <mut_expr>+<binop_add>
<mut_expr> := <minus_expr>+<binop_mut>

<minus_expr> := -? <crochet_expr>

<crochet_expr> := <terminal_expr> ( [ <expr> ] )*

<terminal_expr> := <const>
                 | <ident>
                 | <ident> ( <expr>*, )
                 | [ <expr>*, ] | ( <expr> )

<binop_comp> := <= | >= | > | < | != | ==
<binop_add> := + | -
<binop_mut> := * | // | %
```

2.3 Explicitation des règles *, + et ?

```
<exemple>+
    <exemple_plus> := <exemple> <exemple_plus_rest>
    <exemple_plus_rest> := ε | <exemple_plus>

<exemple>*
    <exemple_etoile> := ε | <exemple> <exemple_etoile>

<exemple>+symbole
    <exemple_plus_symbole> := <exemple> <
   exemple_plus_symbole_rest>
    <exemple_plus_symbole_rest> := ε | symbo<
   exemple_plus_symbole>

<exemple>*symbo
    <exemple_etoile_symbole> := ε | <
   exemple_plus_symbole>

<exemple>?
    <exemple> := ε | exemple*
```

2.4 Factorisation des règles

```
<stmt> := ...
        | if <expr> : <suite>
        | if <expr> : <suite> else : <suite>
        | ...
```

devient :

```
<stmt> := if <expr> : <suite> <stmt_else>
<stmt_else> := ε | else : <suite>
```

```
<terminal_expr> := ...
                 | <ident>
                 | <ident> ( <expr>*, )
                 | ...
```

devient :

```
<terminal_expr> := <ident> <ident_fact>
<expr_rest_ident> := ε | ( <expr>*, )
```

2.5 Ambiguïté avec <simple_stmt>

On a :

```
<simple_stmt> := ...
               | <expr> [ <expr> ] = <expr>
               | <expr>
               | <ident> = <expr>
               | ...
```

Quel règle choisir quand on lit un identificateur ?

Le plus simple serait de remplacer <expr> [<expr>] et <ident> par <expr>.

Problème : on élargit la grammaire. On peut maintenant écrire *<string>* = *<integer>* par exemple.

Supposons que ce problème n'en est pas un, on a alors

```
<simple_stmt> := ...
               | <expr> = <expr>
               | <expr>
               | ...
```

Que l'on peut factoriser, tel que :

```
<simple_stmt> := ...
                | <expr> <simple_stmt_fact>
                | ...
<simple_stmt_fact> := ε | = <expr>
```

La grammaire devient alors LL1.

Revenons sur notre problème. Il n'y a pas de restrictions s'il y a juste <expr>, mais il y en a s'il s'agit d'une affectation <expr>= <expr>, auquel cas membre gauche ne peut être qu'un identificateur, ou un accès via indice.

Il faudrait donc pouvoir regarder après l'analyse du premier <expr>.
Or une implémentation telque : une règle, une méthode/fonction, permet un "stockage de cette information" dans la pile d'execution.

En effet, on lance l'analyse de la première expression en appelant analyseExpr().

Quand cette méthode termine, on revient dans corps de la fonction appelante.

Alors on a une disjonction de cas sur prochain token :

- Si prochain token est un =, alors
  - Si l'expression analysée est un accès par indice ou un identificateur, alors pas de problème,
  - Sinon renvoyer une erreur.
- Sinon, retourner l'expression analysée.

2.6 Grammaire finale

```
<file> := <opt_newline> <def_etoile> <stmt_plus> EOF

<opt_newline> := ε | NEWLINE

<def_etoile> := ε | <deft> <def_etoile>
<deft> := <def> <ident> ( <ident_etoile_virgule> ) : <
   suite>

<stmt_plus> := <stmt> <stmt_plus_rest>
<stmt_plus_rest> := ε | <stmt_plus>

<ident_etoile_virgule> := ε | <ident_plus_virgule>
```

```
<ident_plus_virgule> := <ident> <
   ident_plus_virgule_rest>
<ident_plus_virgule_rest> := ε | , <ident_plus_virgule
   >


<suite> := <simple_stmt> NEWLINE
         | NEWLINE BEGIN <stmt_plus> END

<simple_stmt> := <return> <expr>
               | <print> ( <expr> )
               | <expr> <simple_stmt_fact>
<simple_stmt_fact> := ε | = <expr>

<stmt> := <simple_stmt> <NEWLINE>
        | <for> <ident> <in> <expr> : <suite>
        | <if> <expr> : <suite> <stmt_else>
<stmt_else> := ε | <else> : <suite>

<expr> := <or_expr>
<or_expr> := <and_expr> <or_expr_rest>
<or_expr_rest> := ε | <binop_or> <or_expr>

<and_expr> := <not_expr> <and_expr_rest>
<and_expr_rest> := ε | <binop_and> <and_expr>

<not_expr> := <comp_expr>
<not_expr> := <not> <comp_expr>

<comp_expr> := <add_expr> <comp_expr_rest>
<comp_expr_rest> := ε | <binop_comp> <add_expr>

<add_expr> := <mut_expr> <add_expr_rest>
<add_expr_rest> := ε | <binop_add> <add_expr>

<mut_expr> := <minus_expr> <mut_expr_rest>
<mut_expr_rest> := ε | <binop_mut> <mut_expr>

<minus_expr> := - <crochet_expr>
<minus_expr> := <crochet_expr>

<crochet_expr> := <terminal_expr> <expr_crochet_etoile
   >
<expr_crochet_etoile> := ε | [ <expr> ] <
   expr_crochet_etoile>
```

```
<terminal_expr> := ( <expr> )
                 | <const>
                 | <ident> <expr_rest_ident>
                 | [ <expr_etoile_virgule> ]
<expr_rest_ident> := ε | ( <expr_etoile_virgule> )


<expr_etoile_virgule> := ε | <expr_plus_virgule>

<expr_plus_virgule> := <expr> <expr_plus_virgule_rest>
<expr_plus_virgule_rest> := ε | , <expr_plus_virgule>

<binop_add> := + | -
<binop_mut> := * | // | %
<binop_comp> := <= | >= | > | < | != | ==
<binop_and> := and
<binop_or> := or
<const> := <integer> | <string> | True | False | None
```

La grammaire finale est LL(1), mais recursive droite, donc associative droite.

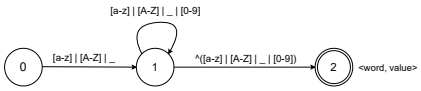
3 Lexeur

Le lexeur est la traduction d'un automate reconnaissant les tokens. Il fonctionne de façon récursive en travaillant toujours sur un seul caractère à la fois. Il est basé sur l'automate suivant:

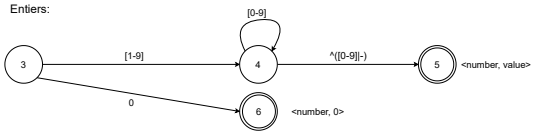
Précisions lexicales:
- word: futur token id ou keyword
- number: entier
- string: chaîne de caractères
- op: opérateur de calcul
    - LP: left parenthesis
    - RP: right parenthesis
    - LB: left bracket
    - RB: right bracket
    - MOD: modulo
    - DD: double dot
    - COM: comma
- relop: opérateur logique
    - LE: less or equal
    - GE: greater or equal
    - LT: less than
    - GT: greater than
    - EQ: equal
    - NE: not equal
- ws: indentations et retours chariot
    - NEWLINE, BEGIN et END
- com: commentaires
- error: caractère non reconnu
    - NR: not recognized

Cet automate permet de "prévoir" les caractères attendus après chaque entrée.
Arriver à un état final revient à retourner à l'état initial dans le code.
Dans le code, on a fusionné les états initiaux.

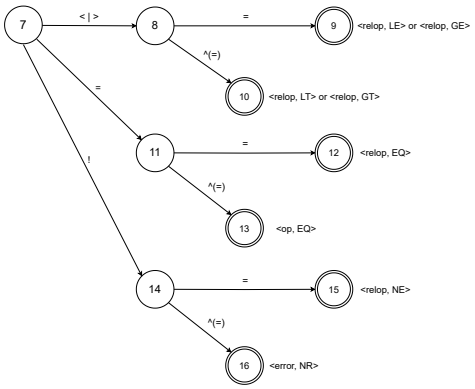
Identificateurs et mots clés:



Entiers:

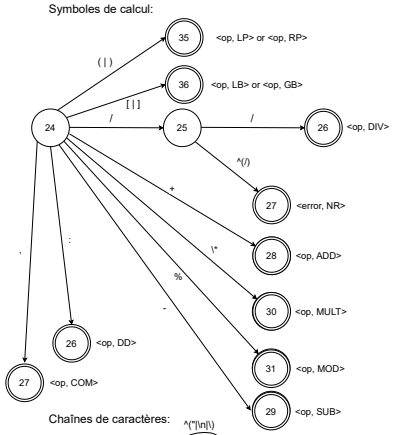


Symboles logiques:

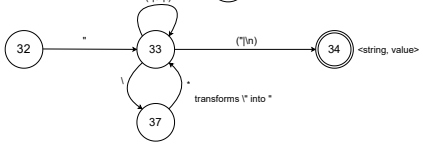


Indentations et symboles d'échappement:

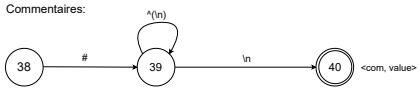
Symboles de calcul:



Chaînes de caractères:



Commentaires:



4 Parseur

Deux parseurs ont été créés :
- Un parseur itératif par Gautier
- Un parseur récursif par Hazar

L'AST (Abstract Syntax Tree) est construit à partir du parseur récursif (fonction sémantique).

5 AST (Abstract Syntax Tree)

Une fonction `visualisation()` écrit dans un fichier `.dot` pour visualiser l'arbre à l'aide de Graphviz.

Les classes `addExpr` et `mutExpr` implémentent une méthode `leftRotate()` pour transformer l'associativité gauche en associativité droite, via des rotations gauche successives.

Une méthode `simplify()` a été implémentée pour simplifier, principalement pour les expressions arithmétiques.
Exemple :

$$- - -a => -a$$
$$1 + 2 - a => 3 - a$$

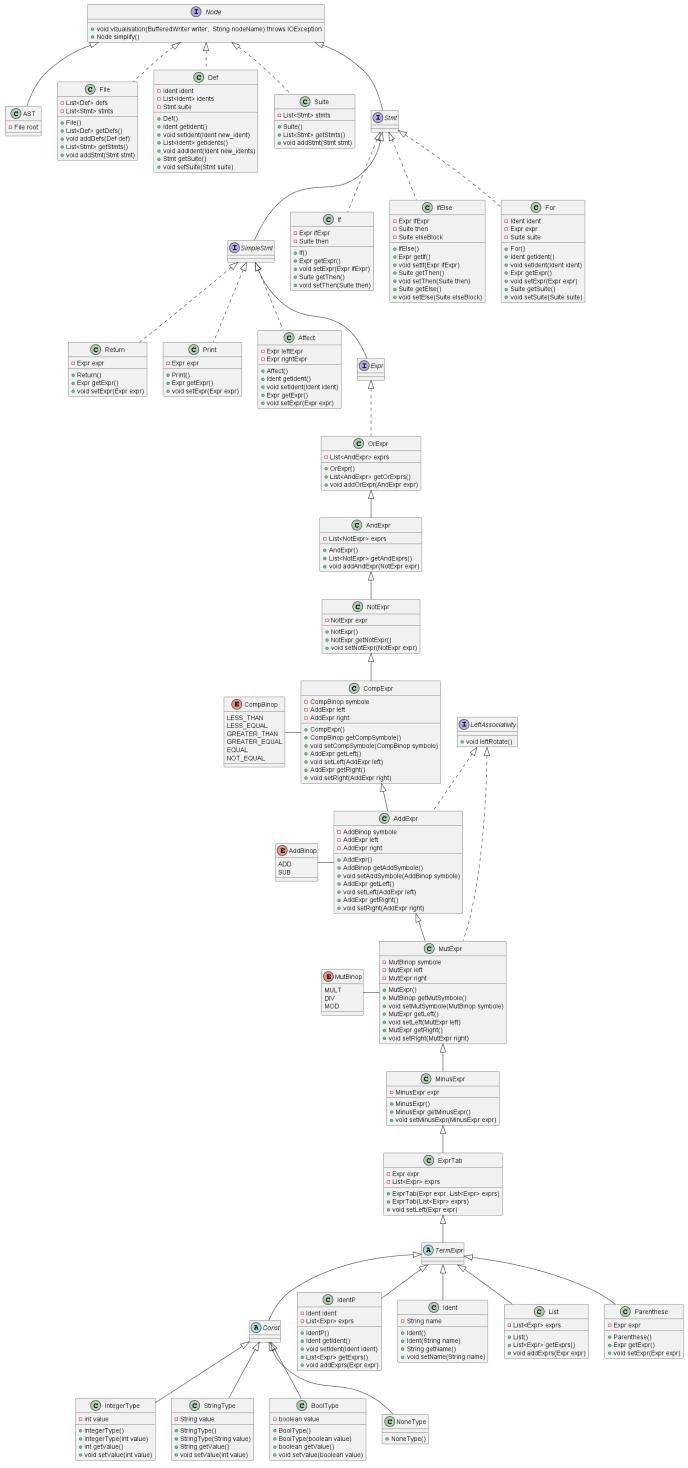


Figure 1: Diagramme de classe de l'AST.

Figure 2: Simplification de l'AST - Page 1.

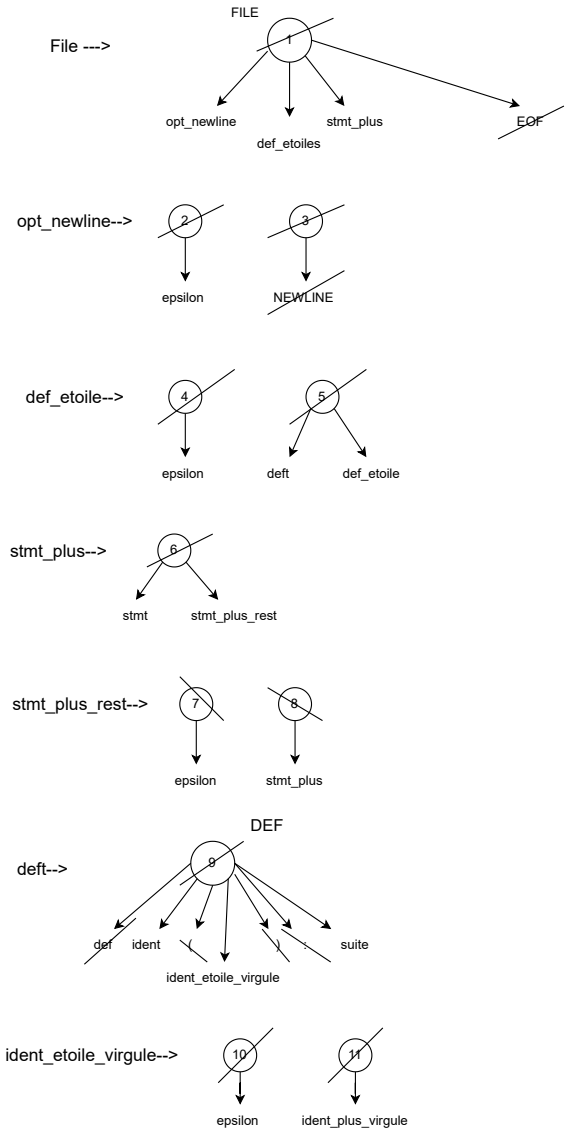


Figure 3: Simplification de l'AST - Page 2.

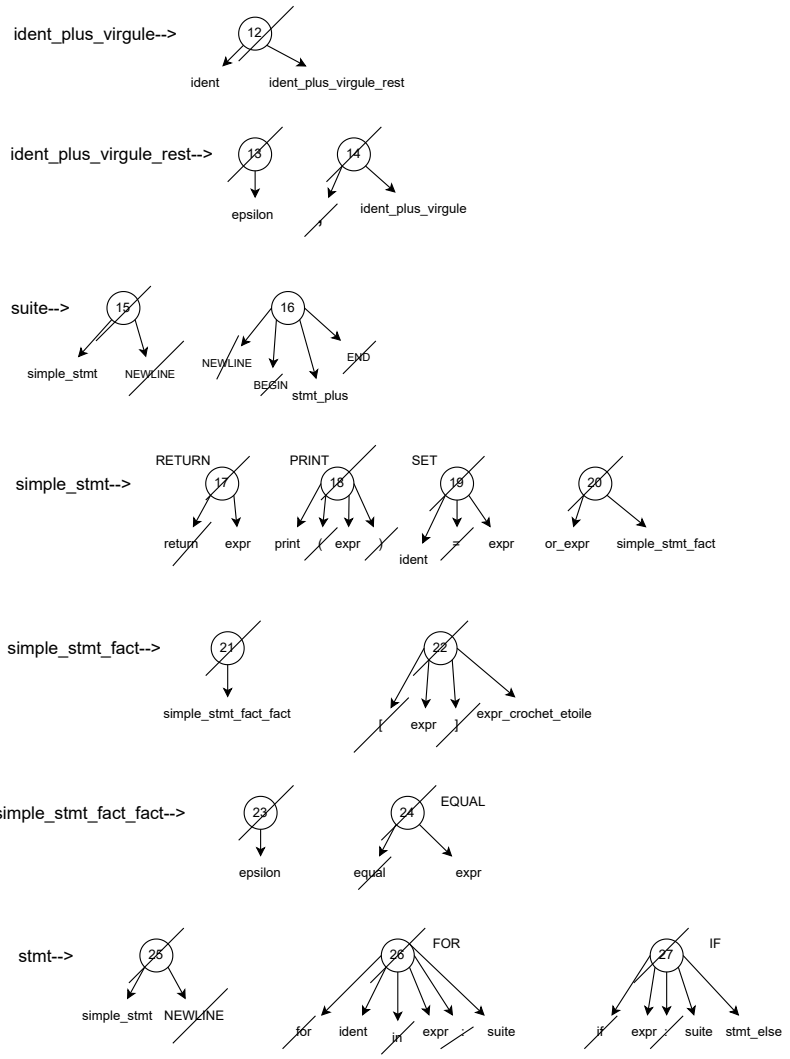


Figure 4: Simplification de l'AST - Page 3.

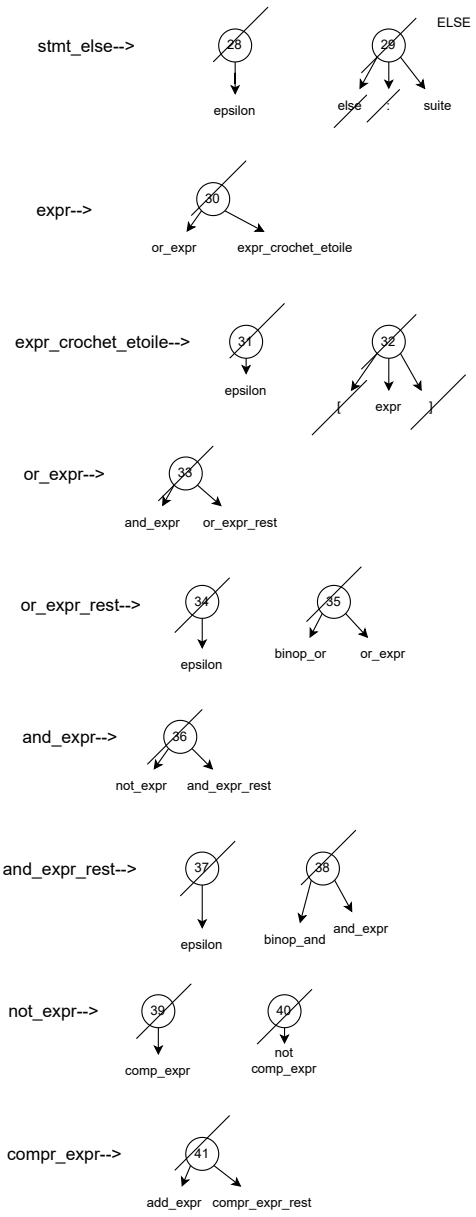


Figure 5: Simplification de l'AST - Page 4.

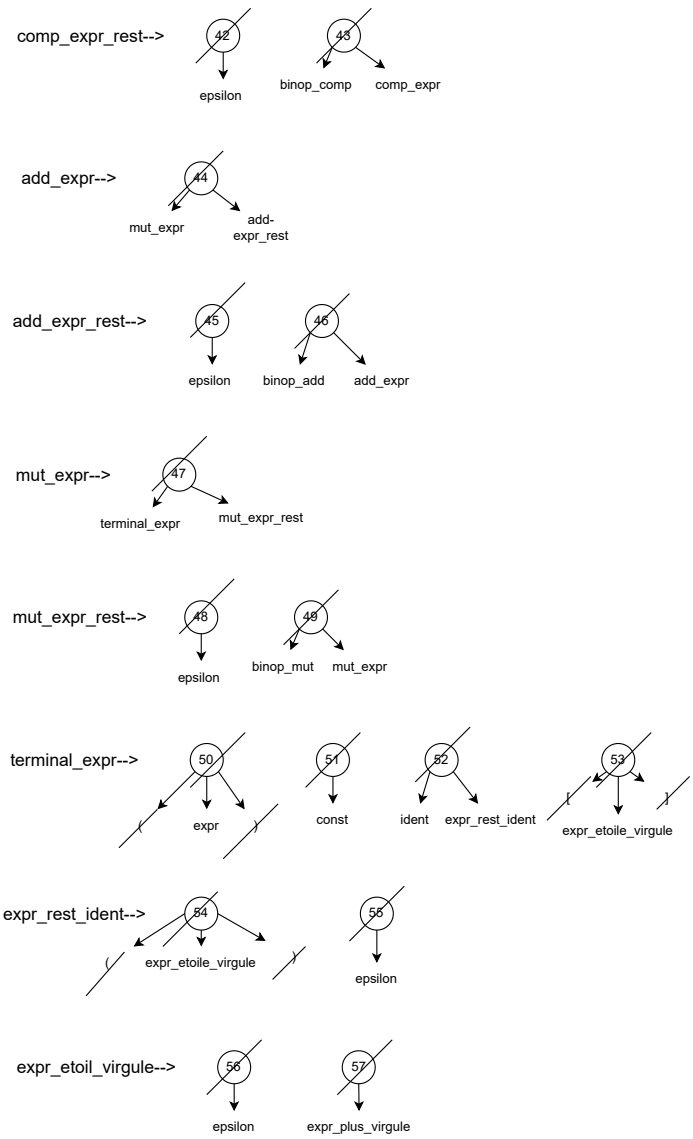


Figure 6: Simplification de l'AST - Page 5.