PCL
Projet Compilation

- 22 Janvier 2025 -

Auteurs :

Martin BICHE
Timothée BRILLAUD
Théophile PICHARD
Zoé VERNICOS

Table des matières

1 Introduction

1.1 Motivation

Les compilateurs sont des traducteurs : ils permettent de traduire un langage source A vers un langage source B. Notamment, ils permettent la traduction de langages haut-niveau, proches des langages humains, vers des langages bas-niveaux interprétables par la machine.

Ils assurent une abstraction de l'architecture sous-jacente : par exemple un compilateur C est indépendant du matériel. Cela facilite la tâche aux programmeurs, qui peuvent se focaliser uniquement sur la logique des programmes, sans s'occuper des détails matériels.

1.2 Aperçu du projet

Le projet de PCL1 a pour objectif de nous apprendre à construire le front-end d'un compilateur, c'est-à-dire :

— **le lexer** (analyse lexicale) : découpage du code source en tokens (unités lexicales)

— **le parser** (analyse syntaxique) : suivi d'une grammaire et création d'un AST (arbre syntaxique abstrait) [1]

Le front-end d'un compilateur comprend également en théorie une dernière phase d'analyse sémantique. L'analyse sémantique est une phase de parcours de l'arbre visant d'une part à construire une (ou, selon l'implémentation, plusieurs) table des symboles, et d'autre part à détecter les erreurs de sémantiques (noms non déclarés, erreur de typage, etc.). Cette dernière phase n'était pas demandée dans le projet.

L'objectif de PCL1 est de préparer au projet de PCL2, qui est la suite de l'écriture du compilateur, c'est à dire l'analyse sémantique et le back-end du compilateur. Nous ne ferons (malheureusement) pas ce projet, car nous nous orientons tous les quatre dans l'approfondissement SLE (logiciels embarqués).

1. Les outils automatiques, comme antlr, génèrent tout d'abord un Parse Tree, que l'on parcourt ensuite pour obtenir un AST. N'utilisant pas d'outils automatique de parsing dans notre projet, nous construisons l'AST directement dans le parser, sans étape intermédiaire. Nous aurions cependant pu décider de construire un Parse Tree, puis soit de le transformer en AST, soit en-place par élagage, soit par parcourt et reconstruction.

2 Difficultés

2.1 Récursivité gauche

L'associativité gauche recquiert que la grammaire soit récursive gauche, ce qui est impossible dans notre cas car notre grammaire est LL(k). L'implémentation de notre parser est ainsi légèrement différent de la grammaire afin de gérer directement l'associativité gauche lors du parsing.

Le langage reconnue par la grammaire reste cependant strictement identique au langage reconnu par notre parser.

2.2 Visualisation de l'AST

Nous avons décidé de visualiser l'AST en générant du code Mermaid. La représentation de l'arbre est tout à fait satisfaisante, à l'exception des chaînes de caractères. Certaines chaînes de caractères, considérées comme des caractères spéciaux par Mermaid, ne s'affichent pas correctement. Mermaid cherche en effet à reconnaître dans les chaînes de caractères un code en Markdown. Selon la documentation de l'outil, cependant, il ne devrait considérer une chaîne de caractère comme un code en Markdown que pour les portions entre “. Une issue sur Github est ouverte et n'a pas encore été résolue à ce jour.

Il n'y a pas de moyen simple de régler ce problème. Il faudrait réécrire le code de la visualisation de l'AST pour utiliser un autre outil de visualisation de graphes.

3 Lexer

Dans le cadre de notre projet de création de compilateur, nous avons choisi d'utiliser un automate pour la transformation du code source en une suite de tokens (Figure 1), car cette tâche est adaptée à une analyse linéaire. L'automate permet de reconnaître de manière efficace les mots-clés et paternes définis dans le langage source.

En utilisant un automate à pile (Figure 2) nous assurons que chaque transition entre états est déterminée de manière unique par le caractère rencontré, ce qui élimine toute ambiguïté dès la conception de l'automate. Lors de l'implémentation, l'automate est largement repris mais adapté pour faciliter son écriture, il ne respecte pas en pratique la définition d'un automate à pile.

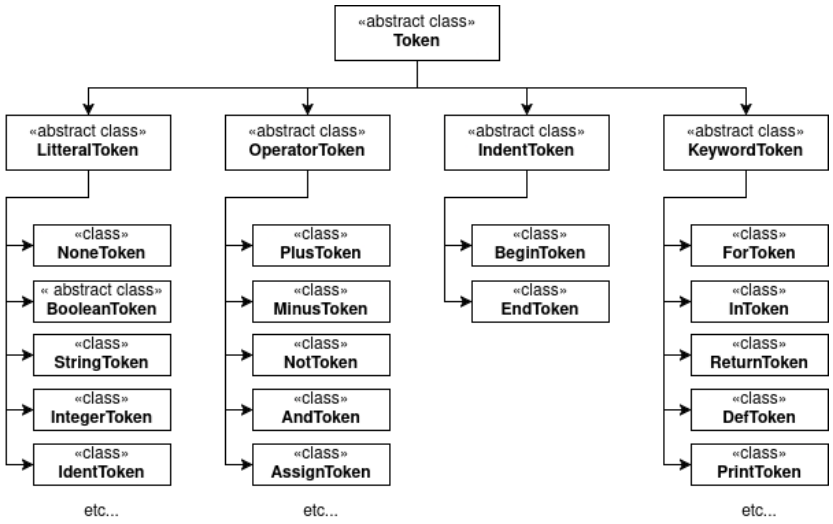

FIGURE 1 – Hiérarchie des tokens.

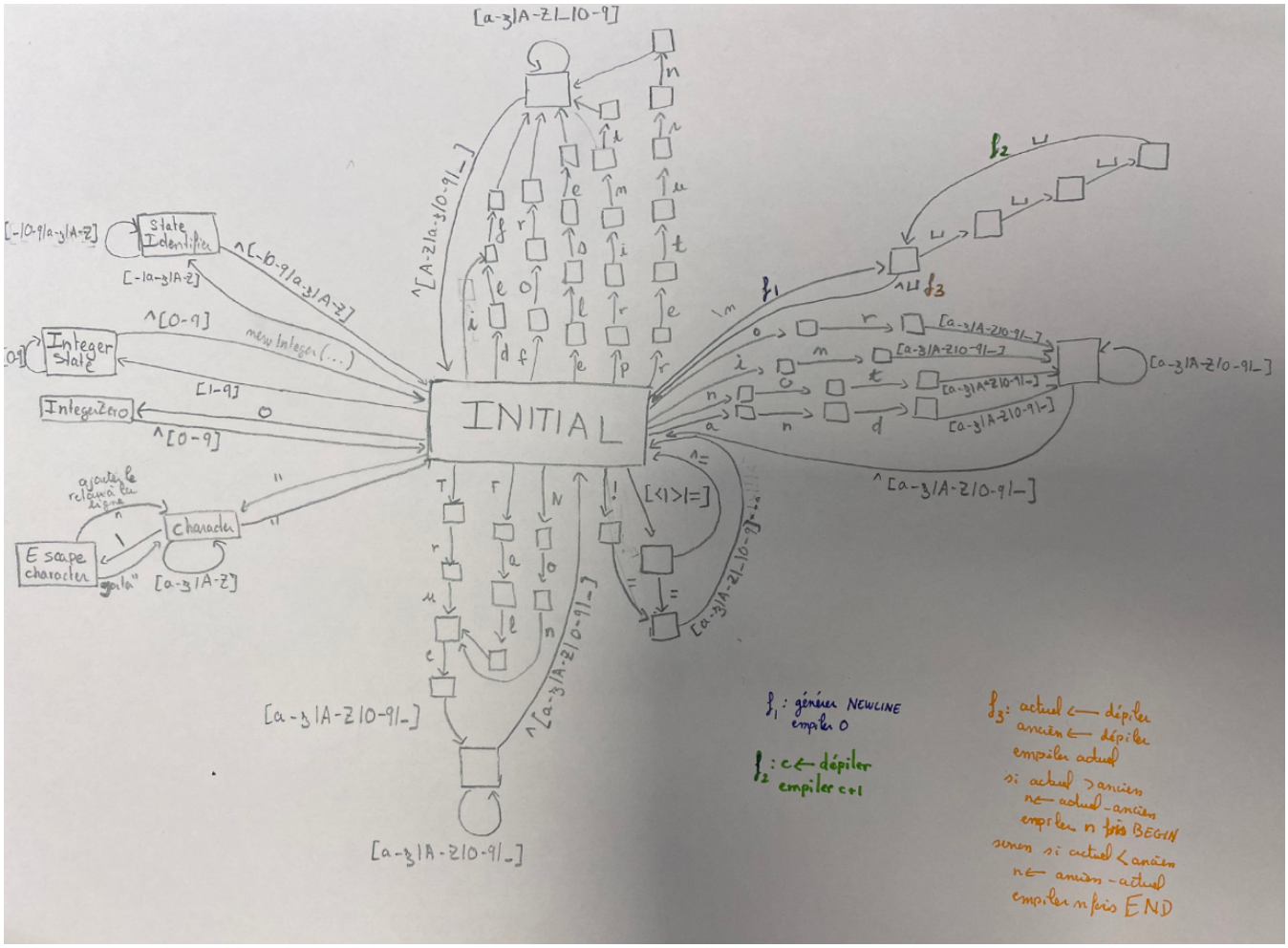

FIGURE 2 – Automate complet.

4 Parser

4.1 Grammaire

La réalisation de la grammaire avait pour objectif d'obtenir une grammaire LL(k) qui serait facile à parse avec une parseur récursif descendant. En pratique, la grammaire présentée est LL(2), majoritairement LL(1) mais LL(2) sur certaines règles. On note qu'il est possible de rendre la grammaire LL(1) mais que cela produirait une grammaire beaucoup plus grande et lourde à implémenter.

```
FILE ->
      | SPACES DEFSTMT "EOF" .
SPACES ->
      | " " SPACES
      | .
DEFSTMT ->
      | FUNCDEF DEFSTMT
      | STMT DEFSTMT
      | .
FUNCDEF ->
      | "def" IDENT "(" ARGLIST ")" ":" FOLLOW .
ARGLIST ->
      | IDENT ARGLIST2
      | .
ARGLIST2 ->
      | "," IDENT ARGLIST2
      | .
STMT ->
      | SIMPLESTMT "NEWLINE"
      | "if" EXP ":" FOLLOW ELSESTMT
      | "for" IDENT "in" EXP ":" FOLLOW .
ELSESTMT ->
      | "else" ":" FOLLOW
      | .
SIMPLESTMT ->
      | "return" EXP
      | "print" "(" EXP ")"
      | IDENT "=" EXP
      | BRACKEXP "=" EXP
      | EXP .
FOLLOW ->
      | SIMPLESTMT "NEWLINE"
      | "NEWLINE" "BEGIN" STMT FOLLOW2 "END" .
FOLLOW2 ->
      | STMT FOLLOW2
      | .
EXP ->
      | OREXP .
OREXP ->
      | ANDEXP OREXP2 .
OREXP2 ->
      | "or" OREXP
      | .
ANDEXP ->
      | NOTEXP ANDEXP2 .
ANDEXP2 ->
      | "and" ANDEXP
      | .
NOTEXP ->
      | "not" COMPEXP
      | COMPEXP .
COMPEXP ->
      | PMEXP COMPEXP2 .
COMPEXP2 ->
      | "==" PMEXP
```

```
        | "!=" PMEXP
        | ">" PMEXP
        | "<" PMEXP
        | ">=" PMEXP
        | "<=" PMEXP
        | .
PMEXP ->
        | FDMEXP PMEXP2 .
PMEXP2 ->
        | "+" PMEXP
        | "-" PMEXP | .
FDMEXP ->
        | NEGEXP FDMEXP2 .
FDMEXP2 ->
        | "*" FDMEXP
        | "//" FDMEXP
        | "%" FDMEXP | .
NEGEXP ->
        | "-" BRACKEXP
        | BRACKEXP .
BRACKEXP ->
        | PEXP BRACKEXP2 .
BRACKEXP2 ->
        | "[" EXP "]" BRACKEXP2
        | .
PEXP ->
        | "(" EXP ")"
        | "[" ARREXP "]"
        | IDENT FUNCEXP
        | NUMBER
        | STRING
        | TRUE
        | FALSE
        | NONE .
ARREXP ->
        | EXP ARREXP2
        | .
ARREXP2 ->
        | "," ARREXP
        | .
FUNCEXP ->
        | "(" ARGEXP ")"
        | .
ARGEXP ->
        | EXP ARGEXP2
        | .
ARGEXP2 ->
        | "," ARGEXP
        | .
```

4.2 Arbre Syntaxique Abstrait (AST)

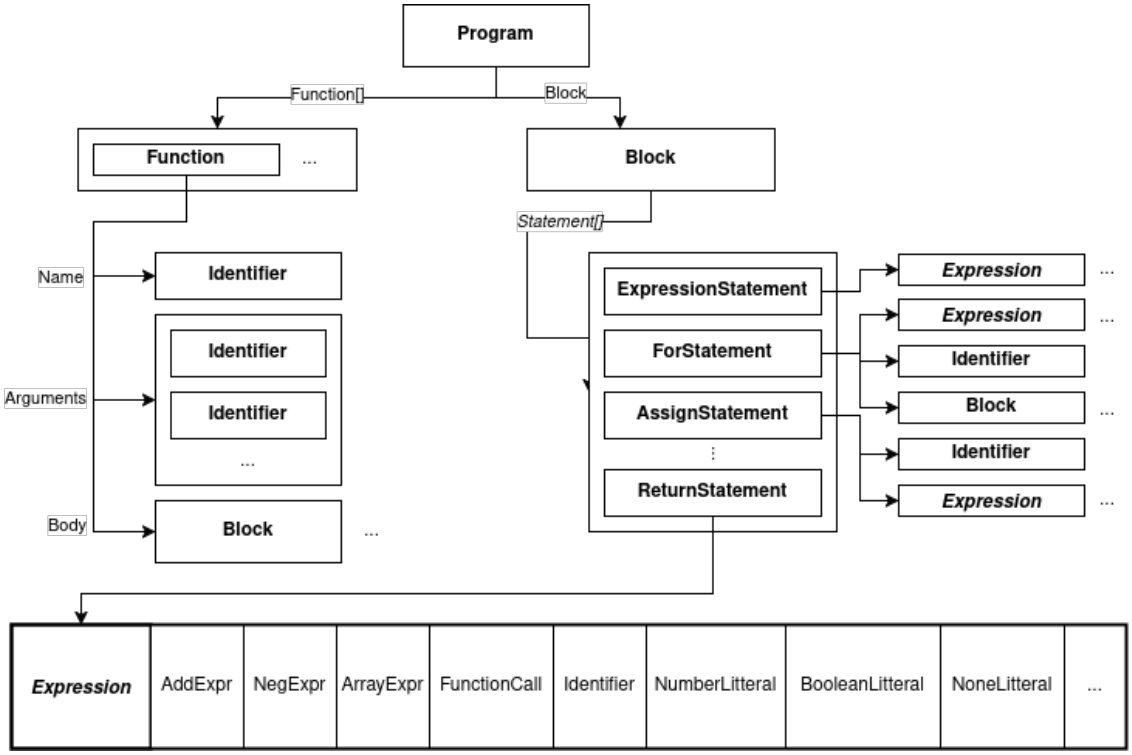

FIGURE 3 – Aperçu de l'Arbre abstrait

L'Arbre Syntaxique Abstrait est assez simplement construit, partant des plus petits composants jusqu'aux plus grands.
On retrouve notamment les littéraux qui sont une transposition simple du token d'entrée (NumberLitteral, BooleanLitteral, ...).
Les expressions sont construites autour d'un accès tableau, d'un appel de fonction ou d'un littéral et des opérateurs par composition d'une ou plusieurs expressions (addition, négation, comparaison, etc).
Les blocs permettent d'indiquer un bloc d'instructions à exécuter, celles-ci sont spécialisées pour chaque instruction définie (ForStatement, AssignStatement, etc).
Une fonction permet d'associer un bloc d'instruction avec un identificateur et les arguments nécessaires à son appel.
Un programme permet de définir les fonctions du fichier et les instructions à exécuter au lancement (Bloc du Programme).

Remarque : Les noeuds apparents dans le visualisateur peuvent être légèrement différents afin de clarifier les éléments reconnus. *Ainsi, dans la figure 5, le noeud* `PARAMETERS` *n'existe pas réellement dans le code mais permet de clarifier le rôle des noeuds enfants.*

5 Exemples

5.1 Exemples minimaux

Les exemples suivants montrent les exemples minimaux permettant de montrer tous les aspects de Mini-Python.

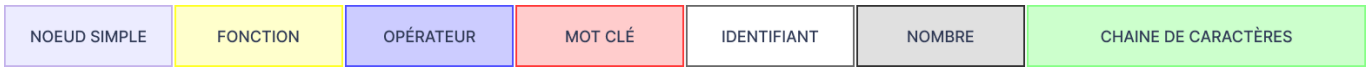

FIGURE 4 – Légende AST

5.1.1 Définition de fonction

```
def hello(name):
    new_name = name + " World!"
    return [new_name, 3 + 2]
```

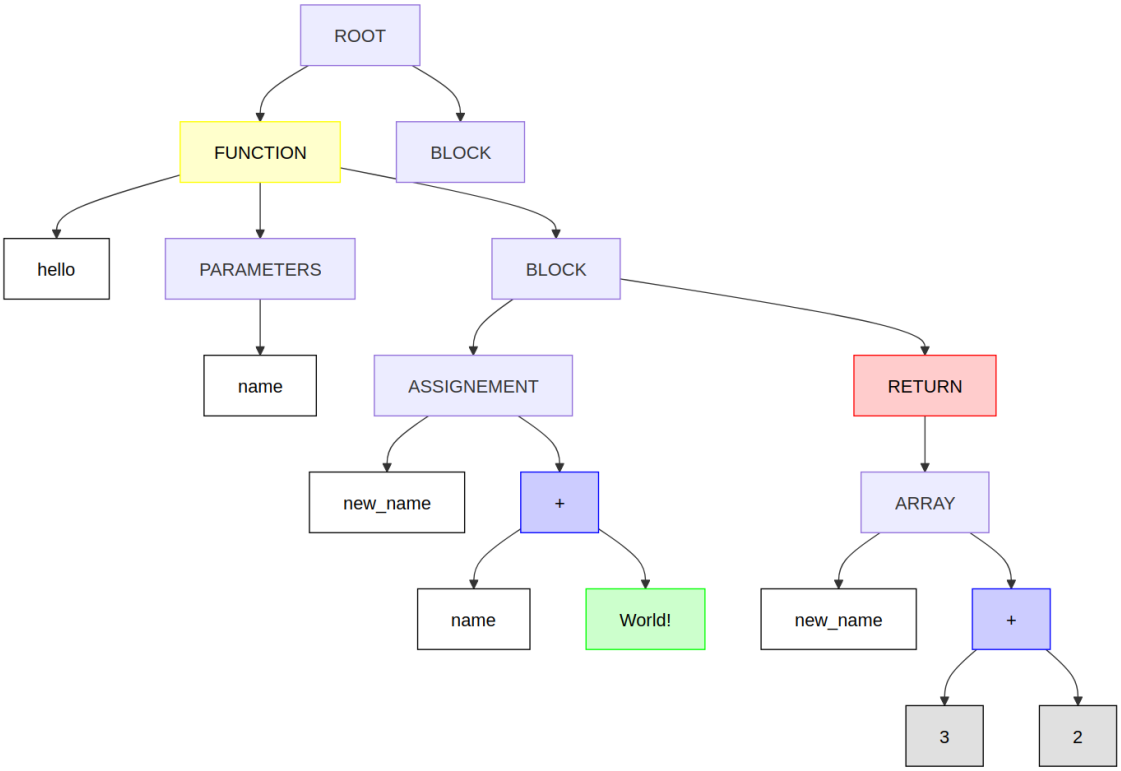

FIGURE 5 – Fonction

5.1.2 Boucles for et condition

```
lst = [1, 2, 3]
sum = 0
for i in range(len(lst)):
    if lst[i] % 2 == 0:
        sum = sum + lst[i]
```

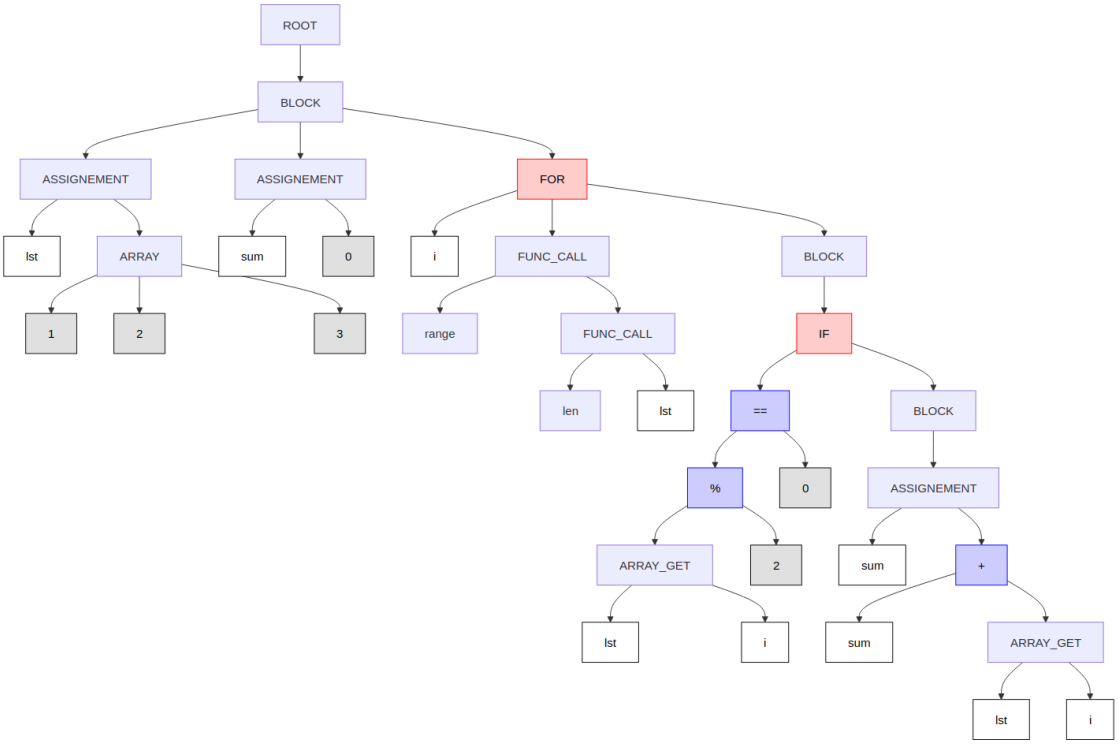

FIGURE 6 – For et If

5.1.3 Tableaux et accès aux tableaux

```
def size(name):
    return 2 * len(name)

names = ["Martin", "Zoe", "Timothee", "Theophile"]
names[size(name) % 4] = "?"
print "Hello, " + names[0]
```

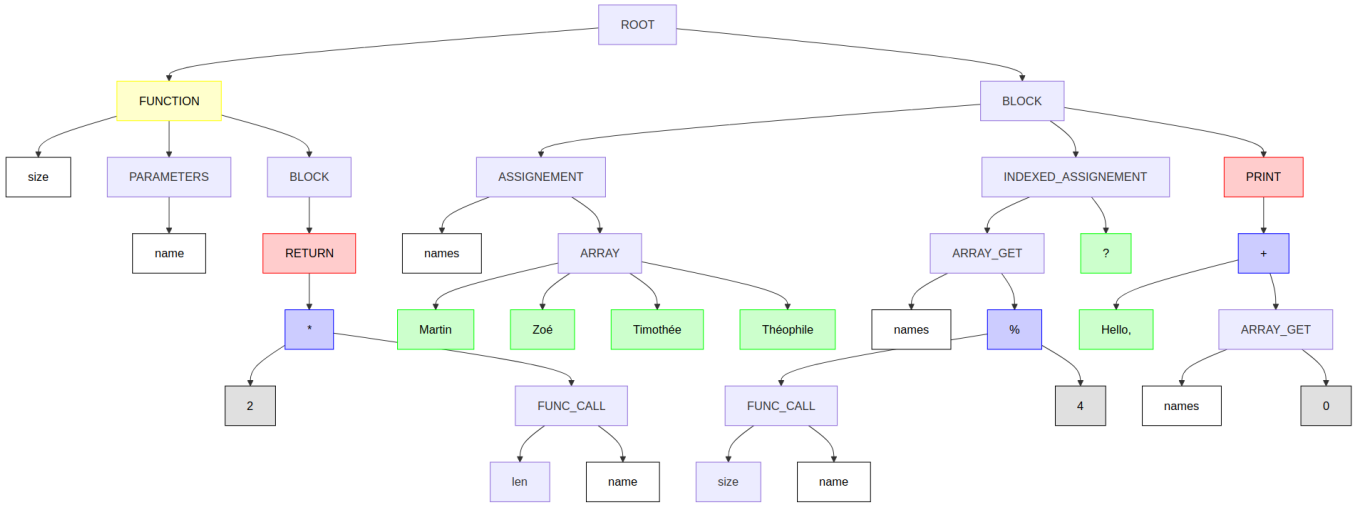

FIGURE 7 – Tableaux et accès aux tableaux

5.1.4 Accents sur les priorités opératoires

```
mes_priorites = m+p*s+c
sont = mes+p+s*c
correctes = 1*2*3*4+5*6*7*s
print mes_priorites + sont + correctes
```

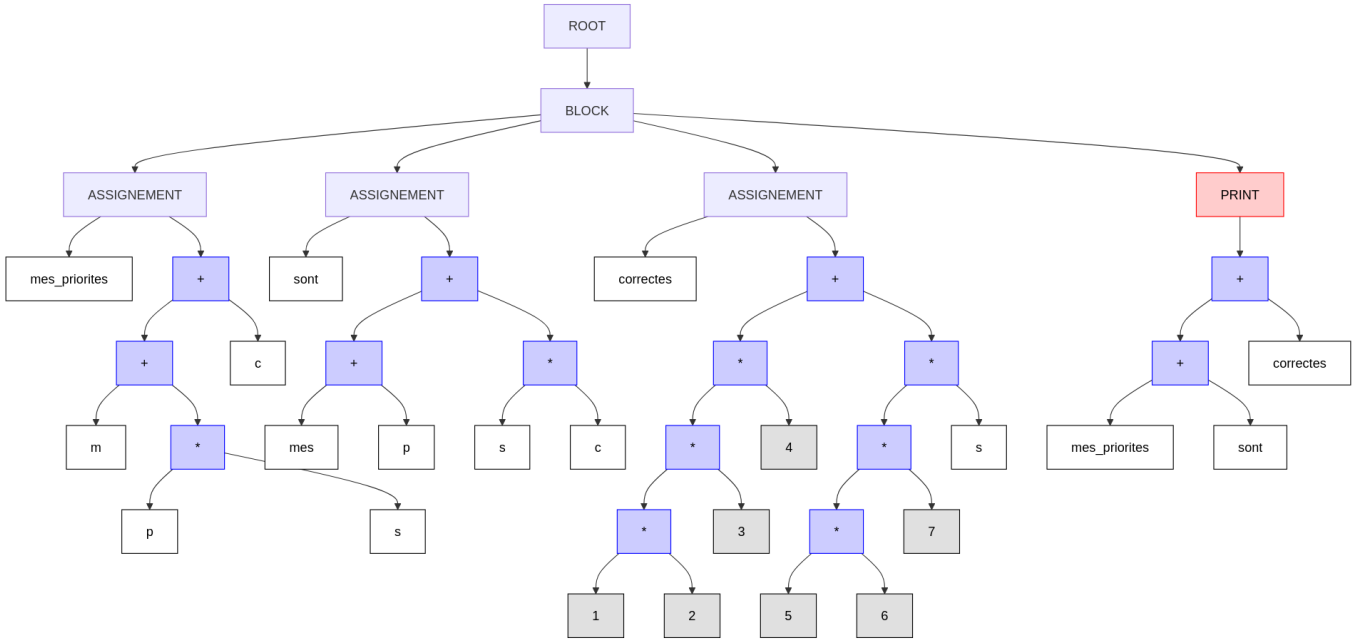

FIGURE 8 – « Mes priorités opératoires sont correctes »

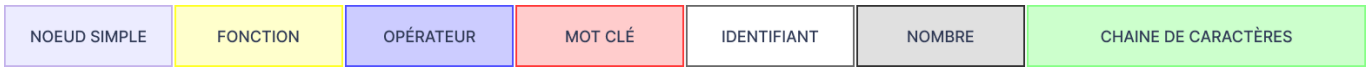

FIGURE 9 – Rappel de la légende AST

5.2 Erreurs

5.2.1 Erreurs du lexer

```
def compute(a, b):
    result = 0
    for i in range(a, b):
        result = result + a * b
    return result

something = compute(42, 666) | 5
print "Here is " + something
```

```
--> lexer error (line 7)
  |
7 | something = compute(42, 666) | 5
  |                              ^
  |                  Unexpected character '|'
```

FIGURE 10 – Erreur du lexer

5.2.2 Erreur unique du parser

```
array = [1, 2, 3]
for elem in array
    print(elem)
```

```
 --> parser error (line 2)
  |
2 | for elem in array
  |                 ^
  | ParserError: Expected colon token at <line=2, column=18, length=1> (got KeywordToken{NEWLINE,span=<line=2, column=18, length=1>})


1 errors found
```

FIGURE 11 – Erreur unique du parser

5.2.3 Erreurs multiples parser

```
def func(a, 2):
    sum = 0
    for i range(5):
        sum = sum + + i
    return sum

func("Hello") = 5
```

```
 --> parser error (line 1)
  |
1 | def func(a, 2):
  |             ^
  | ParserError: An identifier was expected at <line=1, column=13, length=1> (got IntegerToken{value=2, span=<line=1, column=13, length=1>})


 --> parser error (line 3)
  |
3 |     for i range(5):
  |           ^^^^^
  | ParserError: Expected in token at <line=3, column=11, length=5> (got IdentToken{name='range', span=<line=3, column=11, length=5>})


 --> parser error (line 4)
  |
4 |         sum = sum + + i
  |                     ^
  | ParserError: Expected an expression at <line=4, column=21, length=1> (got OperatorToken{PLUS,span=<line=4, column=21, length=1>})


 --> parser error (line 7)
  |
7 | func("Hello") = 5
  | ^^^^^^^^^^^^^^^^^
  | ParserError: Cannot assign to left value expression at <line=7, column=1, length=17>


4 errors found
```

FIGURE 12 – Erreurs multiples du parser

6 Gestion de projet

6.1 Diagramme de Gantt

Le diagramme ci-dessous représente les différentes tâches assignées ainsi que le temps estimé par tâche.

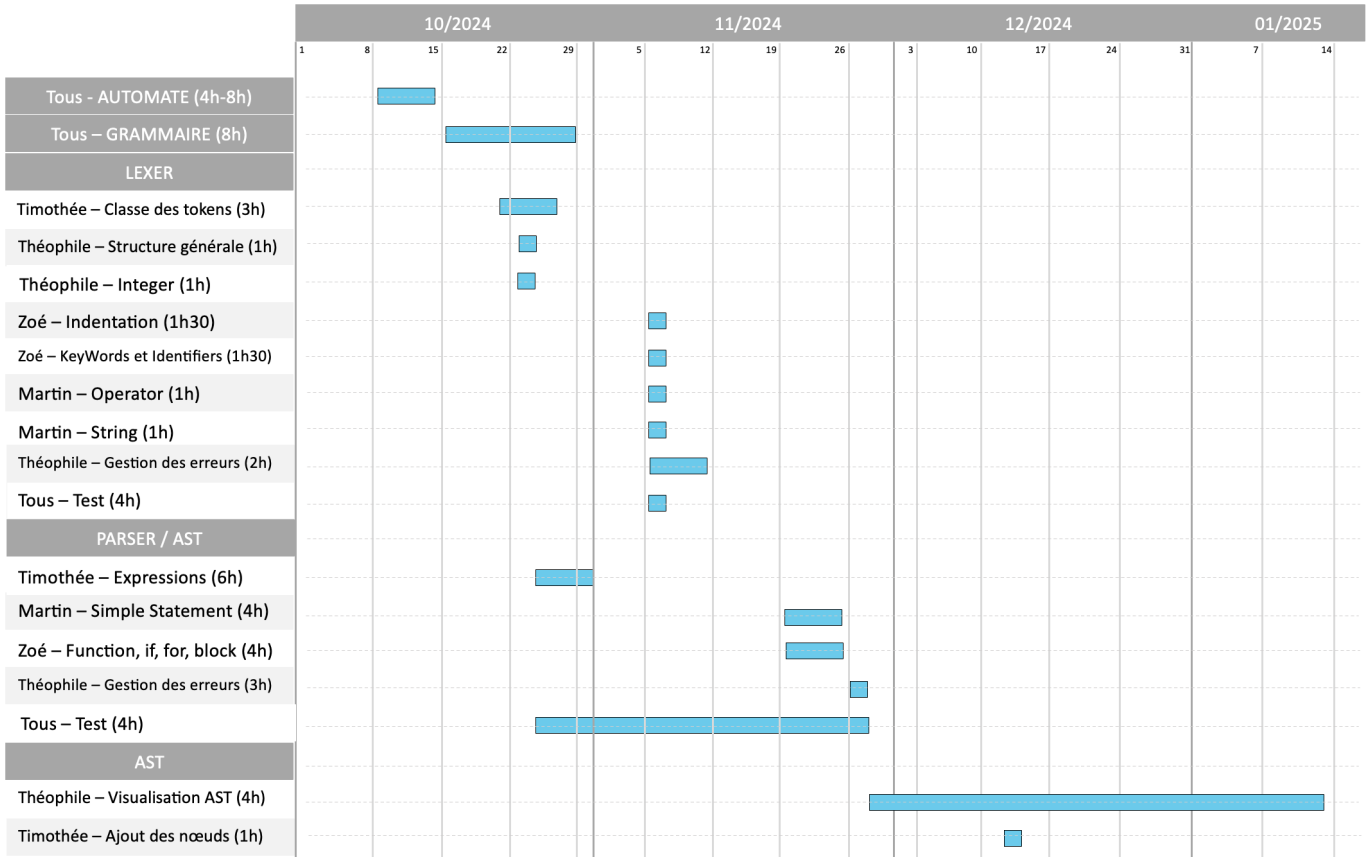

FIGURE 13 – Diagramme de Gantt.

6.2 Descriptif des tâches

- **Automate :** Écriture de l'automate reconnaissant le mini-python, d'abord faisant des sous-automates reconnaissant chaque unité lexicale, puis en rassemblant tous ces sous-automates dans un automate général.
- **Grammaire :** A partir du mini-python, nous avons écrit la grammaire et l'avons rendue LL(2) en supprimant la récursivité gauche et en respectant les priorités opératoires. La grammaire est LL(1) majoritairement (LL(2) seulement au niveau des assignation dans les simple-statements par exemple)
- **Tokens, structure générale, tokenize, structure, integer, indentation, mots-clés & identifier, operator, String :** Création de classes pour définir les unités lexicales. Reconnaissance des différentes unités de langages puis création de nouveaux tokens associés et rajout dans une liste de tokens.
- **Gestion des erreurs :** Création et gestion des différentes erreurs du lexer (unexpected character et indentation error)
- **Expressions, simple statements, Function, for, if, block :** Parsing des tokens reconnus et crées précédemment et analyse syntaxique (vérification qu'il a bien respect de la grammaire).