Rapport PCL

Projet Compilation

Préparé par <David GUICHARD Livie Romanet Celestin Morel Romain Samba>

21 janvier 2025

Table des matières

Contexte du projet

Le projet dit "PCL" (Projet de compilation) s'inscrit dans la suite du programme de la matière compilation et qui a pour but de nous apprendre le fonctionnement de la compilation d'un langage informatique.
Le but de ce projet était alors de faire les premières briques d'un compilateur d'un langage le "mini-python". Notre programme pourra alors prendre en entrée un fichier cible à compiler, en faire l'analyse lexicale puis syntaxique. La sortie de notre programme sera alors un arbre abstrait.
Le projet se passa en plusieurs étapes, ou checkpoint et le rendu final est ce rapport et un git contenant l'intégralité de notre code, fichiers tests et documentation du projet. Nous avons utiliser le language python pour coder notre projet et Markdown pour la documentation.

1 Grammaire

Le but du projet étant la création d'un compilateur à l'aide d'un analyseur LL1, l'une des premières étapes du projet aura été de modifier la grammaire donnée dans le sujet pour la rendre LL1. Voici alors la grammaire produite telle qu'utilisée dans notre code.

```
<file> -> <start_file> <start_func> <start_stmt> EOF
<start_file> -> NEWLINE
             -> ^
<start_func> -> <def> <start_func>
             -> ^
<start_stmt> -> <stmt> <start_stmt_2>
             -> ^
<start_stmt_2> -> <stmt> <start_stmt_2>
               -> ^
<def> -> def identifiant ( <first_para> ) : <suite>
<first_para> -> identifiant <parameters>
             -> ^
<parameters> -> , identifiant <parameters>
             -> ^
<suite> -> <simple_stmt> NEWLINE
        -> NEWLINE BEGIN <start_stmt> END
<simple_stmt> -> return <expr>
              -> identifiant <suite_ident_simple_stmt>
              -> <const> <expr2> <suite_expr>
              -> - <expr> <expr2> <suite_expr>
              -> not <expr> <expr2> <suite_expr>
              -> [ <expr> <depth> ] <expr2> <suite_expr>
              -> ( <expr> ) <expr2> <suite_expr>
              -> print ( <expr> )
<depth> -> , <expr> <depth>
        -> ^
<suite_ident_simple_stmt> -> = <expr>
                          -> <suite_ident_expr>
<suite_expr> -> [ <expr> ] = <expr>
             -> ^
<stmt> -> <simple_stmt> NEWLINE
       -> if <expr> : <suite> <suite_if>
       -> for identifiant in <expr> : <suite>
<suite_if> -> else : <suite>
           -> ^
<expr> -> <const> <expr2>
       -> identifiant <expr2>
       -> - <expr> <expr2>
       -> not <expr> <expr2>
       -> [ <expr> <depth> ] <expr2>
       -> ( <expr> ) <expr2>
```

```
<suite_ident_expr> -> <expr2>
                   -> ( <expr> <depth> ) <expr2>
<expr2> -> [ <expr> ] <expr2>
        -> <binop> <expr> <expr2>
        -> ^
<binop> -> +
        -> -
        -> *
        -> //
        -> %
        -> >=
        -> <=
        -> >
        -> <
        -> !=
        -> ==
        -> and
        -> or
<const> -> number
        -> char
        -> True
        -> False
        -> None
```

Dans cette représentation, chaque non-terminal est écrit entre balises, les éléments de la production sont chacun séparés d'un espace pour la visibilité et le mot vide est écrit à l'aide de '^' On peut également préciser que le langage n'est pas parfaitement LL1 tel qu'il est car nous avons un problème avec le non-terminal <expr2>, mais nous parlerons plus en détail de cela dans la partie liée aux difficultés rencontrées.

2 Générations des différentes structures de données

2.1 Analyse lexicale

Pour cette partie, pour faciliter le code et le rendre plus facile à mettre à jour, nous avons mis en place différentes structures de données (sous forme de classes Python) que nous allons présenter ci-dessous. Pour information, les trois premières classes (Token, Grammaire et TableAnalyse) sont dans le fichier `\grammaire.py`

2.1.1 Classe Token

Le but de l'analyse lexicale est de transformer un fichier de code en une liste de tokens qui vont être utilisés par la suite dans l'analyse syntaxique pour créer une représentation du code qui suit une logique bien précise et compréhensible par un ordinateur. La première structure que l'on va alors créer est la classe Token.

```
class Token:
    def __init__(self,type_token, identificateur):
        self.type_token     = type_token
        self.name           = self.__crea_name__(identificateur)
    def __crea_name__(self, identificateur):
        [...]
    def __str__(self):
        return  f"{self.name}"
```

2.1.2 Grammaire

Pour faciliter le débogage et les retours en arrière en cas de problème dans la grammaire, nous avons décidé de ne pas avoir codé en dur les différentes règles de la grammaire mais plutôt d'avoir notre grammaire décrite dans un fichier texte qui suit une convention précise et d'avoir un objet Grammaire qui est créé à la lecture de ce fichier.
Grâce à cela, on peut aussi avoir un calcul automatique de nombreuses propriétés utiles telles que le calcul des premiers et des suivants.

```
class Grammaire:
    def __init__(self, fichier_regles):
        self.axiome         = None
        self.int_to_token   = {1: "+", 2: "-",[...], 44: "else"}
        self.regles         = self.init_regles(fichier_regles)
        self.non_terminaux  = self.init_non_terminaux()
        self.premiers       = self.calculer_premiers()
        self.suivants       = self.calculer_suivants()
    def __str__(self):
        [...]
    def init_regles(self, fichier_regles) -> dict:
        [...]
```

```
        with open(fichier_regles, 'r') as fichier:
            [...]
    def init_non_terminaux(self):
        [...]
    def calculer_premiers(self):
        [...]
    def calculer_suivants(self):
        [...]
```

2.1.3 TableAnalyse

Nous avons également construit une classe TableAnalyse qui va nous permettre dans le reste du code d'accéder très facilement aux règles que l'on doit appliquer dans l'analyse syntaxique. Cette classe fut particulièrement utile pour corriger le programme et bien comprendre d'où venaient certains bugs car elle rend facile l'accès à la grammaire. Le code était aussi à de nombreux endroits largement plus compréhensible et lisible qu'une solution moins abstraite. Un des attributs de TableAnalyse était un objet Grammaire, ce qui peut faire un peu de surencapsulation d'objets, mais cela évitait d'avoir trop de paramètres lors d'appels de certaines fonctions et comme nous n'avions aucune contrainte de performance pour ce projet, nous avons opté pour la solution la plus simple.

```
class TableAnalyse:
    def __init__(self, grammaire):
        self.grammaire = grammaire
        self.table     = self._construire_table()
    def _construire_table(self):
        [...]
```

2.1.4 ListeToken

À la fin de l'analyse lexicale, on se retrouve avec une liste de Tokens, mais certains tokens, notamment ceux qui sont des char, des identifiants ou bien des nombres ont besoin de plus pour bien les identifier. On va alors regrouper toutes ces structures dans un objet d'une classe Liste_token qui va être remplie au cours de l'analyse lexicale.

```
class Liste_token:
    def __init__(self):
        self.liste_token    = []
        self.dico_idf       = {}
        self.nb_identifiant = 0
        self.dico_char      = {}
        self.nb_char        = 0
        self.dico_number    = {}
        self.nb_number      = 0
        self.dict_lexique   = {"<ERROR>" : -1,"+" : 1,[...],
                               "else": 44}
        self.liste_messages_erreurs = []
    def add_token_in_liste(self,token, etat):
        [...]
    def reconstruire_texte(self):
        [...]
    def reconstruction_last_line(self):
```

```
18          [...]
19      def est_dans_intervalle_int64(self,valeur):
20          [...]
21      def message_erreur(self, type_erreur, token_probleme):
22          [...]
```

2.2 Analyse syntaxique

Pour l'analyse syntaxique, nous allons principalement utiliser les objets créés durant l'analyse lexicale, mais le but de cette deuxième partie est quand même de construire des arbres. Nous aurons alors besoin d'une nouvelle structure.

2.2.1 Arbre

Enfin, la dernière classe mise en place est une classe qui représente la structure d'arbre que l'on va pouvoir utiliser pour construire et afficher nos arbres de dérivation et syntaxiques.

```
1  class Arbre:
2      def __init__(self, value):
3          self.value       = str(value)
4          self.children    = []
5          self.nb_children = 0
6          self.parent      = None
7
8      def add_child(self, child, index=None):
9          [...]
```

3 Présentation de résutats

Voici un exemple de code compilé. Pour cet exemple, nous avons volontairement pris un code très simple pour que l'arbre de dérivation soit visible sur ce rapport. Ce programme est tiré de la liste des programmes qui nous servent de tests pour notre projet.

```
# Test 1: Commentaires et indentation basique
def test1(a):
    if a > 0:
        return 1
    else:
        return 0
```

3.1 Tokenisation du programme

La première étape de la compilation est la création d'une liste de token représentant le code à compilé. Cette étape n'est pas obligatoire mais elle facilite grandement la suite des traitements.

```
liste des tokens : [38, 20, (40, 1), 14, (40, 2), 15, 5, 38, 36, 9,
                    (40,2), 13, (42, 1), 5, 38, 36, 35, (42, 2),
                    38, 37, 44, 5, 38, 36, 35,(42,1), 38, 37, 37,
                    39]
dico des chars : {}
dico des identifiant : {'test1': 1, 'a': 2}
dico des num : {'0': 1, '1': 2}
```

A la fin de cette étape nous disposons alors notament d'une liste des tokens et de dictionnaire ou 'bucket' permettant de stocker les valeur de chaînes de caractères, d'indentifiants de de valeur de nombre dans le code.

3.2 Arbre de dérivation

L'arbre de dérivation représente le processus par lequel une chaîne de caractères (le programme source) est dérivée à partir de la grammaire du langage. Cet arbre montre comment chaque règle de production de la grammaire est appliquée pour générer le programme final.

Dans notre exemple, l'arbre de dérivation illustre :
— La structure hiérarchique des éléments syntaxiques (fonction, paramètres, conditions, etc.).
— Le respect des règles syntaxiques définies dans la grammaire.
Cet arbre est crucial pour valider que le code source respecte les règles syntaxiques du langage. Il constitue une étape intermédiaire essentielle avant la construction de l'arbre syntaxique abstrait.

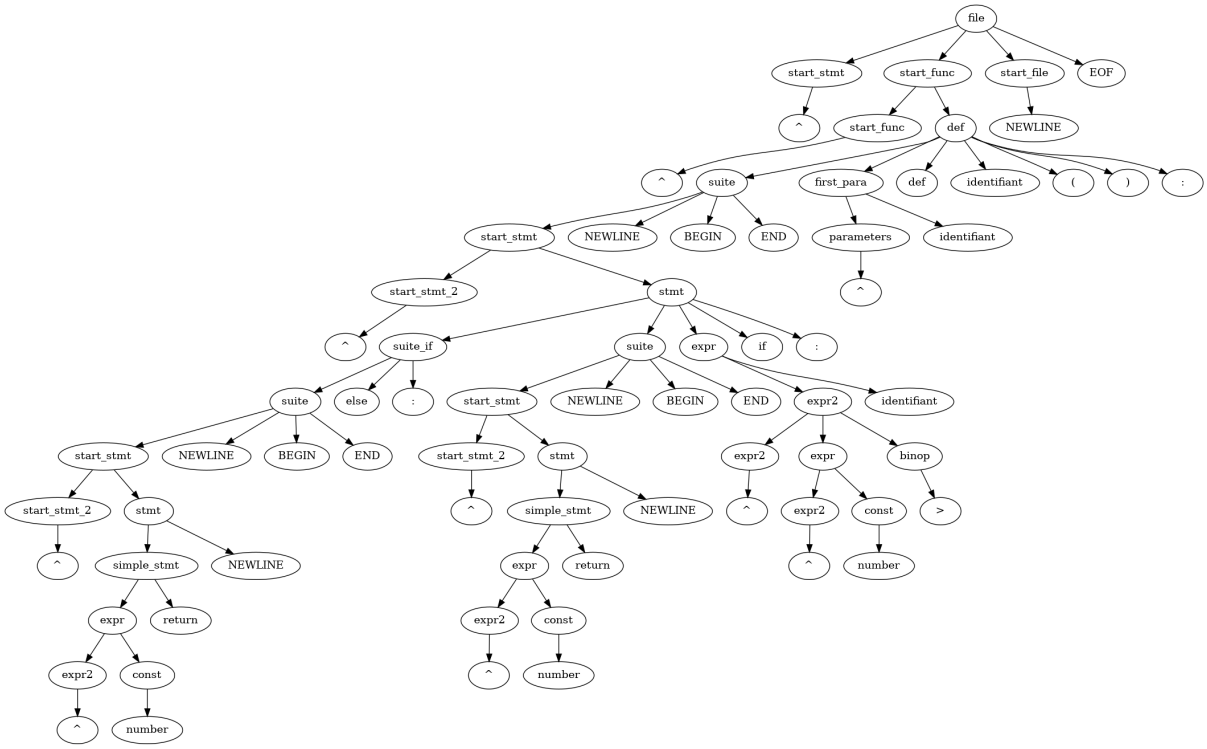

FIGURE 3.1 – Arbre de dérivation du programme présenter en début de partie 3

3.3 Arbre Abstrait

L'arbre syntaxique abstrait (AST) est une représentation simplifiée et structurée du code source. Contrairement à l'arbre de dérivation, il ne contient que les informations essentielles pour l'analyse sémantique et l'exécution du programme.

Dans notre exemple, l'AST extrait les éléments logiques du code, comme :

— La définition de la fonction et ses paramètres.

— Les conditions (ex. : `a > 0`).

— Les instructions (ex. : `return 1` et `return 0`).

Afin de le construire, il s'agit de reprendre la grammaire de l'arbre de dérivation et de simplifier les règles selon ce qui a été défini. Il faut parcourir l'arbre en profondeur de manière récursive.

L'AST est utilisé dans les étapes ultérieures du compilateur, telles que l'optimisation et la génération de code machine. Il supprime les détails superflus (mots-clés, parenthèses, etc.) pour se concentrer sur la structure logique.

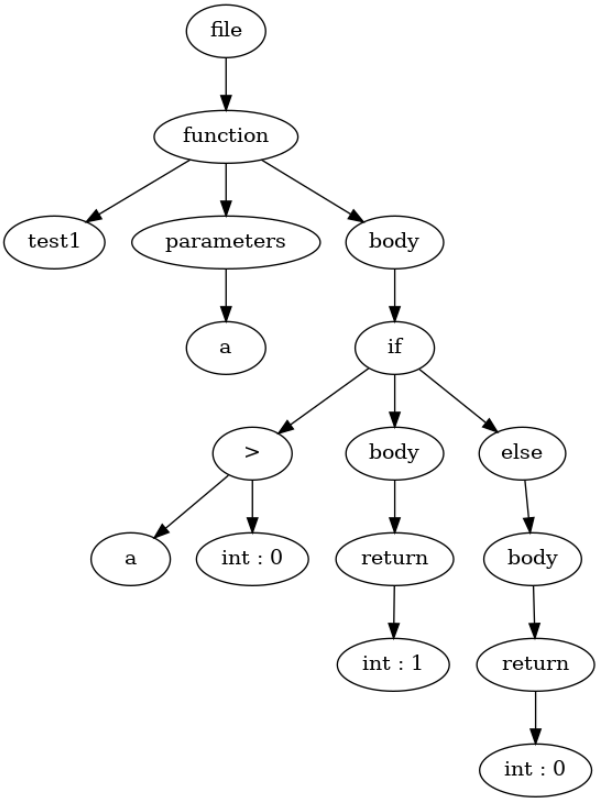

FIGURE 3.2 – Arbre abstrait du programme présenté en début de partie 3

4 Gestion de projet

Ce diagramme de Gantt vient illustrer notre répartition des tâches :

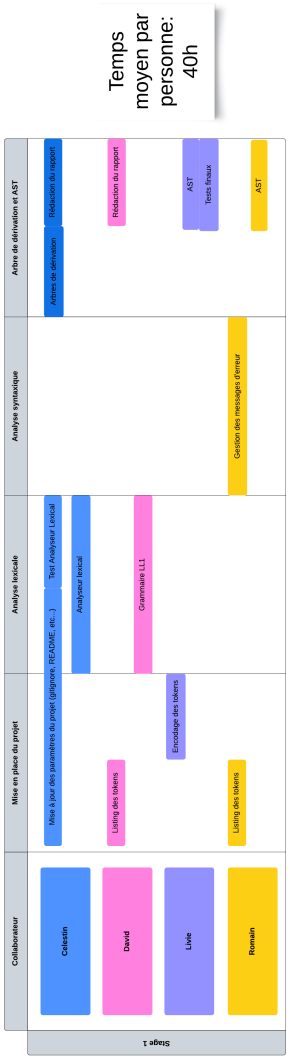

FIGURE 4.1 – Diagramme de Gantt du projet

5 Difficultés rencontrées

Nous avons commencé le projet de compilation fin octobre et avons poursuivi notre travail jusqu'à mi-janvier. Ce projet s'est étendu sur une période significative, nécessitant une gestion rigoureuse et une collaboration étroite entre les membres de l'équipe, avec une répartition des tâches entre quatre personnes. Dans ce contextes plusieurs difficultés de sources différentes sont apparue, dont les principales sont alosr décrites ci-dessous.

5.1 L'organisation

L'une des principales difficultés rencontrées concernait la structuration et la répartition des tâches. Contrairement à d'autres projets où le travail peut être facilement parallélisé, celui-ci suivait une logique séquentielle stricte, avec des jalons interdépendants. Ceci a rendu la distribution équitable du travail complexe. Chaque étape devait être complétée et validée avant de pouvoir passer à la suivante, créant parfois des goulots d'étranglement dans l'avancement du projet.

5.2 Les problèmes liés à la grammaire

La transformation de la grammaire initiale en une grammaire LL(1) s'est révélée être un défi majeur. Pensant avoir réussi cette conversion vers un language LL(1) alors que ce n'était pas le cas, cette erreur d'appréciation a entraîné des complications importantes dans les phases ultérieures du projet, nous obligeant à revenir plusieurs fois sur notre travail initial.

5.3 Intégrer le travail de chacun avec celui des autres

L'intégration des différentes parties du projet s'est avérée délicate. Bien que chaque membre de l'équipe ait développé sa partie en utilisant ses propres tests. La compréhension mutuelle du code de chacun était essentielle mais pas toujours évidente, notamment en raison des différentes approches et styles de programmation adoptés. Cette phase d'intégration a nécessité de plusieurs sessions de mise en commun et d'explications détaillées pour s'assurer que chaque membre comprenne parfaitement le fonctionnement des autres parties du projet.

5.4 Création de test unitaire assez solides

La mise en place d'une suite de tests unitaires pertinente s'est révélée être un exercice complexe. La difficulté principale résidait dans la distinction entre trois types de cas : les programmes échouant à cause de bugs dans notre compilateur, ceux échouant en raison de notre grammaire, et les programmes valides en Python standard mais non conformes aux spécifications du mini-Python. Cette distinction était cruciale pour identifier correctement l'origine des problèmes et y apporter les corrections appropriées. La création de ces tests a nécessité une compréhension approfondie tant des spécificités du mini-Python que des mécanismes de compilation.