Projet IN520

Sommaire :

1. Introduction :

Le but de ce projet est de convertir une `grammaire` en `forme normale de Chomsky et Greibach`. Avec ces deux formes normales le but est de générer le même langage. Ce langage pourra être réduit par une taille maximale de lettre pour un mot.

`On rappelle les termes du sujet :`

— `On définit :`

1. Une lettre majuscule indicée ou non est un `non-terminal`.
2. La lettre majuscule `E` représente epsilon.
3. Une lettre minuscule est un `terminal`.
4. Le séparateur est le caractère :
5. Les règles de la `grammaire` sont de la forme :
   gauche : droite
   — gauche étant un `non terminal`.
   — droite un `terminal` ou un `non terminal`.
6. L'`axiome` se trouve toujours à la première ligne.

— `Forme normale de Greibach :`
Une `grammaire algébrique` est sous
`forme normale de Greibach` si toutes les règles sont de la forme :

1. X : aA1A2...An avec n >= 1
2. X : a
3. S : E seulement si E appartient au langage.

— `Forme normale de Chomsky :`
Une `grammaire` algébrique est sous
`forme normale de Chomsky` si toutes les règles sont de la forme :

1. X : Y Z
2. X : a
3. S : E seulement si E appartient au langage.

Dans toutes les parties où il y a du code nous avons enlevés les docstrings.

2. Chargement de la grammaire :

2.1. Structure de données :

Nous avons choisi d'utiliser une `classe nommée Grammaire` pour stocker et manipuler les `grammaires formelles`. Elle est stockée dans le fichier de création du `parser`. Voici ci-dessous un résumé des fonctionnalités de la classe :

1. `Attributs :`
   — `self.regles :` Une liste qui stocke les règles de la grammaire.Chaque règle est représentée comme un `couple [gauche,[droite]]`, avec :

- — gauche : représente un non terminale.
- — droite : est une liste qui contient les éléments constituant le développement de la règle.

— self.axiome : est l'axiome de la grammaire. Il n'est pas initialisé dans le constructeur mais le sera après grâce au parser.

2. Les méthodes :
   - — ajouter_regle(gauche, droite) : Cette méthode permet d'ajouter une règle à la grammaire.
     - — gauche : Représente la partie gauche de la règle de production. Il s'agit toujours d'un non terminal.
     - — droite : Contient les éléments qui composent la partie droite de la règle. Elle peut inclure des symboles terminaux ou des symboles non terminaux. Si des listes imbriquées sont fournies, elles seront aplaties pour former une liste unique.
   - — __str__ : Fournit une représentation sous forme de chaîne de caractères de la grammaire. Chaque règle est affichée sous la forme : gauche : droite1 droite2 ...

Ci-dessous voici comment nous avons implémenté la classe Grammaire :

```
class Grammaire:
    def __init__(self):
        self.regles = []
        self.axiome = None

    def ajouter_regle(self, gauche, droite):
        droite_aplatie = [item for sublist in droite for
            item in (sublist if isinstance(sublist, list)
            else [sublist])]
        self.regles.append((gauche, droite_aplatie))

    def __str__(self):
        return "\n".join(f"{gauche} : {' '.join(droite)}"
            for gauche, droite in self.regles)
```

2.2. Création du lexer :

Pour implémenté notre lexer nous avons utilisé la bibliothèque PLY (Python Lex-Yacc). Cette bliblothèque est chargé de diviser le texte de la grammaire en tokens distincts. Un token est une unité de base comme un terminal ou un non-terminal.

```
import ply.lex as lex
```

Nous avons défini plusieurs types de tokens pour correspondre aux éléments de la grammaire :

- — NON_TERMINAL : Représente un non-terminal.
- — TERMINAL : Représente un terminal.

— COLON : Utilisé pour séparer la partie gauche de la partie droite dans une `règle de production`.
— EPSILON : Représente la chaîne vide `(E)`, utilisée pour désigner une production qui ne génère aucun symbole.
— NEWLINE : Représente un saut de ligne, permettant de gérer les retours à la ligne dans le fichier qui contient la `grammaire`.
— ignore : Représente les symboles que l'on veut ignorer à savoir les espaces et les tabulations.

Pour définir ces `tokens`, nous avons utilisé des `expressions régulières`. Ces dernières permettent d'identifier avec précision les motifs correspondant à chaque type de `token`. En revanche, pour ignorer les espaces et tabulations, nous avons utilisé le `token spécial 't_ignore'`, qui regroupe directement ces symboles.

```
# Liste des tokens
tokens = (
    'NON_TERMINAL',
    'TERMINAL',
    'COLON',
    'EPSILON',
    'ALTERNATIVE',
    'NEWLINE'
)

    # Definition des tokens
t_NON_TERMINAL = r'[A-Z][0-9]?'
t_TERMINAL = r'[a-z]'
t_COLON = r':'
t_EPSILON = r'E'
t_ignore = ' \t'
t_NEWLINE = r'\n'
```

Nous avons également implémenté une gestion des caractères non valides. Un caractère non valide est tout caractère qui ne correspond à aucun des motifs définis par nos `expressions régulières`.

```
def t_error(t):
    print(f"Caractere illegal '{t.value[0]}' a la ligne {t.
        lexer.lineno}")
    t.lexer.skip(1)
```

On créer notre `lexer`, il va cherché les `tokens` définie ci-dessus pour se créer :

```
    lexer = lex.lex()
```

Comme notre `lexer` est dans un fichier à part, nous avons défini une fonction `build_lexer(grammaire)`. Cette fonction encapsule la création et l'initialisation du `lexer`, permettant ainsi une réutilisation facile dans d'autres parties du programme.

```python
def build_lexer(grammaire):
    lexer.input(grammaire)
    return lexer
```

2.3. Création du parser :

Pour créer un parser nous avons besoin du `lexer` et de ces `tokens`, ainsi que la bibliothèque `PLY`.

```python
import ply.yacc as yacc
from lexer import tokens, build_lexer
```

Voici un résumé des 6 différentes fonctions définies pour le `parser` :

1. `p_grammaire :` définit l'entrée principale du `parser`. Elle attend une liste de règles et initialise la `grammaire` en conséquence.

```python
        def p_grammaire(p):
            '''grammaire : regles'''
            p[0] = p[1]
```

2. `p_regles(p) :` permet de gérer une ou plusieurs règles, séparées par un saut de ligne `NEWLINE`. La classe `Grammaire` est utilisée pour stocker ces règles.

   Si plusieurs règles sont présentes, elles sont fusionnées à l'aide de l'attribut `regles` de l'objet `Grammaire`. Chaque règle est ajoutée à l'aide de la méthode : `ajouter_regle(self, gauche, droite)`.

```python
        def p_regles(p):
            '''regles : regle NEWLINE regles | regle'''
            p[0] = Grammaire()
            gauche, droite = p[1]
            p[0].ajouter_regle(gauche, droite)
            if len(p) == 4:
                p[0].regles.extend(p[3].regles)
```

3. `p_regle(p) :` représente une seule règle de production. Elle associe un `non terminal` à un ensemble de productions, séparés par un :.

```python
        def p_regle(p):
            '''regle : NON_TERMINAL COLON productions
                '''
            p[0] = (p[1], p[3])
```

4. `p_productions(p) :` gère une ou plusieurs productions. Chaque production est ajoutée à une liste, qui devient la partie droite d'une `règle de production`.

```
        def p_productions(p):
            '''productions : production productions |
                production'''
            p[0] = [p[1]] + (p[2] if len(p) == 3 else
                [])
```

5. p_production(p) : représente une production individuelle, qui peut être un terminal, un non-terminal, E, ou une combinaison de ces éléments. L'utilisation récursive permet de représenter des séquences imbriquées de terminaux et non-terminaux.

```
        def p_production(p):
            '''production : TERMINAL | NON_TERMINAL |
                EPSILON | TERMINAL production |
                NON_TERMINAL production'''
            p[0] = [p[1]] + (p[2] if len(p) == 3 else
                [])
```

6. p_error(p) : En cas d'erreur syntaxique, cette fonction affiche un message détaillé.

```
        def p_error(p):
            if p:
                print(f"Erreur syntaxique a la ligne {p
                    .lineno}, a '{p.value}' (type: {p.
                    type})")
            else:
                print("Erreur syntaxique a la fin de l'
                    entree")
```

On créé notre paser, il va chercher les régles définies ci-dessus pour se créer :

```
    parser = yacc.yacc(start="grammaire")
```

Comme notre parser est dans un fichier à part, nous avons défini une fonction build_parser(grammaire). Cette fonction encapsule la création et l'initialisation du parser, permettant ainsi une réutilisation facile dans d'autres parties du programme.

```
    def build_parser(contenu):
        lexer = build_lexer(contenu)
        grammaire = parser.parse(contenu, lexer=lexer)
        grammaire.axiome = grammaire.regles[0][0]
        return grammaire
```

3. Écriture et lecture de fichier :

3.1. Lecture de fichier :

Nous avons défini une fonction `lire_grammaire(fichier)` pour lire les fichiers, elle prend en argument un nom de fichier. Cette fonction utilise le `lexer` et notre parser pour renvoyer la `grammaire`. Elle se situe dans un autre fichier afin de pouvoir l'utiliser dans les fichiers suivant : `grammaire.py` et `generer.py`.

```
1    from parser_yacc import build_parser
2
3    def lire_grammaire(fichier):
4        with open(fichier, "r") as f:
5            contenue = f.read()
6        f.close()
7        return build_parser(contenue)
```

3.2. Écriture de fichier :

1. Nous avons défini 2 fonctions permettant écrire dans un fichier. `ecrire_general(grammaire)` nous permet d'écrire les `formes normales de Chomsky et Greibach`. Comme nous verrons plus tard, l'axiome est toujours remis en première ligne. Cela nous permet donc d'écrire telle quelle notre `grammaire` grâce à sa méthode `__str__`.

```
1        def ecrire_general(grammaire, fichier):
2            with open(fichier, "w") as f:
3                f.write(str(grammaire))
4            f.close()
```

2. `ecrire_generer(grammaire)` nous permet d'écrire les mots générés via les `formes normales de Chomsky et Greibach` cependant les mots qu'on génère n'ont pas les sauts de ligne donc afin de respecter la consigne nous les ajoutons avant de les écrire.

```
1        def ecrire_generer(mots, fichier):
2            with open(fichier, "w") as f:
3                f.write("\n".join(mots) + "\n")
4            f.close()
```

4. Forme normale de Chomsky :

Nous avons implémenté la `forme normale de Chomsky` en faisant la suite d'algorithmes suivante : START->TERM->BIN->DEL->UNIT.
Puis nous remettons l'`axiome` en première ligne grâce à la fonction `axiom_premiere_ligne(grammaire)`.

```python
def axiome_premiere_ligne(grammaire):
    axiome_rule = next((gauche, droite) for gauche, droite
        in grammaire.regles if gauche == grammaire.axiome)
    grammaire.regles.remove(axiome_rule)
    grammaire.regles.insert(0, axiome_rule)
    return grammaire
```

Comme cette fonction est définie dans un autre fichier car utilisée aussi pour la `forme normale de Greibach` nous l'avons défini l'import comme ceci :

```python
    from unit_del_copie import unit, del_,
        axiome_premiere_ligne
```

4.1. START :

`Objectif :`

La fonction garantit que l'`axiome` actuel n'est utilisé que comme point de départ. Si l'axiome apparaît dans la partie droite des règles, la fonction crée un nouveau symbole de départ, qui devient l'`axiome` principal.

`Approche :`

1. La fonction commence par vérifier si l'axiome est utilisé dans les parties droites des règles.
2. Si oui, un nouveau symbole est créé. Une nouvelle `règle de production` est ajoutée à la `grammaire`.
3. La `grammaire` mise à jour est retournée.

```python
def start(grammaire):
    axiome = grammaire.axiome

    utilise_comme_terminal = any(axiome in droite for _,
        droite in grammaire.regles)

    if utilise_comme_terminal:
        compteur = 1
        base_axiome = axiome.rstrip('0123456789')
        nouvel_axiome = f'{base_axiome}{compteur}'
        while any(nouvel_axiome in droite for _, droite in
            grammaire.regles) or nouvel_axiome == 'E':
            compteur += 1
            nouvel_axiome = f'{base_axiome}{compteur}'
        grammaire.ajouter_regle(nouvel_axiome, [axiome])
        grammaire.axiome = nouvel_axiome

    return grammaire
```

4.2. TERM :

Objectif :

La fonction transforme toutes les **règles de production** afin que les **terminaux** présents dans des productions avec plusieurs symboles soient remplacés par des variables **non terminales** uniques.

Approche :

1. La fonction parcourt les règles de production de la **grammaire**.
2. Pour chaque **terminal** apparaissant dans une production de plus d'un symbole, la fonction crée une nouvelle variable **non terminale** et une **règle associée**.
3. La fonction remplace les **terminaux** par ces nouvelles variables dans les **règles originales**.
4. Les nouvelles règles sont ajoutées à la **grammaire**.

```python
def term(grammaire):
    nouvelles_regles = []
    correspondance_terminaux = {}
    compteur_terminaux = 1

    def obtenir_nouvelle_variable(symbole):
        nonlocal compteur_terminaux
        if symbole not in correspondance_terminaux:
            nouvelle_variable = f"T{compteur_terminaux}"
            correspondance_terminaux[symbole] =
                nouvelle_variable
            compteur_terminaux += 1
            nouvelles_regles.append((nouvelle_variable, [
                symbole]))
        return correspondance_terminaux[symbole]

    nouvelles_regles.extend(
        (gauche, [obtenir_nouvelle_variable(symb) if symb.
            islower() else symb for symb in droite])
        if len(droite) > 1 else (gauche, droite)
        for gauche, droite in grammaire.regles
    )

    grammaire.regles = nouvelles_regles
    return grammaire
```

4.3. BIN :

Objectif :

La fonction s'assure que toutes les règles de la **grammaire** sont binaires, c'est-à-dire qu'elles possèdent deux ou moins de symboles à droite, tout en préservant l'équivalence de la **grammaire**.

Approche :

1. La fonction parcourt chaque règle de production de la `grammaire`.
2. Pour les règles avec plus de deux symboles à droite, elle introduit des variables intermédiaires pour décomposer la règle en plusieurs étapes.
3. Les nouvelles variables intermédiaires suivent une convention de nommage unique.
4. Les nouvelles règles intermédiaires sont ajoutées à la `grammaire`.

```
def bin(grammaire):
    nouvelles_regles = []
    compteur_variables = 0

    def obtenir_nouvelle_variable():
        nonlocal compteur_variables
        compteur_variables += 1
        return f"X{compteur_variables}"

    for gauche, droite in grammaire.regles:
        while len(droite) > 2:
            nouvelle_variable = obtenir_nouvelle_variable()
            nouvelles_regles.append((gauche, [droite[0],
                nouvelle_variable]))
            gauche = nouvelle_variable
            droite = droite[1:]
        nouvelles_regles.append((gauche, droite))

    grammaire.regles = nouvelles_regles
    return grammaire
```

4.4. DEL :

Objectif :

La fonction supprime les productions `E` tout en modifiant les autres règles pour qu'elles restent valides et permettent de générer le même langage.

Approche :

1. Identifier tous les symboles `non-terminaux` qui génèrent `E`.
2. Modifier les règles existantes pour inclure toutes les combinaisons possibles de suppression des symboles concernés par `E`.
3. Éliminer les règles où `E` est directement présent.

Comme vu dans la chapitre 4 nous importons la fonction `del_(grammaire)`, car cette dernière nous sera utile pour la transformation en `forme normale de Greibach`.

```python
def del_(grammaire):
    epsilon_productions = {gauche for gauche, droite in
        grammaire.regles if droite == ['E']}
    nouvelles_regles = set()

    def ajouter_nouvelles_regles(gauche, droite):
        for i in range(len(droite)):
            if droite[i] in epsilon_productions:
                nouvelle_droite = droite[:i] + droite[i+1:]
                if nouvelle_droite:
                    regle = (gauche, tuple(nouvelle_droite))
                    if regle not in nouvelles_regles:
                        nouvelles_regles.add(regle)
                        ajouter_nouvelles_regles(gauche,
                            nouvelle_droite)

    for gauche, droite in grammaire.regles:
        if droite != ['E']:
            nouvelles_regles.add((gauche, tuple(droite)))
            ajouter_nouvelles_regles(gauche, droite)

    # Conversion en liste de tuples
    grammaire.regles = [(g, list(d)) for g, d in
        nouvelles_regles]
    return grammaire
```

4.5. UNIT :

`Objectif :`

La fonction remplace les `règles unitaires` par les règles associées à leurs cibles, pour obtenir une `grammaire` équivalente sans `règle unitaire`.

`Approche :`

1. Identifier les `règles unitaires`.
2. Calculer la fermeture des `non-terminaux`, c'est-à-dire l'ensemble des `non-terminaux` atteignables via des `règles unitaires`.
3. Ajouter les règles des `non-terminales` atteignables à leurs sources.
4. Supprimer les `règles unitaires`.

La fonction `unit(grammaire)` nécessite l'appel à `deepcopy` de la bibliothèque `copy`. Sans `deepcopy`, les modifications sur une copie normale affecteraient aussi les données originales. Cela garanti la sécurité et l'intégrité des données, permettant ainsi une manipulation indépendante des règles. Nous avons conçu cette fonction pour limiter la dépendance à `copy` à un seul fichier.

```python
from copy import deepcopy

def copie(liste):
    return deepcopy(liste)
```

Comme vu dans la partie 4 nous importons la fonction `unit(grammaire)`, car cette dernière nous sera utile pour la transformation en `forme normale de Greibach`.

```python
def unit(grammaire):
    import copy
    regles = copy.deepcopy(grammaire.regles)
    unit_productions = [(g, d[0]) for g, d in grammaire.regles if len(d) == 1 and d[0].isupper()]
    nouvelles_regles = [r for r in grammaire.regles if not (len(r[1]) == 1 and r[1][0].isupper())]

    # Initialiser la fermeture pour chaque non terminal, y compris ceux qui ne sont pas presents dans les regles unitaires
    non_terminals = set(g for g, _ in grammaire.regles).union(set(d[0] for _, d in grammaire.regles if d and d[0].isupper()))
    fermeture = {nt: set() for nt in non_terminals}

    for gauche, droite in unit_productions:
        fermeture[gauche].add(droite)

    # Etendre les fermetures
    changed = True
    while changed:
        changed = False
        for gauche in fermeture:
            for droite in list(fermeture[gauche]):
                if droite in fermeture:
                    if not fermeture[gauche].issuperset(fermeture[droite]):
                        fermeture[gauche].update(fermeture[droite])
                        changed = True

    # Ajouter les productions correspondantes sans creer de doublons
    nouvelles_regles_finales = set()
    for gauche in fermeture:
        visites = set()
        stack = [gauche]
        while stack:
            current = stack.pop()
```

```
            if current not in visites:
                visites.add(current)
                for droite in fermeture[current]:
                    for g, d in regles:
                        if g == droite and not (len(d) == 1
                            and d[0].isupper()):
                            nouvelles_regles_finales.add((
                                gauche, tuple(d)))
                    if droite != gauche:
                        stack.append(droite)

    # Ajouter les regles terminales
    for g, d in regles:
        if not (len(d) == 1 and d[0].isupper()):
            nouvelles_regles_finales.add((g, tuple(d)))

    grammaire.regles = [(g, list(d)) for g, d in
        nouvelles_regles_finales]
    return grammaire
```

4.6. Regroupement des fonctions :

Comme vu dans la chapitre 4 on utilise les algorithmes dans l'ordre suivant : START->TERM->BIN->DEL->UNIT->axiom_premiere_ligne(grammaire). La fonction `chomsky(grammaire)` nous permet de faire la transformation d'une `grammaire` en `forme normale de Chomsky`. Toutes les fonctions vues sont dans un fichier `chomsky.py` pour séparer la logique du code.

```
def chomsky(grammaire):
    grammaire = start(grammaire)
    grammaire = term(grammaire)
    grammaire = bin(grammaire)
    grammaire = del_(grammaire)
    grammaire = unit(grammaire)

    # Reorganiser les regles pour que l'axiome soit en
        premiere ligne
    grammaire = axiome_premiere_ligne(grammaire)
    return grammaire
```

5. Forme normale de Greibach :

Nous avons implémenté la `forme normale de Greibach` en faisant la suite d'algorithmes suivante : START -> DEL -> UNIT -> SUPPRESSION DES RÉ-

CURSIONS GAUCHE -> SUPPRESSION DES OCCURRENCES EN TÊTE. Puis nous remettons l'axiome en première ligne grâce à la fonction `axiom_premiere_ligne(grammaire)`. Cette fonction a déjà été détaillé dans le chapitre 4 ainsi elle ne sera pas détaillé ici. Nous avons utilisé le import vu dans la chapitre 4.

```
1    from unit_del_copie import unit, del_,
         axiome_premiere_ligne, start
```

5.1. START :

Nous avons déjà abordé cette fonction dans le sous chapitre 4.1, donc nous ne la retraiterons pas ici.

5.2. DEL :

Nous avons déjà abordé cette fonction dans le sous chapitre 4.4, donc nous ne la retraiterons pas ici.

5.3. UNIT :

Nous avons déjà abordé cette fonction dans le sous chapitre 4.5, donc nous ne la retraiterons pas ici.

5.4. Suppression des récursions gauche :

`Objectif :`
La fonction supprimer les règles récursives directes tout en préservant les règles équivalentes et en garantissant que la `grammaire` reste valide pour les méthodes d'analyse descendante.

`Approche :`

1. Parcourir les règles de production d'un non-terminal et séparer celles où le `non-terminal` apparaît en premier dans le côté droit des autres règles.
2. Introduire un nouveau symbole pour représenter les variantes des règles récursives.
3. Réécrire les règles de manière équivalente :
   - Les anciennes règles sans récursion sont modifiées pour inclure le nouveau `non-terminal` en fin de règle.
   - Les anciennes règles récursives sont transformées pour inclure le nouveau `non-terminal`, permettant d'éliminer la récursion tout en conservant la possibilité de générer les mêmes chaînes.
   - Une règle spéciale est ajoutée pour le nouveau `non-terminal`, permettant de terminer la dérivation si nécessaire.
4. Supprimer les anciennes règles récursives et intégrer les nouvelles règles réécrites.
5. Répéter la même méthode pour chaque ensemble de règles correspondant à un non-terminal.

```python
def suppression_recursion_gauche(grammaire):
    nouvelles_regles = []
    non_terminals = sorted(set(g for g, _ in grammaire.
        regles))

    for Ai in non_terminals:
        regles_recursive = []
        regles_non_recursive = []

        for gauche, droite in grammaire.regles:
            if gauche == Ai:
                if droite[0] == Ai:
                    regles_recursive.append(droite[1:])
                else:
                    regles_non_recursive.append(droite)

        if regles_recursive:
            compteur = 1
            while True:
                Ai_prime = f"{Ai[0]}{compteur}"
                if all(Ai_prime != g for g, _ in grammaire.
                    regles):
                    break
                compteur += 1

            nouvelles_regles.extend((Ai, regle + [Ai_prime])
                for regle in regles_non_recursive)
            nouvelles_regles.extend((Ai_prime, regle + [
                Ai_prime]) for regle in regles_recursive)
            nouvelles_regles.append((Ai_prime, ['E']))
        else:
            nouvelles_regles.extend((Ai, droite) for droite
                in regles_non_recursive)

    grammaire.regles = nouvelles_regles
    return grammaire
```

5.5. Supprimer les occurrences dans l'en-tête :

Objectif :

La fonction réorganise les règles de la `grammaire` pour éliminer les occurrences des `non-terminaux` en début des parties droites des règles.

Approche :

1. Construire une liste triée de tous les `non-terminaux` présents dans la `grammaire`.
2. Pour chaque non-terminal, extraire les règles associées, puis vérifier si d'autres règles commencent par ce `non-terminal`.

3. Lorsqu'une règle commence par un `non-terminal`, la fonction remplace cette occurrence par les règles associées au `non-terminal` en question. Cela permet d'éliminer les dépendances directes tout en préservant l'équivalence de la `grammaire`.
4. Une fois les substitutions effectuées, les nouvelles règles sont intégrées à la `grammaire`. Les anciennes règles associées au `non-terminal` traité sont remplacées.
5. Appliquer ce processus pour tous les `non-terminaux`, garantissant que les parties droites ne commencent plus par un `non-terminal`.

```python
def supprimer_occurences_en_tete(grammaire):
    non_terminals = sorted(set(g for g, _ in grammaire.regles))

    for Ai in non_terminals:
        Ai_regles = [(g, d) for g, d in grammaire.regles if g == Ai]
        for Aj in non_terminals:
            if Ai != Aj:
                Aj_regles = [(g, d) for g, d in grammaire.regles if g == Aj]
                updated_Aj_regles = []
                for gauche, droite in Aj_regles:
                    if droite[0] == Ai:
                        updated_Aj_regles.extend(
                            (gauche, Ai_droite + droite[1:])
                                for _, Ai_droite in Ai_regles
                        )
                    else:
                        updated_Aj_regles.append((gauche, droite))
                grammaire.regles = [rule for rule in grammaire.regles if rule[0] != Aj] + updated_Aj_regles

    return grammaire
```

5.6. Regroupement des fonctions :

Comme vu dans le chapitre 5 nous utilisons les algorithmes dans l'ordre suivant : DEL -> UNIT -> SUPPRESSION DES RÉCURSIONS GAUCHE -> SUPPRESSION DES OCCURRENCES EN TÊTE -> axiom_premiere_ligne(grammaire). La fonction `greibach(grammaire)` nous permet de faire la transformation en `forme normale de greibach`. Toutes les fonctions vues sont dans un fichier `greibach.py` pour séparer la logique du code.

```python
def greibach(grammaire):
    grammaire = start(grammaire)
    grammaire = del_(grammaire)
    grammaire = unit(grammaire)
    grammaire = suppression_recursion_gauche(grammaire)
    grammaire = supprimer_occurences_en_tete(grammaire)

    # Reorganiser les regles pour que l'axiome soit en
        premiere ligne
    grammaire = axiome_premiere_ligne(grammaire)
    return grammaire
```

6. grammaire.py :

Ce fichier créé les `formes normales de Chomsky et Greibach`. Il importe les fonctions `chomsky(grammaire)` (partie 4.5), `greibach(grammaire)` (partie 5.5), `copie(liste)` (partie 4.4), `lire_grammaire()` (partie 3.1), `ecrire_general(grammaire, fichier)` (partie 3.2), et `sys` pour gérer les entrées et exécuter le code.

```python
from chomsky import chomsky
from greibach import greibach
from unit_del_copie import copie
from lire_ecrire import ecrire_general, lire_grammaire
import sys
```

1. La fonction vérifie qu'un argument est fourni via la ligne de commande et que ce dernier est un fichier avec l'extension `.general`.
2. Le fichier spécifié est lu et transformé en une structure de grammaire grâce à la fonction `lire_grammaire(fichier)`.
3. Deux copies de la `grammaire` sont créées afin d'appliquer indépendamment les transformations vers les `formes normales de Chomsky et Greibach`. Elles sont respectivement produite par les fonctions : `chomsky(grammaire)` et `greibach(grammaire)`.
4. Les grammaires transformées sont sauvegardées dans des fichiers avec des extensions adaptées, respectivement `.chomsky` et `.greibach`, en utilisant la fonction `ecrire_general(grammaire, fichier)l`. Les noms des fichiers sont dérivés du fichier d'entrée.

```python
def main():
    if len(sys.argv) != 2 or not sys.argv[1].endswith(".
        general"):
        print("Usage: grammaire.py <fichier.general>")
        sys.exit(1)

    fichier = sys.argv[1]
```

```
    grammaire = lire_grammaire(fichier)

    # Transformations
    grammaire_chomsky = chomsky(copie(grammaire))
    greibach_grammaire = greibach(copie(grammaire))

    # ecriture des grammaires
    nom_base = fichier.rsplit(".", 1)[0]
    ecrire_general(grammaire_chomsky, f"{nom_base}.chomsky")
    ecrire_general(greibach_grammaire, f"{nom_base}.greibach
        ")


if __name__ == "__main__":
    main()
```

7. generer.py :

7.1. Fonction générée :

La fonction `generer_mots(grammaire)` est conçue pour générer tous les mots possibles jusqu'à une longueur maximale spécifiée, en suivant les règles de production définies dans la grammaire donnée.

`Sous-fonction etendre() :`

1. Cette sous-fonction prend une séquence de symboles (terminaux et non-terminaux) et la transforme en de nouvelles séquences selon les règles de production de la grammaire.
2. Les séquences entièrement constituées de terminaux sont considérées comme des mots valides et sont retournées.
3. Lorsqu'un non-terminal est rencontré, il est remplacé par toutes les productions possibles dans les règles correspondantes, créant ainsi de nouvelles combinaisons.
   Une vérification garantit que la longueur des séquences générées ne dépasse pas la limite spécifiée.

`Gestion des cas de récursion :`

1. La récursion est limitée par `longueur_max` pour éviter une génération excessive ou infinie.
2. Une boucle `for` permet de parcourir les règles de production associées à l'axiome pour initialiser la génération des mots.

`Résultat final :`

Tous les mots générés sont collectés dans un ensemble pour éviter les doublons. Ils sont ensuite triés par `ordre lexicographique` avant d'être retournés.

```python
def generer_mots(grammaire, longueur_max):
    def etendre(symboles):
        if all(sym.islower() for sym in symboles):
            return {"".join(symboles)}

        resultats = set()
        for i, sym in enumerate(symboles):
            if sym.isupper():
                for gauche, droite in grammaire.regles:
                    if gauche == sym:
                        nouveaux_symboles = symboles[:i] +
                            droite + symboles[i+1:]
                        if len(nouveaux_symboles) <=
                            longueur_max:
                            resultats.update(etendre(
                                nouveaux_symboles))
                break
        return resultats

    mots = set()
    for gauche, droite in grammaire.regles:
        if gauche == grammaire.axiome:
            mots.update(etendre(droite))

    return sorted(mot for mot in mots if len(mot) <=
        longueur_max)
```

7.2. Main :

Nous allons maintenant voir la fonction `main()` qui gère la lecture d'un fichier `.chomsky` et `.greibach`. Ces fichiers contiennent des `grammaires` qui nous permettent de générer des mots, puis nous les écrivons dans des fichiers `.res`.

1. La fonction appelée `lire_grammaire(fichier)` extrait les règles et l'`axiome` de la `grammaire` depuis le fichier.
2. La fonction `generer_mots(grammaire)` est utilisée pour produire tous les mots possibles jusqu'à la longueur spécifiée, en suivant les règles de la `grammaire`. Cette génération garanti que les mots respectent les productions définies dans la `grammaire`.
3. Les mots générés sont écrits dans un fichier de sortie. Le nom de ce fichier est basé sur le nom du fichier de grammaire d'entrée, la longueur maximale et une extension `.res`.

```python
    def main():
    if len(sys.argv) != 3 or not (sys.argv[2].endswith(".
        greibach") or sys.argv[2].endswith(".chomsky")):
```

```
        print("Usage: generer.py <n> |<fichier.greibach> or
            <fichier.chomsky|")
        sys.exit(1)

    fichier_grammaire = sys.argv[2]
    longueur_max = int(sys.argv[1])

    # Lire la grammaire
    grammaire = lire_grammaire(fichier_grammaire)

    # Generer les mots
    mots = generer_mots(grammaire, longueur_max)
    # ecriture des mots
    ancien_nom = fichier_grammaire.rsplit(".", 1)
    nouveau_fichier = f"{ancien_nom[0]}_{longueur_max}_{
        ancien_nom[1]}.res"
    ecrire_generer(mots, nouveau_fichier)


if __name__ == "__main__":
    main()
```

8. Makefile :

Le fichier `Makefile` est utilisé pour automatiser l'exécution de nos scripts Python et la gestion des fichiers générés par le projet. Le `Makefile` permet d'automatiser les étapes de transformation de la grammaire, de générer des mots, ainsi que de nettoyer les fichiers intermédiaires.

`Les objectifs du Makefile sont :`

1. Exécuter les transformations de la grammaire en utilisant le script Python : `grammaire.py`.

2. Générer des mots grâce au script `generer.py` à partir des fichiers de grammaire au format `.chomsky` et `greibach`.

3. Nettoyer les fichiers générés pour éviter l'encombrement.

`Explication des règles du Makefile :`

1. `Variables` : Les variables définissent les fichiers d'entrée (`INPUTS`), le nom des fichiers sans extension (`INPUTS_NO_DOT`), ainsi que les scripts Python utilisés (`GRAMMAIRE_SCRIPT` et `GENERER_SCRIPT`).

2. `all` : La cible par défaut qui est exécutée lorsque vous tapez simplement `make`. Cette règle appelle la cible `run_examples`.

3. `run_examples` Cette règle exécute les transformations et la génération de mots. Elle traite les fichiers d'entrée spécifiés dans `INPUTS` en les passant aux scripts `grammaire.py` et `generer.py`. Pour chaque fichier, les transformations sont effectuées en `Chomsky` et en `Greibach`, et les mots générés sont enregistrés dans des fichiers de sortie appropriés.

4. `clean` : La règle de nettoyage qui supprime les fichiers générés (par exemple, les fichiers avec les extensions `.chomsky` et `.greibach`). Cela permet de nettoyer l'environnement après l'exécution des transformations.

Contenu du `Makefile` :

```
# Variables pour les fichiers d'entree et les scripts Python
INPUTS = test_cour_1.general test_cour_2.general
    test_complexe.general test_prof_1.general test_prof_2.
    general test_prof_3.general
INPUTS_NO_DOT = test_cour_1 test_cour_2 test_complexe
    test_prof_1 test_prof_2 test_prof_3
GRAMMAIRE_SCRIPT = grammaire.py
GENERER_SCRIPT = generer.py

# Taille par defaut (peut etre personnalisee via une
    variableo d'environnement ou la ligne de commande)
SIZES = 10 5 15 7 8 15

# Cible par defaut (executee lorsque "make" est tape)
all: run_examples

# Regle pour executer les programmes sur les fichiers d'
    exemple
run_examples:
    @echo "Execution des transformations avec $(
        GRAMMAIRE_SCRIPT)..."
    @for file in $(INPUTS); do \
        echo "Traitement de $$file"; \
        python3 $(GRAMMAIRE_SCRIPT) $$file; \
    done
    @echo "Generation de mots avec $(GENERER_SCRIPT)..."
    @set -- $(SIZES); \
    for file in $(INPUTS_NO_DOT); do \
        size=$$1; shift; \
        echo "Generation pour $$file.chomsky avec taille
            $$size"; \
```

```
25          python3 $(GENERER_SCRIPT) $$size $$file.chomsky; \
26          echo "Generation pour $$file.greibach avec taille
                $$size"; \
27          python3 $(GENERER_SCRIPT) $$size $$file.greibach; \
28      done
29
30  # Nettoyage (si necessaire, par exemple suppression des
        fichiers generes)
31  clean:
32      @echo "Nettoyage des fichiers generes..."
33      @rm -f *.chomsky *.greibach
```

9. Conclusion :

Dans ce dévellopement nous avons lu des `grammaires` puis nous les avons transformé en `forme norme de Chomsky et Greibach` en utilisant les algorithmes connu : START, TERM, BIN, DEL, UNIT, Suppression des récursions gauche et supprimer les occurrences dans l'en-tête. Peu de documentations liées aux automates on été trouvée, ce qui à rendu les étapes de création du `lexer` et du `parser` plus difficiles. Le fichier `Makefile` nous permet d'exécuter notre script et d'obtenir les mots via les fonctionnalités du fichier `grammaire.py` et `generer.py`.