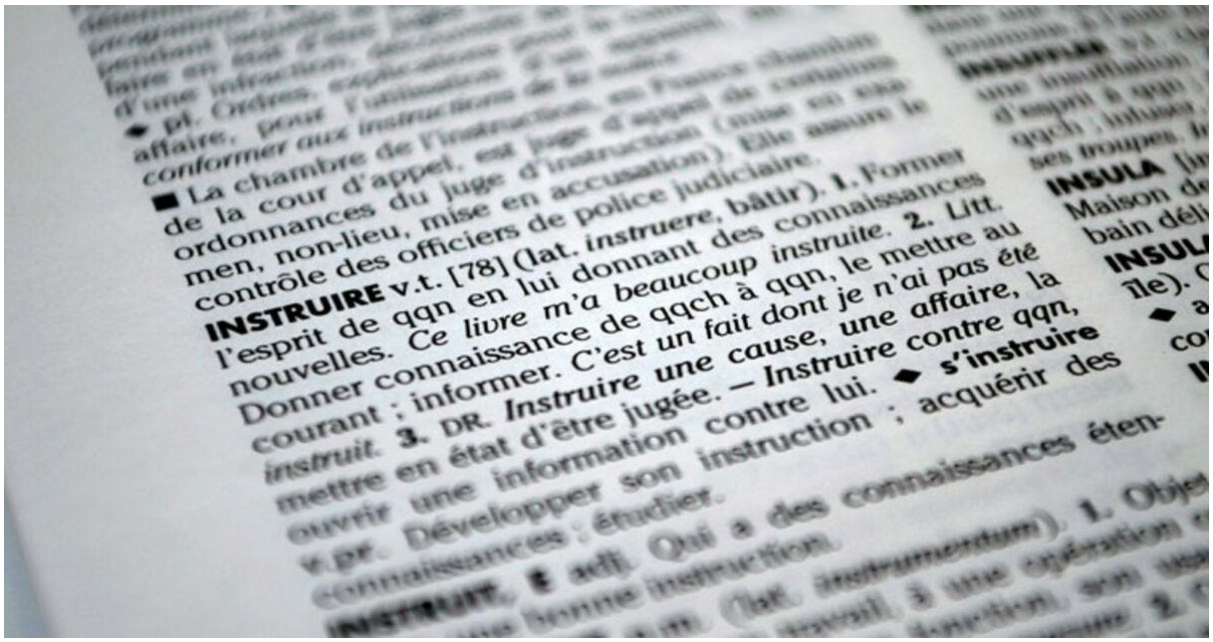
KOCOGLU Lucas
NOIROT-RATHAR Elisée
PINEDE Luka
L2 Classique - Promo 2026

GENERATEUR AUTOMATIQUE DE PHRASE

Rapport de Projet - Algorithmique et structure de données 2



TI301 – DEPARTEMENT INFORMATIQUE
PROFFESEURS : LANDSCHOOT MICHEL / NICOLAS FLASQUE

TABLE DES MATIERES

Introduction :

Dans le cadre du cours d'Algorithmique et structure de données 2, on nous a enseigné de nouvelles notions concernant la structure de données : listes chaînés, listes circulaires, listes doublement chaînés, piles, files, différents types d'arbres binaires, etc. Ces notions constituent les bases essentielles des structures de données.

Pour appliquer ces notions, nous avons été amenées, dans le cadre du projet, de créer un programme qui génère automatiquement des phrases. Le programme comporte 4 types de mots : *Nom*, *Adjectif*, *Adverbe* et *Verbe*. Les mots comportent aussi des sous-types grammaticaux (*Masculin*, *Féminin*, *Singulier*, *Pluriel*, …). Même si les phrases n'ont pas nécessairement de sens, ils doivent être correctement dans la mesure du possible grammaticalement et orthographiquement correct.

Ces données sont conservées dans le fichier **./dictionnaire/dictionnaire_non_accentue.txt**. (*Le projet n'utilise pas de dictionnaire avec accents.*)

Le projet comprend plusieurs structures :

- t_base_tree : Tête de l'arbre des formes de base
- t_base_node : Nœuds de l'arbre des formes de base
- t_flechie_list : Tête de liste des formes fléchies
- t_flechie_node : Nœuds de la liste des formes fléchies
- t_enum_list : Tête de liste des sous-types
- t_enum_node : Nœuds de la liste des sous-types

Sous chaque fonction est commenté leur utilité ainsi que les variables et leur type.

Un git est alimenté pour pouvoir collaborer sur le code. Le répertoire est disponible sur GitHub (https://github.com/ItsLucas93/Projet-L2-S3). Plus de 2000 lignes de codes à travers 120 commits constituent le projet.

Ce projet est composé de **KOCOGLU Lucas**, **NOIROT-RATHAR Elisée**, **PINEDE Luka**.

I. Structures des données

Structure des formes de base

Les arbres de base sont des arbres qui possèdent 27 fils, 26 fils pour les lettres de l'alphabet de a à z, et 1 fils pour des caractères spéciaux (' et -). Les fichiers qui opèrent sur les arbres sont les fichiers *base_tree.c/.h* et *base_node.c/.h* :

```c
// base_tree.h

#include "base_node.h"

struct s_base_tree
{
    struct s_base_node* root[ALPHABET_SIZE];
};

typedef struct s_base_tree s_base_tree, t_base_tree, *p_base_tree;

// base_node.h

#define ALPHABET_SIZE 27
#include "flechie_list.h"

struct s_base_node
{
    char value;
    struct s_base_node* fils[ALPHABET_SIZE];
    int nb_forme_flechie;
    struct s_flechie_list* flechie_list;
};

typedef struct s_base_node s_base_node, t_base_node, *p_base_node;
```

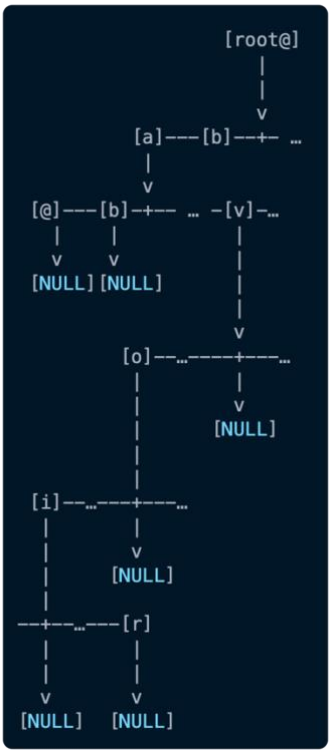
Les structures ressemblent à ceux utilisés en cours.

Dans le code, est généralement appelé des p_base_tree, qui par la fonction createEmptyBaseTree() (*présent dans base_tree.c:9*) reçoit la tête de l'arbre avec les 27 cases du root initialisé à NULL.

Nous avons choisi d'utiliser des tableaux pour les fils des arbres n-aires car facile d'utilisation : l'indice du tableau correspond à sa lettre associée (a = 0 ; b = 1 ; … ; z = 25 ; ' ou - = 26).

Pour la structure des nœuds :

- char value : reçois la lettre du mot (ex : avoir, premier nœud = a)
- struct s_base_node* fils[ALPHABET_SIZE] : pointeur sur les fils
- int nb_forme_flechie : nombre de forme fléchie à ce nœud (ex : avoir, au nœud r le nombre est de 44)
- struct s_flechie_list* flechie_list : reçois la liste des formes fléchies.



Exemple de la représentation de l'arbre, avec le verbe avoir :

Structure des formes fléchies

Les formes fléchies stockent les diverses variants des formes de base.
Les formes fléchies se comportent comme une liste chaînée. Nous l'avons choisi, car elle est simple à implémenter, à visualiser et à manipuler.

```
// flechie_list.h

#include "flechie_node.h"

struct s_flechie_list
{
   struct s_flechie_node* head;
};

typedef struct s_flechie_list s_flechie_list, t_flechie_list, *p_flechie_list;

// flechie_node.h

#include "enum_list.h"

struct s_flechie_node
{
   struct s_flechie_node* next; // node successeur
   char* value; // Mot sous forme fléchie
    t_enum_list *sub_type_list; // liste des sous_types
};

typedef struct s_flechie_node s_flechie_node, t_flechie_node, *p_flechie_node;
```

Le p_flechie_list reçoit la tête de l'arbre par la fonction createFlechieNode() (présent dans flechie_node.c:6).

```
// flechie_node.c

p_flechie_node createFlechieNode()
/*
 * Fonction: createFlechieNode
 * ---------------------
 * Crée un Node pour les formes fléchies
 * next initialisé à NULL
 * liste des sous_type reçoit un pointeur vers une liste de sous_type créée
 *
 * pn: p_enum_node
 */
{
    p_flechie_node nouv = (p_flechie_node) malloc (sizeof (t_flechie_node));

    nouv->next = NULL;
    nouv->sub_type_list = createEmptyEnumList();
    nouv->value = (char*) malloc (sizeof (char) * LENGHT_MAX);

    return nouv;
}
```

Le p_flechie_node stocke le mot en entier, possède un pointeur vers une autre forme fléchie si elle existe. Cette structure possède aussi une tête de liste des sous-types associés à la forme fléchie.

Exemple de structure de nœud p_flechie_node du nœud 'r' du verbe avoir :

s_flechie_node* next	char* value	t_enum_list *sub_type_list
@ -> ai -> aie -> … -> NULL	A	@ -> IPre -> SG -> P3 -> NULL

Structures des sous-types

Les sous-types sont associés à leur forme fléchie, ils décrivent, selon leur type : leur genre, leur nombre, …
Les sous-types ont une structure de type enum. Vus dans le cours en S2 (TI202 – Algorithmique et Structure de donnée 1, cours sur les types et les structures), les types énumérés sont particuliers.

```
// enum_node.h

enum sub_type
{
    // Genre (Masculin / Féminin - ADJ/NOM)
    Mas, Fem,
    // Nombre (Singulier / Pluriel - ADJ/NOM/VER)
    SG, PL,
    // Personne (1e 2e 3e personne - VER)
    P1, P2, P3,

    // Verbe
    // (Infinitif)
    Inf,
    // (Participe passé / Participe Présent)
    PPas, PPre,
    // (Indicatif : Présent / Passé simple / Imparfait / Futur)
    IPre, IPSim, IImp, IFut,
    // (Subjonctif : Présent / Passé (??) / Imparfait / Plus que parfait (??)) |
    SPre, SImp,
    // (Conditionnel : Présent)
    CPre,
    // Impératif (Présent / Passé(x)) | Pas d'impératif passé
    Imp, ImPre,

    // (Invariant pluriel/Invariant genre - ADJ/NOM)
    InvPL,InvGen,

    //Adverbe
    // Adv, <- useless, pas de formes fléchies dans les adverbes puisque invariable

    // NULL
    null,

};

typedef enum sub_type sub_type;
```

Chaque élément de l'énumération est associé à un entier, en interne (implicitement, ici Mas = 0 ; Fem = 1 ; …). Ils sont pratiques d'utilisation, car au lieu de récupérer les indices des cases, on peut les faire correspondre aux sous-types par un test logique.

Exemple : Si enum_node->value = Mas ; et qu'on fait enum_node->value == Mas, cela nous retourne 1. Cela nous facilite la tâche dans la reconnaissance des sous-types.

La structure des sous-types est identique aux listes chaînées :

```c
// enum_list.h

#include "enum_node.h"

struct s_enum_list
{
    struct s_enum_node *head;
};

typedef struct s_enum_list s_enum_list, t_enum_list, *p_enum_list;

// enum_node.h

struct s_enum_node
{
    sub_type value;
    struct s_enum_node *next;
};

typedef struct s_enum_node s_enum_node, t_enum_node, *p_enum_node;
```

Le p_enum_list reçoit la tête de l'arbre par la fonction createEnumNode() (présent dans enum_node.c:7).
Le p_enum_node stocke le sous_type et possède un pointeur vers une autre forme nœud sous-type si elle existe.
La gestion est simple.

Le nœud est créé comme dans le cours avec un malloc et le pointeur du successeur à NULL.

```c
// enum_node.c

p_enum_node createEnumNode()
{
    p_enum_node nouv = (p_enum_node) malloc (sizeof(t_enum_node));
    nouv->next = NULL;

    return nouv;
}
```

II. Création de l'arbre

Dans cette partie 2, nous allons parler de la création de l'arbre. Les fonctions se situent généralement dans le fichier file_manager.c/.h

La création de l'arbre se fait dans la fonction createTypedTrees() (file_manager.c:8).

Extraction du fichier

Regardons tout d'abord la première partie de cette fonction :

```c
// file_manager.h

#define PATH "../dictionnaire/dictionnaire_non_accentue.txt"

// file_manager.c

#include <sys/stat.h>
#include <stdio.h>

void createTypedTrees(p_base_tree tree_nom, p_base_tree tree_adj, p_base_tree tree_adv, p_base_tree tree_verb)
{
    const char *filename = PATH;
    FILE *input_file = fopen(filename, "r");

    struct stat sb;
    if (stat(filename, &sb) == -1) {
        perror("stat");
        exit(EXIT_FAILURE);
    }

    int nbligne = 0;
    for (char c = getc(input_file); c != EOF; c = getc(input_file)) {
        if (c == '\n') {
            nbligne++;
        }
    }
    printf("Nombres de mots : %d\n", nbligne);
    fclose(input_file);

/.../
}
```

La variable PATH est stockée en définition dans le fichier file_manager.h.

Au début de la fonction, on ouvre le fichier en mode lecture, on récupère le nombre de ligne que contient le fichier. Qui dit nombre de ligne veut dire nombre de mots (fléchis). Ensuite, on initialise des variables de type char, qui vont stocker les données des lignes qu'on va récupérer juste après.

Injection dans l'arbre

```c
// file_manager.c

void createTypedTrees(p_base_tree tree_nom, p_base_tree tree_adj, p_base_tree tree_adv, p_base_tree tree_verb)
{
/.../
    input_file = fopen(filename, "r");

    char forme_base[ALPHABET_SIZE];
    char type[100];
    char forme_flechie[ALPHABET_SIZE];

    int i = 0;
    while (i < nbligne) {

        fscanf(input_file, "%s\t%s\t%s", forme_flechie, forme_base, type);

        if (compareType(type, "Ver:"))
        {
            p_base_node ptr_last_node_base = insertBaseTree(tree_verb, forme_base);
            p_flechie_node ptr_flechie_node = insertFlechieList(ptr_last_node_base, forme_flechie);
            addTypeToFlechieList(ptr_flechie_node, type);
        }
        else if (compareType(type, "Adj:"))
        {
            p_base_node ptr_last_node_base = insertBaseTree(tree_adj, forme_base);
            p_flechie_node ptr_flechie_node = insertFlechieList(ptr_last_node_base, forme_flechie);
            addTypeToFlechieList(ptr_flechie_node, type);
        }
        else if (compareType(type, "Adv")) {
            p_base_node ptr_last_node_base = insertBaseTree(tree_adv, forme_base);
            insertFlechieList(ptr_last_node_base, forme_flechie);
        }
        else if (compareType(type, "Nom:")) {
            p_base_node ptr_last_node_base = insertBaseTree(tree_nom, forme_base);
            p_flechie_node ptr_flechie_node = insertFlechieList(ptr_last_node_base, forme_flechie);
            addTypeToFlechieList(ptr_flechie_node, type);
        }
        clearTabChar(forme_base);
        clearTabTypedChar(type);
        clearTabChar(forme_flechie);
        i++;
    }
    fclose(input_file);

}
```

Cette partie de la fonction traite les injections dans les arbres.

Les variables forme_base ; forme_flechie ; type récupèrent les données provenant de la ligne.

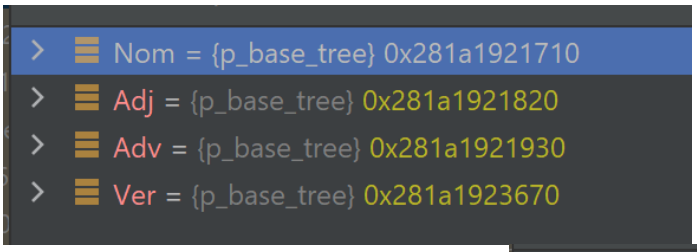
Voici les étapes pour chaque ligne du fichier dictionnaire :

1. On compare le début de la partie des types pour voir dans quel arbre sera injecté le mot.
2. Avec la fonction insertBaseTree() (présent dans base_tree.c:29), on construit les étages s'ils n'existent pas avec le mot et on récupère le pointeur vers le dernier nœud.
3. Avec la fonction insertFlechieList() (présent dans file_manager:91), on construit la liste si elle n'existe pas au bout de ce nœud, sinon on ajoute à cette liste un nouveau nœud et ajoute 1 au compteur de nombre de formes fléchies. On récupère un pointeur vers le nœud stockant la forme fléchie.
4. Avec la fonction addTypeToFlechieList() (présent dans flechie_list.c:109), on construit la liste des sous-types associé à la forme fléchie s'il n'existe pas, sinon on ajoute un nouveau nœud à cette liste.

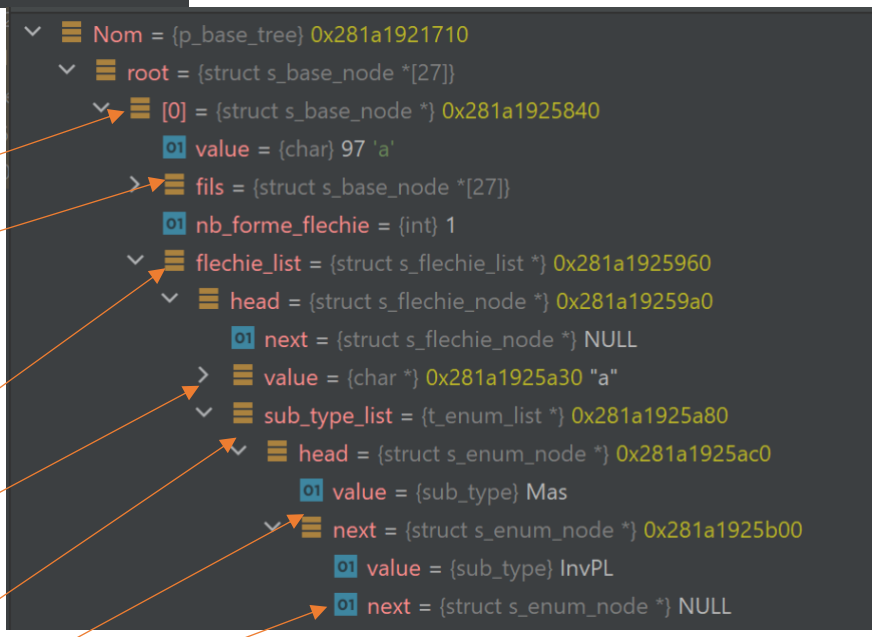
5. À chaque tour, on nettoie les variables forme_base ; forme_flechie ; type.

Remarque : la branche de l'arbre des adverbes n'appelle pas la fonction addTypeToFlechieList().
En effet, les adverbes étant tout de même type, pas besoin de posséder des formes fléchies.

Exemple de représentation (visualisation des variables sous debug mode):



← Arbre selon les types



Nœud de la forme de base

Nœud fils

Liste des formes fléchies

Forme fléchie

Liste des sous-types

Sous-type

III. Recherche / extraction de mots

Ces fonctions sont présentes dans le fichier base_tree.c/.h

Recherche de forme de base

```c
// base_tree.c

void rechercheFormeBase(p_base_tree Verb, p_base_tree Adj, p_base_tree Adv, p_base_tree Nom, const char* chaine)
{
    if (strlen(chaine) == 0 || (strlen(chaine) > 25))
    {
        printf("Chaine incompatible (votre chaine est = 0 ou > à 25) !\n");
        return;
    }

    p_base_node pn = isBaseInTree(Nom, chaine);
    printf("-=-=-=-=-=-=-= Nom =-=-=-=-=-=-=-\n");
    if (pn == NULL)
    {
        printf("Forme de base non trouve.\n");
    }
    else
    {
        printf("%s trouve ! Type : Nom\n"
               "Forme(s) flechie(s) : \n", chaine);
        printFlechieList(pn->flechie_list);
    }

    pn = isBaseInTree(Adj, chaine);
    printf("-=-=-=-=-=-=-= Ajdectif =-=-=-=-=-=-=-\n");
    if (pn == NULL)
    {
        printf("Forme de base non trouve.\n");
    }
    else
    {
        printf("%s trouve ! Type : Adjectif\n"
               "Forme(s) flechie(s) : \n", chaine);
        printFlechieList(pn->flechie_list);
    }

    pn = isBaseInTree(Adv, chaine);
    printf("-=-=-=-=-=-=-= Adverbe =-=-=-=-=-=-=-\n");
    if (pn == NULL)
    {
        printf("Forme de base non trouve.\n");
    }
    else
    {
        printf("%s trouve ! Type : Adverbe\n"
               "Forme(s) flechie(s) : \n%s:Adverbe", chaine, chaine);
    }

    pn = isBaseInTree(Verb, chaine);
    printf("-=-=-=-=-=-=-= Verbe =-=-=-=-=-=-=-\n");
    if (pn == NULL)
    {
        printf("Forme de base non trouve.\n");
    }
    else
    {
        printf("%s trouve ! Type : Verbe\n"
               "Forme(s) flechie(s) : \n", chaine);
        printFlechieList(pn->flechie_list);
    }
    printf("-=-=-=-=-=-=-= Fin de la recherche =-=-=-=-=-=-=-\n");
}
```

Le but de cette fonction est de parcourir les 4 arbres avec la chaîne de caractère saisi par l'utilisateur (qui est implémenté dans le main.c). Tout d'abord, étant donné que c'est une entrée utilisateur, si les caractères sont hors plage, la fonction s'arrête.

Voici les étapes pour chaque arbre :

- Vérifier si le mot est présent dans l'arbre avec la fonction isBaseInTree() (fonction présent dans base_tree.c:94).
  - ⇒ Si c'est le cas, affiches les formes fléchies associé à la forme de base.
  - ⇒ Sinon indique que le mot n'est pas présent dans l'arbre.

Exemple d'exécution :

```
Generateur de phrase
Menu principal. Choisissez votre menu :
1 - Modele 1 : nom - adjectif - verbe - nom
2 - Modele 2 : nom - 'qui' - verbe - verbe - nom - adjectif
3 - Modele 3 : (Modele personnalise) adverbe - nom - adjectif - verbe
4 - Menu de recherche de mots
5 - Quitter le programme.
4
Menu recherche de mots. Faites votre choix :
1 - Rechercher un mot parmi les formes de base
2 - Mot aleatoire parmi les formes de base
3 - Rechercher un mot parmi les formes flechies
4 - Retour au menu principal.
1
Inserez un mot :ecole
 -=-=-=-=-=-= Nom =-=-=-=-=-=-
ecole trouve ! Type : Nom
Forme(s) flechie(s) :
ecole : Feminin Singulier ;
ecoles : Feminin Pluriel ;
-=-=-=-=-=-= Ajdectif =-=-=-=-=-=-
Forme de base non trouve.
-=-=-=-=-=-= Adverbe =-=-=-=-=-=-
Forme de base non trouve.
-=-=-=-=-=-= Verbe =-=-=-=-=-=-
Forme de base non trouve.
-=-=-=-=-=-= Fin de la recherche =-=-=-=-=-=-
```

Extrait du dictionnaire :

ecole	ecole	Nom:Fem+SG
ecoles	ecole	Nom:Fem+PL

Extraction forme de base aléatoire

```c
// base_tree.c

char* extraireRandomBase(p_base_tree Verb, p_base_tree Adj, p_base_tree Adv, p_base_tree Nom, int select_tree)
{
    char* base = (char*) malloc (ALPHABET_SIZE * sizeof(char));
    for (int z = 0; z < ALPHABET_SIZE ; z++) base[z] = '\0';

    int c = 0;

    srand(time(NULL));

    if (select_tree == 0)
    {
        do {
            select_tree = rand() % 5;
        } while (select_tree == 0);
    }

    int letter = rand () % 26;   //aléatoirement l'abre commençant par une lettre
    p_base_node node = NULL;
/.../
}
```

On génère une seed aléatoire pour la fonction random.
Selon le paramètre select_tree, si elle est égale à 0, on va choisir un arbre au hasard pour extraire la forme de base.
Ensuite, on se positionne sur une lettre au hasard.

La fonction va parcourir l'arbre en choisissant le fils de façon aléatoire et qui est non nulle.
Il stocke toutes les lettres des nœuds parcourus dans un variable char.
Lorsqu'on atteint un nœud qui possède un nombre de forme fléchis non nulle, on lance une pièce (1 chance sur 2) : soit on continue le parcours (en ayant vérifié que des fils sont disponibles), soit on s'arrête et on renvoie la chaîne de caractère.

```c
// base_tree.c

char* extraireRandomBase(p_base_tree Verb, p_base_tree Adj, p_base_tree Adv, p_base_tree Nom, int select_tree)
{
/.../ // <-- choix de l'arbre
    int suite = 1;
    int i;

    while (node->nb_forme_flechie == 0 || suite == 1) {//tanque je n'atteint pas une forme fléchie et qu'il existe une suite
        suite = 0;

        i = rand() % 26;
        while (node->fils[i] == NULL) {  //je vais dans un fils aléatoirement
            i = rand() % 26;
        }
        base[c] = node->value; //je stocke le char dans la forme de base
        node = node->fils[i];  // je vais dans le fils
        c++;                   // je passe au caractère suivant

        int est_vide = 1;  //je regarde si il possède des fils
        for (int j = 0; j < 26; j++) {
            if (node->fils[j] != NULL) {
                est_vide = 0;
            }
        }
        if (node->nb_forme_flechie > 0 && est_vide == 0) {// si oui on tire àléatoirement une pièce pour savoir si on continue ou si on s'arrête
            suite = rand() % 2;
        }
    }
    //la boucle s'arrête à l'avant dernière lettre et ne prends pas la dernière lettre, parce que dans le dernier noeud la forme fléchie est présente
    base[c] = node->value;

    base[c + 1] = '\0';
    return base; //je renvoie la forme de base

/.../
}
```

Exemple d'exécution :

```
Nombres de mots : 287976
Initialisation des arbres termine.


Generateur de phrase
Menu principal. Choisissez votre menu :
1 - Modele 1 : nom - adjectif - verbe - nom
2 - Modele 2 : nom - 'qui' - verbe - verbe - nom - adjectif
3 - Modele 3 : (Modele personnalise) adverbe - nom - adjectif - verbe
4 - Menu de recherche de mots
5 - Quitter le programme.
4
Menu recherche de mots. Faites votre choix :
1 - Rechercher un mot parmi les formes de base
2 - Mot aleatoire parmi les formes de base
3 - Rechercher un mot parmi les formes flechies
4 - Retour au menu principal.
2

Mot aleatoire : apaisant
```

Extrait du dictionnaire :

- apaisant apaisant Adj:Mas+SG
- apaisant apaiser Ver:PPre
- apaisante apaisant Adj:Fem+SG
- apaisantes apaisant Adj:Fem+PL
- apaisants apaisant Adj:Mas+PL

IV. Génération des phrases

3 fonctions composent cette partie, présent dans le fichier phrase.c/.h.
Chaque fonction s'occupe d'un modèle différent.
Le modèle 1 se compose de cette manière : Nom – Adjectif – Verbe – Nom
Le modèle 2 se compose de cette manière : Nom – 'qui' – Verbe – Verbe – Nom – Adjectif
Le modèle 3 se compose de cette manière : Adverbe - ',' - Nom - Adjectif – Verbe

```c
// phrase.c

void createSentenceModel1(p_base_tree Ver, p_base_tree Adj, p_base_tree Adv, p_base_tree Nom)
{
    srand(time(NULL));

    int genre = rand() % 2;
    int nombre = rand() % 2;
    int determinant = rand() % 2;

    if (nombre == 0) {
        if (genre == 0) {
            if (determinant == 0) printf("Le");
            else printf("Un");
            for (int i = 0; i < 4; i++) {
                srand(i);
                if (i == 0 || i == 3) createWordGenre(Nom, Mas, SG);
                else if (i == 2) createWordVerb(Ver, (rand() % 4) + 10, SG, P3);
                else if (i == 1) createWordGenre(Adj, Mas, SG);
            }
        } else {
            if (determinant == 0) printf("La");
            else printf("Une");
            for (int i = 0; i < 4; i++) {
                srand(i);
                if (i == 0 || i == 3) createWordGenre(Nom, Fem, SG);
                else if (i == 2) createWordVerb(Ver, (rand() % 4) + 10, SG, P3);
                else if (i == 1) createWordGenre(Adj, Fem, SG);
            }
        }
    /.../
    printf(".\n");
}
```

Les fonctions de génération de mots se calquent sur le même principe de l'extraction aléatoire d'une forme de base, mais à la dernière partie d'aléatoire si continue ou non, on regarde si les formes fléchies associé à ce mot possèdent les types correspondant au genre et au nombre, tiré aléatoirement.

3 fonctions composent la génération de mots :

- createWordGenre() : génère les mots de type Adjectif et Nom. Ils sont typés en fonction de leur genre et de leur nombre (pour la partie orthographique). Remarque : aléatoirement (1 chance sur 2), ils peuvent retourner des mots invariables.
- createWordAdv() : génère un mot de type Adverbe. Rapide, car pas de vérification si un type existe ou non, parcours juste l'arbre de manière aléatoire.
- createWordVerb() : génère les mots de type Verbe en fonction du temps, du nombre (P1/P2/P3) et pluriel (SG/PL).

Remarque : Le code fait en sorte que la phrase commence par une majuscule et se termine par un point.

Exemple d'exécution :

```
Nombres de mots : 287976
Initialisation des arbres termine.


Generateur de phrase
Menu principal. Choisissez votre menu :
1 - Modele 1 : nom - adjectif - verbe - nom
2 - Modele 2 : nom - 'qui' - verbe - verbe - nom - adjectif
3 - Modele 3 : (Modele personnalise) adverbe - nom - adjectif - verbe
4 - Menu de recherche de mots
5 - Quitter le programme.
1
Les lokoums vioques frictionnerent ex.
```

Les loukoums (Nom : Mas+PL) vioques (Adj : InvGen+PL) frictionnerent (Ver : IPSim+PL+P3) ex (Nom : InvGen+PL).

```
Menu principal. Choisissez votre menu :
1 - Modele 1 : nom - adjectif - verbe - nom
2 - Modele 2 : nom - 'qui' - verbe - verbe - nom - adjectif
3 - Modele 3 : (Modele personnalise) adverbe - nom - adjectif - verbe
4 - Menu de recherche de mots
5 - Quitter le programme.
2
Une gryphee qui dehouilla jeune une mafia miro.
```

Une gryphee (Nom : Fem+SG) qui dehouilla (Ver : IPSim+SG+P3) jeune (Ver : IPre+SG+P3) une mafia (Nom : Fem+SG) miro (Adj : InvGen+SG).

```
Menu principal. Choisissez votre menu :
1 - Modele 1 : nom - adjectif - verbe - nom
2 - Modele 2 : nom - 'qui' - verbe - verbe - nom - adjectif
3 - Modele 3 : (Modele personnalise) adverbe - nom - adjectif - verbe
4 - Menu de recherche de mots
5 - Quitter le programme.
3
Formidablement, un bezoard lilial chevillera.
```

Formidablement (Adv), un bezoard (Nom : Mas+SG) lilial (Adj : Mas+SG) chevillera (Ver : IFut+SG+P3).

V. Recherche de mot sous forme fléchie

Difficultés et tentative d'implémentation

Pour la recherche de forme fléchie :

Comme l'indiquait l'énoncé, nous avons voulu utiliser le backtracking.
Pour cela, nous avons utilisé, des piles.
Nous avons voulu utiliser une pile "search", qui va stocker, les pointeurs vers les nœuds.
Ainsi qu'une deuxième pile "un_search", qui va stocker, les pointeurs des nœuds dans laquelle la forme fléchie ne figure pas.
Pour ne pas repasser dans les nœuds dans lesquels nous somment passés.

Et enfin une troisième pile "word", pour stocker les caractères, des nœuds visités. Elle est vidée en même temps que la "search".

On parcourt donc notre arbre à partir du sous-arbre le plus à gauche, jusqu'au sous-arbre le plus à droite, jusqu'à atteindre la forme fléchie qui correspond.

Une partie du code :

Quand on trouve un nœud qui contient une forme, on teste si la forme fléchie donnée est présente, si oui, on s'arrête.
Sinon on regarde s'il existe une suite dans les fils :
Si oui, on stocke l'adresse du nœud dans "un_search" pour ne plus rechercher dedans.
Sinon on dépile "search" dans "un_search". Et on dépile aussi "word".

Si l'on ne l'a pas trouvé après avoir parcouru tout l'arbre, on envoie un message disant que la forme ne se trouve pas dedans.

Ainsi nous parcourons notre arbre jusqu'à atteindre le dernier nœud.

Mais, nous n'avons pas réussi à implémenter cette fonctionnalité en raison d'un manque de temps.

VI. Phase de conclusion

Liste des fonctions implémentés

1. Génération de phrase selon modèle 1 : Nom – Adjectif – Verbe – Nom
2. Génération de phrase selon modèle 2 : Nom – ‘qui’ – Verbe – Verbe – Nom – Adjectif
3. Génération de phrase selon modèle 3 : Adverbe - ',' - Nom - Adjectif - Verbe
4. Recherche de forme de base
5. Génération aléatoire de forme de base
6. ~~Recherche de forme fléchie~~ - *Partie non achevé*

Organisation du travail

Le projet a été l'occasion de pratiquer le travail de groupe par GitHub : la publication de code, la modification de code et la réception du travail du partenaire. Dans l’ensemble, c’est apprendre à travailler à plusieurs et à fusionner les codes en gardant une trace écrite. Cela nous a permis une bonne coordination au niveau de la répartition des tâches et aussi de réorganisation en cas d'imprévu.

Difficultés rencontrées

Nous avons rencontré des difficultés au niveau de la gestion de la mémoire. En effet, le code ne s’exécute pas sur les systèmes macOs et Linux pour des raisons que nous supposons être des problèmes de mémoire, problème qui n'est pas apparu lorsque le code est exécuté sur Windows.

Nous avons aussi eu un problème sur le respect des répartitions des tâches. Malheureusement, en raison d’un manque de participation de la part de l’un des membres du groupe, les tâches ont dû être réassignées, ce qui a engendré du retard au niveau des deadlines.

Le cours nous n’a pas montré comment gérer un fichier. Nous avons dû l’apprendre nous-même, ce qui nous a fait perdre beaucoup de temps.