# Rapport Projet S5 : Shakespearian Monkeys.

LE MÉTAYER Corentin
PÉMÉJA Tanguy

December 14, 2018

**Team n°6**

[1]

[1] source : https://en.wikipedia.org/wiki/Infinite_monkey_theorem_in_popular_culture

# Contents

# 1 Explication brève du projet :

## 1.1 Introduction:

"Shakesperians Monkeys" ou "Paradoxe du singe savant" est un théorème selon lequel un singe qui taperait sur les touches d'une machine à écrire aléatoirement et indéfiniment pourrait "presque sûrement" écrire du texte ayant un sens. Il pourrait alors recréer, par exemple un texte de Shakespeare si il passe un temps infini à écrire des choses aléatoires sur une machine à écrire.
Cependant, ce théorème part du principe que les singes tapent de façon purement aléatoire, or, ça ne serait peut-être pas le cas dans la réalité avec des singes entraînés pour cela, avec une organisation spéciale et des singes ayant pris connaissance du texte.
Ce projet consiste donc à simuler le comportement de ces singes et à essayer de leur faire reproduire, ici, l'intégralité des Sonnets de Shakespeare, ou même un texte d'une qualité supérieure (en étant optimiste).

## 1.2 Base Version :

Nous décidons de créer 3 singes :

- Le singe lecteur : il lit les mots du texte donné et les compte (ici, les sonnets de Shakespeare) pour les "amener" au singe statisticien et au singe imprimeur
- Le singe statisticien : il prend les mots lus par le singe lecteur, en fait l'inventaire et compte ceux qui reviennent plusieurs fois.
- Le singe imprimeur : il prend les mots lus par le lecteur et les imprime.

## 1.3 Achievement 1:

Cependant, le but de ce projet n'est pas de simplement compter les mots, mais de produire du texte "original". Pour cela, nous décidons de créer un nouveau singe, et de faire une modification sur le statisticien :

- Maintenant, le singe statisticien ne compte plus seulement le nombre d'apparition des mots, il fait aussi une liste des mots qui suivent un mot quelconque dans le texte.
- Le singe écrivain : Il prend au hasard un mot du texte, et grâce au statisticien, et choisit un nouveau mot parmi ceux qui le suivent, toujours au hasard, jusqu'à former une phrase. Il envoie ensuite ces mots à l'imprimeur, qui s'occupe de les afficher.

Nous avons donc de nouvelles phrases qui peuvent être créées avec un certain sens (du moins, il n'y a pas d'incohérence entre deux mots qui se suivent).

# 2 Cadre de travail:

Pour ce projet, nous nous sommes surtout partagés le travail sur l'écriture des différents singes. En revanche, le travail sur l'appel des singes, le fichier base (main) et l'écriture des structures s'est plutôt faite en commun. Ce projet s'est fait en language C. Le travail en binôme a pu être facilité par l'utilisation de git, qui est un dépot. Ceci permettait aussi à la forge de faire des tests sur nos programmes.

# 3 Algorithmes et structures:

## 3.1 Base version:

La base version consistait surtout à la création des 3 singes et à les faire "fonctionner" ensemble. Pour cela, nous avons utilisé des files où les mots sont stockés, comme dans une file d'attente dans la réalité (le mot qui vient d'arriver dans la file devra attendre que les autres mots devant lui soient partis avant de pouvoir lui même partir).
Nous en avons utilisé 2. Une file contient les mots du texte ajoutés par le lecteur, pour l'imprimeur et le statisticien. L'autre file n'est utilisée que par le statisticien, elle contiendra des cellules, qui sont des structures (une sorte de paquet) contenant des données utiles : le mot, le nombre d'occurrences de ce mot (modifié par le statisticien au fur et à mesure), et du pointeur "next" qui

"montre"(pointe sur) la cellule suivante.
Enfin, pour stocker ces cellules (et définir l'espace mémoire nécéssaire), nous avons utilisé une memory pool (amas de cellules). Au début de l'algorithme, nous initialisons un certain nombre de cellules vides, ainsi qu'un pointeur qui pointe sur la première cellule libre (ces cellules seront alors remplies au fur et à mesure ).
Pour la condition d'arrêt du programme, nous avons incorporé un système de tours avec un nombre de tours max et un système de grève des singes. Le singe lecteur se met en grève si il a atteint la fin du texte, le singe statisticien se met en grève si il a atteint le mot à la fin de la file a déjà été vu, et le singe imprimeur se met en grève quand la file est vide ou que le prochain mot à imprimer n'a pas encore été vu par le statisticien.
Le programme s'arrête quand tous les singes sont en grève ou qu'on a dépassé le nombre maximum de tours de jeu.

### 3.1.1 Structures

Tout d'abord, nous avons utilisé un tableau d'entiers contenant diverses données:

- la détection de caractère de fin de fichier
- le compteur de mot du lecteur
- le compteur de mot de l'imprimeur
- le compteur du statisticien
- le nombre d'occurences minimum
- le nombre d'occurences maximum
- le nombre de tours déjà passés
- indication si le dernier mot de la file est vu
- indication si la file est vide

Ce tableau est utile car il permet d'éviter d'avoir beaucoup de variables qui sont réutilisées parfois dans une même fonction au même endroit (le code est raccourci).
Cependant, l'inconvénient de cette technique est qu'il est utilisé dans beaucoup de fonctions, et donc que cela pourrait créer quelques dépendances.
Ensuite, pour les autres autres structures :

- la cellule:

Listing 1: cellule

```
1 struct cellule:
2     mot : Chaine de caracteres
3     noccs : Nombre d'occurences du mot dans le texte
4     next : Pointeur vers la cellule suivante
```

La cellule permet d'accéder simplement au mot et son nombre d'occurences dans le texte.

- mot:

Listing 2: mot

```
1 struct mot:
2     word : Chaine de caracteres
3     precedent : Pointeur sur le mot precedent dans la file
4     suivant : Pointeur sur le mot suivant dans la file
5     vu : entier indiquant si le mot a ete vu par le statisticien
```

La structure mot est plus adaptée pour la gestion de la file commune au lecteur, imprimeur et statisticien que les cellules. Elle permet notamment une impression en temps constant.

- la file:

Listing 3: file

```
struct file:
    debut : Pointeur vers mot au debut de la file
    fin : Pointeur vers le mot a la fin de la file
```

La file permet de stocker les mots simplement.

- la pool:

Listing 4: pool

```
struct pool:
    max_mots : Tableau
    mot_libre : Pointeur vers le prochain mot libre
```

max_mots est un tableau initialisé à la taille du maximum de mots dans le texte.
La pool permet de réserver un espace mémoire à la création de cellule. Il existe un pool similaire pour la création de mots.

### 3.1.2 Algorithmes:

Voici le principe des algorithmes utilisés :

- Singe Lecteur :

Listing 5: singe_lecteur

```
procedure singe_lecteur(texte):
    mot:=lire_mot(texte)
    If (pas_fin_fichier):
        ajouter_file(mot, file)
```

Ici, lire_mot va lire le prochain mot du texte. pas_fin_fichier vrai si on a pas atteint la fin du fichier. ajouter_queue ajoute le mot à la fin de la file.

- Singe Statisticien :

Listing 6: singe_statisticien

```
procedure singe_statisticien()
    mot_present:=FAUX
    While(cellule_suivante_non_libre):
        mot1:=mot(file_lecteur)
        mot2:=mot(file_statisticien)
        If (mot1=mot2):
            increm_occur_mot
            mot_present:=VRAI
        cellule_suivante
    If (Not mot_present):
        ajouter_cellule_file_statisticien
```

mot renvoie le dernier mot de la file. increm_occur_mot incrémente le nombre d'occurences du mot dans sa cellule (cf Structures). mot_present indique si le mot est déjà dans la file du statisticien ou pas. ajouter_cellule_file_statisticien crée la cellule contenant ce mot et l'ajoute sur la file du statisticien. cellule_suivante passe à la cellule suivante.

- Singe imprimeur:

Listing 7: singe_imprimeur

```
procedure singe_imprimeur():
    afficher(file)
    defiler(file)
```

Ici, afficher affiche le premier mot de la file (du lecteur), défiler enlève le premier mot de la file (du lecteur).

- Travail des singes :

Listing 8: travail_singes

```
procedure travail_singes():
    texte:=ouvrir_texte
    While((singes_actifs) And (pas_fin_tours)):
        travail_singes
        modif_etat_singes
    calculer_resultats
    afficher_resultats
    fermer_texte
```

Ici, ouvrir_texte ouvre le texte (passé en paramètre). Singes_actifs renvoie l'état des singes (grève ou pas). pas_fin_tours vérifie si on a pas dépassé le nombre de tours max. travail_singes sélectionne un singe au hasard et le fait travailler (exécute un des algorithmes singes ci-dessus). modifs_etat_singes modifie l'état des singes (grève ou pas). calculer_resultats calcule des données utiles à afficher (nombre min et max d'occurences d'un mot, nombre de mots imprimés, lus et différents. affiche_resultats affiche les résultats qu'on vient de calculer.

## 3.2 Achievement1:

L'achievement 1 consistait en l'introduction d'un nouveau singe : le singe écrivain, qui crée du texte à partir grâce au singe statisticien qui ne se contente plus de compter les mots.
En effet, il recense aussi les mots suivant un autre mot du texte. Le singe écrivain prend ensuite aléatoirement un mot du texte, puis en prend un autre au hasard parmi les mots qui peuvent le suivre, et ainsi de suite, jusqu'à créer une phrase (c'est à dire quand le mot choisi n'a pas de successeur).

### 3.2.1 Structures:

Ce nouveau singe a amené à la création d'une nouvelle memory pool et d'une nouvelle file, partagée uniquement par l'écrivain et l'imprimeur.
L'écrivain "écrit" son texte sur sa file, et l'imprimeur imprime les mots de cette file.
Ces files reprennent la syntaxe des files déjà évoquées.
Le statisticien et le lecteur restent quand à eux sur leurs précédentes files. De plus, nous avons modifié la structure des cellules afin de prendre en compte ses successeurs. Pour cela, nous avons créé une nouvelle file pour les successeurs ainsi que de nouveaux pool, l'un pour stocker de nouvelles files de successeurs et l'autre pour stocker les mots successeurs. Il a été nécéssaire de créer une structure "suivant" permettant de stocker un mot et son successeur.

- Suivant:

Listing 9: Suivant

```
struct suivant:
    mot : Chaine de caracteres
    suivant : Le mot suivant
```

Cette structure permet de garder en mémoire le mot vu précédemment et ainsi ajouter son successeur lorsqu'il passe par le statisticien.

- Mot successeur:

Listing 10: Mot_successeur

```
struct mot_successeur:
    mot : Chaine de caracteres
    occ : Entier representant le nombre d'occurences
    prochain : Le prochain mot successeur dans la file de successeurs
```

Mot successeur permet de stocker un successeur d'un mot, et son nombre d'occurrences, ainsi qu'un pointeur sur le prochain successeur du mot.

### 3.2.2 Algorithmes

On reprend les mêmes algorithmes de la base en modifiant le singe statisticien et en ajoutant le singe écrivain.

- Le singe écrivain:

Listing 11: singe_ecrivain

```
procedure singe_ecrivain()
    If (debut_phrase):
        mot:=choisir_mot_texte
        ajouter_file(mot, file_ecrivain)
    Else:
        If (fin_phrase):
            ajouter_file(ponctuation, file_ecrivain)
        Else
            mot:=choisir_mot_successeur
            ajouter_file(mot, file_ecrivain)
```

Dans cette procédure, debut_phrase renvoie Vrai ou Faux si c'est le début de la phrase de l'écrivain. choisir_mot_texte choisit un mot au hasard dans le texte. ajouter_file(mot,file_ecrivain) ajoute le mot sur la file de l'écrivain. De même, ajouter_file(ponctuation,file_ecrivain) ajoute un signe de ponctuation au hasard (sauf si la phrase ne contient qu'un mot, dans ce cas, ce signe de ponctuation est "!") et l'ajoute sur la file de l'écrivain. fin_phrase renvoie Vrai ou Faux si la phrase est terminée ou pas (si le mot qu'il avait choisi a un successeur ou pas). choisir_mot_successeur choisit un mot au hasard parmi les successeurs du mot qu'il avait choisi avant.

- Le singe statisticien modifié :

Listing 12: singe_statisticien_a1

```
procedure singe_statisticien_a1()
    mot_present:=FAUX
    While(cellule_suivante_non_libre):
        mot1:=mot(file_lecteur)
        mot2:=mot(file_statisticien)
        If (mot1=mot2):
            increm_occur_mot
            mot_present:=VRAI
        cellule_suivante
    If (Not mot_present):
        ajouter_cellule_file_statisticien
    ajouter_successeur
```

Ici, en plus de compter le nombre d'occurences des mots on ajoute le successeur du dernier mot vu par le statisticien grâce à ajouter_successeur.

# 4 Complexité des algorithmes

## 4.1 Base version:

- Le singe lecteur :
  Le singe lecteur a une complexité en temps et en espace constante car il n'a pas à parcourir toute la file ou modifier sa taille, c'est une complexité très satisfaisante.

- Le singe statisticien :
  La complexité en temps du singe statisticien est en revanche plus problématique, en effet, la fonction ajout_cellule_file_statisticien parcourt dans le pire des cas toute la file du statisticien. Pour une file de taille n,sachant qu'on a une boucle while qui dans le pire des cas nous fera faire cette opréation n fois ,on aura alors C(n)=$\Theta(n^2)$, complexité quadratique.
  Sa complexité en espace est constante pour les mêmes raisons qu'avant (la taille de la file est définie au début du programme).
- Le singe imprimeur:
  La complexité en temps et en espace du singe imprimeur est constante car il se contente de prendre le dernier mot de la file et de l'imprimer.
- Travail des singes:
  Au final, pour la complexité du programme en général, il s'agit juste de la somme des complexités de tous les singes réunis (on ne prends pas en compte le nombre de tours car il est fixé au départ, de plus le programme peut se terminer même sans cette condition avec les singes qui se mettent en grève). On a donc une complexité en temps quadratique majoritairement à cause du singe statisticien, ce qui est acceptable (mais pas optimal). La complexité en espace est constante, ce qui est très satisfaisant.

## 4.2 Achievement 1:

- Le singe écrivain:
  Sa complexité en temps et en espace est constante car il ne se contente que d'ajouter un mot (choisi aléatoirement) sur sa file.
- Le nouveau singe statisticien:
  La complexité en temps du singe statisticien a peu changé par rapport à la base.
  En effet, la fonction ajouter_successeur a pour effet de parcourir la file une première fois pour ajouter le dernier mot vu, et va reparcourir la file jusqu'à trouver le successeur de ce mot. Donc, dans le pire des cas, il parcourt toute la file de cellules puis toute la file de mots successeurs, ce qui donne une complexité, au final, quadratique : C(n) $=n^2 + 2n=O(n^2)$ (avec n la taille d'une file).
  En revanche, sa complexité en espace reste la même.
- Travail des singes:
  Au final, même avec la modification du singe statisticien la complexité ne change pas, Elle est toujours quadratique en temps, ce qui est acceptable et constant en espace, ce qui est très satisfaisant.

# 5 Mise en oeuvre:

Les algorithmes ci-dessus ont étés en réalité écrits en language C. Cependant, l'implémentation des algorithmes en language C ne s'est pas totalement faite de manière totalement naturelle. Voici quelques problèmes que nous avons rencontrés :

## 5.1 Le texte donné:

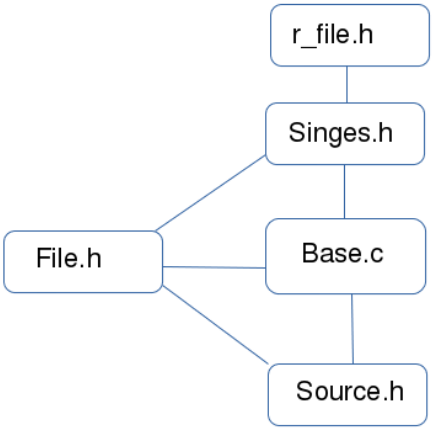
Lors de l'exécution du programme pour l'achievement 1, nous nous sommes rendus compte qu'il n'y avait aucun signe de ponctuation dans la phrase. Ceci était dû au texte lui même qui ne contenait en réalité aucun mot sans successeur. Nous avons donc du rajouter une condition pour qu'il puisse finir ses phrases. Nous avons simplement fait choisir pour chaque tour un nombre aléatoire entre 1 et 10 au singe pour savoir s'il devait terminer sa phrase.

## 5.2 L'aléatoire

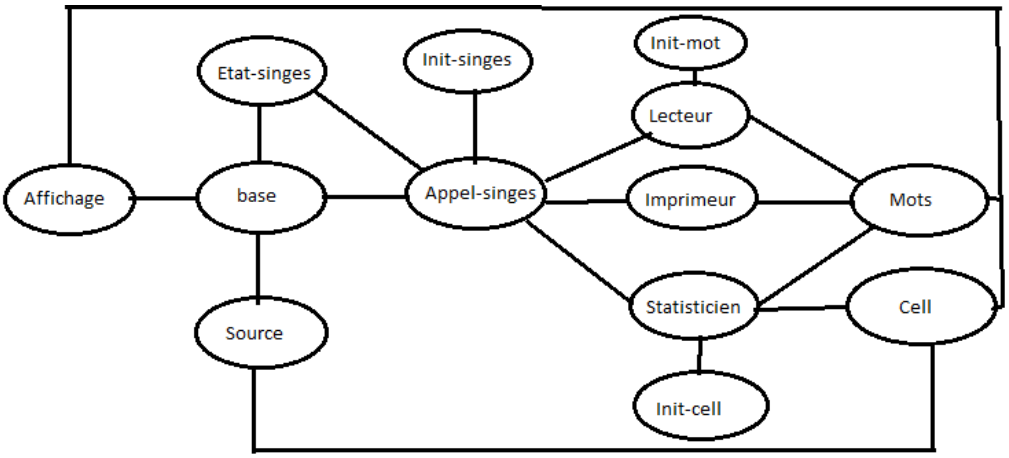
Nous nous sommes rendus compte lors de la génération d'un nombre aléatoire avec la fonction *rand*() que cela renvoyait toujours le même nombre.
C'est pour cela que nous avons du utiliser une graine aléatoire (seed) passée en paramètre, pour obtenir des valeurs vraiment aléatoires.

## 5.3 Organisation des fichiers:

Au départ du projet, nous avions prévu de mettre chaque achievement dans des dossiers bien séparés. Nous avons d'ailleurs gardé cette idée jusqu'à la fin du projet.
En revanche, pour l'organisation des fichiers contenant du code dans les dossiers, nous nous sommes rendus compte qu'elle était peu compréhensible et assez instable pour les autres achievements. Nous avons donc repensé le design de notre logiciel afin de l'améliorer.
En effet, au début du projet, notre design ressemblait à ceci :




Avec le fichier base.c qui contenait le main, le fichier Singes.h contenait le code de tous les singes, ainsi que leurs appels. Le fichier r_file.h contenait la fonction qui lisait le prochain mot du texte. File.h contenait toutes les structures et enfin Source.h contenait le programme pour obtenir la seed afin d'avoir de vraies valeurs aléatoires.
Le code compilait et fonctionnait parfaitement, mais ce choix de design nous paraissait inapproprié pour la suite. En effet, avec un nombre aussi faible de fichiers, il y a forcément beaucoup de dépendances dans un même fichier.
Nous avons alors opté pour un design de ce type :




Avec Singe_lecteur.c qui contient le code du singe lecteur, idem pour les fichiers Singe_imprimeur.c et Singe_statisticien.c. Affichage contient toutes les fonctions nécéssaires pour afficher les données utiles. Source contient la fonction pour obtenir la seed, ainsi que la fonction de calcul de multiplicité des mots.
Etats-singes, Mots et Cell contiennent différentes structures qui étaient présentent dans Struct.h auparavant. Init-singes, Init-mot et Init-cell servent à initialiser toutes ces structures, juste au moment où on en a besoin.
Ce design est intéressant car il permet l'ajout de singes de façon simple et claire (car on a un meilleur découpage des fichiers). De plus, ce design est mieux organisé, avec des fichiers plus nombreux, mais plus courts.Et surtout, ce design a aussi permis de supprimer plusieurs dépendances entre les fichiers, ce qui rend la modification des fichiers plus simple.

### 5.4 Makefile et compilation séparée:

Pour ce dernier design, nous avons aussi organisé d'une meilleure façon les fichiers .h et .c afin de pouvoir compiler séparément les fichiers principalement.
C'est là que le Makefile prends tout son intérêt. en compilant séparément les fichiers, on peut demander au Makefile de créer un exécutable à partir de toutes les compilations séparées. Et donc, si on fait une modification sur un fichier, il suffira de recompiler ce fichier, et le Makefile n'aura pas à tout recompiler, ce qui apporte un gain de temps.

## 6 Description des tests:

### 6.1 Singe lecteur:

Les tests du singe lecteur vérifient si :

- Le compteur de mots du lecteur correspond au nombre de mots du texte
- Les mots sont correctement placés dans la file
- Le dernier mot du texte est le bon

Si tous ces tests réussissent, on affiche Success, sinon, Failure.

### 6.2 Singe imprimeur:

Les tests du singe imprimeur vérifient sur 3 mots mis sur une file si :

- Le mot est bien enlevé de la file après l'avoir imprimé.
- Le mot précédent est ensuite bien modifié.
- Ces test sont ensuite répétés 2 fois pour être sûr que les tests fonctionnent

Si tous ces tests réussissent, on affiche Success, sinon, Failure.

## 6.3 Singe statisticien:

Les tests du singe statisticien vérifient sur les 3 premiers mots du texte si :

- Le mot dans la cellule est le bon
- Le nombre d'occurences dans le texte de ce mot est bon

Si tous ces tests réussissent, on affiche Success, sinon, Failure.

## 6.4 File

Pour les tests d'une file, on ajoute un mot sur une file, puis on vérifie si :

- Le mot sur la file est bien celui qu'on vient d'ajouter
- Le mot précédent est bien correct (NULL ici)
- La variable "vu" est bien à la valeur initialisée

On fait ce test pour le cas d'une file vide et d'une file non vide.
Puis, on teste la cellule : on crée une nouvelle cellule contenant un mot et un nombre d'occurrences. Puis, on vérifie si :

- Le nombre d'occurence du mot est correct
- La cellule suivante correcte

Ensuite, on teste l'ajout d'un élément sur la file. On crée un mot et une file, puis on ajoute ce mot sur la file, et on vérifie si :

- Le mot est bien au début de la file
- Le mot suivant est correct (NULL ici)

On rajoute ensuite le même mot sur la file et on refait les tests
Après celà, on teste l'ajout d'une cellule sur la file. On crée une nouvelle cellule, une nouvelle file, on ajoute la cellule sur la file, puis on vérifie si la cellule suivante dans la file a bien étée modifiée par l'ajout de la nouvelle cellule.
Enfin, on teste la supression d'un élément d'une file. On initialise une nouvelle file et ajoute 2 mots sur cette file, puis on supprime le $1^{er}$ élément de la file. Ensuite, on vérifie si :

- Le mot dans la file correspond bien au $2^{eme}$ mot ajouté
- L'élément suivant est correct (NULL ici)

On enlève ensuite encore un élément de la file, puis on vérifie qu'il n'y a plus d'éléments dans la file.

## 6.5 Singe écrivain:

Pour le singe écrivain, on compte le nombre de phrases qu'il produit avec le texte, puis, on vérifie si :

- Il crée bien une phrase lors d'un tour
- Il finit ses phrases

Si tous ces tests réussissent, on affiche Success, sinon, Failure.

## 6.6 Singe statisticien modifié

Pour ce singe, on crée une file de cellules, une file de successeurs, on ajoute 2 cellules avec les 2 premiers mots du texte sur la file de cellule. Puis en ajoutant le premier mot comme suivant du deuxième, on vérifie si :

- Le début de la file successeur est correct
- Le début de la file successeur a été initialisé correctement

– Le bon mot s'est ajouté sur la file de successeurs

Puis on refait ces tests sur les 3 mots suivants dans le texte pour vérifier que le mot s'ajoute bien à la file de mots successeurs même si cette file n'est pas vide.
Si tous ces tests réussissent, on affiche Success, sinon, Failure.

# 7 Résultats et conclusion:

## 7.1 Base Version:

L'exécution des algorithmes a renvoyé des résultats cohérents par rapport aux attentes. Par exemple pour un texte de 122 mots d'un sonnet de Shakespeare, nous obtenons (en ignorant tous les mots que l'imprimeur a imprimé):

– Nombre de mots lus :122
– Nombre de mots imprimés :122
– Nombre de mots différents :91
– Plus grande multiplicité :8
thy
– Plus petite multiplicité :1

## 7.2 Achievement 1:

Pour cet achievement, la seule différence provient du texte écrit par l'imprimeur, "imaginé" par l'écrivain. Voici donc un exemple de texte généré par ce singe : "*as the ? by time decease his memory , we desire increase that thereby beauty's rose might , thereby beauty's rose , his memory but as : increase that art , to thy light's flame with self-substantial fuel making a famine where abundance lies thy foe to . world or else this glutton be to thy beauty's field thy brow . due ! eat the gaudy spring within ; waste in niggarding pity . ornament and thee when , ornament and : sweet self*"

## 7.3 Conclusion:

Pour conclure, ce projet, dont l'objectif était de "simuler" une approche moins aléatoire du théorème du "Paradoxe du singe savant", s'est révélé être riche en enseignements. Cela nous a permis de réfléchir au design du logiciel (nom des variables, des fichiers, organisations des fichiers, etc.). Mais aussi d'en savoir plus sur la création d'un Makefile et de réaliser l'importance de la compilation séparée. Sans oublier de renforcer les bases sur les structures et les pointeurs. Ainsi que sur la création de tests solides.