# Rapport d'Algorithmique
## *Programme à inventer des mots et des phrases*

Loann Pottier, Arthur Blaise

20 janvier 2019

# Table des matières

# Introduction

Ces derniers mois, en cours d'algorithmique, nous avons réalisé un programme codé en Pascal, dont le but était de créer des mots. L'intérêt de ce projet est de mettre en pratique tout ce que l'on a vu pendant premier semestre en cours et obtenir quelque chose de plus ou moins optimisé, ainsi que de réfléchir aux manières d'atteindre notre but, avec nos connaissances actuelles. Ce qui requiers évidemment d'apprendre à se documenter pour combler les lacunes que nous pourrions avoir, ainsi que de savoir réfléchir aux différents moyens d'arriver à un but. Le sujet était divisé en 4 étapes, et pour chacune d'entre elles nous avons dû réfléchir sur comment obtenir de manière optimisée le résultat attendu. De ce fait dans ce rapport nous expliquerons toutes ces étapes, avec un maximum de détails possibles.

# Explications

## 2.1 Le cœur du programme

Pour commencer, voici comment s'articule le code. Tout part d'un simple fichier contenant le programme mère, *projet.pas*. C'est ce fichier qui articule tout le programme, c'est lui qui appelle les fonctions secondaires dans le bon ordre en fonction des choix de l'utilisateur, et c'est lui qu'il faut compiler puis lancer. Tout le reste est géré automatiquement, pas besoin de la moindre manipulation. C'est pas beau la technologie ?

Comme je le disais, ce programme mère est donc conçu pour pouvoir lire les arguments passés par l'utilisateur lors de l'exécution du fichier, et exécuter le code correspondant à ces choix. Si aucun type de génération n'est donné par l'utilisateur, ce même programme affichera l'aide principale, avec la liste des arguments utilisables. Tout le reste est conçu comme des paquets secondaires (parfois appelé **libraries**), importés par le programme via le mot-clé **uses** et dont seuls les fonctions principales sont utilisées. Ce cœur semble donc très petit, puisqu'effectivement il ne gère que la manipulation des arguments de commande. A peine 25 lignes...

Ensuite, c'est là que les choses se complexifient. Chaque question du sujet possède son propre fichier, lui-même organisé comme étant un paquet constitué d'une multitude de constantes, procédures, fonctions et autres choses diverses et variées. Un sixième fichier appelé *useful.pas* contient diverses fonctions communes à certains fichiers, et permettant une meilleure gestion des données. On y retrouve par exemple la fonction **StrInArray**, permettant de savoir si une chaîne de caractère se trouve présente dans un tableau de chaines, ou **readFile** qui renvoie les lignes d'un fichier, tels que ceux contenant les mots à analyser. Concernant les autres fichiers, voici leurs explications plus détaillées.

## 2.2 Aléatoire total

La première partie du projet est relativement simple. Il s'agit ni plus ni moins de mettre des lettres aléatoires ensemble pour former un "mot", bien que dans la plupart des cas ce mot ne soit même pas prononçable (essayez de prononcer "*qqyvkcnqieikpvpbîçg*", je vous souhaite ben du courage). Nous avons voulu corser un peu les choses en ajoutant un nombre aléatoire de lettres, tout en gardant le nombre minimum donné par l'utilisateur grâce à l'argument '-s'. Ce qui veut dire que l'argument 'taille' donné à la fonction 'creemotaleatoire' ne correspond pas exactement au nombre de lettres, mais au nombre minimum à atteindre pour qu'un mot soit valide.

Intéressons-nous maintenant au fonctionnement interne du programme. Nous avons donc une constante nommée 'alphabet', sous la forme d'une **widestring**, qui contient l'ensemble des caractères utilisables. Cependant, on remarque en y regardant de plus près que cet alphabet contient un caractère supplémentaire, le caractère numéro 44, le 'ø', et pourtant il n'apparaît jamais dans les mots générés. En fait ce caractère est utilisé pour les trois premières parties du projet comme représentant un début ou une fin de mot. Dans cette partie précisément, le programme va générer des caractères aléatoirement, et lorsqu'il tombe sur celui-ci, si le mot est assez long, va le supprimer et arrêter la génération du mot. De cette manière nous nous assurons que le mot soit de longueur variable, avec une moyenne tournant autour de 44 lettres lorsque l'argument '-s' n'est pas utilisé.

## 2.3 Digramme

Cette partie du programme est autrement plus complexe que la première. Ici il s'agit de générer une lettre en fonction de la lettre précédente, grâce à une table de probabilité. Le défi est donc de pouvoir lire une table de probabilité, avec un certain nombre de colonne, ainsi que de tirer une lettre au hasard en respectant les coefficients, mais surtout il faut être en mesure de générer cette table avec le moins de temps de traitement possible. Chose qui se révèle particulièrement ardue en Pascal, langage créé il y a près de 40 ans à des fins éducatives et possédant des limites facilement atteintes. Néanmoins nous avons réussi à faire en sorte que le programme du digramme soit capable de générer une table de 44*44 en seulement une dizaine de secondes, avec une liste d'un peu plus de 336 700 mots.

Au niveau interne, la procédure *p2_main* est la première appelée. C'est cette procédure qui déclenche toutes les autres procédures et fonctions, afin d'afficher un certain nombre de mots générés grâce à l'aléatoire-contrôlé. Si un fichier est donné en argument grâce au mot-clé '-r', il sera lu par la procédure *updateTable*, qui lit ligne après ligne ce fichier et récupère les statistiques de chaque lettre en fonction de la lettre précédente. Tout le tableau de probabilité est stocké dans une variable temporaire de type *tableOfProba*, qui référence pour chaque lettre (rangée) la probabilité que chaque lettre (colonne) la suive. Ici aussi nous utilisons le caractère 'ø' afin de générer le début et la fin d'une phrase : ainsi la probabilité qu'une lettre débute un mot est référencée par la probabilité que cette lettre suive 'ø', et il en est de même pour la fin des mots. Cela permet de ne pas avoir n'importe quelle lettre pour commencer un mot, et aussi de ne pas finir un mot de manière complètement aléatoire ou à une limite définie.

Une fois que ces données sont enregistrées dans le fichier *letters.csv*, ayant pour séparateur la virgule, la fonction *readTable* est prête à lire ce fichier et à le décortiquer pour en générer une nouvelle *tableOfProba*. C'est cette variable-ci qui est utilisée jusqu'à la fin du programme, indépendamment de si le fichier a été régénéré ou non. La troisième et dernière fonction appelée par le programme principal est la fonction *genWord*, qui s'occupe de générer un mot à partir de la table de probabilité. Pour ce faire, on commence un mot par le caractère 'ø', puis on utilise la fonction *genLetter* qui récupère une lettre pseudo-aléatoire en fonction de la lettre précédente. On récupère cette lettre, on l'ajoute à la fin du mot, puis on recommence le processus avec cette lettre, jusqu'à ce qu'on obtienne à nouveau 'ø' **et** que la longueur du mot soit supérieure à celle demandée via l'argument '-s'. A la fin, on enlève tous les caractères 'ø' du mot, puis on renvoie le mot final à la procédure principale, qui l'affiche directement grâce à un **writeln**. Le tout n'aura pris qu'une fraction de secondes, invisible à l'œil nu.

## 2.4 Trigramme

La partie du programme concernant le trigramme est sensiblement la même que pour le digramme. En fait la seule différence a été d'adapter le code afin qu'il génère une lettre à partir des deux précédentes, et non uniquement de la précédente. Cette technique permet réellement d'avoir des mots beaucoup plus lisibles, voir même parfois de retomber sur des mots existants. C'est incroyable de se dire qu'avec une série de nombres et deux cents lignes de code en pascal, on peut recréer des mots français, comme par exemple 'dépliées', 'barit', ou encore 'ordres'.

Lorsqu'on regarde à l'intérieur du code, on a l'impression à première vue qu'il s'agit du même que pour le digramme. En y regardant de plus près, on remarque néanmoins certaines différences, et non des moindres : le type de variable *tableOfProba* ne contient plus 44 lignes, mais 1936, pour représenter toutes les combinaisons de deux lettres possible. Cette modification se répercute dans le fichier enregistrant les probabilités, appelé *letters2.csv*, et possédant lui aussi 1936 lignes pour 44 colonnes.

Si on continue de lire un peu plus bas, on voit aussi que la fonction *child_updateTable* a été modifiée pour ne plus construire les probabilités à partir de deux lettres, mais de trois. Une rapide formule mathématique nous permet de calculer la ligne du fichier correspondante à une combinaison de deux lettres, à partir de l'index de ces deux lettres dans l'alphabet. Les autres fonctions n'ont pas été modifiées, jusqu'à *genLetter* qui prend deux lettres en paramètre au lieu d'une, et *genWord* qui commence le mot par 'øø' au lieu de 'ø'. Rien de bien impressionnant, mais grâce à toutes ces petites modifications on obtient une génération de mots beaucoup plus réaliste que le digramme, et sans même comparaison possible avec la méthode aléatoire de la première partie.

## 2.5 Génération de phrases

Cette partie est très différente des autres, puisque l'enjeu n'est pas le même. Il ne s'agit non pas de générer des lettres aléatoirement pour former un mot, mais de générer une phrase à partir de listes de mots données. Pas de table de probabilité à calculer, ici on se concentre sur la syntaxe de la phrase et l'accord des mots entre eux. Le défi majeur de cette partie est effectivement l'accord des mots, car il faut être en mesure de détecter quel est le genre et le nombre du sujet utilisé afin de mettre les verbes et adjectifs à la bonne personne. De plus, pour rajouter un peu de piquant dans l'aventure, nous avons décidé de laisser le choix à l'utilisateur de rentrer sa propre syntaxe, en plus des trois proposées dans l'énoncé ! Ce qui signifie qu'un verbe ne s'accorde non pas uniquement à la troisième personne du singulier ou du pluriel, comme c'est le cas lorsqu'on prend un sujet lambda, mais à n'importe quelle autre personne ! Si l'utilisateur décide de placer un 'Je' avant un verbe, il faut que le verbe soit conjugué à la première personne du singulier, peu importe la conjugaison utilisée.

En parlant des verbes, nous nous sommes contenté de formes relativement simples, à savoir le présent, le futur simple et l'imparfait de l'indicatif. Nous aurions sans doute pu ajouter d'autres formes plus complexes telles que le passé composé si nous avions un peu plus de temps, mais nous avons décidé de déjà faire une base solide avant de rentrer dans le domaine du bonus.

Maintenant, place à l'explication du code. Pour commencer, comme dans tous les autres modules, c'est la procédure *p4_main* qui est appelée, le paramètre '-n' décidant du nombre de phrases à générer. Un menu s'ouvre alors, invitant l'utilisateur à sélectionner une syntaxe pour la phrase. S'il choisit une syntaxe personnalisée, une autre fonction prend le relais le temps de lui expliquer comment créer une syntaxe fonctionnelle. Ensuite, c'est la fonction *replaceKeys* qui est appelée. Comme son nom l'indique, cette fonction lit la syntaxe donnée et remplace les arguments par les mots aléatoires nécessaires : '-s' par un sujet, '-v' par un verbe, '-a' par un adverbe etc. Pour l'accord de ces mots, mis à part les adverbes qui heureusement sont invariables, on fait appel à d'autres fonctions séparées. La génération du mot brut est faite grâce à la fonction '*getRandomWord*, qui va lire le nombre de lignes que possède un fichier, tirer un nombre aléatoire entre 1 et ce nombre de lignes, puis retourner le mot correspondant à cette ligne. Simple et efficace.

Penchons-nous sur le traitement de ces mots bruts. Lorsqu'il s'agit d'un nom propre, il n'y a pas grand chose à faire, donc la fonction *upperName* se contente de placer les majuscules aux bons endroits, à savoir sur le premier caractère et après chaque tiret. *getSubject* tire aléatoirement un nom propre ou un nom commun, et s'il s'agit d'un nom commun, refait un tirage au hasard pour choisir s'il doit être singulier ou pluriel. Une série de **if/else** lui permet de mettre le nom au pluriel, puis de lui attribuer le bon article. Une condition particulière lui permet d'éluder l'article si besoin, pour obtenir "l'anniversaire" au lieu de "la anniversaire" par exemple. *getAdj* commence déjà à être un peu plus complexe : la fonction essaie de repérer dans les quelques mots précédents un article. La fonction *getGenre* lui permet d'analyser cet article et de savoir à quel type il correspond, parmi les 4 possibles, mais si aucun article n'est trouvé, on prend par défaut le masculin singulier. Ensuite il suffit d'ajouter un 's' ou un 'e' lorsqu'il y en a besoin.

La fonction la plus complexe est certainement *conjugMain*. Cette fonction prend un verbe dans la liste des verbes bruts, puis doit l'analyser afin de savoir dans quel groupe ce verbe se situe (en fonction des deux dernières lettres, '-er' ou '-ir'). Mais avant cela il faut décortiquer les mots précédents et essayer d'y trouver un article ou un pronom, afin de savoir à quelle personne conjuguer le verbe. La fonction *lastItem* nous est d'une grande aide, car elle permet de donner le dernier item non vide d'une liste, et couplée à la fonction *splitStr* qui découpe la phrase grâce aux espaces, elle fait des merveilles. Si aucun article n'est trouvé, on regarde le mot juste avant le verbe, et s'il s'agit d'un pronom, on peut enfin connaître la personne à utiliser. Si tous ces tests échouent, on prend par défaut la troisième personne du singulier. Enfin le verbe est passé aux deux dernières fonctions du programme, en même temps que le temps à utiliser (tiré au hasard) et la personne trouvée. Ces fonctions s'appellent *conjug1* et *conjug2*, qui respectivement s'occupent du premier et du deuxième groupe. Une suite de **case of** permet d'ajouter la terminaison correcte à la fin de la racine, et le tour est joué.

Au final cette génération de phrase est très lente (environ 3 secondes pour générer 100 phrases de type sujet+verbe+adjectif), mais au vu du nombre d'analyses à faire pour obtenir une phrase propre avec une syntaxe aussi modulable, cette légère lenteur semble justifiée. Par exemple avec la syntaxe 'Je -v -a -s', nous obtenons "Je complimenterai tendancieusement Erico !", ou encore "Je coupellais indiscrètement des parachutages...". Avec la syntaxe '-s -v -a -s', qui demande plus de temps de traitement, "Les incidences syncristalliseront picturalement une mouscaille.", ou "Le mobilier reconnecte a fortiori les broques.". Et le nombre de combinaisons possibles permet *à fortiori* de ne jamais tomber deux fois sur la même phrase.

# Conclusion

A bien y réfléchir, ce projet a beau avoir été particulièrement long, et sur un langage que je n'affectionne pas particulièrement, il n'en reste pas moins intéressant. Le fait d'utiliser ce genre de langage, avec autant de contraintes, force à réfléchir autrement, à trouver des solutions différentes de celles qu'on a l'habitude d'utiliser, à comprendre comment il fonctionne pour essayer de l'utiliser au mieux. La première approche que nous avions choisi, qui était de coder d'abord en Python puis de le traduire en Pascal, était assez illusoire, mais nous l'avons compris bien plus tard. Ce qui nous a forcé à reprendre le code de zéro, à tout repenser, tout réécrire, sans passer par autre intermédiaire qu'une feuille de papier. C'est une expérience extrêmement importante, que nous n'oublierons pas de si tôt. Malgré de longues heures d'arrachage de cheveux face à des bugs inexplicables, le fait d'avoir pu mener ce projet à bien rapporte une satisfaction proportionnellement égale à la sueur que nous y avons laissé.

Concernant les sources, il est très compliqué – si ce n'est impossible – de reconstituer une liste correcte, car il s'agit majoritairement de plein de petites recherches sur des centaines de pages, à la recherche de fragments d'informations. Néanmoins nous pouvons au moins citer les deux sites les plus utilisés : l'incontournable StackOverflow[1], outils et forum connu de n'importe quel programmateur ; et le wiki officiel de freePascal[2], rassemblant des explications et des exemples sur la quasi-totalité des fonctions du langage. Pour le reste, il est malheureusement impossible de donner de réelle liste qui soit digne d'intérêt.

# Bibliographie

[1] Stack Overflow, *Question and answer site for professional and enthusiast programmers*
`https://stackoverflow.com`

[2] Free Pascal Wiki, *Knowledge base for Free Pascal/Lazarus*
`http://wiki.freepascal.org/`