Rapport du Projet d'OCaml : Les phrases réflexives

Cher lecteur, je suis un titre reflexif, ainsi je vous invite a compter soigneusement toutes les lettres afin de me croire quand je vous affirme que je contient exactement dix 'a', un 'b', dix 'c', huit 'd', quarante-trois 'e', neuf 'f', quatre 'g', six 'h', trente-cinq 'i', cinq 'j', un 'k', cinq 'l', six 'm', vingt-huit 'n', treize 'o', deux 'p', huit 'q', dix-neuf 'r', dix-huit 's', trente-trois 't', vingt-trois 'u', six 'v', un 'w', onze 'x', un 'y', et enfin trois 'z'.

Adrien BLASSIAU

4 avril 2018

Table des matières

1 Introduction

L'objectif de ce projet est de créer un générateur de phrases réflexives, c'est à dire un générateur de phrases qui décrivent leur propre contenu. Ici on se restreint au nombre d'occurrences des différentes lettres contenues dans la phrase et pas aux symboles.

Voici un exemple de phrase réflexive :

"Vous avez vraiment raison de compter toutes les lettres de cette longue phrase avant de la croire quand elle vous affirme qu'elle contient exactement quinze 'a', un 'b', neuf 'c', huit 'd', quarante-et-un 'e', sept 'f', cinq 'g', quatre 'h', vingt-trois 'i', un 'j', un 'k', neuf 'l', cinq 'm', vingt-huit 'n', douze 'o', six 'p', dix 'q', dix-sept 'r', treize 's', trente 't', vingt-quatre 'u', neuf 'v', un 'w', six 'x', un 'y', et pour finir cinq 'z'."

Pour former de telles phrases, on découpe ce projet en plusieurs étapes :
— la première est de générer des corps de phrases possiblement réflexifs, cela au moyen de gabarits imposés dans le sujet
— la deuxième est d'essayer à partir de ces gabarits de former une phrase réflexive.

Pour cela, on utilisera tout d'abord la méthode de **Douglas Hofstadter**, informaticien américain, puis ensuite l'amélioration proposée par **Jacques Pitrat**, chercheur au CNRS dans le domaine de l'intelligence artificielle.

2 Informations pratiques

L'ensemble des informations relatives à la compilation et l'exécution du programme se trouvent dans le fichier README.txt fourni avec le projet. Des tests unitaires vous permettront d'observer le travail effectué.

3 Résolution du problème : le générateur de phrases réflexives

3.1 Présentation générale du générateur

Notre générateur de phrases réflexives `generateur` utilise une des deux méthodes proposées pour trouver des phrases réflexives, représentées par les fonctions `boucle_pitrat` et `boucle_hofstadter`.

L'exercice est plus complexe qu'il n'y parait. En effet ces phrases sont assez rares à trouver et coûteuses en ressources à former, notamment si l'on cherche à en construire une qui décrivent ses 26 lettres.

La démarche reste pourtant assez simple. On génère une phrase de départ et on la modifie jusqu'à obtenir une phrase réflexive. Cependant certaines situations bouclent ou d'autres encore ne proposent pas de solution en un temps raisonnable. Ainsi, pour chaque méthode, on impose des limites de manière à ce que le programme ne tourne pas indéfiniment et puisse s'arrêter, même si la solution était proche.

Détaillons un peu plus le fonctionnement du générateur : il génère un corps de phrase aléatoire et propose à une des deux méthodes de la rendre réflexive. Au bout d'un certain nombre d'itérations, la méthode appelée renvoie true si elle a réussi à rendre la phrase réflexive pendant le nombre de tours maximum imposé et false si elle a échoué. Dans ce dernier cas, le générateur lui renvoie un nouveau corps de phrase généré aléatoirement et la méthode peut recommencer son travail. `iter_corps` impose une limite de corps de phrase générés de manière à ce que le programme s'arrête même s'il n'a pas trouvé de solution.

Pour terminer, ce générateur prend en arguments les paramètres suivants : la langue dans laquelle on travaille (belge 'b', français 'f' ou suisse 's'), le nombre de lettres sur laquelle notre phrase doit être réflexive (0 à 26), la méthode choisie (Hofstadter ou Pitrat), le nombre de corps différents que l'on souhaite tester, une liste vide qui contiendra les solutions et le nombre de phrases trouvées, initialisé à 0.

Voyons ces méthodes plus en détails et les problématiques induites par leur réalisation. Commençons pas l'initialisation de la phrase.

3.2 1ère étape de l'initialisation : la génération aléatoire du corps de la phrase par le générateur

On souhaite créer nos phrases en choisissant à chaque fois une possibilité parmi celles proposées dans cette expression de type string list list :

```
let gabarits_debut =
[["cher lecteur,";"ami lecteur,";"chers lecteurs,";""];
["vous avez"];
["bien";"vraiment";"parfaitement";""];
["raison de compter"];
["et de recompter";""];
["soigneusement";"avec soin";""];
["toutes";""];
["les lettres de cette"];
["longue";"curieuse";"bizarre";""];
```

```
["phrase avant de la croire quand elle"];
["vous";""];
["affirme qu'elle contient"];
["exactement";""]] ;;

let gabarits_fin =
[["et finalement ";"et pour finir ";"et enfin ";"et pour terminer ";"et "]];;
```

La création de corps de phrases aléatoires avec ces gabarits est une étape capitale dans notre générateur. En effet, peu de ces phrases pourront devenir réflexive, il faudra donc veiller à ce que leur **création soit rapide**. Autre problème, on veut que cette **création soit aléatoire** de manière à ne pas rencontrer deux fois le même corps. Ces deux problèmes sont assez faciles à résoudre.

Tout d'abord, le choix des gabarits se fait par la fonctionnelle `List.map` appliquée au deux listes de listes de gabarits présentées plus haut. La fonction interne au map est `choix_alea_gabarits` qui appelle directement le fonction `pre_choix_alea_gabarits` qui itère sur les chaînes de caractères de chaque sous listes et qui en choisit une aléatoirement. On obtient une liste de chaînes de caractères que l'on convertit ensuite avec la fonction `tradmot`.

On aura généré au préalable un nombre aléatoire compris entre 0 et le taille de la sous liste `l` exclus. Cela se fait aisément avec la fonction :

```
Random.int : (Random.int(List.length(l)))
```

Pour ne pas avoir tout le temps la même série de phrases aléatoires, on initialise le générateur de nombres aléatoires avec une graine aléatoire choisie en se basant sur les paramètres du système comme l'heure ou l'ID du processus courant. La fonction qui réalise cette opération est la suivante : `self_init` du module Random.

Voici quelques exemples de corps de phrase généré aléatoirement :

"Cher lecteur, vous avez vraiment raison de compter et de recompter soigneusement les lettres de cette bizarre phrase avant de la croire quand elle affirme qu'elle contient exactement""et finalement "

"Vous avez bien raison de compter et de recompter soigneusement toutes les lettres de cette bizarre phrase avant de la croire quand elle affirme qu'elle contient exactement"..."et enfin "

"Chers lecteurs, vous avez bien raison de compter les lettres de cette curieuse phrase avant de la croire quand elle affirme qu'elle contient exactement"..."et pour finir "

"Vous avez bien raison de compter avec soin toutes les lettres de cette longue phrase avant de la croire quand elle vous affirme qu'elle contient"..."et "

3.3 2ème étape de l'initialisation : l'affectation de nombres littéraux aléatoires comme point de départ

Après avoir reçu le corps de phrase généré aléatoirement, les deux méthodes continuent l'initialisation de la phrase, qui avait commencé par sa génération aléatoire au moyen des gabarits.

Cette deuxième étape consiste à affecter aléatoirement à chaque lettre de la phrase un nombre aléatoire littéral, car on ne connaît pas encore leur nombre d'occurrences dans la phrase et on ne peut pas l'obtenir car la phrase n'est constituée que de son corps ! Cette étape permet ainsi de former une phrase tout entière comme celle du titre de ce rapport par exemple.
Pour réaliser de telles phrases, on utilise la fonction `creation_index`. Là encore, la fonction `Random.int` est

sollicitée. On obtient finalement une phrase de la forme :
`"Vous avez bien raison de compter les lettres de cette phrase bizarre avant de la croire quand elle vous affirme qu'elle contient vingt-huit 'a', trois 'b', douze 'c', dix 'd', soixante-deux 'e', huit 'f', six 'g', cinq 'h', trente-cinq 'i', un j, un k, huit 'l', quatre 'm', trente-cinq 'n', treize 'o', six 'p', quatorze 'q', vingt-trois 'r', seize 's', trente-sept 't', dix-neuf 'u', neuf 'v', un 'w', sept 'x', un 'y', et pour finir sept 'z'.`
Cette étape est déterminante pour la suite. Si l'une des méthodes n'arrive pas à trouver de solution, on peut revenir à cette étape et régénérer un nouveau point de départ qui pourra peut-être être concluant.

Mais cette génération aléatoire pose plusieurs problèmes : comment transformer un chiffre en son écriture littérale, comment passer d'une liste de mots à une chaîne de caractères, comment stocker le nombre d'occurrences de chacune de nos lettres et modifier la phrase en conséquence, ... Les prochaines parties vont nous permettre d'y voir un peu plus claire.

Sous-problème récurent n°1 : la traduction littérale des nombres en français, belge et suisse

Une des parties primordiales de ce projet est le retranscription d'un nombre numérique en un nombre littéral. Cela doit pouvoir se faire en français, en belge ou en suisse et de manière rapide car l'opération est souvent réalisée. La fonction qui se charge de cela est la fonction `litt` qui prend 2 arguments : le nombre et sa langue de traduction.

On pourrait opérer de manière simple en écrivant littéralement tout les nombres de 1 à 100 dans une liste et en affectant avec un `match with` l'écriture correspondant à un nombre, mais cela prendrait trop de temps.

Notre fonction découpe donc chaque nombre en deux morceaux : ce que l'on appelle sa dizaine et son unité. Puis le filtre se base sur ces deux caractéristiques pour construire le nombre en sélectionnant la bonne dizaine correspondante puis en choisissant l'unité. Une fois l'unité trouvée, la fonction renvoie l'écriture littéral. Ce découpage nous assure la terminaison de l'algorithme.

On peut illustrer le principe sur l'exemple suivant : on cherche à traduire 97 en belge. La dizaine vaut 9 et l'unité 7. On filtre tout d'abord sur la dizaine. On arrive au cas où la dizaine vaut 7 ou 9. Ensuite, la langue n'est pas le français donc le filtrage nous donne 'nonante' pour la dizaine, et pas 'quatre-vingt'. Filtrage suivant : le chiffre des unités n'est pas 1, donc la liaison n'est pas un '-et-' mais simplement '-' que l'on concatène à 'nonante'. Après, si on écrit en français, on cherche l'unité + 10, soit 17. Ici c'est du belge donc on envoie seulement l'unité 7 au programme `litt` via appel un récursif qui trouve le mot 'sept', le concatène et renvoie 'nonante-sept'.

Sous-problème récurent n°2 : le passage d'une chaîne de caractères à une liste de ses mots et vice-versa

Dans la majorité du projet, on aime travailler uniquement sur des chaînes de caractères en itérant sur chacune des lettres avec l'opérateur chaîne.(i) . Pour certaines opérations spécifiques, on passe par une liste de mots mais les opérations de conversion chaîne vers liste où liste vers chaîne sont coûteuses donc on évite de les utiliser. Deux fonctions appelant diverses sous-fonctions réalisent ce travail : la fonction `tradmot` (liste -> chaîne) et la fonction `tradliste` (chaîne -> liste).
`tradmot` utilise la fonctionnelle `List.fold_left` et une fonction interne qui permet la concaténation des différents mots en gérant bien les espaces de manière à ce que cela soit possible : la chaîne suivante : `tradmot ["";"";"";"Bonjour";"";"le";"monde";""]` renvoie ... "Bonjour le monde" !
`tradliste` opère dans l'autre sens en itérant sur toutes les lettres et notamment en les convertissant en string avec l'opération suivante : `make 1 lettre` .

Sous-problème récurent n°3 : le stockage de l'information dans un index

Le parcours de la phrase pour compter l'occurrence des différentes lettres est une opération coûteuse. Elle est en $\mathcal{O}(n)$ avec n la taille de la phrase. À première vue, on stocke les lettres dans un index de type **liste de type (char * int) list** comme celui donnée dans l'énonce. Seulement cette structure **pose un problème** : l'accès à chacune des lettres nécessite au pire des cas 26 itérations, ce qui est assez conséquent.
De plus, on remarque que **les lettres les plus présentes** dans l'écriture littérale des chiffres de 1 à 100 en français **ne sont pas les premières lettres de l'alphabet** !
Voici une liste de type (char * int) list présentant le décompte des lettres de l'alphabet dans une phrase comprenant tous les nombres littéraux de 1 à 100 :

```
[('t', 157);('e', 145);('n', 121);('i', 112);('u', 90);('a', 80);('q', 60);
('r', 60);('s', 48);('x', 46); ('o', 36);('g', 30);('v', 30);('d', 22);
('c', 18);('z', 18); ('f', 10);('h', 10);('p', 10);('b', 0);('j', 0); ('k', 0);
('l', 0); ('m', 0) ;('y', 0);('w', 0)]
```

La lettre 't' et 'n' par exemple sont après le milieu de l'alphabet, leur accès prendrait du temps avec une liste.
On remarque aussi quelque chose de surprenant : **certaines lettres n'apparaissent pas du tout dans l'écriture littérale des nombres**. Ces lettres sont : **'b', 'j', 'k', 'l', 'm', 'y', et 'w'**.
Ainsi, on souhaiterait réduire le temps de parcours de l'index de la phrase.

On propose l'utilisation d'un Arbre Binaire de Recherche (ABR) possédant plusieurs caractéristiques :
- les couples sont rangés en suivant l'ordre lexicographique
- les couples sont aussi rangés en essayant de mettre plus haut les couples possédants des lettres qui reviennent plus fréquemment
- son parcours infixe nous donne les couples dans l'ordre alphabétique.
- il est composé de 5 niveaux, donc l'accès se fait en 5 essais maximum.

Voici donc le type utilisé :

```
type arbre = Vide | Noeud of ((char*int)*arbre*arbre);;
```

Et voici la représentation de l'arbre (sans le compte des lettres) :

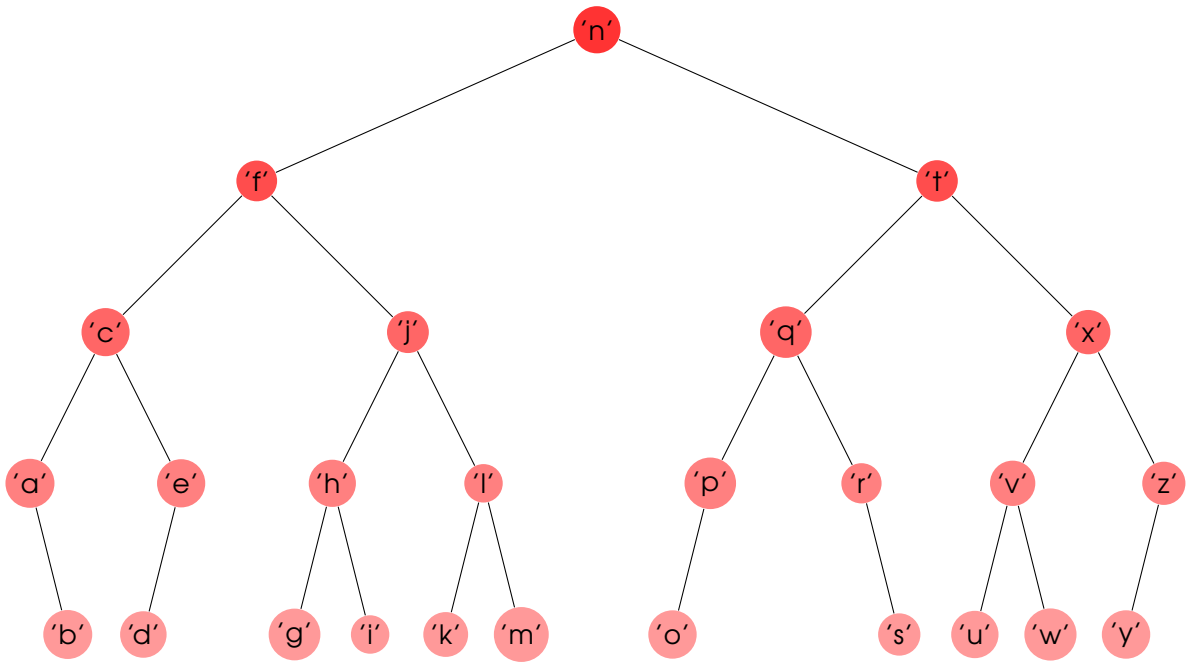

FIGURE 1 – Arbre binaire de recherche de l'alphabet

En tirant parti du caractère particulier de l'ABR, l'accès aux différentes lettres de l'alphabet est ainsi réduit. On peut aussi facilement transformer cette arbre en liste de type (char * int) list en réalisation un parcours infixe sur l'arbre. C'est l'objectif de la fonction `infixe`. On alternera donc les traitements sur l'arbre ou sur la liste en fonction du coût du traitement et de sa fréquence.

3.4 3ème et dernière étape de l'initialisation : première itération des méthodes

On procède maintenant à une première modification de la phrase en itérant une fois sur toutes les lettres de la phrase. C'est ici que les premières différences apparaissent entre les deux méthodes.
Dans le cas de la méthode d'Hofstadter, on compte les occurrences de toutes les lettres puis on modifie la phrase en conséquence. C'est une étapes rapide qui est directement faite dans la fonction `hofstadter`.
Dans le cas de Pitrat, la fonction `initialise_pitrat` compte chacune des lettres une par une en modifiant la phrase directement après le compte de chaque lettre.
On obtient maintenant la phrase dite **initialisée**.

Présentons maintenant la dernière étape avant la possible transformation en phrase réflexive de notre phrase initialisée.

3.5 Découpage de la phrase en deux parties : la partie mobile et la partie fixe

Soit la phrase suivante, obtenue après la phase d'initialisation :

"Vous avez bien raison de compter les lettres de cette phrase bizarre avant de la croire quand elle vous affirme qu'elle contient vingt-huit 'a', trois 'b', douze 'c', dix 'd', soixante-deux 'e', huit 'f', six 'g', cinq 'h', trente-cinq 'i', un j, un k, huit 'l', quatre 'm', trente-cinq 'n', treize 'o', six 'p', quatorze 'q', vingt-trois 'r', seize 's', trente-sept 't', dix-neuf 'u', neuf 'v', un 'w', sept 'x', un 'y', et pour finir sept 'z'.

On a fait plus haut l'observation que certaines lettre n'apparaissent pas dans l'écriture littérale des nombres. De plus, après l'initialisation, tout le début de la phrase ne sera jamais modifié pendant la création de la phrase réflexive.
Ainsi, on propose de découper la phrase en deux morceaux :

— la partie fixe, c'est-à-dire la partie qui ne varie plus après la phase d'initialisation de la phrase
— la partie mobile, c'est-à-dire la partie qui va varier après l'initialisation de la phrase

Ainsi cela donne sur la phrase exemple :
— partie fixe : "Vous avez bien raison de compter avec soin toutes les lettres de cette longue phrase avant de la croire quand elle vous affirme qu'elle contient 'a', trois 'b','c','d','e','f','g','h','i', un 'j', un 'k', huit 'l', quatre 'm','n','o','p','q','r','s','t','u','v', un 'w', 'x', un 'y', et pour finir 'z'.
— partie mobile : vingt-huit douze dix soixante-deux huit six cinq trente-cinq trente-cinq treize six quatorze vingt-trois seize trente-sept dix-neuf neuf sept sept.

La création de la phrase fixe et de la phrase mobile se fait avec la fonction `fragment_phrase` qui utilise la fonctionnelle `List.fold_right`. Si l'on souhaite former la phrase toute entière, on utilise aussi `fragment_phrase`. La fonctionnelle utilise trois fonctions différentes, une pour chaque opération : `concat_index_fixe` `concat_index_mobile` et `concat_index_entier`.

Ainsi deux index sont nécessaires : un index entier ou mobile qui comptabilise l'occurrence de toutes les lettres dans toutes la phrase et un index fixe qui comptabilise l'occurrence de toutes les lettres dans la partie fixe de la phrase.

3.6 Les méthodes de Pitrat et d'Hofstadter en action

Nos phrases sont initialisées et découpées, on passe maintenant à l'application des deux méthodes. Mais avant cela, voyons certaines opérations omniprésentes lors du fonctionnement de ces méthodes.

Le comptage des lettres dans une phrase, la mise à jour et la comparaison d'index

Compter, **mettre à jour** et **comparer** sont les trois opérations majeures de ce projet. Quelque soit la méthode, on souhaite pouvoir compter le nombre de lettre dans la phrase, mettre à jour l'index pour ensuite pouvoir mettre à jour la phrase et comparer l'index avant et après modification pour voir si on a atteint un point fixe, signe que l'on a trouvé une phrase réflexive.
Ainsi plusieurs manières de compter les lettres d'une phrase et de mettre à jour l'index se complètent dans ce projet.

Pour la première méthode mais aussi pour la deuxième, on veut pouvoir recenser toutes les lettres dans une phrase : on utilise pour cela la fonction `compte_mot_h` qui appelle `compte_mot_global` qui appelle elle même la fonction `compte_lettre_arbre` qui met à jour l'index en incrémentant le compte d'une lettre. Pour la deuxième méthode, on veut partir de la phrase et compter toute les occurrences d'une lettre en particulier, cela se fait via la fonction `compte_lettre` qui appelle la fonction `compte_lettre_dans_mot`. Ensuite, la fonction `maj_index_courant` permet de mettre à jour l'index en modifiant l'occurrence d'une lettre par son nouveau compte, elle est donc souvent utilisée après `compte_lettre`.

Enfin, la fonction `compare_arbre` a le rôle de comparer deux index entre eux afin de voir si la modification d'une ou plusieurs lettres a permis d'atteindre un point fixe. Elle compare donc les noeuds de l'arbre deux à deux en s'arrêtant et renvoyant false si elle rencontre deux comptes de lettre différents. Dans le cas contraire, elle renvoie true.

Présentation d'une itération des deux méthodes

La méthode d'Hofstadter Une itération de la fonction `hofstadter` est simple. Elle compte l'occurrence de toutes les lettres de la phrase mobile et modifie la phrase mobile et l'index entier en conséquence, en s'appuyant sur l'index fixe. Si le nouvel index entier est différent du précédent, on recommence. Sinon, on a trouvé une solution et on arrête le générateur. Voir en annexe 1 un résumé de la méthode sous forme d'algorigramme.

La méthode de Pitrat Une itération de la fonction `pitrat` est un peu plus complexe. La fonction `corps_pitrat` compte l'occurrence d'une première lettre de la phrase mobile et modifie la phrase mobile et l'index entier en conséquence, en s'appuyant sur l'index fixe. Si le nouvel index entier est différent du précédent, on recommence sur une nouvelle lettre. Sinon, on a trouvé une solution et on arrête le générateur. On boucle ainsi sur toute les lettres de l'alphabet sauf les lettres qui ne varient plus depuis l'initialisation, le 'b', 'j', 'k', 'l', 'm', 'y', et 'w'. Ce test se fait avec la fonction `est_fixe`. Une fois l'alphabet terminé, on recommence. Voir en annexe 1 un résumé de la méthode sous forme d'algorigramme.

Le bouclage

Chaque itération principale des deux méthodes est réalisée jusqu'à une limite de 100 fois, comme précisé dans l'énoncé. De plus, si on n'a pas trouvé de solution au bout de ces 100 itérations, les fonctions `boucle_pitrat` et `boucle_hofstadter` permettent de générer une nouvelle affectation de nombres littéraux aléatoires au même corps de phrase et ainsi repartir sur de nouvelles bases. On réalise cette opération 100 fois.

3.7 Remarques

Bouclage

Certains corps de phrase ne possèdent pas de solution. Ainsi, quels que soient les nombres littéraux affectés à la phrase lors de l'initialisation, la phrase mènera à un bouclage. Voici un exemple de situation qui peut provoquer un bouclage : on affiche "... quatre 'a'..." alors qu'il y en a 5. On écrit donc "...cinq 'a'...". Or on vient de retirer le 'a' qui était dans le mot "cinq" donc on repasse à "...quatre a..." où il y a en fait 5 'a' ...

Point de départ et itération

Pour un corps de phrase généré aléatoirement par le générateur, deux paramètres sont à prendre en compte afin d'optimiser la fréquence de réussite du programme :

— le nombre de point de départ `nb_essai`

— le nombre d'itération `iter` des algorithmes de résolution.

Plus ces deux paramètres sont élevés, plus la chance de trouver une phrase réflexive est grande. On choisit 100 pour les deux paramètres dans le programme.

On remarque que l'algorithme de Pitrat est plus lent mais produit des solutions au bout de moins d'itérations `iter` que Hofstadter car la méthode est plus efficace. On pourrait donc baisser la limite de `iter` à 50.
Quant à l'algorithme de Hofstadter, il est plus rapide mais moins efficace, on pourrait donc augmenter ce nombre d'itérations à 200.
Le nombre de points de départ est aussi important. Certains corps ne marchent que sur peu de points de départ, il est donc important d'en tester un certain nombre avant de passer à un nouveau corps de phrase.

3.8 Expériences

Expérience n°1 On teste le générateur avec la méthode de Hofstadter pour 26 lettres, en français, sur 2000 corps. Cette expérience n'est pas réalisée avec Pitrat car l'algorithme est bien plus lent.

```
generateur boucle_hofstadter 26 'f' 2000 [] 0;;
```

On obtient 10 phrases réflexives. Le programme a tourné pendant 5 minutes :

"Chers lecteurs, vous avez bien raison de compter soigneusement les lettres de cette bizarre phrase avant de la croire quand elle affirme qu'elle contient quinze 'a', trois 'b', huit 'c', sept 'd', quarante 'e', sept 'f', six 'g', sept 'h', vingt-cinq 'i', un 'j', un 'k', neuf 'l', quatre 'm', vingt-quatre 'n', neuf 'o', six 'p', neuf 'q', dix-huit 'r', dix-huit 's', vingt-neuf 't', vingt-trois 'u', huit 'v', un 'w', six 'x', un 'y', et quatre 'z'."

"Vous avez parfaitement raison de compter les lettres de cette bizarre phrase avant de la croire quand elle affirme qu'elle contient quinze 'a', deux 'b', huit 'c', six 'd', trente-huit 'e', huit 'f', cinq 'g', sept 'h', vingt-quatre 'i', un 'j', un 'k', neuf 'l', cinq 'm', vingt-sept 'n', huit 'o', six 'p', neuf 'q', quatorze 'r', onze 's', vingt-neuf 't', vingt-et-un 'u', huit 'v', un 'w', cinq 'x', un 'y', et finalement six 'z'."

"Ami lecteur, vous avez vraiment raison de compter soigneusement les lettres de cette curieuse phrase avant de la croire quand elle vous affirme qu'elle contient quinze 'a', un 'b', neuf 'c', huit 'd', quarante-deux 'e', huit 'f', six 'g', cinq 'h', vingt-cinq 'i', un 'j', un 'k', neuf 'l', six 'm', vingt-huit 'n', neuf 'o', sept 'p', neuf 'q', quatorze 'r', dix-sept 's', vingt-sept 't', vingt-six 'u', dix 'v', un 'w', sept 'x', un 'y', et enfin quatre 'z'."

"Vous avez raison de compter soigneusement les lettres de cette curieuse phrase avant de la croire quand elle affirme qu'elle contient onze 'a', un 'b', dix 'c', huit 'd', trente-neuf 'e', sept 'f', six 'g', six 'h', vingt-cinq 'i', un 'j', un 'k', neuf 'l', cinq 'm', vingt-huit 'n', neuf 'o', quatre 'p', huit 'q', douze 'r', treize 's', vingt-cinq 't', vingt-deux 'u', huit 'v', un 'w', six 'x', un 'y', et finalement cinq 'z'."

"Cher lecteur, vous avez parfaitement raison de compter et de recompter soigneusement les lettres de cette phrase avant de la croire quand elle affirme qu'elle contient quinze 'a', un 'b', dix 'c', dix 'd', quarante-six 'e', sept 'f', cinq 'g', cinq 'h', vingt-et-un 'i', un 'j', un 'k', neuf 'l', six 'm', vingt-six 'n', neuf 'o', huit 'p', neuf 'q', dix-sept 'r', quatorze 's', trente 't', vingt-deux 'u', sept 'v', un 'w', huit 'x', un 'y', et quatre 'z'."

"Vous avez parfaitement raison de compter soigneusement les lettres de cette longue phrase avant de la croire quand elle affirme qu'elle contient quinze 'a', un 'b', six 'c', huit 'd', quarante-deux 'e', sept 'f', sept 'g', cinq 'h', vingt-quatre 'i', un 'j', un 'k', neuf 'l', six 'm', vingt-six 'n', dix 'o', neuf 'p', huit 'q', seize 'r', dix-sept 's', vingt-neuf 't', vingt-trois 'u', huit 'v', un 'w', sept 'x', un 'y', et pour terminer quatre 'z'."

"Chers lecteurs, vous avez vraiment raison de compter et de recompter toutes les lettres de cette curieuse phrase avant de la croire quand elle affirme qu'elle contient exactement quatorze 'a', un 'b', treize 'c', dix 'd', cinquante-et-un 'e', sept 'f', quatre 'g', cinq 'h', vingt-deux 'i', un 'j', un 'k', neuf 'l', six 'm', vingt-sept 'n', neuf 'o', six 'p', neuf 'q', dix-huit 'r', quinze 's', trente-deux 't', vingt-six 'u', huit 'v', un 'w', neuf 'x', un 'y', et cinq 'z'."

"Ami lecteur, vous avez parfaitement raison de compter soigneusement les lettres de cette curieuse phrase avant de la croire quand elle vous affirme qu'elle contient exactement seize 'a', un 'b', neuf 'c', huit 'd', quarante-six 'e', huit 'f', cinq 'g', neuf 'h', vingt-neuf 'i', un 'j', un 'k', neuf 'l', huit 'm', vingt-six 'n', dix 'o', sept 'p', six 'q', dix-huit 'r', dix-sept 's', trente-quatre 't', vingt-huit 'u', huit 'v', un 'w', huit 'x', un 'y', et pour terminer trois 'z'."

"Ami lecteur, vous avez vraiment raison de compter et de recompter les lettres de cette bizarre phrase avant de la croire quand elle affirme qu'elle contient exactement dix-sept 'a', deux 'b', dix 'c', treize 'd', quarante-sept 'e', six 'f', cinq 'g', quatre 'h', vingt-six 'i', un 'j', un 'k', neuf 'l', sept 'm', vingt-quatre 'n', dix 'o', huit 'p', huit 'q', vingt-deux 'r', douze 's', trente-trois 't', vingt-et-un 'u', neuf 'v', un 'w', dix 'x', un 'y', et pour finir cinq 'z'."

"Chers lecteurs, vous avez bien raison de compter soigneusement toutes les lettres de cette phrase avant de la croire quand elle affirme qu'elle contient douze 'a', deux 'b', onze 'c', neuf 'd', quarante-trois 'e', neuf 'f', six 'g', cinq 'h', vingt-cinq 'i', un 'j', un 'k', neuf 'l', quatre 'm', vingt-neuf 'n', treize 'o', sept 'p', neuf 'q', dix-sept 'r', dix-sept 's', vingt-huit 't', vingt-trois 'u', huit 'v', un 'w', cinq 'x', un 'y', et pour finir cinq 'z'.")

Expérience n°2 On a laisser tourner le générateur avec la méthode de Hofstadter puis de pitrat pour 15 lettres, en français, sur 30 corps (les mêmes pour les deux méthodes). On remarque généralement que la méthode de Pitrat est plus efficace que la méthode de Hofstadter. De plus, on remarque que les deux méthodes ne trouvent pas forcément des solutions pour les mêmes corps.

```
generateur boucle_hofstadter 15 'f' 30 [] 0;;
generateur boucle_pitrat 15 'f' 30 [] 0;;
```

On obtient 8 phrases réflexives pour la méthode de Pitrat et 7 pour la méthode de Hofstader :

Méthode de Pitrat :

"Chers lecteurs, vous avez vraiment raison de compter et de recompter toutes les lettres de cette bizarre phrase avant de la croire quand elle vous affirme qu'elle contient treize 'a', deux 'b', neuf 'c', neuf

'd', quarante-et-un 'e', six 'f', un 'g', trois 'h', douze 'i', un 'j', un 'k', neuf 'l', cinq 'm', seize 'n', douze 'o'."

"Vous avez bien raison de compter soigneusement toutes les lettres de cette phrase avant de la croire quand elle vous affirme qu'elle contient onze 'a', deux 'b', six 'c', sept 'd', trente-cinq 'e', trois 'f', deux 'g', trois 'h', quatorze 'i', un 'j', un 'k', huit 'l', quatre 'm', quinze 'n', treize 'o'."

"Vous avez parfaitement raison de compter soigneusement les lettres de cette curieuse phrase avant de la croire quand elle vous affirme qu'elle contient exactement douze 'a', un 'b', huit 'c', huit 'd', trente-six 'e', cinq 'f', deux 'g', six 'h', dix-huit 'i', un 'j', un 'k', huit 'l', six 'm', seize 'n', neuf 'o'."

"Vous avez raison de compter les lettres de cette bizarre phrase avant de la croire quand elle affirme qu'elle contient exactement treize 'a', deux 'b', six 'c', huit 'd', trente-trois 'e', trois 'f', un 'g', quatre 'h', douze 'i', un 'j', un 'k', huit 'l', quatre 'm', douze 'n', onze 'o'."

"Vous avez bien raison de compter et de recompter soigneusement toutes les lettres de cette bizarre phrase avant de la croire quand elle affirme qu'elle contient treize 'a', trois 'b', sept 'c', huit 'd', quarante 'e', trois 'f', deux 'g', quatre 'h', seize 'i', un 'j', un 'k', huit 'l', cinq 'm', treize 'n', douze 'o'."

"Ami lecteur, vous avez parfaitement raison de compter soigneusement les lettres de cette phrase avant de la croire quand elle affirme qu'elle contient douze 'a', un 'b', sept 'c', huit 'd', trente-cinq 'e', six 'f', deux 'g', trois 'h', quinze 'i', un 'j', un 'k', neuf 'l', six 'm', dix-sept 'n', neuf 'o'."

"Ami lecteur, vous avez parfaitement raison de compter et de recompter les lettres de cette curieuse phrase avant de la croire quand elle vous affirme qu'elle contient treize 'a', un 'b', huit 'c', six 'd', trente-neuf 'e', six 'f', un 'g', quatre 'h', seize 'i', un 'j', un 'k', neuf 'l', six 'm', quinze 'n', huit 'o'."

"Cher lecteur, vous avez raison de compter et de recompter toutes les lettres de cette curieuse phrase avant de la croire quand elle affirme qu'elle contient exactement treize 'a', un 'b', douze 'c', neuf 'd', quarante-deux 'e', cinq 'f', un 'g', quatre 'h', onze 'i', un 'j', un 'k', neuf 'l', cinq 'm', dix-huit 'n', onze 'o'.")

Méthode de Hofstadter :

"Chers lecteurs, vous avez vraiment raison de compter et de recompter toutes les lettres de cette phrase avant de la croire quand elle affirme qu'elle contient exactement treize 'a', un 'b', neuf 'c', sept 'd', quarante-et-un 'e', six 'f', un 'g', trois 'h', onze 'i', un 'j', un 'k', neuf 'l', six 'm', dix-neuf 'n', onze 'o'."

"Ami lecteur, vous avez bien raison de compter et de recompter toutes les lettres de cette phrase avant de la croire quand elle vous affirme qu'elle contient onze 'a', deux 'b', neuf 'c', huit 'd', trente-huit 'e', cinq 'f', un 'g', quatre 'h', treize 'i', un 'j', un 'k', neuf 'l', cinq 'm', dix-sept 'n', onze 'o'."

"Ami lecteur, vous avez vraiment raison de compter avec soin les lettres de cette longue phrase avant de la croire quand elle vous affirme qu'elle contient treize 'a', un 'b', neuf 'c', huit 'd', trente-cinq 'e', quatre 'f', deux 'g', trois 'h', seize 'i', un 'j', un 'k', dix 'l', cinq 'm', dix-sept 'n', onze 'o'."

"Vous avez bien raison de compter avec soin les lettres de cette longue phrase avant de la croire quand elle vous affirme qu'elle contient douze 'a', deux 'b', sept 'c', huit 'd', trente-cinq 'e', quatre 'f', deux 'g', trois 'h', treize 'i', un 'j', un 'k', neuf 'l', trois 'm', quatorze 'n', treize 'o'."

"Ami lecteur, vous avez parfaitement raison de compter et de recompter toutes les lettres de cette bizarre phrase avant de la croire quand elle affirme qu'elle contient quinze 'a', deux 'b', sept 'c', neuf 'd', quarante-deux 'e', sept 'f', un 'g', deux 'h', onze 'i', un 'j', un 'k', neuf 'l', six 'm', seize 'n', neuf 'o'."

"Ami lecteur, vous avez raison de compter et de recompter soigneusement toutes les lettres de cette

phrase avant de la croire quand elle affirme qu'elle contient onze 'a', un 'b', huit 'c', huit 'd', trente-huit 'e', quatre 'f', deux 'g', cinq 'h', quinze 'i', un 'j', un 'k', neuf 'l', six 'm', seize 'n', dix 'o'."

"Ami lecteur, vous avez raison de compter soigneusement les lettres de cette phrase avant de la croire quand elle vous affirme qu'elle contient douze 'a', un 'b', huit 'c', huit 'd', trente-cinq 'e', quatre 'f', deux 'g', quatre 'h', douze 'i', un 'j', un 'k', neuf 'l', cinq 'm', seize 'n', onze 'o'.")

4 Conclusion

Pour conclure, la recherche de phrases réflexives c'est avérée être un travail passionnant mais parfois frustrant, le domaine étudié étant particulièrement chaotique. On obtient tout de même de bons résultats, même pour 26 lettres.
Le générateur fonctionne, quelque soit la méthode utilisée, l'objectif est donc rempli.
Cependant, la méthode de Pitrat, certes plus efficace, est bien plus lente que celle de Hofstadter est son utilisation pour plus de 20 lettres n'est pas recommandée car la recherche prend beaucoup de temps.
Ainsi, malgré les efforts d'optimisation mis en place durant ce projet, des améliorations pourraient être apportées à la méthode de Pitrat, mais aussi à la méthode d'Hofstadter. Pour obtenir des résultats plus rapidement, on pourrait plus travailler sur le choix des paramètres d'itérations et du nombre d'essais.

5 Annexes

Résumé en algorigrammes

Les algorigrammes suivants résument simplement le processus :

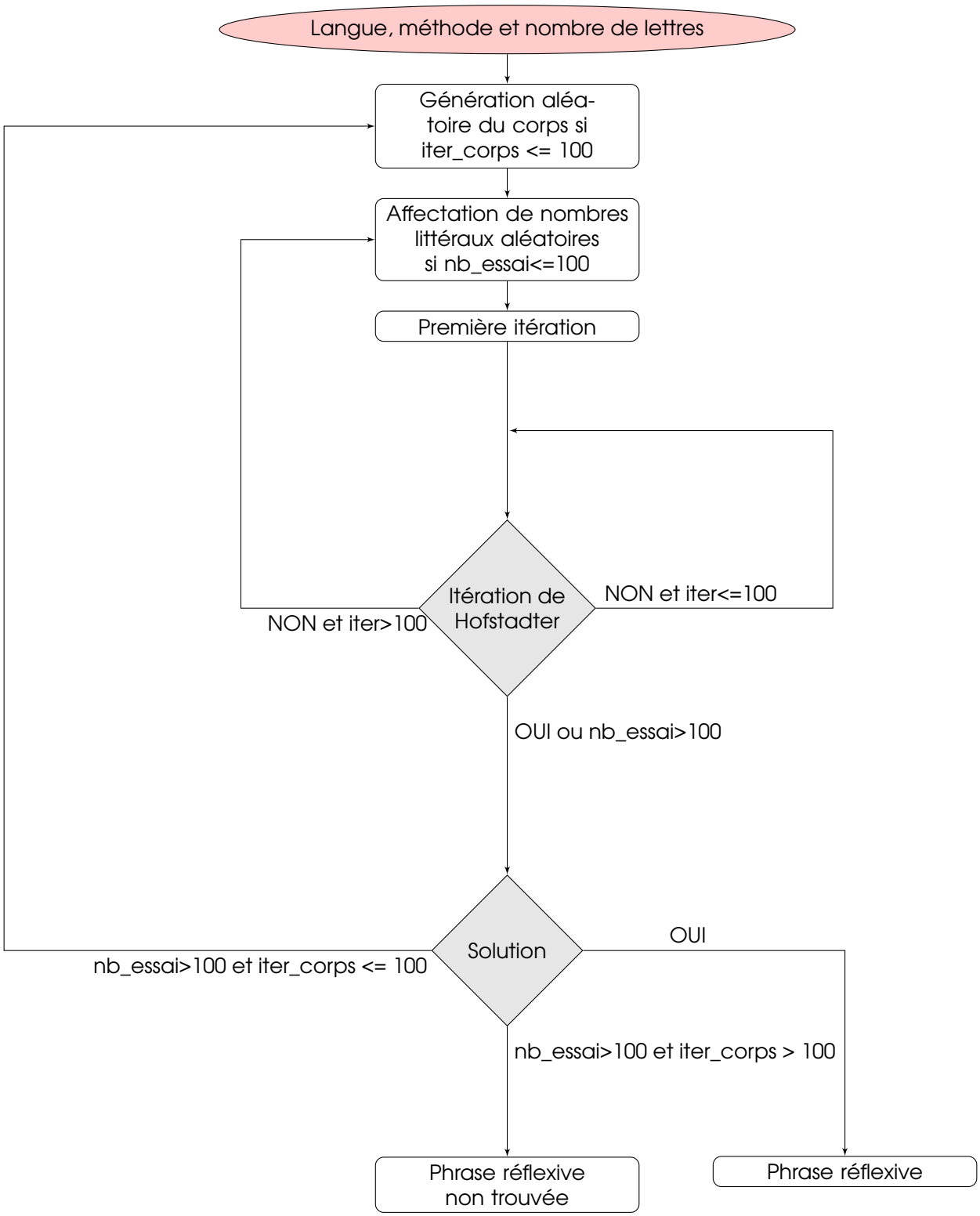

FIGURE 2 – Algorigramme de la méthode d'Hofstadter (`nb_essai` itérations sur les nombres littéraux initiaux, `iter` itérations de la méthode d'Hofstadter et`iter_corps` itérations sur les corps)

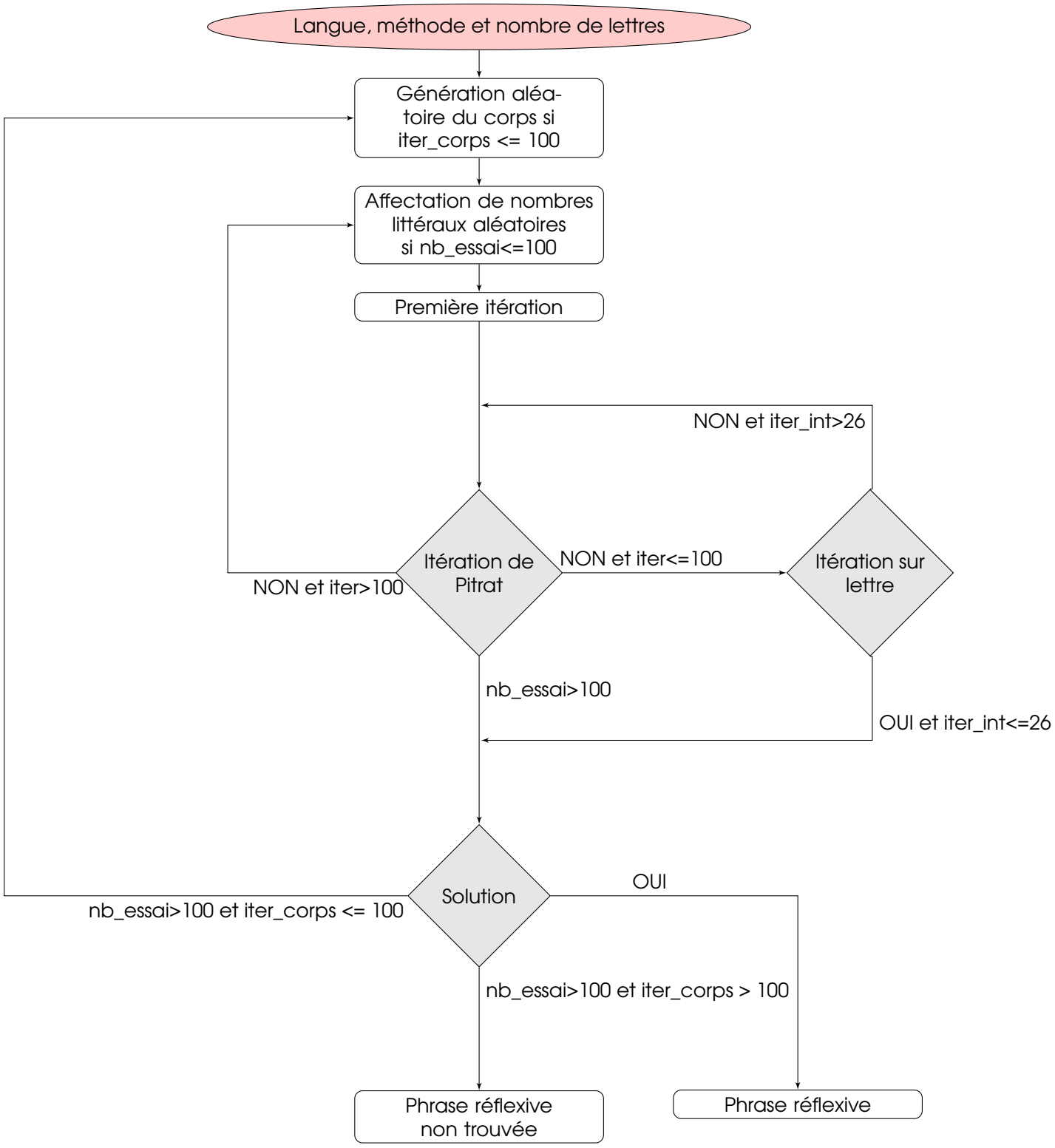

FIGURE 3 – Algorigramme de la méthode de Pitrat (`nb_essai` itérations sur les nombres littéraux initiaux, `iter` itérations dans la méthode de Pitrat, `iter_int` itérations sur les lettres de l'alphabet et `iter_corps` itérations sur les corps)

Interfaces des fonctions

```
(* print_list_int_string

@param l une liste de couple (int,chaine de caractere)
@return affiche les r sultats du g n rateur

val print_list_string : (int * string) list * int -> unit = <fun>
*)

(** Tests :

Voir les derniers test unitaires ...
*)


(** chaine_vers_liste_de_chaine

@param s une cha ne de caract res, n un compteur pour it rer sur cette cha ne,
    buf un buffer qui r cup re chacun des mots afin de les placer dans la liste et
     l la liste de cha nes de caract res renvoy e
@return transforme une cha ne de caract res en une liste de ses mots en coupant
    au niveau des espaces.
  quivalent   de String.split_on_char " " s.

val chaine_vers_liste_de_chaine : string -> int -> string -> string list -> string
    list = <fun>
*)

(** Tests :
chaine_vers_liste_de_chaine "Bonjour le monde" 0 "" [];; -> ["Bonjour"; "le"; "
   monde"]
chaine_vers_liste_de_chaine "" 0 "" [];; -> [""]
*)



(** tradliste

@param s une cha ne de caract res
@return utilise la fonction ci-dessus, donc transforme une cha ne de caract res
    en une liste de ses mots en coupant au niveau des espaces.

val tradliste : string -> string list = <fun>
*)

(** Tests :
tradliste "Bonjour  le    monde" ;; -> ["Bonjour"; "le"; "monde"]
tradliste "";; -> [""]

*)


(** tradmot
```

```
@param l une liste de cha nes de caract res
@return la cha ne de caract res associ e    la liste de cha nes de caract res
    entr e en concat nent chacune des cha nes

val tradmot : string list -> string = <fun>
*)

(** Tests :
tradmot ("";"Bonjour";"le";"monde");; -> "Bonjour le monde"
tradmot ("Bonjour";"le";"";"monde");; -> "Bonjour le monde"
tradmot ("Bonjour";"le";"monde";"";"";"";"");; -> "Bonjour le monde"
tradmot ("Bonjour";"";"le";"monde");; -> "Bonjour le monde"
tradmot ("";"Bonjour";"le";"monde");; -> "Bonjour le monde"
tradmot ("";"";"";"Bonjour";"";"le";"monde";"");; -> "Bonjour le monde"
*)



(** infixe

@param abr un arbre binaire de recherche
@return l'index affixe associ     l'arbre binaire de recherche entr

val infixe : arbre -> (char * int) list = <fun>

*)

(** Tests :
infixe abr;; -> (('a', 0); ('b', 0); ('c', 0); ('d', 0); ('e', 0); ('f', 0); ('g',
   0); ('h', 0); ('i', 0); ('j', 0); ('k', 0); ('l', 0); ('m', 0); ('n', 0); ('o',
   0); ('p', 0); ('q', 0); ('r', 0); ('s', 0); ('t', 0); ('u', 0); ('v', 0); ('w',
   0); ('x', 0); ('y', 0); ('z', 0))
infixe Vide;; -> ()

o   abr est l'arbre binaire de recherche d fini plus haut.
*)



(** creation_index

@param alea un bool en qui vaut true si on veut un remplissage de l'arbre par des
    nombres al atoires et false pour un remplissage avec des 0, abr un arbre
    initialiser
@return l'arbre initialis

val creation_index : arbre -> bool -> arbre = <fun>

*)

(** Tests :
creation_index abr true ;; ->
Noeud
 (('n', 45),
  Noeud
```

```
    (('f', 97),
     Noeud
      (('c', 60), Noeud (('a', 55), Vide, Noeud (('b', 19), Vide, Vide)),
       Noeud (('e', 71), Noeud (('d', 44), Vide, Vide), Vide)),
     Noeud
      (('j', 4),
       Noeud
        (('h', 98), Noeud (('g', 64), Vide, Vide),
         Noeud (('i', 98), Vide, Vide)),
       Noeud
        (('l', 1), Noeud (('k', 56), Vide, Vide),
         Noeud (('m', 71), Vide, Vide)))),
   Noeud
    (('t', 86),
     Noeud
      (('q', 13), Noeud (('p', 94), Noeud (('o', 24), Vide, Vide), Vide),
       Noeud (('r', 2), Vide, Noeud (('s', 7), Vide, Vide))),
     Noeud
      (('x', 83),
       Noeud
        (('v', 22), Noeud (('u', 50), Vide, Vide),
         Noeud (('w', 71), Vide, Vide)),
       Noeud (('z', 42), Noeud (('y', 1), Vide, Vide), Vide))))

creation_index abr false ;;->
Noeud
 (('n', 0),
  Noeud
   (('f', 0),
    Noeud
     (('c', 0), Noeud (('a', 0), Vide, Noeud (('b', 0), Vide, Vide)),
      Noeud (('e', 0), Noeud (('d', 0), Vide, Vide), Vide)),
    Noeud
     (('j', 0),
      Noeud
       (('h', 0), Noeud (('g', 0), Vide, Vide), Noeud (('i', 0), Vide, Vide)),
      Noeud
       (('l', 0), Noeud (('k', 0), Vide, Vide), Noeud (('m', 0), Vide, Vide)))),
  Noeud
   (('t', 0),
    Noeud
     (('q', 0), Noeud (('p', 0), Noeud (('o', 0), Vide, Vide), Vide),
      Noeud (('r', 0), Vide, Noeud (('s', 0), Vide, Vide))),
    Noeud
     (('x', 0),
      Noeud
       (('v', 0), Noeud (('u', 0), Vide, Vide), Noeud (('w', 0), Vide, Vide)),
      Noeud (('z', 0), Noeud (('y', 0), Vide, Vide), Vide))))


creation_index Vide false ;; -> Vide
creation_index Vide true ;; -> Vide

o  abr est l'arbre binaire de recherche d fini plus haut.
*)
```

```
(** compte_lettre_arbre

@param abr un arbre binaire de recherche et lettre la lettre mise     jour dans l'
    arbre
@return l'arbre avec la lettre voulue incr ment e de 1

val compte_lettre_arbre : arbre -> char -> arbre = <fun>

*)

(** Tests :
compte_lettre_arbre Vide 'a';; -> Vide
compte_lettre_arbre abr 'a';; ->
Noeud
 (('n', 0),
  Noeud
   (('f', 0),
    Noeud
     (('c', 0), Noeud (('a', 1), Vide, Noeud (('b', 0), Vide, Vide)),
      Noeud (('e', 0), Noeud (('d', 0), Vide, Vide), Vide)),
    Noeud
     (('j', 0),
      Noeud
       (('h', 0), Noeud (('g', 0), Vide, Vide), Noeud (('i', 0), Vide, Vide)),
      Noeud
       (('l', 0), Noeud (('k', 0), Vide, Vide), Noeud (('m', 0), Vide, Vide)))),
  Noeud
   (('t', 0),
    Noeud
     (('q', 0), Noeud (('p', 0), Noeud (('o', 0), Vide, Vide), Vide),
      Noeud (('r', 0), Vide, Noeud (('s', 0), Vide, Vide))),
    Noeud
     (('x', 0),
      Noeud
       (('v', 0), Noeud (('u', 0), Vide, Vide), Noeud (('w', 0), Vide, Vide)),
      Noeud (('z', 0), Noeud (('y', 0), Vide, Vide), Vide))))
*)



(** compte_mot_global

@param abr une arbre binaire de recherche, mot la cha ne de caract res courante
    dont on cherche     recenser les lettres, n un compteur pour it rer sur les
    lettres de mot
@return l'arbre mis     jour apr s avoir compt  les occurrences des diff rentes
    lettres de mot par le biais de la fonction compte_lettre_arbre (voir ci-dessus)

val compte_mot_global : arbre -> string -> int -> arbre = <fun>
*)

(** Tests :
compte_mot_global abr "Bonjour" 0 ;;
-> Noeud
```

```
 (('n', 1),
  Noeud
   (('f', 0),
    Noeud
     (('c', 0), Noeud (('a', 0), Vide, Noeud (('b', 1), Vide, Vide)),
      Noeud (('e', 0), Noeud (('d', 0), Vide, Vide), Vide)),
    Noeud
     (('j', 1),
      Noeud
       (('h', 0), Noeud (('g', 0), Vide, Vide), Noeud (('i', 0), Vide, Vide)),
      Noeud
       (('l', 0), Noeud (('k', 0), Vide, Vide), Noeud (('m', 0), Vide, Vide)))),
  Noeud
   (('t', 0),
    Noeud
     (('q', 0), Noeud (('p', 0), Noeud (('o', 2), Vide, Vide), Vide),
      Noeud (('r', 1), Vide, Noeud (('s', 0), Vide, Vide))),
    Noeud
     (('x', 0),
      Noeud
       (('v', 0), Noeud (('u', 1), Vide, Vide), Noeud (('w', 0), Vide, Vide)),
      Noeud (('z', 0), Noeud (('y', 0), Vide, Vide), Vide))))
compte_mot_global Vide "Bonjour" 0 ;; -> Vide

o  abr est l'arbre binaire de recherche d finit plus haut.
*)


(** compte_mot_h

Voir d finition au dessus de compte_mot_global, ici on impose que le compteur d'
   it ration soit   0

val compte_mot_h : arbre -> string -> arbre = <fun>
*)



(** compte_lettre_dans_mot

@param mot le mot dont on veut compter le nombre  d occurrences  de la lettre
   lettre_pitrat, compte qui r cup re le nombre de lettre_pitrat et n un compteur
    pour initialiser le compte de d part
@return le nombre d'occurrences de la lettre lettre_pitrat dans mot

val compte_lettre_dans_mot : string -> int -> char -> int -> int = <fun>
*)

(** Tests :
compte_lettre_dans_mot "Bonjour" 0 'o' 0;; -> 2
compte_lettre_dans_mot "Bonjour" 1 'o' 0;; -> 3
*)
```

```
(** compte_mot_h

Voir d finition au dessus de compte_lettre_dans_mot, ici on impose que le compteur
    d'it ration soit    0

val compte_lettre : string -> int -> char -> int = <fun>
*)



(** compte_lettre_dans_mot

@param abr l'arbre binaire de recherche que l'on souhaite mettre    jour, lettre la
    lettre dont le compte est mis    jour et nombre son nouveau compte
@return l'arbre mis    jour
val maj_index_courant : arbre -> char -> int -> arbre = <fun>
*)

(** Tests :
maj_index_courant abr 'a' 32;; ->
Noeud
 (('n', 0),
  Noeud
   (('f', 0),
    Noeud
     (('c', 0), Noeud (('a', 32), Vide, Noeud (('b', 0), Vide, Vide)),
      Noeud (('e', 0), Noeud (('d', 0), Vide, Vide), Vide)),
    Noeud
     (('j', 0),
      Noeud
       (('h', 0), Noeud (('g', 0), Vide, Vide), Noeud (('i', 0), Vide, Vide)),
      Noeud
       (('l', 0), Noeud (('k', 0), Vide, Vide), Noeud (('m', 0), Vide, Vide)))),
  Noeud
   (('t', 0),
    Noeud
     (('q', 0), Noeud (('p', 0), Noeud (('o', 0), Vide, Vide), Vide),
      Noeud (('r', 0), Vide, Noeud (('s', 0), Vide, Vide))),
    Noeud
     (('x', 0),
      Noeud
       (('v', 0), Noeud (('u', 0), Vide, Vide), Noeud (('w', 0), Vide, Vide)),
      Noeud (('z', 0), Noeud (('y', 0), Vide, Vide), Vide))))


o  abr est l'arbre binaire de recherche d finit plus haut.
*)


(** litt

@param nombre un entier    convertir et langue la langue de conversion ('f' pour
    fran ais, 'b' pour belge et 's' pour suisse)
@pre: 0<nombre<=100
@return l'  criture   litt rale d'un nombre dans la langue choisie
@raises Failure "erreur nombre rentr " si le nombre est entre entre -1 et -9 et
```

```
Failure "Votre nombre doit etre compris entre 0 exclus et 100 inclus" si le nombre
   n'est pas compris entre 0 et 100


val litt : int -> char -> string = <fun>
*)

(** Tests :
litt (-110) 's';; -> Exception: Failure "Votre nombre doit etre compris entre 0
   exclus et 100 inclus".
litt 101 'b';; -> Exception: Failure "Votre nombre doit etre compris entre 0 exclus
   et 100 inclus".
litt 0 'f';; -> ""
litt (-1) 'f';; -> Exception: Failure "erreur nombre rentre"
litt 80 'f';; -> "quatre-vingts"
litt 80 's';; -> "huitante"
litt 78 'b';; -> "septante-huit"
litt 13 'f';; -> "treize"
*)



(** pre_choix_alea_gabarits

@param l une liste d'lments , n un compteur qui permet de faire d filer les
      lments de la liste et choix le choix d'lments de la liste
@return un lment pris au hasard dans cette liste
@raises: Failure "liste vide" si l est vide

val pre_choix_alea_gabarits : 'a list -> int -> int -> 'a = <fun>
*)

(** Tests :
pre_choix_alea_gabarits ("cher lecteur,";"ami lecteur,";"chers lecteurs,";"") 0 2;;
    -> "chers lecteurs,"
pre_choix_alea_gabarits ("vous avez") 0 0;; -> "vous avez"
pre_choix_alea_gabarits ("toutes";"") 0 1;; -> ""
*)



(** choix_alea_gabarits

Voir d finition au dessus de pre_choix_alea_gabarits, ici on impose que le
   compteur d'it rations soit 0 et que le choix soit fait al atoirement parmi
   un des lments de la liste.

val choix_alea_gabarits : 'a list -> 'a = <fun>
*)



(** initiale_maj

@param s une cha ne de caract res
@return La m me cha ne de caract re s avec une majuscule sur la premi re lettre
```

```
val initiale_maj : string -> string = <fun>
*)

(** Tests :
initiale_maj "";; -> ""
initiale_maj "ceci est une chaine de caracteres";; -> "Ceci est une chaine de
    caracteres"
*)



(** compare_arbre

@param arbre1 et arbre2 les deux arbres     comparer et le_nombre le nombre de
    lettres     comparer en partant de 'a'
@return true si les deux arbres sont les mmes et false sinon

val compare_arbre : arbre -> arbre -> int -> bool = <fun>
*)

(** Tests :
compare_arbre abr1 abr2 10 ;; -> false
compare_arbre abr1 abr2 9 ;; -> true


avec :
let abr1 = Noeud
 (('n', 0),
  Noeud
   (('f', 0),
    Noeud
     (('c', 0), Noeud (('a', 0), Vide, Noeud (('b', 0), Vide, Vide)),
      Noeud (('e', 0), Noeud (('d', 0), Vide, Vide), Vide)),
    Noeud
     (('j', 0),
      Noeud
       (('h', 0), Noeud (('g', 0), Vide, Vide), Noeud (('i', 0), Vide, Vide)),
      Noeud
       (('l', 0), Noeud (('k', 0), Vide, Vide), Noeud (('m', 0), Vide, Vide)))),
  Noeud
   (('t', 0),
    Noeud
     (('q', 0), Noeud (('p', 0), Noeud (('o', 0), Vide, Vide), Vide),
      Noeud (('r', 0), Vide, Noeud (('s', 0), Vide, Vide))),
    Noeud
     (('x', 0),
      Noeud
       (('v', 0), Noeud (('u', 0), Vide, Vide), Noeud (('w', 0), Vide, Vide)),
      Noeud (('z', 0), Noeud (('y', 0), Vide, Vide), Vide)))));;

let abr2 = Noeud
 (('n', 0),
  Noeud
   (('f', 0),
    Noeud
```

```
      (('c', 0), Noeud (('a', 0), Vide, Noeud (('b', 0), Vide, Vide)),
       Noeud (('e', 0), Noeud (('d', 0), Vide, Vide), Vide)),
     Noeud
      (('j', 1),
       Noeud
        (('h', 0), Noeud (('g', 0), Vide, Vide), Noeud (('i', 0), Vide, Vide)),
       Noeud
        (('l', 0), Noeud (('k', 0), Vide, Vide), Noeud (('m', 0), Vide, Vide)))),
   Noeud
    (('t', 0),
     Noeud
      (('q', 0), Noeud (('p', 0), Noeud (('o', 0), Vide, Vide), Vide),
       Noeud (('r', 0), Vide, Noeud (('s', 0), Vide, Vide))),
     Noeud
      (('x', 0),
       Noeud
        (('v', 0), Noeud (('u', 0), Vide, Vide), Noeud (('w', 0), Vide, Vide)),
       Noeud (('z', 0), Noeud (('y', 0), Vide, Vide), Vide))));;
*)



(** decoupe_liste

@param n la taille de la liste souhait e et l la liste     r duire de taille
@return la liste r duite

val decoupe_liste : int -> 'a list -> 'a list = <fun>
*)

(** Tests :
decoupe_liste 5 tab;; -> (('a', 0); ('b', 0); ('c', 0); ('d', 0); ('e', 0))
decoupe_liste (-1) tab;; -> ()

avec :
let tab =
(('a', 0); ('b', 0); ('c', 0); ('d', 0); ('e', 0); ('f', 0); ('g', 0);('h', 0); ('i
    ', 0); ('j', 0); ('k', 0); ('l', 0); ('m', 0); ('n', 0);('o', 0); ('p', 0); ('q
    ', 0); ('r', 0); ('s', 0); ('t', 0); ('u', 0);('v', 0); ('w', 0); ('x', 0); ('y
    ', 0); ('z', 0)) ;;

*)



(** est_fixe

@param l une lettre de l'alphabet
@return true si la lettre est une lettre dite fixe c'est     dire une lettre dont le
     nombre ne varie plus apr s la phase d'initialisation de hofstadter ou pitrat.
    En d'autre terme, cette lettre ne figure dans aucune    criture    litt rale des
    nombres compris entre 0 et 100.

val est_fixe : char -> bool = <fun>
*)
```

```
(** Tests :
est_fixe 'a' ;; -> false
est_fixe 'l' ;; -> true

*)


(** fragment_phrase

@param arb un arbre binaire de recherche, f une fonction qui    un couple (lettre,
    nombre  d occurrence ) et    une cha ne de caract re associe une nouvelle
    cha ne de caract re et le_nombre le nombre de lettres de l'alphabet
    s lectionn
@return une cha ne de caract res particuli re


val fragment_phrase : arbre -> (char * int -> string -> string) -> int -> string =
    <fun>
*)

(* Tests :
fragment_phrase abr concat_index_fixe 12 ;; -> "adix-neufbcdefghiquatrejcinquante-
    sixkunl"

fragment_phrase abr concat_index_entier 14 ;; -> "cinquante-cinq a, dix-neuf b,
    soixante c, quarante-quatre d, soixante-et-onze e, quatre-vingt-dix-sept f,
    soixante-quatre g, quatre-vingt-dix-huit h, quatre-vingt-dix-huit i, quatre j,
    cinquante-six k, un l, soixante-et-onze m, quarante-cinq n"

fragment_phrase abr concat_index_mobile 13 ;; -> "cinquante-cinq  soixante quarante
    -quatre soixante-et-onze quatre-vingt-dix-sept soixante-quatre quatre-vingt-dix-
    huit quatre-vingt-dix-huit      "

avec :

let debut = tradmot(List.map choix_alea_gabarits gabarits_debut) ;;
let fin = tradmot(List.map choix_alea_gabarits gabarits_fin) ;;

et avec f :

let concat_index_entier (a,b) reste = (litt b 'f')^" "^(String.make 1 a)^(if reste
    ="" then "" else ","^" "^(if a='y' then fin else "")^reste) ;;
let concat_index_mobile (a,b) reste = (if not(est_fixe a) then (litt b 'f') else
    "")^" "^reste ;;
let concat_index_fixe (a,b) reste = (if (est_fixe a) then (litt b 'f') else "")^(
    String.make 1 a)^(if a='y' then fin else "")^reste ;;

et encore :

let abr =
Noeud
 (('n', 45),
  Noeud
   (('f', 97),
    Noeud
     (('c', 60), Noeud (('a', 55), Vide, Noeud (('b', 19), Vide, Vide)),
```

```
      Noeud (('e', 71), Noeud (('d', 44), Vide, Vide), Vide)),
    Noeud
     (('j', 4),
      Noeud
       (('h', 98), Noeud (('g', 64), Vide, Vide),
        Noeud (('i', 98), Vide, Vide)),
      Noeud
       (('l', 1), Noeud (('k', 56), Vide, Vide),
        Noeud (('m', 71), Vide, Vide)))),
  Noeud
   (('t', 86),
    Noeud
     (('q', 13), Noeud (('p', 94), Noeud (('o', 24), Vide, Vide), Vide),
      Noeud (('r', 2), Vide, Noeud (('s', 7), Vide, Vide))),
    Noeud
     (('x', 83),
      Noeud
       (('v', 22), Noeud (('u', 50), Vide, Vide),
        Noeud (('w', 71), Vide, Vide)),
      Noeud (('z', 42), Noeud (('y', 1), Vide, Vide), Vide)))) ;;


*)



(** hofstadter

@param index_fixe l'index de la partie fixe de la phrase, index_reel l'index de
    toute la phrase, phrase la phrase    transformer en phrase r flexive, iter le
    nombre d'it rations,  fin un bool en qui indique si l'on a trouv  ou non,
    le_nombre le nombre de lettre    inclure dans la phrase r flexive, d but le
    d but de phrase g n r e al atoirement, concat_index_fixe la fonction de
    cr ation de la phrase fixe et concat_index_mobile la fonction de cr ation de
    la phrase mobile.
@return au bout de 100 it rations maximum, retourne un bool en indiquant si on a
    obtenu une phrase r flexive et l'index de la phrase.

val hofstadter : arbre -> arbre -> string -> int -> bool -> int -> string ->
      (char * int -> string -> string) -> (char * int -> string -> string) -> bool
    * arbre = <fun>

*)

(** Tests :
hofstadter Vide index_reel phrase_entiere 0 false 26 debut concat_index_fixe
    concat_index_mobile ;; ->
(false,
 Noeud
  (('n', 22),
   Noeud
    (('f', 5),
     Noeud
      (('c', 6), Noeud (('a', 15), Vide, Noeud (('b', 3), Vide, Vide)),
       Noeud (('e', 39), Noeud (('d', 8), Vide, Vide), Vide)),
```

```
      Noeud
       (('j', 1),
        Noeud
         (('h', 4), Noeud (('g', 4), Vide, Vide),
          Noeud (('i', 23), Vide, Vide)),
        Noeud
         (('l', 8), Noeud (('k', 1), Vide, Vide),
          Noeud (('m', 3), Vide, Vide)))),
    Noeud
     (('t', 30),
      Noeud
       (('q', 8), Noeud (('p', 7), Noeud (('o', 13), Vide, Vide), Vide),
        Noeud (('r', 21), Vide, Noeud (('s', 16), Vide, Vide))),
      Noeud
       (('x', 7),
        Noeud
         (('v', 8), Noeud (('u', 21), Vide, Vide),
          Noeud (('w', 1), Vide, Vide)),
        Noeud (('z', 6), Noeud (('y', 1), Vide, Vide), Vide))))))


avec :
let index_affiche = creation_index abr true ;;
let debut = "Vous avez bien raison de compter les lettres de cette phrase bizarre
   avant de la croire quand elle vous affirme qu'elle contient" ;;
let fin = "et pour finir " ;;
let milieu_1 = fragment_phrase (index_affiche) concat_index_entier 26 ;;
let phrase_entiere = debut^" "^milieu_1^"." ;;
let index_reel = compte_mot_h (creation_index abr false) phrase_entiere ;;
*)


(** boucle_hofstadter

@param nb_essai le nombre d'essai sur un m me corps de phrase, le_nombre le nombre
    de lettre  inclure dans la phrase  transformer en phrase r flexive, debut
   le d but de phrase g n r e al atoirement, concat_index_entier la fonction
   de cr ation de la phrase toute enti re, concat_index_mobile la fonction de
    cr ation de la phrase mobile, concat_index_fixe la fonction de cr ation de la
   phrase fixe et phrase_mobile la phrase mobile
@return au bout d'un maximum de 100 situations initiales g n r es al atoirement
    , retourne un bool en indiquant si la phrase trouv e est r flexive ou non, la
    phrase trouv e ainsi que le nombre de situations initiales g n r es.

val boucle_hofstadter : int -> int -> string -> (char * int -> string -> string
   ) -> (char * int -> string -> string) -> (char * int -> string -> string) -> '
   a -> bool * int * string = <fun>

*)

(** Tests :
boucle_hofstadter 0 26 debut concat_index_entier concat_index_mobile
   concat_index_fixe () ;; ->
(true,56, "Vous avez bien raison de compter les lettres de cette phrase bizarre
   avant de la croire quand elle vous affirme qu'elle contient onze a, trois b,
   sept c, huit d, trente-cinq e, cinq f, quatre g, sept h, vingt-sept i, un j, un
```

```
    k, huit l, trois m, vingt-et-un n, douze o, huit p, six q, seize r, dix-sept s,
    vingt-huit t, dix-neuf u, huit v, un w, six x, un y, et pour finir six z.")

*)


(** initialise_pitrat

@param le_nombre le nombre de lettres     inclure dans la phrase r flexive ,
    corps_phrase le corps de la phrase g n r   al atoirement , arbre_entier l'
    arbre de recherche binaire qui va comptabiliser l'ensemble des lettres dans la
    phrase, iter_int le nombre d'it ration , phrase_totale l'ensemble de la phrase
    et concat_index_entier la fonction de cr ation de la phrase toute enti re
@return l'arbre qui rend compte du nombre de lettre dans la phrase

val initialise_pitrat : int -> string -> arbre -> int -> string -> (char * int ->
    string -> string) -> arbre = <fun>
*)

(** Tests :
initialise_pitrat 26 debut index_aleatoire 97 "" concat_index_entier ;; ->
Noeud
 (('n', 35),
  Noeud
   (('f', 8),
    Noeud
     (('c', 12), Noeud (('a', 28), Vide, Noeud (('b', 3), Vide, Vide)),
      Noeud (('e', 62), Noeud (('d', 10), Vide, Vide), Vide)),
    Noeud
     (('j', 1),
      Noeud
       (('h', 5), Noeud (('g', 6), Vide, Vide),
        Noeud (('i', 35), Vide, Vide)),
      Noeud
       (('l', 8), Noeud (('k', 1), Vide, Vide), Noeud (('m', 4), Vide, Vide)))),
  Noeud
   (('t', 37),
    Noeud
     (('q', 14), Noeud (('p', 6), Noeud (('o', 13), Vide, Vide), Vide),
      Noeud (('r', 23), Vide, Noeud (('s', 16), Vide, Vide))),
    Noeud
     (('x', 7),
      Noeud
       (('v', 9), Noeud (('u', 19), Vide, Vide),
        Noeud (('w', 1), Vide, Vide)),
      Noeud (('z', 7), Noeud (('y', 1), Vide, Vide), Vide))))

avec :

let debut = "Vous avez bien raison de compter les lettres de cette phrase bizarre
    avant de la croire quand elle vous affirme qu'elle contient" ;;

let index_aleatoire = (creation_index abr true) ;;


*)
```

```
(** corps_pitrat

@param lettre la lettre courante modifi e , le_nombre le nombre de lettre
    inclure dans la phrase r flexive , index_fixe_infixe l'index de la phrase fixe
    sous forme de liste , index_fixe l'index de la phrase fixe , index_courant l'index
     de la phrase courante, phrase_mobile la phrase mobile et concat_index_mobile la
     fonction de cr ation de la phrase mobile.
@return r alise une modification compl te de la phrase en la modifiant lettre par
     lettre. A chaque it ration , on regarde si la phrase obtenue n'est pas
    r flexive auquel cas on stoppe le processus

val corps_pitrat : char -> int -> (char * int) list -> arbre -> arbre -> string ->
    (char * int -> string -> string) -> bool * arbre * string = <fun>

*)

(** Tests :
corps_pitrat ' ' 26 (infixe index_fixe) index_fixe index_courant phrase_mobile
    concat_index_mobile ;; ->('z', false,
 Noeud
  (('n', 96),
   Noeud
    (('f', 47),
     Noeud
      (('c', 29), Noeud (('a', 53), Vide, Noeud (('b', 46), Vide, Vide)),
       Noeud (('e', 36), Noeud (('d', 24), Vide, Vide), Vide)),
     Noeud
      (('j', 58),
       Noeud
        (('h', 70), Noeud (('g', 32), Vide, Vide),
         Noeud (('i', 66), Vide, Vide)),
       Noeud
        (('l', 4), Noeud (('k', 14), Vide, Vide),
         Noeud (('m', 10), Vide, Vide)))),
   Noeud
    (('t', 35),
     Noeud
      (('q', 36), Noeud (('p', 89), Noeud (('o', 7), Vide, Vide), Vide),
       Noeud (('r', 22), Vide, Noeud (('s', 53), Vide, Vide))),
     Noeud
      (('x', 64),
       Noeud
        (('v', 94), Noeud (('u', 65), Vide, Vide),
         Noeud (('w', 56), Vide, Vide)),
       Noeud (('z', 38), Noeud (('y', 45), Vide, Vide), Vide)))),
 "cinquante-trois  vingt-neuf vingt-quatre trente-six quarante-sept trente-deux
    soixante-dix soixante-six      quatre-vingt-seize sept quatre-vingt-neuf trente-
    six vingt-deux cinquante-trois trente-cinq soixante-cinq quatre-vingt-quatorze
     soixante-quatre  trente-huit ")

avec :

let index_fixe = (creation_index abr true) ;;
let index_courant = (creation_index abr true) ;;
```

```
let phrase_mobile = "dix  cinq dix vingt-huit cinq cinq six vingt-huit     vingt
    douze six trois dix-huit dix-neuf vingt-six dix-sept huit  onze  six ";;

*)


(** pitrat

@param le_nombre le nombre de lettres     inclure dans la phrase, index_fixe l'index
     de la partie fixe de la phrase, index_courant l'index de toute la phrase,
    phrase_mobile la phrase mobile, iter le nombre d'it rations, fin un bool en
    qui indique si l'on a trouv  ou non et concat_index_mobile la fonction de
    cr ation de la phrase mobile.
@return au bout de 100 it rations maximum, retourne un bool en indiquant si on a
    obtenu une phrase r flexive et l'index de la phrase.

val pitrat : int -> arbre -> arbre -> string -> int -> bool -> (char * int ->
    string -> string) -> bool * arbre =
  <fun>


*)

(** Tests :
pitrat 2 index_fixe index_courant phrase_mobile 0 false concat_index_mobile ;; ->
(true,
 Noeud
  (('n', 84),
   Noeud
    (('f', 17),
     Noeud
      (('c', 63), Noeud (('a', 10), Vide, Noeud (('b', 3), Vide, Vide)),
       Noeud (('e', 85), Noeud (('d', 27), Vide, Vide), Vide)),
     Noeud
      (('j', 28),
       Noeud
        (('h', 70), Noeud (('g', 31), Vide, Vide),
         Noeud (('i', 70), Vide, Vide)),
       Noeud
        (('l', 8), Noeud (('k', 79), Vide, Vide),
         Noeud (('m', 88), Vide, Vide)))),
   Noeud
    (('t', 37),
     Noeud
      (('q', 58), Noeud (('p', 35), Noeud (('o', 95), Vide, Vide), Vide),
       Noeud (('r', 80), Vide, Noeud (('s', 32), Vide, Vide))),
     Noeud
      (('x', 95),
       Noeud
        (('v', 51), Noeud (('u', 19), Vide, Vide),
         Noeud (('w', 19), Vide, Vide)),
       Noeud (('z', 79), Noeud (('y', 42), Vide, Vide), Vide)))))

avec :
let index_aleatoire = (creation_index abr true) ;;
```

```
let index_courant = initialise_pitrat 2 debut index_aleatoire 97 ""
    concat_index_entier ;;
let milieu_2 = fragment_phrase (index_courant) concat_index_fixe 2 ;;
let index_fixe = compte_mot_h (creation_index abr false) (debut^milieu_2) ;;
let phrase_mobile = fragment_phrase (index_courant) concat_index_mobile 2 ;;

*)


(** boucle_pitrat

@param nb_essai le nombre d'essai sur un m me corps de phrase, le_nombre le nombre
     de lettres    inclure dans la phrase r flexive, d but le d but de phrase
     g n r  al atoirement, concat_index_entier la fonction de cr ation de la
    phrase toute enti re, concat_index_mobile la fonction de cr ation de la phrase
     mobile, concat_index_fixe la fonction de cr ation de la phrase fixe et
    phrase_mobile la phrase mobile.
@return au bout d'un maximum de 100 situations initiales g n r es al atoirement
    , retourne un bool en indiquant si la phrase trouv e est r flexive ou non, la
     phrase trouv e ainsi que le nombre de situations initiales g n r es.

val boucle_pitrat : int -> int -> string -> (char * int -> string -> string) -> (
    char * int -> string -> string) -> (char * int -> string -> string) -> string
    list -> bool * int * string = <fun>
*)


(** Tests :
boucle_pitrat 0 26 debut concat_index_entier concat_index_mobile concat_index_fixe
    () ;; -> (95, "Vous avez bien raison de compter les lettres de cette phrase
    bizarre avant de la croire quand elle vous affirme qu'elle contient onze a,
    trois b, sept c, huit d, trente-cinq e, cinq f, quatre g, sept h, vingt-sept i,
    un j, un k, huit l, trois m, vingt-et-un n, douze o, huit p, six q, seize r, dix
    -sept s, vingt-huit t, dix-neuf u, huit v, un w, six x, un y, et pour finir six
    z.")

*)



(** generateur

@param boucle_f une fonction de g n ration de phrases r flexives (boucle_pitrat
    ou boucle_hofstadter), le_nombre le nombre de lettres    inclure dans la phrase
    r flexive, langue la langue de traduction des chiffres litt raux, iter_corps
    le nombre de corps diff rents al atoires sur lesquels on veut tester la
    r flexivit  des phrases et list_solutions la liste des phrases r flexives
    obtenues et nombre_phrases_trouvees le nombre de phrases reflexives trouv es
@return retourne une liste constitu e de toutes les phrases r flexives trouv es
    ainsi que du nombre_essai pour les trouver.

val generateur : (int -> 'a -> string ->(char * int -> string -> string) ->(char *
    int -> string -> string) -> (char * int -> string -> string) -> 'b list -> bool
    * 'c * string) ->'a -> char -> int -> ('c * string) list -> int -> ('c * string)
     list * int = <fun>
*)
```

```
(** Tests :

generateur boucle_pitrat 10 'f' 5 () 0;; -> (((0,
   "Ami lecteur, vous avez bien raison de compter avec soin toutes les lettres de
      cette bizarre phrase avant de la croire quand elle vous affirme qu'elle
      contient douze 'a', trois 'b', sept 'c', sept 'd', trente-trois 'e', trois 'f
      ', un 'g', deux 'h', treize 'i', un 'j'.");
  (7,
   "Ami lecteur, vous avez vraiment raison de compter et de recompter avec soin
      toutes les lettres de cette phrase avant de la croire quand elle affirme qu'
      elle contient quatorze 'a', un 'b', huit 'c', six 'd', trente-quatre 'e',
      trois 'f', un 'g', trois 'h', treize 'i', un 'j'.");
  (0,
   "Vous avez bien raison de compter et de recompter toutes les lettres de cette
      bizarre phrase avant de la croire quand elle affirme qu'elle contient dix 'a
      ', trois 'b', six 'c', dix 'd', trente-et-un 'e', trois 'f', un 'g', deux 'h
      ', douze 'i', un 'j'.")),
 3)

generateur boucle_hofstadter 10 'f' 5 () 0;; ->(((57,
   "Cher lecteur, vous avez raison de compter et de recompter soigneusement toutes
      les lettres de cette bizarre phrase avant de la croire quand elle vous
      affirme qu'elle contient exactement quatorze 'a', deux 'b', neuf 'c', neuf 'd
      ', quarante-deux 'e', six 'f', deux 'g', trois 'h', neuf 'i', un 'j'.");
  (6,
   "Chers lecteurs, vous avez raison de compter et de recompter toutes les lettres
      de cette bizarre phrase avant de la croire quand elle vous affirme qu'elle
      contient exactement douze 'a', deux 'b', dix 'c', neuf 'd', trente-huit 'e',
      cinq 'f', un 'g', quatre 'h', neuf 'i', un 'j'.");
  (23,
   "Cher lecteur, vous avez vraiment raison de compter toutes les lettres de cette
      bizarre phrase avant de la croire quand elle affirme qu'elle contient
      exactement quatorze 'a', deux 'b', neuf 'c', sept 'd', trente-cinq 'e',
      quatre 'f', un 'g', trois 'h', dix 'i', un 'j'.")),
 3)
*)
```