Algorithmique 2 : structures récursives, linéaires et binaires

Projet

Index

Généralités

Le projet consiste à développer en C un exécutable qui produit la *liste des mots* qui apparaissent dans une *réunion de lexiques donnés*, chacun des mots listés étant accompagné d'indicateurs montrant à quels lexiques il appartient et de la liste strictement croissante des numéros de ligne auquel il apparait dans un *texte donné*.

La notion de mot doit être comprise par défaut comme une suite de longueur maximale de caractères qui ne sont ni des caractères d'espacement ni des caractères de ponctuation (pour les caractères de base du C, qui n'appartiennent donc pas aux catégories `isspace` ou `ispunct`).

Un lexique peut être signifié sous deux formes sur la ligne de commande. Soit au moyen d'une chaine de caractères à l'intérieur de laquelle figurent les mots du lexique, séparés par des caractères d'espacement ou de ponctuation. Soit au moyen d'un nom de fichier texte, les mots figurant dans ce cas dans ce fichier. Pour distinguer les deux formes, les chaines signifiant un nom de fichier sont précédées d'un tiret sur la ligne de commande.

MÀJ 17-11
le mots → les mots

Par défaut, le texte est lu sur l'entrée standard.

Le résultat de l'exécutable, l'*index*, est produit par défaut sur la sortie standard. Il est subdivisé en colonnes, séparées par le caractère tabulation. La première colonne est dévolue aux mots, celle d'après à la réunion des lexiques donnés sous la forme de chaines, quand bien même cette réunion serait vide, celles d'après aux lexiques donnés sous la forme de noms de fichiers, la dernière aux listes des numéros de ligne. Les champs des deux premières colonnes de la première ligne sont vides. Figurent dans les champs suivants de la première ligne les noms de fichier. À partir de la deuxième ligne, figurent, dans le champ de la première colonne, le mot, dans celui de la colonne d'après, un indicateur d'appartenance à la réunion des lexiques donnés sous la forme de chaines, dans ceux des colonnes suivantes, les indicateurs d'appartenance aux lexiques donnés sous la forme de noms de fichier, dans celui de la dernière colonne, la liste des numéros de ligne. L'indicateur d'appartenance est le caractère « x ». Les numéros de ligne sont séparés par une virgule. Par défaut, la liste n'est pas ordonnée.

MÀJ 17-11
au mots → aux mots, quant → quand

Le nom de l'exécutable est imposé : `lidx`, pour *line indexer*.

Pour fixer les idées, supposons que le fichier texte `surleboutdesdoigts.txt` (d'après Paul Vincensini) ait pour contenu (hors numéros de ligne)

```
1 Je compte les jours
2 Sur mes doigts
3 J'y compte aussi mes amis
4 Mes amours
5 Un jour
6 Je ne compterai plus que mes doigts
7 Sur mes doigts.
```

le fichier texte `lex0.txt`

```
compte sur mes doigts
```

et le fichier texte `lex1.txt` (d'après Charles Aznavour)

```
Mes amis,
 mes amours,
  mes emmerdes...
```

Alors la commande :

```
$ ./lidx - lex0.txt "chaque jour" compte - "lex1.txt" < surleboutdesdoigts.txt
```

MÀJ 28-11 Ajout de `$` et de « `./` ».

produit sur la sortie standard, à l'ordre près sur les lignes 2 et suivantes :

MÀJ 17-11 lignes des mots → 2 et suivantes

```
          	          	lex0.txt  	lex1.txt  	¶
chaque    	x         	          	          	¶
jour      	x         	          	          	5¶
compte    	x         	x         	          	1,3¶
sur       	          	x         	          	¶
mes       	          	x         	x         	2,3,6,7¶
doigts    	          	x         	          	2,6,7¶
Mes       	          	          	x         	4¶
amis      	          	          	x         	3¶
amours    	          	          	x         	4¶
emmerdes  	          	          	x         	¶
```

les flèches grisées signifiant les caractères tabulation (avec une largeur de taquet fixée ici à 10) et le symbole ¶, la fin d'une ligne de texte.

Des options peuvent permettre d'obtenir des résultats différents. Par exemple `-u`, pour convertir tous les caractères lus en majuscules (*uppercase*), et `-S`, pour afficher la liste dans l'ordre lexicographique (*sort*). La commande :

```
$ ./lidx -u -S "chaque jour compte" - lex0.txt - lex1.txt \
> < surleboutdesdoigts.txt
```

MÀJ 28-11 (Suite.) Du coup, c'est trop long : on prolonge en passant à la ligne.

produit ainsi sur la sortie standard :

```
          	          	lex0.txt  	lex1.txt  	¶
AMIS      	          	          	x         	3¶
AMOURS    	          	          	x         	4¶
CHAQUE    	x         	          	          	¶
COMPTE    	x         	x         	          	1,3¶
DOIGTS    	          	x         	          	2,6,7¶
EMMERDES  	          	          	x         	¶
JOUR      	x         	          	          	5¶
MES       	          	x         	x         	2,3,4,6,7¶
SUR       	          	x         	          	2,7¶
```

Mise en œuvre

Votre programme doit être écrit en C et doit supporter les options de compilation usuelles : `-std=c11`, `-Wall`, `-Wconversion`, `-Werror`, `-Wextra`, `-Wpedantic`, `-Wwrite-strings` et `-O2`.

Il doit être correctement indenté (2 espaces), espacé (une espace après un mot-clé comme **`if`** ou **`while`**, une virgule, aucune espace après une parenthèse ouvrante et avant une parenthèse fermante, une espace de chaque côté d'un opérateur binaire...), ne doit comporter aucune tabulation. Aucune des lignes de texte qui y figurent ne doit excéder 80 caractères ; si une instruction longue figure sur plusieurs lignes, les lignes en excès sont double indentées par rapport à la première.

Les identificateurs des types, sous-programmes et paramètres doivent être correctement nommés. Vous éviterez toute duplication de code.

À la base de votre travail, vous utilisez nécessairement l'un des deux modules efficaces, présentés et travaillés en cours et en TP : `hashtable` ou `bst` dans sa version AVL. Vous ne modifierez en aucun cas leurs parties interface. Vous utiliserez également le module `holdall` dans le même but que celui des exemples présentés en cours et proposés à l'occasion des séances de TP. Vous n'êtes autorisé à n'y introduire que la fonction `holdall_sort` décrite dans la fiche n° 8 de TP. MÀJ 17-11 Refonte complète du paragraphe.

Vous pouvez développer des modules additionnels, par exemple pour gérer les options ou pour développer la partie récupération de mots dans un flot texte avec leurs numéros de ligne. MÀJ 17-11 Nouveau paragraphe.

Toutes les zones mémoires explicitement allouées devront être désallouées avant la fin de l'exécution, même en cas d'erreur. Jamais deux chaines de caractères de même valeur ne devront être allouées.

Les mots lus pourront être tronqués. La longueur du préfixe retenu pour chaque mot lu sera au minimum égale à 63. Toute troncature devra donner lieu à un message d'avertissement. Le résidu de la troncature ne devra en aucun cas être considéré comme un mot. MÀJ 28-11 Formulation plus claire.

Tout message d'avertissement ou d'erreur sera envoyé vers la sortie erreur.

Deux options devront obligatoirement être supportées :
`--help` : afficher l'aide selon le format usuel ;
`-S` ou `--sort` : afficher la liste dans l'ordre lexicographique.

Les options suivantes pourront être développées :
`--case=VALUE` : convertir tous les caractères lus en minuscule si `VALUE` vaut `lower`, en majuscules si `VALUE` vaut `upper`, les laisser tels quels si `VALUE` vaut `as-is` ;
`-l` : équivalent à `--case=lower` ;
`-s` : équivalent à `--case=as-is` ;
`-u` : équivalent à `--case=upper` ;
`-i FILE` ou `--input=FILE` : au lieu de l'entrée standard, lire le texte dans le fichier `FILE` ;
`-o FILE` ou `--ouput=FILE` : au lieu de la sortie standard, écrire résultat dans le fichier `FILE`.

Votre programme doit répondre aux diverses exigences : il pourra être testé avec des outils automatiques.

Modalités

Vous pouvez réaliser ce projet à deux, pas plus.

Votre projet sera compilable et exécutable sur les machines des salles de travaux pratiques du département d'informatique et sous Xubuntu.

Votre projet devra être rendu avec tous les fichiers source nécessaires, un fichier `makefile`, un rapport de développement, un jeu d'exemples illustrant le mieux possible ses fonctionnalités et performances. Le rapport sera fourni sous forme électronique (fichier au format `pdf`) qui précisera l'objet de chacun des modules, décrira l'implantation, commentera les exemples, explicitera les limites éventuelles du programme.

Vous remettrez votre projet au plus tard le vendredi 10 janvier 2020 à 12 h dans une archive au format `tar.gz` de nom *identifiant*`_projet.tar.gz`, où *identifiant* est l'identifiant de 8 caractères de l'un des membres du groupe. Cette archive sera envoyée en pièce jointe à un courriel à l'adresse de votre chargé de TP, de sujet « Algo2 projet ». Un seul envoi sera effectué par groupe : l'éventuel autre membre du groupe sera mis en copie du message.

Vous soutiendrez le projet individuellement, oralement, sur une machine des salles de travaux pratiques et sous Xubuntu.

La notation sera sensible à la propreté du code, à son homogénéité, aux performances de l'exécutable, à la clarté des explications fournies dans le rapport et lors de la soutenance, ainsi qu'à un passage sans encombre au grill `valgrind`.

Grille d'évaluation

Si la moindre erreur de compilation intervient lors de la soutenance, la note attribuée au projet sera de 0/20 : c'est la note juste pour un travail de programmation. 0/20 de même pour le non respect de la mise en forme standard,pour tout plagiat ou pour méconnaissance du projet présenté.

MÀJ 16-12 Mise forme non standard ⇒ 0/20.

En dehors de ces cas rédhibitoires la grille d'évaluation sur 20 points est :

critère	*points*
nom de l'archive correct	0,5
archive propre	0,5
makefile de production de l'exécutable `lidx` correct	0,5
option `--help` correcte au regard de ce qui est développé par l'exécutable	0,5
gestion correcte de la ligne de commande	1
test `valgrind` probant	2
développement d'un module spécifique de lecture de mots. Le module figure dans un dossier à part. Il est utilisé par `lidx`. Figurent également dans ce dossier un source de démonstration et son fichier `makefile`	2
développement d'un module spécifique d'ajout d'entiers du type `longint` à un ensemble initialement vide. Tout entier ajouté est supposé supérieur ou égal à ceux déjà dans l'ensemble. Une opération permet l'affichage de la suite croissante des entiers dans l'ensemble. Le module figure dans un dossier à part. Il est utilisé par `lidx`. Figurent également dans ce dossier un source de démonstration et son fichier `makefile`	2
développement d'un module spécifique d'ajout d'entiers à un ensemble initialement vide. Tout entier ajouté est supposé être inférieur ou égal à une valeur maximale fixée à l'initialisation de l'ensemble. Une opération permet l'affichage du graphe d'appartenance à l'ensemble des entiers de 0 à la valeur maximale. Le module figure dans un dossier à part. Il est utilisé par `lidx`. Figurent également dans ce dossier un source de démonstration et son fichier `makefile`	2
uniformité du style, lisibilité, non redondance, absence de nombres magiques	2
complexités satisfaisantes de l'ensemble	2
rapport et soutenance orale, dont justification du choix des structures de données, illustration, descriptif des modules développés et spécification des fonctions, explicitation des limites, problèmes rencontrés lors du développement	5

MÀJ 16-12 Détail des développements attendus.

Foire aux questions

► 1 Il n'y a pas des erreurs dans les lignes de commande que vous donnez ? Des fois il y a des guillemets autour des noms de fichiers, des fois non. Même chose pour les mots des lexiques. Et vous dites à chaque fois que ce sont des chaines de caractères !

Programmez ! et regardez ce que vous récupérez par `argv` ! Pour le premier exemple, c'est `argv[0] == "./lidx"`, `argv[1] == "-"`, `argv[2] == "lex0.txt"`, `argv[3] == "chaque jour"`, `argv[4] == "compte"`, `argv[5] == "-"` et `argv[6] == "lex1.txt"`. Ce ne sont que des chaines de caractères !

► 2 Peut-on utiliser `fscanf` pour lire les mots ?

Non ! `fscanf`, avec le format `"%s"`, ignore les caractères de la catégorie `isspace`. Donc les caractères fin de ligne. Impossible de gérer la numérotation des lignes.

► 3 Peut-on utiliser `fgets` pour lire les mots ? ou sa version étendue, `fgets_x`, proposée dans l'un des exercices qui figure sur le support de cours ?

`fgets` comme `fgets_x` lisent une ligne de texte. Vous aurez donc à *revenir* sur les caractères stockés dans la zone mémoire dont vous avez passé l'adresse de début (`fgets`) ou dont vous récupérez l'adresse de début (`fgets_x`) pour y déceler tous ceux qui sont des catégories `isspace` ou `ispunct` et en déduire les différents mots présents dans la zone. Ce n'est donc pas très efficace. Et pour `fgets`, vous devrez gérer le cas où toute la ligne n'est pas lue d'un coup et que la zone mémoire se termine par un caractère qui n'est pas des catégories `isspace` ou `ispunct`...

► 4 Donc il faut utiliser `fgetc` pour lire les mots ?

Sur les caractères de base, oui...

► 5 Pour aligner verticalement les mots, les « x », les noms des lexiques et les listes des numéros de ligne, il faut utiliser des espaces ?

Non ! Les champs affichés sont séparés par le caractère de tabulation. Si vous voulez les voir alignés, chargez le résultat sous un tableur ; élargissez ensuite les colonnes et/ou faites pivoter les étiquettes des colonnes. Ou utilisez la commande `tabs` ; pour les deux exemples de production donnés plus haut, « `tabs 1,+10,+10,+10,+10` » convient.

► 6 Peut-on mettre l'aide dans un fichier texte et afficher le contenu de ce fichier lorsque l'utilisateur demande de l'aide avec `--help` ?

Non : votre exécutable doit être autosuffisant, donc avec l'aide intégrée.

► 7 Vous m'avez recommandé d'utiliser un fourretout à la place d'une liste simplement chainée (en tout cas, c'est ce que j'ai compris). Ce qui révèle très efficace dans le cas où le lexique a un grand nombre de mots. Cependant, pourquoi le fourretout est meilleur qu'une liste simplement chainée dont les cellules contiennent chacun un mot (sans doublons) et les lignes où ils apparaissent dans le cas d'un lexique à grand nombre mots sachant que le fourretout est aussi une liste simplement chainée ?

Peut-être me suis-je mal exprimé ou alors n'ai-je pas été assez clair.

Le module `holdall` a été introduit avec un objectif précis : stocker des liste d'objets alloués que l'on sera amené à désallouer lorsqu'ils n'auront plus d'utilité. Contrairement aux modules qui implantent « proprement » les TDA introduits dans le cours — `set_mp`, `stack`, `table`, `hashtable`, `bt`, `bst` —, le module `holdall` autorise des opérations « sales » de *modification* des propres objets qu'il stocke ; il le propose *via* les fonctions `holdall_apply` et `holdall_apply_context`, dans lesquelles `fun` peut agir sur les objets pointés puisque son paramètre est de type `void *`. Cela permet en particulier de passer une fonction de désallocation (nommée `rfree` dans les codes qui figurent dans les sources de l'archive `algo2_src.tar.gz`).

Sinon, oui, aux deux fonctions précédentes près, un fourretout n'est pas beaucoup plus qu'une liste dynamique simplement chainée. Et comme une liste dynamique simplement chainée, il ne peut assurer à lui seul des performances correctes dans le cadre du projet. Le performances correctes sont à trouver du côté des tables de hachage et des arbres binaires de recherche AVL. Les fourretouts doivent apparaitre comme une aide efficace à l'utilisation d'une table de hachage : voir les dossiers `algo2_src/cm/9/` et `algo2_src/tp/8/`. Ils devraient pouvoir également l'être pour une utilisation d'un AVL.

► 8 Je ne comprends pas bien ce que sont les noms des fichiers qui correspondent à des lexiques. Ils doivent commencer par un tiret ? ils doivent avoir une extension `.txt` ?

Non pour l'extension. Les noms de fichiers sont libres. Le fichier associé doit toutefois être lu comme un fichier texte.

Non plus pour le tiret. Ce que je veux dire par « les chaines signifiant un nom de fichier sont précédées d'un tiret », c'est que le tiret est l'argument qui précède l'argument correspondant au nom du fichier. Conformément aux exemples et à la réponse à la question 1.

De manière plus générale, le tiret est utilisé dans les commandes que l'on trouve sous Linux pour introduire une option. Une option courte est introduite par un seul tiret. Une option longue (merci GNU) est introduite par deux tirets ; elle est plus lisible, plus facilement mémorisable ; et elle n'a quelques fois pas d'équivalent sous forme courte. Mais la « condamnation » du tiret à jouer les faire-valoir amène à un problème : celui des noms de fichiers qui commencent par un tiret... Ce problème est résolu par l'astuce suivante : il suffit de faire précéder le nom de tout fichier qui commence par un tiret par... un tiret ! Un exemple. La commande

```
$ ./lidx - --help - ./--help --help - -
```

est une demande d'aide. Mais c'est grâce à la troisième occurrence de `--help` : sa première comme sa deuxième désignent le fichier `"--help"`. Un troisième fichier est désigné : `"-"`.

► 9 Et si l'utilisateur donne plusieurs fois la même option sur la ligne de commande avec des valeurs différentes ou même identiques ? Doit-on traiter ce cas comme une erreur ? Sinon, laquelle des valeurs est la bonne ?

La convention usuelle est de ne pas râler et de ne considérer que la dernière valeur. Voici, dans l'abondante littérature sur les options, un extrait qui résume bien la philosophie générale :

> *Options begin with a dash and consist of a single character. GNU-style long options consist of two dashes and a keyword. The keyword can be abbreviated, as long as the abbreviation allows the option to be uniquely identified. If the option takes an argument, then the keyword is either immediately followed by an equals sign (=) and the argument's value, or the keyword and the argument's value are separated by whitespace. If a particular option with a value is given more than once, it is the last value that counts.*

► 10 Pourquoi « 63 » comme longueur limite ? Ça fait « magique » non ?

« Ma » magie vient de la norme C11 :

> *5.2.4.1 TRANSLATION LIMITS*
> *The implementation shall be able to translate and execute at least one program that contains at least one instance of every one of the following limits:*
> — [...]
> — *63 significant initial characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character)*

Voilà. Cela dit, j'ai omis dans l'extrait qui précède l'appel de note de bas de page qui suit « *limits:* ». Je m'empresse donc de rectifier :

> *5.2.4.1 TRANSLATION LIMITS*
> [...] *limits:*[18)]

18) Implementations should avoid imposing fixed translation limits whenever possible.

À vous de voir...

► 11 Je peux utiliser des modules non standard ? vus en L3, ou GNU.

Les seuls modules non standard que vous pouvez utiliser sont ceux qui vous ont été fournis dans le cours Algorithmique 2 et ceux que vous aurez été amené à développer.

► 12 Ai-je le droit de rajouter un champ `tail` à l'implémentation du module `holdall` afin d'insérer en queue et non plus en tête ? Dans l'intérêt d'afficher les mots dans leur ordre d'apparition dans les lexiques.

Non. Mais vous pouvez à cet effet utiliser la nouvelle version du module que je viens de déposer (mardi 10 décembre à 18 h 10). Il vous suffit de définir à la compilation de la macroconstante `HOLDALL_INSERT_TAIL`, et donc d'ajouter l'option `-DHOLDALL_INSERT_TAIL` à la variable `CFLAGS` du fichier `makefile`.

Cette nouvelle version propose également une syntaxe et une sémantique plus évoluées pour la fonction `holdall_apply_context` : la fonction est dorénavant capable de prendre en compte un contexte supplémentaire, comme par exemple une variable de type `FILE *`, laquelle peut être associée au fichier dans lequel des résultats doivent être écrits… Cela doit permettre d'éviter deux écueils dans le projet : l'utilisation d'une variable globale de type `FILE *` ; le détournement de `stdout` par `freopen`.

► 13 Sauf à convertir tous les mots en majuscules, les résultats que j'obtiens ne sont pas triés dans l'ordre lexicographique : les « A… » ne sont pas avec les « a… », même chose pour les « B… » et les « b… », et toutes les autres lettres. J'ai déjà rencontré le même problème en TP avec le tri avec les piles. Il y a quelque chose à faire ?

Par défaut en C, l'ordre lexicographique est celui basé sur le seul ordre des caractères de base : l'ordre ASCII. Le résultat que vous obtenez est donc certainement tout à fait normal.

Si vous voulez quelque chose qui se rapproche de l'ordre lexicographique usuel — celui du dictionnaire donc — il va vous falloir utiliser les fonctions standard `setlocale` et `strcoll`. La première pour aller chercher la couleur locale — `fr_FR.UTF8` sur les machines des salles de TP — avec les paramètres `LC_COLLATE` et `""`. La deuxième à utiliser en lieu et place de `strcmp`.

► 14 Avez-vous des textes à nous proposer sur lesquels on peut tester nos exécutables ?

Vous en trouverez quelques-uns dans l'archive `textes.tar.gz` sur UniversiTICE. Bien évidemment, le poème de Vincensini, `surleboutdesdoigts.txt`, et les deux lexiques `lex0.txt` et `lex1.txt` utilisés dans les exemples. Pour aller plus loin, deux couples de fichiers, `tartuffe.txt` et `tartuffe_.txt`, de tailles moyennes (environ 100 ko), `lesmiserables.txt` et `lesmiserables_.txt`, de tailles plus imposantes (environ 3 Mo), et un lexique, `lex2.txt`. Si le lexique `lex2.txt` n'est pas très gros, il contient des mots des plus communs :

```
*Un, une, des/de/d'.
*Le/L', la/l', les.
```

autrement dit les articles indéfinis et définis de la langue française, avec leurs formes élidées ou contractées. L'emploi de ce lexique doit permettre de tester la bonne gestion des listes de numéros de lignes : longues sur de gros texte ; avec éventuellement plusieurs occurrences d'un même mot sur une même ligne. *Tartuffe* et *Les Misérables* sont deux classiques de la littérature française. Ils sont donnés ici en deux versions : une version ASCII (donc sans accents, sans cédilles, sans voyelles à « e » collé, sans guillemets en chevron…) et une version UTF8 (dans laquelle chaque caractère est codé sur un ou plusieurs octets). Le nom du fichier correspondant à la deuxième forme contient un tiret bas.

En plus des deux exemples introductifs, votre programme doit supporter

```
$ ./lidx -u -S - lex2.txt < tartuffe.txt
$ ./lidx -u -S - lex2.txt < lesmiserables.txt
```

S'il produit un résultat (exact) pour

```
$ ./lidx -u -S - lesmiserables.txt < lesmiserables.txt > out.txt
```

en quelques fractions de secondes, c'est très bien. Sinon, c'est que vous vous y êtes mal pris et que votre programme repose sur des structures de données inadaptées ou implantées de manière peu performantes.

N'utilisez les textes UTF8 que si vous avez développé une fonction de lecture des mots *ad hoc* : il faut nécessairement utiliser les en-têtes standard `<wchar.h>` et `<wctype.h>` pour gérer des caractères étendus, et recourir au type **`wint_t`**, aux macroconstante `WEOF` et `MB_LEN_MAX`, aux fonctions `fgetwc`, `wctob`, `wctomb`, `iswpunct`, `iswspace`, `towlower`, `towupper`… Ne croyez pas que les « petites » fonctions sur les caractères de base conviennent : elle ne peuvent assurer une reconnaissance correcte de la catégorie des caractères (les guillemets en chevron sont bien des symboles de ponctuation !) ; la coupure à une longueur donnée comptée en nombre de caractères de base risque de couper un caractère étendu.

► 15 Pour les guillemets, il ne suffit pas de les rajouter comme symboles de ponctuation ? De plus, elles sont codées sur un octet.

Pour le codage des guillemets sur un octet, vous pensez à l'ISO-8859-1 ? le codage tellement bien pensé qu'il ne dispose pas de « œ » et « Œ » ? Sans œil ? ni nœud ? NI CHEF-D'ŒUVRE ? Et que, s'il dispose de « ÿ », il ne dispose pas de la majuscule associée ? Ou de leurs pauvres améliorations occidentales ISO-8859-15 et WINDOWS-1252 ? On va éviter.

D'autre part, il ne vous appartient pas de décider qui est ou n'est pas `punct` : vous devez en référer, par défaut, à `ispunct` ou, avec les caractères étendus, à `iswpunct`. Point barre.

► 16 Pardon, mais je ne vois pas à quoi doit servir le 3e module : « Développement d'un module spécifique d'ajout d'entiers à un ensemble initialement vide. Tout entier ajouté est supposé être inférieur ou égal à une valeur maximale fixée à l'initialisation de l'ensemble. » Je comprends que le 2nd module va servir à stocker les numéros de ligne dans l'ordre croissant. Mais celui-là, c'est quand on veut donner un nombre maximum de ligne à stocker ?

Non : l'appartenance aux lexiques ! Pour chaque mot appartenant à la réunion des lexiques, il faut coder son appartenance aux différents lexiques. Les noms des lexiques, hormis celui issu des chaines, sont connus : ils figurent sur la ligne de commande. Reste à coder l'appartenance du mot : une information du type 0/1 par lexique suffit. Le nombre maximum d'informations de ce type requises, c'est le nombre de fichiers plus un.

Si cette proposition de codage ne vous convient pas, implantez-en une autre : vous la justifierez dans votre rapport. Mais, de grâce, « dégagez »-la dans un module.

Pour être plus clair encore, le « détail des développements attendus » est là pour ne pas vous prendre par surprise : il est anormal de voir, au bout de trois semestres de codage en C, comme boucle principale d'une fonction `main` d'un projet de lecture de mots et de calcul d'informations sur eux (puis de publication de ces informations) quelque chose comme

```
int c;
while ((c = getchar()) != EOF) {
  et là, j'ai mis tout ce qui concerne la formation des mots ainsi que le
      calcul des informations sur eux. C'est incompréhensible ? Je peux même pas
      imaginer en donner un invariant de boucle ? Ah... Oui, mais ça marche !
      (Et c'est même double indenté comme demandé !)
}
```

Le découpage en modules, la prise de hauteur donc, ça doit être un acquis du semestre d'Algorithmique 2. Montrez-le.

— *Quoi d'autre ?*