TP 2 : Expressions régulières

Exercice 1 : Utilisation de grep

L'utilitaire grep permet de rechercher des occurrences d'une expression régulière dans un fichier.

Une occurrence d'une expression régulière est une suite de caractères formant un mot (au sens mathématique du terme) appartenant au langage de l'expression.

On utilise grep de la façon suivante :

```
grep -E 'expression' nom_fichier
```

Si aucun fichier n'est spécifié, la recherche se fait dans l'entrée standard.

Grep travaille ligne par ligne : une occurrence à cheval sur deux lignes n'est pas considérée.

Par défaut, grep affiche chaque ligne dans laquelle il y a au moins une occurrence.

Si on ajoute l'option `-0`, grep n'affiche que les occurrences

Syntaxe de base :

- concaténation : il suffit de mettre à la suite
- étoile de Kleene : `*`
- union : `|`
- parenthèses : `()`

Ajouts :

- `+` : comme l'étoile mais on interdit d'avoir 0 répétition
- `.` : n'importe quel caractère
- `[a3ju]` : le caractère `a`, `3`, `j` ou `u`
- `[^a3ju]` : n'importe quel caractère sauf `a`, `3`, `j` ou `u`
- `[a-eC-Z]` : une lettre minuscule entre `a` et `e` ou majuscule entre `C` et `Z`
- `\d` ou `[0-9]` : un chiffre
- `\s` : un caractère d'espacement (espace, tabulation, retour à la ligne)
- `\S` : un caractère qui n'est pas un caractère d'espacement
- `\w` : équivalent à `[a-zA-Z0-9_]`
- `\W` : tout ce qui n'est pas un `\w`
- `^` : le début d'une ligne
- `$` : la fin d'une ligne
- `?` : 0 ou 1 occurrence de ce qui précède
- `{n}` : n occurrences de ce qui précède
- `{m, n}` : entre m et n occurrences de ce qui précède
- `{m,}` : au moins m occurrences de ce qui précède
- `{,n}` : au plus n occurrences de ce qui précède

L'utilisation de grep n'est pas à connaître par cœur. Mais c'est un très bon entraînement pour les expressions régulières et c'est un outil très utile dans la vie informatique de tous les jours.

Le fichier `francais.txt` contient un dictionnaire français de 336531 mots, avec flexions (pluriel, féminin, conjugaisons...) à raison d'un mot par ligne. Les accents et cédilles ont été supprimés.

> **1)** Quels sont les mots qui contiennent au moins un « w » et au moins un « q » ?

> **2)** Quels sont les mots qui contiennent au moins 6 fois la lettre « i » ?

> **3)** Quels sont les mots qui contiennent au moins 10 voyelles ?

> **4)** Quels sont les mots qui commencent par un « p » et contiennent au moins 9 voyelles ?

> **5)** Quels sont les mots d'exactement 12 lettres ne contenant ni « a » ni « e » ni « i » ?

> **6)** Quels sont les mots dans lesquels chaque groupe consécutif de 2 lettres contient au moins un « s » ?

Exercice 2 : Grep et tubes

La plupart des utilitaires en ligne de commande prennent leur entrée sur l'entrée standard et émettent leur résultat sur la sortie standard, ou permettent en tout cas de faire cela. Pour grep, par exemple :

- le résultat est émis sur la sortie standard ;
- l'entrée peut être un fichier, mais sera l'entrée standard (par défaut cette entrée standard correspond aux text qui est écrit dans le terminal) si aucun fichier n'est précisé.

L'entrée et la sortie standard peuvent être *redirigés* :

- `grep -E 'q[a-z]*q' > fichier.txt` cherchera des occurrences dans ce qu est écrit dans le terminal et les écrira dans le fichier `fichier.txt`.
- `fichier.txt > grep -E 'q[a-z]*q'` cherchera des occurrence dans `fichier.txt` et les écrira dans dans le terminal.
- `fichier1.txt > grep -E 'q[a-z]*q' > fichier2.txt` cherchera des occurrence dans `fichier1.txt` et les écrira dans `fichier2.txt`.

Imaginons maintenant que l'on souhaite compter le nombre de lignes du fichier `moby_dick.txt` contenant (au moins) une occurrence de `wallow`.

- Un appel à `grep -E 'wallow' moby_dick.txt > tmp.txt` extrait les lignes qui nous intéressent.
- Pour compter ces lignes, on peut ensuite utiliser le programme `wc` (word count) qui peut compter le nombre de lignes dans un fichier : `wc --lines tmp.txt`
- Il faut ensuite supprimer le fichier `tmp.txt` qui ne sert plus à rien.

À la place, on peut effectuer tout ça en une ligne, sans créer de fichier inutile, en redirigeant la sortie standard de `grep` (les lignes ayant au moins une occurrence) vers l'entrée standard de `wc` (pour que les lignes soient comptées)

On utilise pour cela un *tube* (ou *pipe* en anglais) :

```
grep -E 'wallow' moby_dick.txt | wc --lines
```

7) Combien y a-t-il de lignes contenant au moins une occurrence de « the » dans `moby_dick.txt` ?

8) Combien de lignes de `moby_dick.txt` qui contiennent au moins deux occurrences de « the » ?

9) Combien y a-t-il d'occurrences de « the » dans moby_dick.txt ?

10) Que calcule la commande suivante ? `grep -E -o '(\W|^)the(\W|$)' moby_dick.txt | wc --lines`

11) Combien de lignes de ce fichier contiennent au moins deux fois le mot « the » ?

On pourra supposer que le mot « the » n'apparaît jamais deux fois d'affilée.

Pour cet exercice, trois autres utilitaires peuvent être utiles : `uniq`, `sort` et `head`.

12) Donner les séquences maximales de voyelles n'apparaissant qu'une seule fois dans `francais.txt`. Une séquence est maximale si elle n'est ni précédée ni suivie d'une voyelle.

13) Combien y a-t-il de mots distincts de quinze lettres ou plus dans Moby Dick ?

14) Afficher la liste des séquences (distinctes) de deux mots identiques consécutifs (séparés par une espace) dans moby_dick.txt.

La réponse attendue est :

```
Ay Ay
dat dat
had had
that that
```

15) Afficher les mots contenant au moins 8 voyelles et au plus 4 consonnes dans `francais.txt`.

16) Afficher la liste des 15 séquences maximales d'au moins quatre voyelles consécutives les plus courantes dans `francais.txt`, avec pour chaque séquence son nombre d'occurrences.

Exercice 3 : Représentation des expressions régulières en OCaml

On représente les expressions régulières à l'aide du type suivant :

```
type regex =
| Vide
| Epsilon
| Lettre of char
| Somme of regex * regex
| Produit of regex * regex
| Etoile of regex
```

Vous trouverez dans le fichier fourni une fonction `parse : string -> regex` qui prend en entrée une représentation textuelle d'une expression régulière et renvoie l'arbre (ou plutôt un arbre) de type regex correspondant.

- Les caractères `(` et `)` sont interprétés comme des parenthèses déterminant la structure de l'expression régulière et non comme des caractères « normaux ».
- Le caractère `&` est interprété comme le symbole ε.
- Le caractère `#` est interprété comme le symbole $\varnothing$.
- Les espaces sont ignorés.

17) Essayer cette fonction `parse` sur quelques exemples.

18) Écrire une fonction `string_of_regex : regex -> string` qui est la réciproque de `parse`.

Dans un premier temps, vous pouvez utiliser plus de parenthèses que nécessaire.

En question bonus, vous pouvez faire en sorte d'utiliser le bon nombre de parenthèse pour les règles de priorités des opérateurs vues en cours.

19) Écrire une fonction `est_vide : regex -> bool` qui prend en entrée une expression régulière et renvoie un booléen indiquant si le langage associé à l'expression est vide.

20) Écrire une fonction `un_mot : regex -> string` telle que l'appel `un_mot e` renvoie `Some s`, où `s` est un mot (quelconque) du langage de `e` si ce langage est non vide, et `None` si le langage est vide.

`String.make 1 c` renvoie une chaîne de caractère composée uniquement du caractère `c`.

21) Montrer que toute expression régulière est équivalente :
- soit à l'expression $\varnothing$;
- soit à une expression ne faisant pas intervenir le symbole $\varnothing$.

22) Écrire une fonction `extrait_vide : regex -> regex` prenant en entrée une expression régulière e et renvoyant e' équivalente à e telle que e' ne contienne pas le constructeur `Vide`, ou bien soit réduite au constructeur `Vide`.