Professeur: Nastaran FATEMI
Assistant: Christopher MEIER

Labo 1

Bot-Tender: Tokenizer

Introduction

Dans le cadre de leurs études et travaux à l'HEIG-VD, certains étudiants, assistants, et autres dipsomanes insolents sont amenés à fréquenter un établissement (que nous nommerons ici impartialement le Chill-Out) dans le but de décompresser après de dures journées de labeur. Au fil des années, l'espace autogéré a su fournir un service de qualité et a donc commencé à gagner en popularité, si bien que le personnel - pour aussi compétent qu'il soit - se sent désormais quelque peu surchargé. D'autres chuchotent même que certains bénévoles, dans leur grand altruisme, sont contraints de mettre de côté leurs tâches auxiliaires, faisant ainsi le choix de servir les clients au détriment de leur travail, de la préparation de leurs repas, ou encore du lavage de leur voiture, par exemple.

Pour votre plus grand plaisir, vous êtes mandaté par le proéminent Yann M. pour réaliser un petit *chat bot* permettant aux clients de commander leurs produits, de tenir à jour leurs comptes, et d'offrir quelques statistiques intéressantes sur les consommateurs et leurs consommations respectives. Vous voilà particulièrement flatté d'avoir été sélectionné pour pouvoir apporter votre pierre à l'édifice de ce qui sera sans aucun doute le futur de vos braves successeurs, quelle veine ! La tête engourdie par l'euphorie, les yeux légèrement exorbités, vous acceptez fièrement la tâche et donnez de toute votre personne pour fournir un travail de qualité.

Objectifs

Ce laboratoire est la partie 1 d'une série de labos. L'objectif de la partie 1 est de lire des phrases entrées par l'utilisateur en ligne de commande, puis de les normaliser, de contrôler leur orthographe, et de les découper en tokens. En outre, il vous est aussi demandé d'écrire une fonction permettant de calculer le nombre de *clinks* nécessaires pour que tout le monde puisse faire santé à tout le monde lors d'un santé général. La figure 1 montre des exemples d'utilisations.

Indications

Le code source du labo vous est fourni dans un repo git. Ce code contient une pré-implémentation du laboratoire, vous offrant la structure de base de ce que vous allez devoir implémenter.

Les parties suivantes (prochains laboratoires) vont utiliser le même repo (avec de nouveau commit). Afin de simplifier l'intégration des prochaines parties, il est conseillé d'effecturer les operations suivantes : D'abord créer un repo vide (sur Github ou autre). Ensuite, depuis le clone local de votre repo, ajouter une nouvelle `remote` nommée `upstream` [1] qui correspond au repo fourni. Pour finir il suffit de faire un pull et merge depuis `upstream` [2].

1. `git remote add upstream https://git-ext.iict.ch/scala/scala2023-labo-bottender.git`
2. `git pull upstream master`

```
> Je veux 12 bières et 4 croissants, stp.
(je,JE)
(vouloir,VOULOIR)
(12,NUM)
(biere,PRODUCT)
(et,ET)
(4,NUM)
(croissant,PRODUCT)
(svp,SVP)
(EOL,EOL)
====================
```

(a) Tokenisation d'une simple commande

```
> J'aimerais 2 bières stp !
(je,JE)
(vouloir,VOULOIR)
(2,NUM)
(biere,PRODUCT)
(svp,SVP)
(EOL,EOL)
====================
```

(b) Normalisation du terme "aimerais" en "vouloir" et "stp" en "svp"

```
> Bonjour, je suis _Michel !
(bonjour,BONJOUR)
(je,JE)
(etre,ETRE)
(_michel,PSEUDO)
(EOL,EOL)
====================
```

(c) Identification à l'aide d'un pseudonyme (qui doit commencer par "_")

```
> bonuour ja veut 8 bbères et 5 crsoiasabsdts stppp...
(bonjour,BONJOUR)
(je,JE)
(vouloir,VOULOIR)
(8,NUM)
(biere,PRODUCT)
(et,ET)
(5,NUM)
(croissant,PRODUCT)
(svp,SVP)
(EOL,EOL)
====================
```

(d) Correction orthographique, très utile pour les clients alcoolisés

```
> Santé !
Nombre de *clinks* pour un santé de 2 personnes : 1.
Nombre de *clinks* pour un santé de 3 personnes : 3.
Nombre de *clinks* pour un santé de 4 personnes : 6.
Nombre de *clinks* pour un santé de 5 personnes : 10.
Nombre de *clinks* pour un santé de 6 personnes : 15.
```

(e) Compteur de *clinks*

FIGURE 1 – Exemples d'utilisations

— Ce laboratoire est à effectuer **par groupe de 2**. Tout plagiat sera sanctionné par la note de 1.

— Le partie 1 du laboratoire est à rendre pour le Dim. 19.03.2023 à 23h59 sur Cyberlearn, une archive nommée `SCALA_labo1_NOM1_NOM2.zip` contenant les sources de votre projet devra y être déposée.

— Il n'est pas nécessaire de rendre un rapport, un code propre et correctement commenté suffit. Faites cependant attention à bien expliquer votre implémentation.

— Faites en sorte d'éviter la duplication de code.

— Préférez des implémentations récursives de fonctions.

— Préférez l'utilisation de `val` par rapport à `var`.

Description

La série de laboratoires (parties 1 et 2) se base sur le principe d'un compilateur en version très simplifiée. En effet, un compilateur va utiliser un Tokenizer (aussi appelé Lexer) pour lire un code en entrée, un Parser pour construire un arbre afin de parcourir les opérations, un Analyzer pour vérifier la validité des opérations, un TypeChecker afin de vérifier les types des opérations, etc. L'arbre de parsing contient théoriquement des expressions (+, -, ...) et des déclarations (méthodes, classes, ...).

Notre Bot-tender utilisera un Tokenizer pour découper les entrées utilisateurs en token, un Parser pour construire l'arbre syntaxique des tokens, et un Analyzer très simplifié pour évaluer les actions à prendre selon les tokens récupérés. **Dans la partie 1, vous allez développer uniquement le Tokenizer, ainsi que quelques fonctions auxiliaires**.

Voici un bref résumé de la fonction des différents fichiers fournis pour cette partie 1 :

— `MainTokenizer.scala` est le point d'entrée du programme, il lit les entrées utilisateurs et les envoie au `TokenizerService`.

— `Chat/`

  — `Tokens.scala` définit les tokens du programme, ainsi que le type `Token` (qui est un enum). Un token représente un mot valide par rapport au dictionnaire, et est identifié par une valeur de type `Token`.

  — `TokenizerService.scala` reçoit une entrée utilisateur et convertit les chaînes de caractères en tokens après avoir corrigé et normalisé les différents mots de la phrase.

  — `Tokenized.scala` permet de récupérer un à un les tokens séparé par le `TokenizerService`.

— `Utils/`

  — `ClinksCalculator.scala` contiendra les fonctions nécessaires au calcul du nombre de *clinks* pour un nombre `n` de personnes, à savoir une implémentation de la fonction factorielle, ainsi que celle d'une combinaison de `k` éléments parmi `n`.

  — `Dictionary.scala` contient le dictionnaire de l'application, qui sera utilisé pour valider, corriger, et normaliser les mots entrés par l'utilisateur. Il s'agit d'un objet de type `Map` qui contient comme clés des mots définis comme étant valides, et comme valeurs leurs équivalents normalisés (par exemple nous souhaitons normaliser les mots "veux" et "aimerais" en un seul terme qui sera plus tard reconnu par le Parser : "vouloir").

  — `SpellCheckerService.scala` permettra pour un mot donné de trouver le mot syntaxiquement le plus proche dans le dictionnaire, à l'aide de la *distance de Levenshtein*.

Un example de fichier de test vous est également fournit (`BotTenderTokenizerInputSuite.scala`). Il va vous permettre de debugger vos différentes entrées. C'est particulierement utile car l'exécution en mode debug avec VS Code ne permet pas les entrées utilisateurs.

Implémentation

Étape 1, apéritif : le calculateur de *clinks*

Afin de pouvoir calculer le nombre de *clinks* nécessaires dans un santé général pour que tout le monde puisse souhaiter "Santé !" à tout le monde, nous souhaitons calculer une combinaison de `k` éléments parmi `n`, où `k` vaut 2 (car un santé se déroule entre deux personnes) et `n` correspond au nombre de personnes. Ainsi, pour n = 1 personne il y aura 0 clink, pour n = 2 il y en aura 1, pour n = 3 il y en aura 3, pour n = 4 il y en aura 6, etc.

Pour rappel, la formule de la combinaison de `k` éléments parmi `n` est la suivante :

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

À partir de là, il vous est demandé **d'écrire** la fonction factorielle dans le fichier `ClinksCalculator.scala` et d'utiliser cette dernière pour calculer la combinaison de `k` éléments parmi `n`. Vous pourrez tester votre implémentation en entrant "Santé !" dans la console (le code de cette fonctionnalité étant situé dans le fichier `MainTokenizer.scala`).

```
Bienvenue au Chill-Out !

> Santé !

Nombre de *clinks* pour un santé de 2 personnes : 1.
Nombre de *clinks* pour un santé de 3 personnes : 3.
Nombre de *clinks* pour un santé de 4 personnes : 6.
Nombre de *clinks* pour un santé de 5 personnes : 10.
Nombre de *clinks* pour un santé de 6 personnes : 15.
```

Étape 2, soupe de lettres : le correcteur orthographique

Pour rappel, le correcteur orthographique de cette application sera situé dans le fichier `SpellCheckerService.scala` et permettra pour un mot donné de trouver le mot syntaxiquement le plus proche dans le dictionnaire fourni (`Dictionary.scala`) à l'aide de la *distance de Levenshtein*. Par exemple le mot "diète" sera automatiquement corrigé par le mot "bière", alléluia.

La distance de Levenshtein correspond au nombre d'opérations (insertion d'un caractère, suppression d'un caractère, et substitution d'un caractère) nécessaires pour passer d'un mot donné à un autre. Par exemple, la distance entre "diète" et "bières" est de 3 (substitution de 'd' en 'b', substitution de 't' en 'r', et ajout de 's'). Vous trouverez un pseudo-code itératif, ainsi que des explications supplémentaires dans l'article Wikipedia ainsi que dans cette vidéo. Pour cette partie, vous obtiendrez la moitié des points si vous implémentez la version itérative de l'algorithme, tandis que la totalité des points vous seront accordés si vous le transformez en algorithme **fonctionnel récursif**. Vous pourrez valider votre implémentation notamment à l'aide de ce calculateur en ligne.

Pour réaliser le correcteur orthographique, il vous est demandé :

1. d'écrire une fonction permettant de calculer la distance de Levenshtein entre deux chaînes de caractères données (fonction `stringDistance(...)`) ;

2. de rechercher pour un mot mal orthographié donné (`misspelledWord`) l'occurrence la plus proche dans le dictionnaire (fonction `getClosestWordInDictionary(...)`) ; voici une approche possible :
   (a) contrôler que le mot donné n'est pas un nombre ou un pseudonyme (un mot qui commence par `"_"`), dans quel cas il faut simplement le retourner ;
   (b) calculer pour chacun des mots du dictionnaire (les clés de l'objet `Map`) la distance de Levenshtein avec le mot de base `misspelledWord` ;
   (c) retourner le mot normalisé du dictionnaire possédant le score le plus bas (s'il y a plusieurs résultats, nous retournerons le premier arrivé dans l'ordre alphabétique des clés).

Étape 3, plat de résistance : le Tokenizer

Le but du Tokenizer est de découper une phrase donnée en plusieurs mots, de les normaliser, puis de déterminer leurs équivalents en tokens pour les fournir au programme qui les interprétera. Le programme peut donc interpréter uniquement les tokens qui sont définis.

Dans un vrai compilateur, le processus de tokenisation est un processus de bas niveau qui parcourt un à un tous les caractères de l'entrée jusqu'à arriver au prochain espace ou `EOF` / `EOL` (End of file / line - caractère de fin de l'entrée), dans quel cas il se met en pause jusqu'à recevoir un signal `nextToken`. Voici un cas concret :

1. l'utilisateur entre la phrase "bonjour je veux 2 bières svp" (dans cette exemple la ponctuation est volontairement omise) ;
2. la fonction `nextToken()` est appelée par le programme ;
3. le Tokenizer lit un à un les caractères de l'entrée jusqu'à arriver à un espace ou un EOL, puis retourne le token au programme : "bonjour" ;
4. le programme interprète le token "bonjour" et agit en conséquence ;
5. lorsque le token a été correctement traité, la fonction `nextToken()` est à nouveau appelée par le programme et lit le token suivant "je", puis le programme traitera ce token, et ainsi de suite jusqu'à arriver au caractère final `EOL`.

Cette méthode qui procède pas à pas est utilisée par les compilateurs afin d'optimiser le processus et de le rendre plus léger, évitant ainsi de surcharger la mémoire en y stockant tous les tokens d'un seul coup (qui peuvent se chiffrer en millions selon la longueur de certains codes). Dans notre cas, puisque les entrées ne seront jamais suffisamment grandes pour cette surcharge, nous avons approcher le problème avec une solution de plus haut niveau en séparant directement la phrase avec la méthode `split(...)` pour stocker les tokens dans une liste. Dans ce cas précis, la méthode `nextToken()` utilisera un compteur qui permettra de retourner le token suivant dans la liste, et ce jusqu'à arriver en fin de liste (dans quel cas le token `EOL` sera retourné pour signaler au programme la fin du processus).

Les classes `TokenizerService` et `Tokenized` située dans les fichiers éponymes contiennent chacune une méthode à implémenter :

1. `TokenizerService.tokenize` est automatiquement appelé par le `MainTokenizer` et retourne un `Tokenized`. Voici quelques pistes :
   — afin de vous simplifier la tâche, pensez à rapidement éliminer les caractères de ponctuation (`.`, `,`, `!`, `?`, `*`) et à remplacer les apostrophes et espaces multiples par des espaces simples (afin que le `j'` soit interprété comme un `j`, et donc un `je` selon le dictionnaire) ;
   — à un certain moment, il vous faudra rechercher le ou les mots dans le dictionnaire afin de récupérer leurs équivalents normalisés et pouvoir les tokeniser ; si un mot n'existe pas dans le dictionnaire (à savoir, si l'objet `Map` ne contient pas ce mot comme clé), vous le transformerez automatiquement en l'occurrence la plus proche dans le dictionnaire (à l'aide de la méthode `getClosestWordInDictionary(...)` précédemment implémentée) ;

- pour rappel, chaque token sera retourné sous la forme d'un tuple de type `(String, Token)` (`Token` étant le type déclaré dans le fichier `Tokens.scala`) qui contiendra la valeur du mot comme premier élément, ainsi que la valeur du token (un enum) comme deuxième élément. Nous avons fait ce choix d'implémentation afin de pouvoir facilement stocker des nombres et des pseudonymes (par exemple `("28", NUM)` pour le nombre `28`, ou `("_michel", PSEUDO)` pour le pseudo `Michel`).
- toujours dans le but de vous simplifier la tâche, la conversion d'un mot en token se fera à l'aide de conditions ; en pseudo-code, l'algorithme ressemblerait à ceci :
  ```
  Si le mot actuel vaut "hello", alors retourne le tuple ("bonjour", Token.BONJOUR)
  Si le mot actuel vaut "veux", alors retourne le tuple ("vouloir", Token.VOULOIR)
  Si le mot actuel vaut "_michel", alors retourne le tuple ("_michel", Token.PSEUDO)
  ...
  ```

2. `Tokenized.nextToken` qui retourne le prochain tuple `(String, Token)` de l'entrée, ou alors le tuple `("EOL", Tokens.EOL)` si la fin de la ligne a été atteint.

Il est tout à fait possible d'ajouter des méthodes à la classe si vous le jugez nécessaire et/ou plus propre.

Le `MainTokenizer` appelle naïvement la méthode `nextToken()` jusqu'à obtenir le token EOL afin de tester votre implémentation.