

## Labo 2 : Bot-tender Parser

Basé sur l'oeuvre de Miguel Santamaria

### Objectifs

L'objectif de ce laboratoire est d'utiliser le Tokenizer codé dans la partie 1 pour construire un arbre syntaxique de tokens à l'aide d'un Parser, et pour évaluer les actions à prendre selon l'arbre. Il devra en outre être possible de gérer des produits ("bière", "croissant", etc.) et des types/marques associés (bière "Boxer" (défaut), bière "PunkIPA", croissant "maison" (défaut), croissant "Cailler", etc.). Finalement, vous devrez gérer une authentification utilisateur simplifiée, ainsi que les soldes des différents utilisateurs s'étant authentifiés au cours d'une même instance de l'application. (voir figure 1 pour un exemple)

### Indications

Le code source du labo vous est fourni sur Cyberlearn. Celui-ci contient la pré-implémentation du laboratoire 1 (que vous devez remplacer par la vôtre, ou par celle d'un autre groupe), ainsi qu'une pré-implémentation du laboratoire 2, vous offrant la structure de base de ce que vous allez devoir implémenter.

- Ce laboratoire est à effectuer **par groupe de 2**, en gardant les mêmes formations que la partie 1. Tout plagiat sera sanctionné par la note de 1.
- Le laboratoire 2 est à rendre pour le 17.04.2022 à 23h59 sur Cyberlearn, une archive nommée **SCALA\_lab2\_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

À titre de rappel, l'application Bot-Tender (laboratoire 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 2, vous allez développer le Parser et l'Analyzer.**

Dans le cadre de ce laboratoire, il vous est demandé de gérer *au moins* les requêtes suivantes :

- requête d'authentification : authentifie l'utilisateur actuel, nécessaire pour initialiser/connaître son solde et pouvoir opérer des commandes;
  - exemple de saisie : *Bonjour, je suis \_Michel.*;
  - exemple de retour : *Bonjour, Michel!*;

```
Bienvenue au Chill-Out !
> Bonjour, je suis assoiffé !
Eh bien, la chance est de votre côté car nous offrons les meilleures bières de la région !
> J'aimerais commander 3 bières PunkIPAs !
Veuillez d'abord vous identifier.
> Je suis _Michel.
Bonjour, michel !
> Combien coûte 1 bière PunkIPA ?
Cela coûte CHF 3.0.
> J'aimerais connaître mon solde.
Le montant actuel de votre solde est de CHF 30.0.
> Je veux commander 2 bières PunkIPAs et 1 bière Ténébreuse.
Voici donc 2 punkipa et 1 tenebreuse ! Cela coûte CHF 10.0 et votre nouveau solde est de CHF 20.0.
> Je voudrais commander 1 croissant.
Voici donc 1 croissant maison ! Cela coûte CHF 2.0 et votre nouveau solde est de CHF 18.0.
> Bonjour ! Je suis _Bobby.
Bonjour, bobby !
> Je suis affamé !
Pas de soucis ! Nous pouvons vous offrir des croissants faits maisons !
> Je veux commander 2 croissants cailler.
Voici donc 2 croissant cailler ! Cela coûte CHF 4.0 et votre nouveau solde est de CHF 26.0.
> Je veux connaître mon solde.
Le montant actuel de votre solde est de CHF 26.0.
> Je suis _Michel.
Bonjour, michel !
> Je veux connaître mon solde.
Le montant actuel de votre solde est de CHF 18.0.
> J'aimerais commander 18 bières Farmer.
Voici donc 18 farmer ! Cela coûte CHF 18.0 et votre nouveau solde est de CHF 0.0.
> 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.
> Quitter
Adieu.
```

FIGURE 1 – Exemple d'exécution typique

- requête de commande (**nécessite que l'utilisateur soit authentifié**) : opère une commande ;
  - exemples de saisie : *J'aimerais commander 2 bières Farmers et 1 bière Jackhammer.* ou *Je veux commander 1 bière Farmer ou 1 bière PunkIPA.* ;
  - exemple de retour : *Voici donc 2 Farmers et 1 Jackhammer ! Cela coûte CHF 5.00 et votre nouveau solde est de CHF 25.0.*
- requête de prix : opère une demande de prix pour un produit donné ;
  - exemples de saisie : *Quel est le prix de 2 bières Boxers ?* ou *Combien coûtent 4 bières Farmers et 6 croissants ?* ;
  - exemple de retour : *Cela coûte CHF 2.0.*
- requête de solde (**nécessite que l'utilisateur soit authentifié**) : demande le montant du solde courant de l'utilisateur ;
  - exemple de saisie : *J'aimerais connaître mon solde.* ;
  - exemple de retour : *Le montant actuel de votre solde est de CHF 25.0.*

Les retours affichés en console sont libres et peuvent donc différer des exemples donnés ci-dessus, tant qu'ils restent claires et explicites pour l'utilisateur.

Vous trouverez de plus dans le code de base un exemple fonctionnel permettant de refléter et d'interpréter les plus grands états d'âme de l'utilisateur :

```
Bienvenue au Chill-Out !  
> Je suis assoiffé !  
Eh bien, la chance est de votre côté car nous offrons les meilleures bières de la région !  
> Je suis affamé !  
Pas de soucis ! Nous pouvons vous offrir des croissants faits maisons !
```

## Grammaire

Voici la grammaire complète liée à ces requêtes que vous allez implémenter :

- Pseudo := 'identifiant qui commence par \_'
- Nombre := 'nombre entier'
- Produit := "croissant" | "bière"
- Marque := "maison" | "cailler" | "farmer" | "boxer" | "wittekop" | "punkipa" | "jackhammer" | "ténébreuse"
- Politesse := "je" ("aimerais" | "veux" | "voudrais")
- EtatAme := ("je" "suis" "affamé") | ("je" "suis" "assoiffé")
- Identification := "je" ("suis" | "me" "appelle") Pseudo
- Produits := Nombre Produit [Marque] {"et" | "ou"} Nombre Produit [Marque]}
- Commande := Politesse "commander" Produits
- Solde := Politesse "connaître" "mon" "solde"
- Prix := ("combien" ("coûte" | "coûtent")) | ("quel" "est" "le" "prix" "de") Produits
- Phrase := ["bonjour"] (EtatAme | Identification | Commande | Solde | Prix)

Lorsque nous avons affaire à un "ou" dans une phrase, la logique utilisée sera de toujours retourner l'option dont le prix est le moins élevé. Ainsi, lorsqu'un utilisateur commande ou demande par exemple le prix de 2 Boxers ou 4 Ténébreuses, l'application choisira la première option car c'est la moins chère, à savoir les Boxers.

Les séquences de ET et OU ont une associativité à gauche. Voici un exemple d'interprétation d'une séquence ET/OU pour calculer le prix d'une commande :

```
((((punkipa et boxer) ou farmer) ou tenebreuse) et boxer) => (((3 et 1) ou 1) ou 4) et 1)
(((punkipa + boxer ) ou farmer) ou tenebreuse) et boxer) => ((( 4 ) ou 1) ou 4) et 1)
((( farmer ) ou tenebreuse) et boxer) => (( 1 ) ou 4) et 1)
(( farmer ) et boxer) => (( 1 ) et 1)
( farmer + boxer ) => ( 2 )
```

## Structure

Voici un bref résumé de la fonction de tous les fichiers fournis pour cette partie 2 (les packages et fichiers en vert étant ceux qui ont été **modifiés** ou **ajoutés** par rapport à la partie 1) :

- **MainParser.scala** est le point d'entrée du programme, il lit les entrées utilisateurs, les tokenize, les envoie au Parser puis à l'Analyzer et affiche finalement les résultats.
- **MainTokenizer.scala** est l'ancien 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**.
  - **Tree.scala** définit les nœuds et les feuilles de l'arbre syntaxique.
  - **AnalyzerService.scala** contient la logique de l'application et retourne la bonne réponse selon le nœud ou la feuille fourni.
  - **Parser.scala** traite les tokens par ordre d'apparition à l'aide de la méthode **nextToken** de **Tokenized**, puis construit un arbre syntaxique pour traiter les requêtes à l'aide du fichier **Tree.scala**.
- **Data** (ce package contient des fichiers liés aux données de l'application, puisque pour des raisons de facilité nous n'utilisons pas de base de données dans cette partie du laboratoire)
  - **ProductService.scala** contient liste des produits disponibles, ainsi que leurs types/marque et leurs prix. Chaque produit possède un type/marque par défaut qui sera géré.
  - **AccountService.scala** contient la liste des utilisateurs enregistrés dans l'application et de leurs soldes courants.
  - **SessionService.scala** permet de connaître l'utilisateur actuellement connecté (s'il y en a un). Dans ce labo, il n'y a qu'une seule session par execution. Par contre dans un prochain labo, il sera possible d'avoir plusieurs sessions par execution.
- **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*.

## Implémentation

### Étape 0 : intégration du dernier labo

Cette étape initiale consiste à fusionner le code du labo 1 (le votre ou celui d'un autre groupe) avec la pré-implémentation du labo 2.

### Étape 1 : mise à jour des tokens

Cette première étape consiste comme son nom l'indique à mettre à jour les différents tokens dans le fichier `Token.scala` afin que le Parser puisse traiter toutes les différentes possibilités de la grammaire, puis de mettre à jour en conséquence le dictionnaire et le Tokenizer afin de retourner les bons tuples (`String`, `Token`).

### Étape 2 : complétion des classes annexes

Cette étape a pour but de modéliser et d'implémenter les données de l'application, situées dans le package `Data` :

- `ProductService.scala` : l'implémentation de ce trait va contenir un attribut qui va permettre à l'application d'accéder à la liste des produits, leurs types/marques, et leurs prix *de la manière la plus aisée et la plus optimisée possible*. Il vous est aussi demandé de trouver un moyen de gérer les types/marques par défaut. Voici la sélection de produits qui devront obligatoirement être gérés :
  - Bières :
    - Boxer (*par défaut*) : CHF 1.00
    - Farmer : CHF 1.00
    - Wittekop : CHF 2.00
    - PunkIPA : CHF 3.00
    - Jackhammer : CHF 3.00
    - Ténébreuse : CHF 4.00
  - Croissants :
    - Maison (*par défaut*) : CHF 2.00
    - Cailler : CHF 2.00
- `AccountService.scala` : l'implémentation de ce trait va permettre de gérer les comptes utilisateurs existants ainsi que leurs solde ; vous devez en outre implémenter ici :
  1. un attribut `accounts` qui contiendra la liste de tous les utilisateurs qui se sont connectés durant l'instance actuelle de l'application, ainsi que leur solde courant (CHF 30.00 par défaut), tout ceci *de la manière la plus optimisée possible* ;
  2. la méthode `purchase` qui permet de soustraire le montant donné du compte de l'utilisateur donné.

### Étape 3 : construction de l'arbre/Analyzer

Le fichier `Tree.scala` contiendra les différents noeuds et feuilles de l'arbre syntaxique qui sera construit par le Parser. En vous basant sur l'implémentation des états d'âme (assoiffé/affamé) qui vous ont été fournis dans ce fichier, ajoutez les noeuds manquants. Voici une idée des types de noeud que vous pourriez implémenter : un noeud pour chaque requête (identification, commande, etc.), un noeud "et" et un noeud "ou" (pour les requêtes de produits multiples), et un noeud représentant un produit et son type / sa marque.

Le fichier `AnalyzerService.scala` contient la logique de l'application (calculs, authentification, etc.). La class

`AnalyzerService` contient deux méthodes qui permettent respectivement de calculer le retour d'un noeud (`computePrice`), et de retourner le texte de sortie à écrire en console (`reply`) selon le noeud. La première méthode s'occupera de calculer un prix lorsque le noeud représente une commande d'un certain nombre d'**un** produit, une requête de prix, un "et", et un "ou", tandis que la seconde méthode retournera un string, appellera ou non la méthode `computePrice` selon le type du noeud traité et, si besoin, mettra à jour la session et/ou le compte (les noms d'utilisateur contenu dans la session et le compte ne doivent pas être préfixé d'un "\_").

Pour rappel, la session (dont l'implémentation vous est fournie dans le fichier `SessionService.scala`) permet de spécifier quel utilisateur est actuellement connecté.

À partir de là, et en vous basant sur la théorie vue en cours, nous vous laissons le soin d'implémenter les fichiers `Tree.scala` et `AnalyzerService.scala`. Prenez en compte le fait que certains noeuds possèdent des valeurs et/ou des enfants.

## Étape 4 : implémentation du Parser

Toujours en vous basant sur l'exemple fourni, observez et comprenez le contenu du fichier `Parser.scala`. Cette classe s'occupe de "manger" les différents tokens (à savoir, lire un token, le traiter, et passer au tokens suivant) dans l'ordre donné, puis de créer la ou les bonnes feuilles de l'arbre (depuis le fichier `Tree.scala`). Une instance de cette classe est appelée depuis le Main, et le parsing est effectué grâce à sa méthode `parsePhrase`. L'object `ExprTree` retourné est ensuite passé à la méthode `reply` de l'analyser.

Une fois l'exemple étudié et compris, complétez la suite de ce fichier, puis testez finalement votre application! Prêtez une attention toute particulière à la factorisation de votre code dans cette étape, en le séparant notamment en fonctions.