SCALA 2021

Professeur: Nastaran Fatemi
Assistant: Christopher Meier

Labo 2 : Bot-tender (partie 2)

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 example)

Indications

Le code source du labo vous est fourni sur Cyberlearn. Celui-ci contient l'implémentation du laboratoire 1 (que vous pouvez remplacer à votre guise par la vôtre), 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 26.04.2020 à 23h59 sur Cyberlearn, une archive nommée `SCALA_-labo2_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 l'utilisation de `match case` si nécessaire.
- 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-tenter 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
- Commande := Politesse "commander" Nombre Produit [Marque] {("et" | "ou") Nombre Produit [Marque]}
- Solde := Politesse "connaître" "mon" "solde"
- Prix := ("combien" ("coûte" | "coûtent")) | ("quel" "est" "le" "prix" "de") Nombre Produit [Marque]{("et" | "ou") Nombre Produit [Marque]}
- 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 :

```
((((boxer et punkipa) ou farmer) ou tenebreuse) et boxer) => ((((1 et 1) ou 1) ou 4) et 1)
((((boxer + punkipa ) ou farmer) ou tenebreuse) et boxer) => ((((   2  ) 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 italique étant ceux qui ont été *modifiés* ou *ajoutés* par rapport à la partie 1) :

- *Main.scala* est le point d'entrée du programme, il lit les entrées utilisateurs, les tokenize, les envoie au Parser.scala et affiche finalement les résultats.
- *Chat*
  - Tokens.scala définit les tokens du programme, ainsi que le type Token (qui est un alias à Int). Un token représente un mot valide par rapport au dictionnaire, et est identifié par une valeur de type Token.
  - Tokenizer.scala reçoit une entrée utilisateur et convertit les chaînes de caractères en tokens après avoir corrigé et normalisé les différents mots de la phrase.
  - *Tree.scala* définit les nœuds et les feuilles de l'arbre syntaxique. Il s'agit ici aussi de la classe qui occupe le rôle d'Analyzer, car c'est en effet elle qui contiendra la logique de l'application et qui va aussi afficher la bonne sortie en console selon le noeud ou la feuille traité.
  - *Parser.scala* traite les tokens par ordre d'apparition à l'aide de la méthode nextToken du Tokenizer, puis construit un arbre syntaxique pour traiter les requêtes à l'aide du fichier Trees.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 le cadre de ce laboratoire)
  - *Products.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.
  - *UsersInfo.scala* contient l'identifiant de l'utilisateur actuellement connecté, ainsi que la liste des utilisateurs enregistrés dans l'application et de leurs soldes courants.
- Utils
  - ClinksCalculator.scala contient 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").
  - SpellChecker.scala permet 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 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 `Tokens.scala` afin que le Parser puisse traiter toutes les différentes possibilité de la grammaire, puis de mettre à jour en conséquence le dictionnaire et le Tokenizer afin de retourner les bons tuples `(String, Token)`. Si vous reprenez le code que vous avez réalisé dans la partie 1, prêtez attention à y ajouter les tokens `AFFAME` et `ASSOIFFE` qui servent dans l'exemple fourni avant d'exécuter celui-ci.

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` :

- `Products.scala` : cet objet 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. La modélisation vous est laissée totalement libre, mais voici cependant une 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
- `UsersInfo.scala` : cet objet contient déjà le nom de l'utilisateur actuellement actif comme attribut ; vous devez ou outre implémenter ici :
  1. un deuxième 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*. Vous aurez sans doute à rajouter des méthodes d'accès et d'écriture aux attributs. La modélisation vous est laissée totalement libre ;
  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. C'est aussi ici que se trouve la logique de l'application (calculs, authentification, etc.). En observant l'implémentation des états d'âme (assoiffé/affamé) qui vous ont été fournis dans ce fichier, vous remarquerez tout d'abord que le `trait ExprTree` représente un noeud de l'arbre, et que les différents types de noeuds sont déclarés à la fin de l'objet. 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 `trait ExprTree` contient en outre 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 second méthode retournera un string qui appellera ou non la méthode `computePrice` selon le type du noeud traité.

À partir de là, et en vous basant ur la théorie vue en cours, nous vous laissons le soin d'implémenter le fichier `Tree.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 d'appeler 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`. C'est ici que la méthode `reply` de l'objet `ExprTree` retourné sera appelée. Le `Parser` ne doit pas gérer les erreurs de parsing qui est en soit une tâche complexe.

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.