HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Professeur : Nastaran Fatemi
Assistant : Miguel Santamaria

Labo 2 : Calculatrice version 2.0

Objectifs

L'objectif de ce laboratoire est de développer une calculatrice en console, afin de mettre en pratique les notions avancées de la programmation en Scala.

Indications

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

- Ce laboratoire est à effectuer **seul ou par groupe de 2 ou 3**, en gardant les formations utilisées dans le Labo 1. Tout plagiat sera sanctionné par la note de `1`.
- Le laboratoire est à rendre pour le `25.04.2018` à `12h00` sur Cyberlearn, une archive nommée `SCALA_-labo2_NOM1_NOM2.zip (7z, rar, ...)` contenant les sources de votre projet devra y être déposé.
- 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.

Descriptions

Dans ce labo, vous allez utiliser toutes les opérations que vous avez déjà implémenté dans le labo précédent (`+`, `-`, `*`, `/`, `%` (modulo), `^` (puissance), factoriel `!`, etc), y compris `GCD`, etc.

Cette nouvelle version de la calculatrice se base sur le principe d'un compilateur en version très simplifiée. En effet, un compilateur va utiliser un `Tokenizer (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, ...).

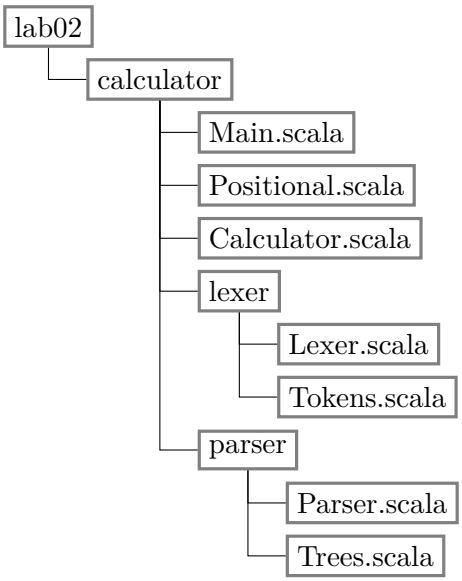
Notre calculatrice n'utilisera que des expressions, un `Tokenizer` (`Lexer`) pour lire les opérations, un `Parser` pour construire l'arbre syntaxique et un `Analyzer` très simplifié pour évaluer les opérations, afin de retourner le résultat de chaque opération.

Pour faire simple :

- `Main.scala` est le point d'entrée du programme, lit les entrées utilisateurs et les envoie au `Calculator.scala`.
- `Positional.scala` permet d'attacher une position aux `Tokens`.
- `Calculator.scala` est le point d'entrée de la calculatrice, reçoit l'entrée utilisateur, et appelle le `Parser`, qui appellera le `Lexer`.
- Le `Lexer` :
  - `Tokens.scala` définit les `Tokens` du programme. Un `Token` représente un symbole valide du programme, comme un nombre, une variable, un opérateur, un mot clé du langage (`GCD`, ...), etc.
  - `Lexer.scala` lit l'entrée utilisateur et convertit les chaînes de caractères en `Tokens`.
- Le `Parser` :

— `Trees.scala` définit les nœuds et les feuilles de l'arbre syntaxique. Dans la deuxième partie du labo, vous créerez les nœuds et feuilles pour simplement afficher l'arbre. Dans la troisième partie, vous remplacerez l'affichage par l'évaluation des opérations.

— `Parser.scala` est le point central de la calculatrice. C'est lui qui va traiter les `Tokens` afin de pouvoir construire l'arbre puis traiter les opérations. Il faudra tenir compte de la priorité des opérateurs. La priorité ajoutée par les parenthèses vous est déjà fournie.



Implémentation

Etape 1 : Le `Tokenizer`

Le `Tokenizer` permet de découper le fichier source (en l'occurrence, les entrées utilisateurs), pour les fournir au programme sous forme d'objets appelés `Tokens`. Le programme peut donc interpréter uniquement les `Tokens` que vous aurez défini.
Les `Tokens` ne représentent pas uniquement les opérateurs (`+`, `-`, `*`, `/`, `%`, `^`, `!`, etc), mais également les symboles de contrôles (parenthèses, accolades, virgules, etc), ainsi que les mots clés du langage (`GCD( , )`, `EGCD( , )`, etc), concrètement, tout ce que vous n'avez pas besoin pour faire les calculs, et qui ne doit pas retourner d'erreurs.
Il y a deux sortes de `Tokens`, ceux qui n'ont pas de paramètre, et ceux qui en ont un.

— Ceux qui n'ont pas de paramètre s'implémentent tous de la même manière. Les mots clés du langage n'ont pas de paramètre, car leurs "paramètres" se trouvent entre les `Tokens` parenthèses (et parfois séparé par une virgule).

— Pour ceux qui ont un paramètre, comme par exemple, le `Token` représentant les nombres, il est plus judicieux de garder `NUM(valeur)` que simplement son type `NUM`, il en va de même pour les variables (identifiants `ID(value)` contre simplement son type `ID`). Pour ceci, vous pouvez surcharger la méthode `toString`.

Une fois tous les `Tokens` du langage défini, il faut implémenter le `Tokenizer` qui se trouve dans le fichier `lexer.scala`.

Dans le `Tokenizer`, la fonction `nextToken` est automatiquement appelée par des fonctions fournies, permettant de naviguer dans les entrées utilisateurs et d'y associer le `Token` correspondant. Complétez le `match-case` pour traduire les entrées utilisateurs en `Tokens`. Par exemple, si le symbole correspond à `(`, il faut y associer le `Token` `LPAREN` à l'aide de la fonction `setToken`.

Traitement des nombres et des mots clés du langage

Pour simplifier le problème, nous vous demandons de traiter au moins les nombres entiers ; un bonus sera collecté si vous traitez des nombres à virgule flottants. Un nombre entier valide commence par `[1-9]`, puis, peut être suivi par un nombre indéfini de chiffres `[0-9]` ; un nombre à virgule flottant peut contenir **au maximum** un point. On considère qu'une variable (ou un mot clé du langage) est valable uniquement si son premier caractère est une lettre, suivi de caractères alphanumériques.

- Si le caractère correspond à `0`, c'est le chiffre 0.
- Si le caractère correspond à un chiffre différent de `0`, il faut lire jusqu'au prochain espace ou `EOF` pour récupérer le nombre.
- Si le caractère est alphanumérique, il faut lire jusqu'au prochain espace ou `EOF` pour récupérer la chaîne de caractères alphanumériques complète. Cette chaîne représente soit une variable (pour la mémoire), soit un mot clé du langage. La fonction `keywordOrID` vous permet de facilement traiter les chaînes de caractères alphanumériques et d'y associer le `Token` correspondant, par exemple, si la chaîne est "egcd" ou "EGCD", il faut y associer le `Token` EGCD, mais si la chaîne est "asd0923da", il faut y associer le `Token ID` qui représente un identifiant (ou variable).
- Dans tous les autres cas, c'est une erreur (la fonction `fatalError` est disponible, ainsi que le `Token BAD`).

Etape 2 : Le `Parser`

Le `Parser` permet de faire le lien entre les `Tokens` et les opérations correspondantes. Il va les traduire en un arbre syntaxique qui représente la priorité des opérateurs. Ceci permettra, dans un premier temps, de pouvoir vérifier ce qui a été implémenté jusqu'ici, en affichant l'arbre à l'aide de la fonction `printTree`, puis dans un deuxième temps, d'implémenter les opérations réelles (principalement les fonctions du labo précédent, sauf `prime` et `quadratic solve`), afin de calculer les résultats de chaque opération à l'aide de la fonction `computeSource`.

Il faut donc commencer par implémenter les nœuds de l'arbre (`case class`) du fichier `Trees.scala`, au même titre que les `Tokens` pour le `Tokenizer`, les nœuds seront utilisés par le `Parser` pour construire l'arbre. Il existe deux feuilles possibles, qui vous sont déjà fournies, il s'agit des nombres et des identifiants. Nous vous fournissons également l'affectation mémoire (affectation de variable), et l'opération `+`, afin que vous compreniez plus facilement la structure et les paramètres de chaque nœud.
Ne vous occupez pas encore de la fonction `compute`. Elle permettra de traduire l'arbre en opération mathématique afin de calculer les résultats de chaque entrée utilisateur.

Il faut désormais implémenter le `Parser`. La fonction `parseExpr` est appelée automatiquement, elle va permettre de parcourir de manière récursive tous les `Tokens` afin de les traiter par ordre de priorité. Lorsqu'un `Token` est identifié, on le supprime de la liste, et on ajoute le nœud correspondant à l'arbre syntaxique. On traite ensuite le sous-arbre de gauche et le sous-arbre de droite de manière récursive. Il faut également tenir compte de la priorité des opérateurs. Du plus prioritaire au moins prioritaire :

- `parseSimpleExpr` : un nombre, une variable, un mot clé, des parenthèses, etc.
- L'opérateur Factoriel.
- L'opérateur Puissance.
- L'opérateur Modulo.
- Les opérateurs Multiplication et Division
- Les opérateurs Additions et Soustraction.
- `parseEquals` : l'opérateur Affectation

Il y a certains opérateurs qui ont la même priorité que d'autres. Afin que tout le monde ait le même ordre de priorité de tous les opérateurs, nous imposons une associativité à gauche.

La méthode `parseEquals` est la seule qui est implémentée complètement. Vous allez implémenter le traitement des autres `Tokens` en tenant compte de la priorité des opérateurs. Vu que les appels sont récursifs, vous allez implémenter d'abord les opérateurs les moins prioritaires, en partant de la fonction `parseEquals`, il faudra appeler la fonction pour l'addition et la soustraction, qui appellera la fonction pour la multiplication et la division, et ainsi de suite, jusqu'à `parseSimpleExpr` qui permettra de traiter des nœuds particuliers (les parenthèses, les nombres négatifs ou les mots clés du langage) ainsi que les feuilles de l'arbre.

La fonction `expected` permet de lever une erreur afin de vous aider à déboguer votre programme. Elle prend un nombre indéfini de paramètres, qui représentent les `Tokens` attendu par le programme à ce moment là.

Vous pouvez maintenant implémenter la fonction `execute` du fichier `Calculator.scala` pour appeler la fonction `printTree` du fichier `Parser.scala` (l'import est déjà effectué), pour afficher l'arbre syntaxique. Exécutez le programme afin de vérifier que (1 + 3^4!) * 5 - 2 % 6 donne bel et bien l'arbre suivant :
`Minus(Times(Plus(NumLit(1),Pow(NumLit(3),Fact(NumLit(4)))),NumLit(5)),Mod(NumLit(2),NumLit(6)))`

Etape 3 : L'évaluation des opérations

Il ne reste plus qu'à traduire l'arbre des opérations en calcul mathématique. Pour ceci, il faut implémenter la fonction `compute` du fichier `Trees.scala`. Il faut parcourir toutes les feuilles et nœuds de l'arbre et agir en conséquence. Par exemple, si l'on rencontre la feuille représentant un nombre, il faut retourner la valeur du nombre (on retourne un Int, Double, etc, tout simplement).
Lorsqu'il s'agit d'un nœud, par exemple, l'opération `+`, il faut retourner le calcul récursif de la partie gauche de l'opération additionné au calcul récursif de la partie droite de l'opération. Reprenez également toutes les fonctions que vous avez implémenté dans le labo précédent (sauf `prime` et `quadratic solve`).
Il faut tenir compte de plusieurs éléments :

- Le retour d'une affectation de variable.
- Dans les autres cas, retourner la valeur de l'opération, évaluée de manière récursive.
- Récupérer les variables de la mémoire lorsque c'est nécessaire.
- La division par 0 (Exception).
- La racine carrée n'est pas valide pour un nombre négatif (Exception).

Finalement, modifiez la fonction `execute` du fichier `Calculator.scala` pour appeler la fonction `computeSource`. Vous pouvez utiliser des constantes pour définir une réponse, par exemple :

```
def execute(): Unit = computeSource match {
    case Double.NegativeInfinity => println("Memory updated !")
    case result => println("Result : " + result)
  }
}
```