Youssef AGHZERE – Mohamed CHERGUI

Rapport TP01 – Complexes

Ce TP représente le lancement dans les TPs de Scala. En effet, on a pu découvrir l'environnement Scala en allant de l'installation de Scala sur Intellij et passant par la compréhension du schéma d'un projet Scala qui est constitué en principe des fichiers de classes Scala (fichiers .scala), et des worksheets Scala (fichiers .sc) pour faire des tests au fur et à mesure, et jusqu'à arriver à faire des implémentations et des classes. Dans ce cadre, nous étions amenés à appliquer ce que nous avions appris comme base du langage, à savoir les constructeurs, les variables, les méthodes de base telles que toString() et la surcharge de différents opérateurs (unaires et binaires) qui permettent la manipulation des nombres complexes.

Nous avons pu prendre petit à petit le langage en s'inspirant de différentes sources que ce soient les diapositives du cours ou les forums en ligne.

Par contre, nous avons rencontré quelques problèmes de respect des quelques normes de Scala. Les deux principaux étaient la non-utilisation des points-virgules ' ; ' et de ' return '. Néanmoins, on a essayé de relire notre code à chaque fois pour vérifier et donc pouvoir adopter un certain reflexe dans ce sens. Un problème qui n'était pas fréquent est l'utilisation des boucles d'itération. Certes elle n'était pas demandée dans ce TP, mais nous avons pensé à cette contrainte le long du TP et commencer à s'habituer. Sinon, des difficultés du genre l'écriture de quelques types de base nous a gêné un peu comme par exemple Int au lieu de 'int ...

Rapport TP02 – Ensembles de fonctions

Le TP "**Ensembles fonctions**" a pour objectif de manipuler des ensembles d'entiers au sens mathématique en utilisant des fonctions de type "*Int => Boolean*". Ces fonctions permettent de déterminer si un entier fait partie de l'ensemble ou non.

Le code fourni définit plusieurs fonctions pour atteindre cet objectif : La fonction "*singleton()*" permet de créer un singleton, c'est-à-dire un ensemble contenant un seul élément. La fonction "*union()*" permet de renvoyer l'union de deux ensembles, tandis que la fonction "intersection" permet de renvoyer l'intersection de deux ensembles. La fonction "*difference()"* permet de renvoyer la différence entre deux ensembles, tandis que la fonction "*complement()*" permet de renvoyer le complément d'un ensemble.

La fonction "*filtrer()*" permet de filtrer un ensemble selon un prédicat donné, tandis que la fonction "*pourTout()*" permet de vérifier si un prédicat est vrai pour tous les éléments d'un ensemble. La fonction "*ilExiste()"* permet de vérifier qu'il existe au moins un élément de l'ensemble qui vérifie un prédicat. Enfin, la fonction "image()" permet de renvoyer l'ensemble image par une fonction d'un autre ensemble.

Le code fourni inclut également plusieurs fonctions utilitaires, telles que "*contient()*" qui indique si un ensemble contient un élément donné, "*chaine()*" qui renvoie un ensemble sous forme de chaîne de caractères(en utilisant la fonction *mkString()*), et "*afficheEnsemble()*" qui affiche un ensemble à l'écran.

En résumé, le TP "Ensembles fonctions" nous a permis de manipuler des ensembles d'entiers de manière simple et efficace grâce à l'utilisation de fonctions de type "*Int => Boolean*". Nous avons également vu plusieurs opérations couramment utilisées en mathématiques sur ces ensembles, telles que l'union, l'intersection, la différence et le complément. Nous avons également appris à filtrer un ensemble et à vérifier si un prédicat est vrai pour tous ou au moins un élément d'un ensemble. **Du point de vue du langage Scala**, nous avons travaillé avec des fonctions de type "*Int => Boolean*" et défini des fonctions récursives. Nous avons également utilisé la syntaxe du "collections avec for" de Scala pour créer une liste en itérant sur les entiers compris entre -limite et +limite.

Rapport TP03 – Listes

Dans ce TP, nous avons été amenés à créer des fonctions de base pour manipuler des listes en Scala en utilisant que les méthodes (::, head, tail, isEmpty) de la classe List (éventiellement avec de récursivité terminal c'est si possible).

Nous avons commencé par définir la fonction "*dernier()*", qui renvoie le dernier élément d'une liste s'il existe, ou génère une exception NoSuchElementException sinon. Nous avons également implémenté la fonction "*kieme()*", qui renvoie l'élément d'indice k d'une liste s'il existe, ou génère une exception IndexOutOfBoundsException sinon.

Ensuite, nous avons défini la fonction "*taille()*", qui renvoie le nombre d'éléments dans une liste, et la fonction "*contient()*", qui vérifie si un élément donné est présent dans une liste. Nous avons également implémenté la fonction *"supprimerKieme() / ajouterKieme()"*, qui renvoie une nouvelle liste en supprimant/ajoutant l'élément d'indice k de la liste d'origine.

Nous avons également défini la fonction *"identique()"*, qui vérifie l'égalité de deux listes, ainsi que la fonction *"filtrer()"*, qui renvoie une nouvelle liste en ne gardant que les éléments qui satisfont un prédicat donné. Enfin, nous avons implémenté la fonction image, qui renvoie une nouvelle liste image par une fonction donnée de la liste courant.

Dans ce TP, nous avons utilisé plusieurs méthodes de la classe List de Scala, telles que ::, head, tail et isEmpty. Nous avons également utilisé les exceptions NoSuchElementException et IndexOutOfBoundsException pour gérer les cas d'erreur dans nos fonctions. Enfin, nous avons utilisé la récursion terminal pour implémenter la plupart de nos fonctions, ce qui nous a permis de parcourir et de manipuler les éléments d'une liste de manière itérative.

Dans ce TP, nous avons appris à utiliser la récursion terminal pour implémenter des fonctions de base pour manipuler des listes en Scala sans avoir besoin d'utiliser des varible mutable et les boucles. Nous avons également appris à utiliser certaines méthodes de la classe List, telles que ::, head, tail et isEmpty, ainsi que les exceptions NoSuchElementException et IndexOutOfBoundsException pour gérer les erreurs. Enfin, nous avons appris à créer des fonctions pour effectuer des opérations courantes sur les listes, telles que la récupération d'un élément spécifique, la suppression d'un

élément, l'ajout d'un élément, la vérification de l'égalité de deux listes et le filtrage d'éléments selon un prédicat.

Malheureusement lorsqu'on a géré les exceptions on n'a pas respecté les principes de programmation fonctionnelle, précisément l'utilisation des fonctions à effet de bord.

Pour terminer, Grâce à ce TP, nous avons compris comment à partir de quelques méthodes de base on peut construire tout un concept et des méthodes correspondant (en l'occurence, le concept des listes).

Rapport TP04 – Parenthèses

Dans ce TP, nous avons implémenté trois fonctions permettant de vérifier si une chaîne de caractères contient autant de parenthèses ouvrantes que fermantes et disposées dans le bon ordre. La première fonction, "*equilibre()"*, vérifie si une chaîne donnée est équilibrée en utilisant la récursivité et le pattern matching. La deuxième fonction, "*equilibreGenerique()"*, est une version générique de la première qui prend en paramètres les caractères ouvrant et fermant. Enfin, la troisième fonction, "*equilibreXml()"*, utilise la fonction générique pour vérifier si une chaîne est équilibrée avec les caractères < et > comme caractères ouvrant et fermant qui génèrent les balises pour les fichiers XML en informatique.

Dans la fonction "*equilibre()"*, nous utilisons la récursivité pour parcourir la chaîne de caractères. Nous utilisons également le pattern matching pour vérifier si le caractère courant est une parenthèse ouvrante ou fermante, et pour mettre à jour le compteur de parenthèses ouvertes. Si le compteur de parenthèses ouvertes est égal à 0 à la fin de la chaîne, cela signifie que la chaîne est équilibrée et nous renvoyons true. Si le compteur n'est pas égal à 0, cela signifie que la chaîne n'est pas équilibrée et nous renvoyons false.

La fonction "*equilibreGenerique()"* est une version générique de "*equilibre()*" qui prend en paramètres les caractères ouvrant et fermant. Nous utilisons cette fonction pour définir la fonction "*equilibreXml()"*, qui vérifie si une chaîne est équilibrée avec les caractères < et > comme caractères ouvrant et fermant.

En résumé, ce TP nous a permis de mettre en pratique la récursivité et le pattern matching (vesion amélioré d'une switch case) en Scala pour résoudre un problème de vérification d'équilibre de parenthèses. Nous avons également vu comment créer une fonction générique en Scala qui peut être utilisée pour définir des fonctions spécialisées.

Le pattern matching dans cet exercice nous a aidé donc à réduire une grosse partie de code écrit avec des instruction comme "if" et "else".

Rapport TP05 – ASCII-ART

Le TP ASCII-art a pour objectif de créer une classe permettant de générer de l'art ASCII à partir de chaînes de caractères. Pour ce faire, la classe ASCIIart prend en entrée une chaîne de caractères contenant la définition de l'art ASCII souhaité, avec une largeur, une hauteur et une liste de chaînes de caractères représentant chaque ligne de l'art ASCII.

La classe ASCIIart dispose déjà d'une méthode "*listeToMap()*" qui permet de créer un tableau associant chaque lettre de l'alphabet à son art ASCII correspondant, sous forme de liste de chaînes de caractères. Cette méthode fonctionne en découpant chaque ligne de l'art ASCII en sous-chaînes de caractères de la largeur de l'art ASCII, puis en transposant le résultat pour obtenir une liste de listes de chaînes de caractères, où chaque liste de chaînes de caractères correspond à l'art ASCII d'une lettre. Ensuite, cette liste est convertie en un tableau associatif en utilisant la méthode "zip" de la classe "List", qui permet de combiner deux listes en une liste de couples (lettre, art ASCII). Enfin, le tableau obtenu est converti en un "Map" en utilisant la méthode "toMap", et le caractère "?" est défini comme valeur par défaut pour les lettres non définies en utilisant la méthode "*withDefaultValue()*".

La classe ASCIIart possède également une méthode "*apply()*" permettant de générer l'art ASCII d'un mot donné. Cette méthode fonctionne en utilisant la méthode "*flatMap()*" de la classe "List", qui permet de transformer chaque lettre du mot en son art ASCII correspondant, puis en concaténant les lignes de chaque art ASCII grâce à la méthode "*mkString*" de la classe "String". La méthode "*apply()*" utilise également la méthode "transpose" de la classe "List" pour transposer les lignes de chaque art ASCII afin de les aligner verticalement.

En résumé, ce TP nous permet d'apprendre à utiliser certaines méthodes de la classe List pour traiter des données sous forme de liste. Nous avons vu comment utiliser les méthodes de base de cette classe, comme head, tail, isEmpty, et comment les combiner pour créer des fonctions plus complexes. Nous avons également appris à gérer les exceptions. En utilisant ces méthodes de manière récursive, nous avons pu créer des fonctions capables de traiter des listes de manière itérative. Enfin, nous avons également vu comment utiliser les fonctions d'ordre supérieur, telles que map et filter, pour manipuler les éléments d'une liste de manière plus concise et efficace.

Rapport TP06 – Conway

La classe Conway fournie dans le code source a pour objectif de calculer et de représenter les valeurs de la suite de Conway. Pour ce faire, elle utilise une approche basée sur les LazyLists, qui permettent de représenter une liste de valeurs qui s'étend de manière infinie.

La première étape consiste à écrire la fonction "*lire()*", qui prend en entrée une liste d'entiers et renvoie la liste au rang suivant selon les règles de la suite de Conway. Pour ce faire, la fonction utilise une fonction imbriquée "*imbrique_lire()*" qui parcourt la liste d'entiers et utilise une variable "n" pour compter le nombre d'occurrences consécutives d'un même entier. Lorsqu'un entier différent est rencontré, "n" et l'entier en cours de traitement sont ajoutés à la liste de sortie et "n" est remis à 1 pour compter les occurrences de l'entier suivant.

La seconde étape consiste à définir la LazyList "*rangs*" qui contient l'ensemble des rangs possibles de la suite de Conway. Pour cela, la classe utilise une fonction imbriquée "i*mbrique_rang*" qui appelle

récursivement la fonction "*lire()*" sur chaque élément de la LazyList jusqu'à obtenir une liste infinie de rangs.

La troisième étape consiste à définir la méthode "*apply()*" qui prend en entrée un entier représentant le rang souhaité et renvoie la valeur de la suite de Conway à ce rang sous forme de chaîne de caractères. La méthode utilise simplement la LazyList "rangs" pour accéder au rang souhaité et utilise la méthode "mkString" pour convertir la liste d'entiers en chaîne de caractères.

En résumé, la classe Conway utilise une approche basée sur les LazyLists pour représenter la suite de Conway de manière infinie et utilise la fonction "*lire()*" pour calculer les rangs suivants à partir d'un rang donné. La méthode "*apply()*" permet d'accéder à un rang donné et de le représenter sous forme de chaîne de caractères.

Rapport TP07 – Graphe

Le TP Graphe a pour objectif de manipuler des graphes non orientés et non valués. Un graphe est représenté par un ensemble de nœuds et un ensemble d'arcs.

La première étape consiste à définir l'opérateur "+" permettant d'ajouter un arc à un graphe. Cela est réalisé en créant une nouvelle instance de la classe Graphe en utilisant l'opérateur "+" sur les ensembles de noeuds et d'arcs pour ajouter le nouvel arc aux ensembles existants.

La deuxième étape consiste à définir l'opérateur "+" permettant de faire l'union de deux graphes. Cela est réalisé en créant une nouvelle instance de la classe Graphe en utilisant l'opérateur "++" sur les ensembles de noeuds et d'arcs pour ajouter les noeuds et arcs de l'autre graphe aux ensembles existants.

La troisième étape consiste à définir la méthode "*voisins()*" permettant de renvoyer les voisins d'un nœud donné. Pour cela, on utilise la méthode *flatMap* sur l'ensemble d'arcs pour mapper chaque arc sur un ensemble de noeuds. Si l'extrémité 1 de l'arc correspond au nom du noeud en question, on ajoute l'extrémité 2 au Set. Si l'extrémité 2 de l'arc correspond au nom du noeud en question, on ajoute l'extrémité 1 au Set. Si aucune des extrémités ne correspond au noeud en question, on ajoute un Set vide. Ensuite, on utilise la méthode *diff* pour enlever le noeud en question du Set de voisins.

La quatrième étape consiste à définir la méthode permettant de connaître le degré d'un nœud (nombre de voisins). Pour cela, on utilise simplement la méthode size sur le Set de voisins du noeud en question.

La cinquième étape consiste à définir la méthode permettant de calculer, si le chemin est possible, la distance minimale entre deux points. Pour cela, on peut utiliser l'algorithme de parcours en largeur . On commence par initialiser une queue avec le noeud de départ et un tableau de booléens pour marquer les noeuds visités. Tant que la queue n'est pas vide, on récupère le premier élément et on parcourt tous ses voisins. Si l'un d'eux est le noeud d'arrivée, on renvoie la distance actuelle. Sinon, on ajoute les voisins non visités à la queue et on met à jour la distance pour chacun d'eux. Si aucun voisin n'est le noeud d'arrivé, la distance entre ces deux noeuds est de 1. Si un voisin est le noeud d'arrivée, la distance est de 0. Sinon, la distance est égale à la distance minimale entre le noeud de

départ et l'un de ses voisins, plus 1. On peut donc implémenter cette fonction de manière récursive, en utilisant la fonction voisins définie précédemment et en comparant chaque voisin au noeud d'arrivée. Si aucun des voisins n'est le noeud d'arrivée, on retourne la distance minimale entre le noeud de départ et l'un de ses voisins, plus 1. Sinon, on retourne 0.

Pour trouver les composantes connexes d'un graphe, on peut utiliser une approche similaire. On commence par définir une fonction récursive *composantesConnexesAux* qui prend en entrée un noeud et une liste de noeuds déjà visités, et qui renvoie l'ensemble des noeuds accessibles depuis ce noeud en parcourant le graphe en largeur. On peut alors utiliser cette fonction pour trouver les composantes connexes de chaque noeud du graphe, en éliminant les noeuds déjà visités de la liste de noeuds à traiter.

Rapport TP08 – N-Reines

Pour résoudre le problème des N-Reines, le code source fourni utilise une approche récursive. La fonction principale "*solutions()*" est définie de manière à être appelée récursivement, en ajoutant une reine à la grille à chaque itération.

La première étape de la résolution consiste à vérifier si une colonne donnée est acceptable pour y placer une reine, compte tenu de la solution partielle actuelle. Pour cela, la fonction "*estOk()*" prend en entrée la colonne à vérifier et la liste des colonnes actuellement occupées par les reines. Elle utilise une fonction récursive "*estOkRec()*" pour parcourir la liste des colonnes occupées et vérifier si la colonne donnée est compatible. Si la colonne donnée est la même que celle d'une reine déjà présente ou si elle se trouve sur la même diagonale qu'une reine déjà présente, la fonction "*estOk()*" renvoie false, sinon elle renvoie true.

La seconde étape consiste à trouver toutes les solutions possibles au problème des N-Reines. Pour cela, la fonction "*solutions()*" utilise une boucle "for" pour parcourir toutes les colonnes de la grille et vérifie, à l'aide de la fonction "*estOk()*", si la colonne est acceptable pour y placer une reine. Si la colonne est acceptable, une nouvelle reine est placée sur la grille et la fonction "solutions" est appelée récursivement avec la solution mise à jour. Si aucune colonne acceptable n'est trouvée, la fonction "*solutions()*" renvoie l'ensemble des solutions trouvées jusque-là.

La troisième étape consiste à compter le nombre de solutions trouvées. Pour cela, la fonction "*nombreSolutions()*" utilise la fonction "solutions" définie précédemment et renvoie simplement la taille de l'ensemble des solutions trouvées.

Enfin, la quatrième étape consiste à afficher une solution sous forme de chaîne de caractères. Pour cela, la fonction "*afficheSolution()*" prend en entrée une liste représentant une solution et utilise une boucle "for" pour parcourir cette liste et remplacer chaque élément par un caractère "X" ou "O", selon qu'il y a une reine ou non à cet endroit. La fonction "*afficheToutesSolutions()*" utilise la fonction "*afficheSolution()*" pour afficher toutes les solutions trouvées.

En résumé, le code source utilise une approche récursive pour résoudre le problème des N-Reines en vérifiant la compatibilité de chaque colonne pour y placer une reine et en ajoutant récursivement une reine à chaque itération jusqu'à trouver une solution complète. La fonction "*estOk()*" est utilisée pour vérifier si une colonne est acceptable pour y placer une reine en fonction de la solution partielle actuelle. La fonction "solutions" trouve toutes les solutions possibles en appelant récursivement la fonction sur chaque colonne acceptable. La fonction "nombreSolutions" compte le nombre de

solutions trouvées en utilisant la fonction "solutions". Enfin, la fonction "afficheSolution" transforme une solution en une chaîne de caractères affichable et la fonction "*afficheToutesSolutions()*" utilise "*afficheSolution()*" pour afficher toutes les solutions trouvées.

Conclusion

En guise de conclusion, cette gamme de TP nous a permis progressivement de découvrir le langage fonctionnel Scala, et y s'approfondir ensuite. Le véritable défi lors de l'effectuation du code était surtout l'obligation d'éviter les paradigmes de la programmation impérative. En effet, nous étions face à des programmes qui, en principe, semblent être très simple à faire en utilisant des boucles, des variables mutables, sauf que le paradigme du langage Scala nous oblige à chaque fois trouver des solutions basées principalement sur des fonctions imbriquées et de la récursivité.

En outre, avec ce nouveau paradigme, on a pu écrire des codes avec un langage purement basé sur l'algorithmique. En effet, le TP des listes illustre le fait qu'on a pu écrire des fonctions qui manipulent les listes en n'utilisant que quatre méthodes basiques. Cela nous a donné une idée de comment un langage et les bibliothèques se construisent.

Nous avons gagné a la fin :

- Un code plus clair, plus sûr et plus robuste : la programmation se fait localement sans avoir besoin stocker et changer les valeurs des variables durant l'exécution.
- Parallélisation simplifiée, utilisation de multithreads facile sans besoin de synchronisation d'accès aux variables.