UNIVERSITÉ DE RENNES 1

Calculabilité

Rapport de travaux pratiques sur les Pretty Printer

Glen Le Baill

Antoine Bouffard

Introduction

Un pretty printer a pour but de rendre, à partir d'un AST[1] reçu en paramètres, une chaîne de caractères ayant la même sémantique que cet AST, et respectant la syntaxe et l'indentation du langage de programmation convenu. Ainsi, un pretty printer est un formateur syntaxique qui ne change pas la sémantique du programme.

Programme

Structure du programme

Dans le fichier principal `prettyPrinter.scala`, le squelette du projet qui nous a été fourni comporte une structure en plusieurs parties. Les plus importantes sont les parties Expressions, Méthodes auxiliaires pour les chaînes de caractères (indentation), Commandes, et Programmes. Les parties Expressions et Commandes correspondent chacune à une interface Scala englobant un groupe d'instructions du langage While ayant des caractéristiques communes, et comporte donc le traitement applicatif lié au thème représenté par l'interface.

La partie Expressions s'applique aux expressions du langage While, comprenant les types de base `{ Const, VarExp, Nil }` ainsi que les méthodes opérant sur les arbres du langage While `{ Hd(), Tl(), Eq() }`.

La partie Commandes s'applique quant à elle à l'instruction conditionnelle `if` ainsi qu'aux boucles du langage While et du langage For `{ while, for }` ainsi qu'à l'affectation `Set` et au type `Nop`.

Enfin la partie Programmes utilise les deux parties précédentes dans le but de retourner un programme While syntaxiquement correct. C'est donc en quelque sorte le contrôleur de l'application qui est en charge d'orchestrer les différents modules.

Méthodes

Les méthodes de l'application sont ainsi départagées entre les quatre sections listées ci-dessus.

1 Abstract Syntax Tree : Arbre de syntaxe abstraite.

Comme nous l'avons vu, la logique applicative du programme se concentre dans la section Programmes. La méthode principale de l'application, `prettyPrint()`, s'y trouve donc. Le reste du programme se compose en grande partie de méthodes auxiliaires nécessaires à la réalisation de cette méthode principale, respectant ainsi certaines préconisations de génie logiciel.

Séparer cette méthode principale en différentes méthodes auxiliaires permet en effet de factoriser le code et de respecter les principes DRY[2], apportant comme bénéfices notamment un code clair et une maintenabilité facilitée.

Bénéficiant de ces bonnes pratiques, la méthode principale est très concise :

```
    /**
     * @param program : un AST décrivant un programme du langage WHILE
     * @param is : une liste de spécifications d'indentation
     * @return une chaîne représentant la syntaxe concrète du programme
     */
    def prettyPrint(program: Program, is: IndentSpec): String = {
        program match {
            case Progr(in, body, out) => {
                "read " + prettyPrintIn(in) + "\n%\n" +
listeStringToString(appendStringBeforeAll(makeIndent(indentSearch("PROGR", is)),
prettyPrintCommands(body, is))) + "%\n" + "write " + prettyPrintOut(out)
            }
        }
    }
```

Pour analyser le déroulement du programme, nous allons donc garder le focus sur cette méthode `prettyPrint()` et y jouer le débogueur, sommairement pour ne pas ennuyer le lecteur.

Dans le pattern matching, la première méthode à être appelée est `prettyPrintIn()`, qui se contente d'afficher avec la bonne syntaxe les variables d'entrée du programme. Ensuite, plusieurs méthodes sont imbriquées. L'exécution de ces méthodes se faisant à partir de la plus imbriquée jusqu'à celle la plus à l'extérieur, nous allons commencer l'analyse par `indentSearch()`.

Cette méthode appartient à la section des méthodes auxiliaires destinées à la réalisation des traitements des chaînes de caractères. Son rôle est de retourner l'entier correspondant au nombre d'espaces nécessaires pour l'indentation correcte du contexte reçu en paramètres.

Ensuite, `makeIndent()`, qui appartient à la même section du programme, est chargée de convertir cet entier en nombre d'espaces dans une chaîne de

2 DRY : Don't repeat yourself

caractères. `makeIndent()` et `indentSearch()` réalisent donc l'indentation du programme.

La méthode `appendStringBeforeAll()` est elle en charge de concaténer cette indentation obtenue au début de chaque élément de son deuxième paramètres, une `List[String]` contenant dans l'ordre chaque instruction du programme bien indentée. Cette `List[String]` a été préalablement obtenue en appelant une méthode de la section Commandes du programme, qui elle-même se servait de méthodes auxiliaires et appelait les méthodes de la section Expressions de l'application.

`AppendStringBeforeAll()` retourne donc une `List[String]` syntaxiquement correcte et bien indentée, qui est transformée en `String` par la méthode `listeStringToString()`. Enfin, les variables de sortie sont listées et le résultat final est retourné par `prettyPrint()`.

Nous avons donc démontré que `prettyPrint()` est la racine d'un arbre de méthodes chargées de réaliser l'objectif de l'application : rendre un AST syntaxiquement correct en While. L'appel à cette seule racine suffit donc à mettre en mouvement l'arbre entier.

Difficultés rencontrées

La section du programme nous ayant causé le plus de difficultés est la section Commandes. En effet les deux méthodes de cette section, `prettyPrintCommand()` et `prettyPrintCommands()`, ne sont pas triviales à réaliser du fait de l'appel récursif mutuel qu'elle se font. De plus l'indentation étant à gérer, les appels imbriqués de méthodes sont nombreux et rendent la lisibilité moins évidente. Enfin, la gestion des `;`, qui ne doivent pas apparaître à la fin de la dernière instruction d'un bloc d'instructions, nous a causé quelques difficultés.

Les sections Méthodes auxiliaires pour les chaînes de caractères, Expressions et Programmes ne nous ont en revanche pas posé de difficultés particulières.

Conclusion

Ce projet de pretty printer a été pour nous l'occasion d'améliorer nos compétences en programmation fonctionnelle et récursive. En effet au début du projet, nous avions fortement tendance à programmer dans un paradigme impératif, avec des variables mutables et des boucles for. Le design du langage Scala et la prise de conscience de l'élégance et de la concision du style récursif nous ont décidé à réécrire toutes nos méthodes en respectant cette guideline.

Par ailleurs, nous avons amélioré nos connaissances à propos des AST et des langages While et Scala. Nous avons de plus entamé une réflexion sur le but des AST dans le processus de compilation ou d'interprétation d'un code source. Nous avons ainsi compris que l'AST a le rôle de lien entre le code source de haut niveau écrit par le programmeur et le code binaire exécuté par la machine.