TP2 : Analyse syntaxique – ANTLR

1 Installation

Sur les machines de Polytech, ANTLR est déjà installé. Lors de la première utilisation, lancer dans un terminal la commande suivante :

```
antlr4 -v 4.13.2
```

Pour l'installer sur vos machines, suivre les instructions sur `https://www.antlr.org/download.html`. Sous Linux, une installation est possible avec `pip install antlr4-tools`

2 Tutoriel

Le but de ce TP est de construire un programme qui évalue une expression en NPI. On va commencer par construire un analyseur lexical et syntaxique pour obtenir un arbre de syntaxe abstraite (AST). Pour cela on va utiliser un outil : ANTLR.

La grammaire du TD (question 4.3) est fournie dans le fichier `NPI.g4`. Ce fichier est composé de deux parties :

- les règles de dérivation de la grammaire
- des expressions régulières pour reconnaître les terminaux

On peut remarquer que les règles de la grammaire sont commentées (après les #), ce qui permettra d'obtenir des méthodes spécifiques à chaque règle. Pour plus d'informations, on pourra se référer à la documentation

https://github.com/antlr/antlr4/blob/master/doc/getting-started.md

On peut générer un AST avec ANTLR directement au terminal avec la commande `antlr4-parse`, qui prend en argument le fichier de la grammaire, et le nom de l'axiome. Par exemple, après la commande

```
antlr4-parse NPI.g4 main -tree
```

on peut taper une expression qui sera analysée par ANTLR. Pour terminer l'analyse, il faut sauter une ligne puis utiliser `Ctrl+D` (linux) ou `Ctrl+Z` (windows).

Le résultat de la commande est l'AST, écrit en format textuel. Pour l'obtenir sous forme graphique, on peut plutôt utiliser la commande :

```
antlr4-parse NPI.g4 main -gui
```

Tester ces commandes sur $1\,2+3*$, $1\,2\,3+$ et $1\,2\,3+*/$.

3 Évaluation

Le but de cette section est de construire un programme qui évalue une expression en NPI. On va procéder en parcourant l'AST généré par ANTLR. La commande

```
antlr4 NPI.g4 -visitor
```

génère plusieurs classes et interfaces Java. L'interface la plus importante est `NPIVisitor`, qu'on va implémenter pour créer l'évaluateur. Un exemple est fourni par la classe `NPIBaseVisitor`. Ces classes contiennent des méthodes appelées *visiteurs* qu'on va lancer sur des nœuds de l'AST et qui renvoient un élément d'un certain type générique T.

L'interface `NPIVisitor` et la classe `NPIBaseVisitor` sont génériques (elles dépendent d'un type T). Ici, on souhaite obtenir des visiteurs qui renvoient un entier, qui sera le résultat de l'évaluation de la sous-expression qu'on considère.

Dans cette section, on vous demande d'implémenter la classe `NPIEvaluator`. Similairement à la classe `NPIBaseVisitor`, elle hérite de la classe `AbstractParseTreeVisitor<Integer>` et implémente l'interface `NPIVisitor<Integer>`. Il y a donc trois méthodes que vous devez compléter, qui correspondent au traitement de chaque type de noeud de l'AST.

4 Vers un code assembleur

Le but de cette section est de transformer une expression NPI en une suite d'instructions assembleur qui évalue l'expression. Dans un premier temps, on va implémenter un algorithme similaire à celui du TD (question 4.2), où on utilise une pile pendant l'évaluation.

4.1 Pile

Implémenter une classe `NPIStack` qui hérite de la classe `AbstractParseTreeVisitor<Void>` et qui implémente l'interface `NPIVisitor<Void>`. Cette classe possède une pile comme attribut, sur laquelle on va empiler et dépiler des éléments quand on appelle les visiteurs. Chaque visiteur fait donc uniquement des modifications de la pile, mais ne renvoie rien. Quand on a fini d'évaluer une expression, on affichera le résultat sur la sortie standard.

4.2 String

On souhaite maintenant construire le code assembleur, qui fonctionne sur le même principe que la section précédente : on va simuler une pile dans la mémoire, en remplaçant les empilements par des ST et des dépilements par des LD.

Implémenter la classe `NPIString` dont les visiteurs renvoient un objet de type String contenant un programme assembleur.

Adapter la classe `Main` pour écrire le programme obtenu dans le fichier `prog.asm`. Vérifier ensuite en lançant :

```
python3 simproc.py
```

que le contenu de `sorties.txt` contient bien le résultat attendu.

4.3 Program

La construction précédente peut être rendue plus propre d'un point de vue orienté objet. Cela vous permettra aussi de manipuler plus facilement le programme obtenu quand vous occuperez de la partie "génération de code" du projet. On vous fournit les classes :

- `Instruction` qui représente les instructions, dont héritent `UAL`, `UALi`, `Mem`, `IO` et `CondJump`, `Ret`, `JumpCall` et `Stop`.
- `Program` qui représente les programmes, c'est-à-dire les suites d'instructions.

Reprendre la section précédente en implémentant une classe `NPIProgram` contenant des visiteurs qui renvoient un objet de type Program et non String.