Le but de ce TP est de construire un arbre abstrait pour les programmes Ava lors de l'analyse syntaxique, permettant ultérieurement des analyses sémantiques modulaires. Pour ce faire on reprend la grammaire algébrique du TP2 (éventuellement en la modifiant si elle n'est pas du tout adaptée[1]) et on l'attribue pour construire un arbre.

1 Description du matériel fourni

Copiez chez vous l'archive `tp4.tgz` qui est sur le portail. Elle contient entre autres des scripts, des spécifications `.lex` et `.cup` (répertoire `spec`) et des fichiers sources (répertoire `src`).

Spécifications

Le répertoire `spec` contient une spécification d'analyseur lexical et une spécification d'analyse syntaxique pour Ava (générer les analyseurs par `ant genAnLex` et `ant genAnSynt`). Ces fichiers sont fournis pour que les étudiants n'ayant pas finalisé le TP2 puisse faire le TP4. Les autres peuvent effacer ces spécifications et les remplacer par les leurs, après validation par l'enseignant.

Paquetages `arbreAbstrait` et `decompilateur`

Paquetage `ava.arbreAbstrait` Le paquetage `ava.arbreAbstrait` est décrit informellement par un diagramme de classes dans l'annexe A (classe principale `Programme`). Ce diagramme, très incomplet, a pour but d'expliciter la hiérarchie de classes. Pour les détails on consultera la javadoc (`ant genDoc`). Le paquetage `ava.arbreAbstrait` contient tous les types requis pour construire un arbre abstrait (du moins ceux choisi par l'enseignant, d'autres solutions sont possibles). Vous n'avez plus qu'à assembler les briques, après en avoir complété certaines. Le style de programmation utilisé est défensif : les méthodes lèvent des exceptions de type `IllegalArgumentException` si elles sont mal utilisées (paramètre null, etc).

Le paquetage contient aussi des interfaces `Visiteur` et `Visitable` que vous utiliserez explicitement pour le TP5.

Paquetage `ava.decompilateur` Décompiler signifie reconstruire un programme source à partir d'un arbre abstrait. L'analyse sémantique consiste alors à construire une représentation textuelle de l'arbre. La classe `ava.decompilateur.Decompilateur` effectue ce travail en parcourant l'arbre abstrait (on verra comment lors du TP5 : utilisation d'un `Visiteur`). La syntaxe décompilée est volontairement fantaisiste, pour montrer que l'arbre abstrait est indépendant de toute syntaxe.

Ce décompilateur doit vous aider à tester votre arbre abstrait en scrutant l'affichage de sa représentation textuelle[2]. On stockera les exemples testés dans `test/OK`. Le décompilateur permet aussi de faire apparaître les priorités des opérateurs fixées par la grammaire.

Classes principales et scripts

Les scripts habituels sont fournis, avec de nouveaux venus :

- `execEnLigneDecompilateur.sh` construit l'arbre abstrait d'un programme entré dans la console, et affiche sur la console le résultat de sa décompilation ;
- `execSurFichierDecompilateur.sh` qui prend en ligne de commande le nom d'un fichier à analyser, construit l'arbre abstrait pour le programme contenu dans le fichier, et affiche sur la console le résultat de sa décompilation.

Aucun script de la forme `execTest` n'est fourni pour le décompilateur, cette forme de test n'étant pas adaptée ici.

[1] Auquel cas il est impératif de repasser les tests OK et KO pour l'analyse syntaxique.

[2] C'est une très mauvaise manière de tester. D'une part scruter l'écran ne réalise pas un test automatique. D'autre part, ce n'est pas un test unitaire : si la décompilation indique une erreur, comment dire si elle vient du décompilateur ou de l'arbre ? Un test rigoureux et unitaire devrait comparer l'arbre construit à l'arbre réellement obtenu, mais on ne le fera pas dans ce TP.

2 Travail à réaliser

2.1 Paquetage arbre abstrait

Pour vous familiariser avec ce paquetage, vous êtes invités à compléter le code des classes `Boucle`, `ExpressionBinaire` et `ImpressionChaineSansSautDeLigne` (très simple, surtout en s'inspirant des autres classes fournies).

2.2 Analyse sémantique

Commencer par s'assurer que l'analyseur lexical associe bien des valeurs aux terminaux concernés (entier, identificateur et chaîne).

Ensuite, attribuer la grammaire (grammaire S-attribuée : attributs synthétisés seulement) pour construire l'arbre abstrait (ne pas oublier d'importer le paquetage `ava.arbreAbstrait` et toute autre classe Java dont vous auriez l'usage). Procéder de manière itérative sachant que le décompilateur prend en paramètre un objet de type `Programme` non nul[3]. Ce `Programme` est construit à partir du nom du programme (objet `String` non null) et de ses listes de déclarations (`ListeDecl`) et d'instructions (`ListeInstr`), non nulles elles-aussi. On commencera donc ainsi[4] :

```
terminal String CHAINE;
... // autres terminaux
non terminal String entete;
non terminal Programme programme;
... // autres non terminaux
... // priorités

programme ::= entete:chaine declarations instructions
  {: RESULT = new Programme(chaine,
                            new ListeDeclaration(),
                            new ListeInstruction());   :}
;

entete ::= PROG CHAINE:chaine FININSTR {: RESULT = chaine; :}
;
```

On peut à ce stade tester la construction d'un arbre abstrait et sa décompilation pour un programme sans déclarations ni instructions.

Ensuite on ajoutera progressivement des instructions et/ou déclarations, en pensant bien à tester au fur et à mesure. Noter que les programmes testés n'ont pas à être sémantiquement corrects.

[3] Ce qui explique que si on lance le décompilateur sans avoir associé d'objet `Programme` à l'axiome, on obtient une levée d'exception de type DecompilateurException.

[4] À adapter à votre grammaire bien entendu !

A Diagramme de classes simplifié pour `ava.arbreAbstrait`