TP 1 : écriture de grammaires pour des langages de programmation

1 Utilisation d'ANTLR4

1.1 Avec le script bash fourni `test.sh` (recommandé)

Téléchargez le script et rendez-le exécutable.

Lors de la conception d'un langage, donnez lui un nom (par exemple 'bibi'), et créez un fichier 'bibi.g4'. Votre fichier pourra alors commencer par la ligne 'grammar bibi;'

Nœud de départ : il doit avoir le même nom dans les différents exercices (par défaut "prog", sinon mettez à jour la variable `start` du script).

Librairie dans le script, mettez à jour la variable `path` où vous souhaitez télécharger la libraire antlr (archive java). (Ce même fichier vous sera utile pour les autres séances, inutile d'en conserver une copie par sujet de TP).

Test pour tester votre grammaire `nom.g4` sur un fichier `fic`, utilisez la commande suivante : `./test.sh nom fic` (nottez l'absence de l'extension `.g4`).

1.2 Avec IntelliJ

Vous pouvez aussi utiliser IntelliJ et le plugin ANTLR4. La fenêtre "Antlr4 Preview" permet de tester votre grammaire, et d'afficher l'arbre syntaxique qui est reconnu par le programme.

Vous devez pour cela spécifier une règle à tester (clic droit → 'test rule ...'), pour que le reconnaisseur sache à quelle catégorie syntaxique s'adresser pour commencer l'analyse. (dans le cadre de la reconnaissance d'un programme, on pourra avoir une règle `program` qui sera celle à appliquer. Mais on peut auparavant tester que les expressions sont bien reconnues, etc...)

2 Dans ce TP

Dans ce TP, il vous est demandé d'écrire des grammaires correspondant à des langages de programmation, selon différents paradigmes. On commencera par

quelques expressions et instructions dans un style impératif, pensez à un petit programme écrit en C ou C++.

Ensuite, il s'agit d'étendre votre grammaire pour qu'elle supporte l'écriture de classes, avec attributs et méthodes, de façon à pouvoir reconnaître un petit programme Java.

Enfin, écrivez une grammaire pour un langage fonctionnel, en vous basant sur OCaml (uniquement son fragment fonctionnel).

2.1 Méthode

Sont indiqués pour les trois grammaires quels sont les éléments que doit contenir le langage. Il est conseillé de commencer par un petit langage qui fonctionne et de l'étendre petit à petit, plutôt que d'essayer d'écrire toute la grammaire avant de la tester.

Vous pouvez bien sûr rajouter tous les éléments qui vous semblent pertinents, et faire des tests sur des programmes plus complexes que ceux proposés.

tests Un exemple pour chaque langage est fourni sur Celene. Mais dans un premier temps, vous pouvez utiliser le mode intéractif (sur IntelliJ ou en ligne de commande) en testant des expressions ou instructions au fur et à mesure.

Attention Si votre test est reconnu sans erreur, rien n'assure que l'arbre syntaxique obtenu est celui souhaité, vérifiez bien que votre arbre rend compte de la structure de votre programme, et que tout ce qui est écrit dans la source apparaît dans l'arbre.

2.2 Rappels sur ANTLR

Pour démarrer :

- (Vous pouvez vous inspirer du fichier 'trees.g4' de la démo du cours pour un rappel du format de fichier attendu)
- Dans le fichier `langage.g4`, entrez la ligne `grammar langage;`
- Pour la reconnaissance des lexèmes, utilisez des expressions régulières (notamment pour les identifiants). Par exemple `INT:[0-9]+;`
- N'oubliez pas de traiter les espaces blancs (et les sauts de ligne[1] avec une ligne comme : `WS:[ \t\r\n] -> skip;`
- Pour les règles de la grammaire, utilisez la syntaxe basée sur BNF vue en cours (les noms de règles pour les lexèmes commencent par une majuscule, et ceux de règles pour la syntaxe par une minuscule).
- Chaque règle (lexicale ou syntaxique) se termine par un point-virgule.

[1] Si vous voulez reconnaître un langage similaire à Python, ce traitement sera différent, bien sûr.

3 Un langage impératif

Dans un premier temps, écrivez une grammaire comportant les éléments suivants :

1. Lexèmes de base, comprenant :
   - Types (bool, int, float,...)
   - Identifiants (suite de caractères alphanumériques, ne pouvant pas commencer par un chiffre)
   - Entiers
   - Flottants (sous la forme `x.y` ou `x.`)
2. Expressions
   - Opérateurs booléens et arithmétiques usuels.
   - Tableaux et éléments de tableaux
3. Instructions
   - Impression
   - Lecture
   - Modification de tableau
   - Déclarations et affectations
   - Conditionnelle
   - Boucle 'Tant que'
4. Programmes. Un programme est une liste d'instructions.

Vous devez faire en sorte que les instructions terminent par un point-virgule, mais pas les blocs d'instructions liés aux boucles et conditionnelles.

Testez votre reconnaisseur sur un petit programme impératif simple. Un exemple est fourni sur Celene (`imp.c`).

4 Un langage objet

4.1 Méthodes

On veut que les méthodes soient déclarées exactement comme en java

1. Il y a toujours un type de retour dès la déclaration de la méthode
2. Le nom de la méthode respecte les mêmes contraintes que ceux des identifiants
3. Le corps de la méthode est constitué d'expressions.

4.1.1 Classes

1. Toute instruction est à l'intérieur d'une classe
2. Il peut y avoir plusieurs classes dans le fichier
3. Il y a une classe principale (celle qui contient la méthode `main`)
4. Une classe peut hériter d'une autre (avec le mot clef `extends`)
5. On peut instancier les objets (`new ...`)
6. On peut accéder à une méthode ou un attribut d'un objet (`class.met(...)`, `class.attr`)

Bonus. Interfaces, imports, packages.

4.1.2 Test

Essayez votre reconnaisseur sur un programme java quelconque. Un exemple est fourni sur Celene (`Obj.java`).

5 Un langage fonctionnel

Rappel Pour un langage fonctionnel (on prendra OCaml comme exemple), il n'y a qu'une catégorie syntaxique : les expressions. Certaines instructions — qu'on peut trouver dans des langages non fonctionnels — comme `print` sont des expressions de type `unit`. D'autres, comme les boucles, ne seront pas présentes mais traitées à travers l'évaluation de fonctions récursives.

- Expressions usuelles (comme dans les langages précédents), mais comme en Caml, le test d'égalité s'écrit `=` et non `==`).
- Fonctions (`fun x -> ...`)
- Applications de fonctions (`f x`)
- Liaisons (`let x = ...`, `let x = ... in ...`)
- Indications de typage explicite facultatives (`let f:int->int =...` ou `let x:int = ...`)

Bonus Ajoutez la possibilité d'écrire les fonctions `let f x =...`(en plus de l'écriture `let f = fun x -> ...`)

Bonus Ajoutez les modules et les signatures de modules.

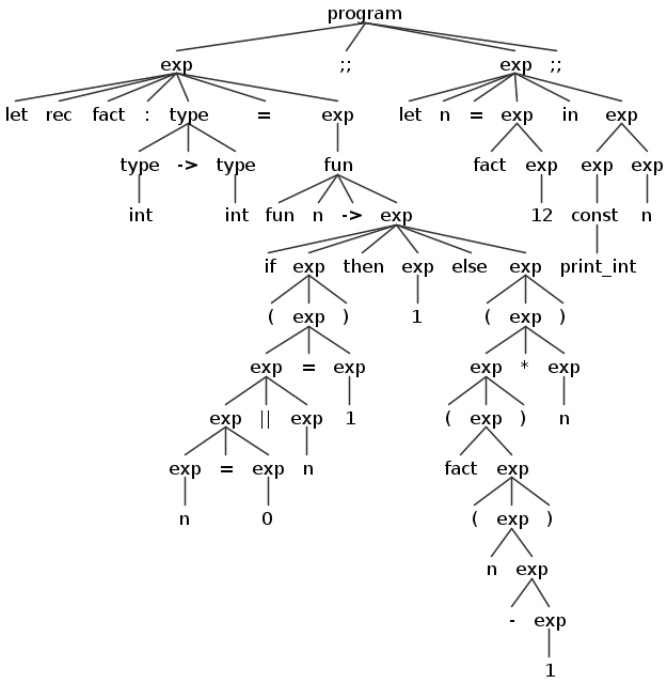

Figure 1: Un arbre possible pour le programme donné en exemple (`fun.ml` sur Celene)