ISTIC - Département Informatique

DM de Compilation

Prise en main de l'outil ANTLR

Bihel Simon - Daniel Lesly-Ann

28 septembre 2016

Introduction

Antlr est un outil permettant de générer des analyseurs lexicaux ou syntaxiques dans un langage cible, notamment en Java. Il peut être utilisé entre autre, pour construire un traducteur, un interpréteur ou un compilateur à partir de règles de grammaire définies par l'utilisateur. Il permet aussi de générer facilement des arbres syntaxiques, de gérer les erreurs et d'associer des actions aux règles de grammaire.

1 Fonctionnement de l'outil antlr

Antlr utilise une stratégie d'analyse syntaxique *top-down* appelée LL(*), qui analyse le flot de données de gauche à droite . Cette stratégie permet d'accepter toutes les grammaires LL, qui sont un type de grammaire hors-contexte. Lors de l'analyse LL(*) un nombre arbitraire d'unités lexicales peuvent être lues afin de déterminer la bonne production. Cette méthode offre un bon compromis entre efficacité et expressivité. Cependant, elle ne permet pas d'analyser les grammaires présentant des règles récursives à gauche. Néanmoins, la récursivité à gauche peut être facilement éliminée, de plus la notation EBNF proposée par Antlr permet de contourner ce problème.

En effet, l'outil Antlr lit une grammaire en notation EBNF, qui a une syntaxe proche des expressions régulières, et introduit notamment l'utilisation des symboles '*', ' ?' et '+' pour la définition règles. Cette notation permet de définir facilement des règles récursives à droite et d'éviter ainsi la récursivité à gauche.

Les règles de grammaire peuvent être différenciées en deux types.

— Les règles lexicales ne contiennent que des littéraux (qui peuvent contenir des symboles EBNF) ou d'autres règles lexicales. Par convention dans Antlr ces règles commencent par une majuscule.
— Les règles syntaxiques peuvent contenir des littéraux, des règles lexicales ou des règles syntaxiques. Par convention dans Antlr ces règles commencent par une minuscule.

Antlr permet d'associer des options aux règles de grammaire. L'action, définie entre accolades, est une instruction ou une suite d'instructions, écrite dans le langage cible et qui sera exécutée quand la règle sera rencontrée. Elles permettent entre autre, comme dans l'exemple fourni, d'affecter une valeur à un attribut en fonction d'attributs déjà évalués, à la manière d'une grammaire attribuée.

L'outil offre aussi la possibilité de construire des arbres de syntaxe abstraite à l'aide de règles de réécriture. Dans un AST, les feuilles représentent les tokens et les branches représentent les constructions syntaxiques. Les règles de réécriture permettent , pour une règle donnée, de définir comment générer la sortie. On peut aussi utiliser simplement des opérateurs '!' pour ne pas inclure le nœud ou le sous-arbre et '^' pour créer un nœud racine d'AST.

2 Exemple : évaluateur d'expression

L'exemple fourni permet d'évaluer des expressions mathématiques utilisant des identificateurs, les opérateurs '=', '+', '-', '*', ainsi que le parenthésage. Il permet aussi de construire un arbre de syntaxe abstraite pour chaque ligne de l'entrée. Les expressions sont évaluées ligne par ligne et les identificateurs sont stockés dans une table de hachage.

Comme illustré dans le listing 1, on peut ajouter la division dans la règle de la multiplication, les deux opérations ayant la même priorité. L'associativité à gauche est respectée naturellement par Antlr puisqu'il utilise une analyse LL, de gauche à droite.

Dans le cas de la puissance, il est nécessaire d'ajouter une règle pour respecter la priorité. Il faut aussi prendre en compte l'associativité à droite en forçant l'analyseur à évaluer la partie à droite de l'opérateur. On peut faire cela en utilisant l'opérateur ' ?' au lieu de '*' qu'on a utilisé précédemment, comme on peut le voir dans le listing 1. Cet opérateur permet de n'avoir au maximum qu'une seule sous-règle et force ainsi la règle à être récursive, ce qui permet la création de l'arbre de la racine aux feuilles.

Listing 1– Fichier Expr.g

```
expr:    multExpr (('+'^|'-'^) multExpr)*
    ;

multExpr
    :    powExpr (('*'^|'/'^) powExpr)*
    ;

powExpr
    :    atom ('^'^ powExpr)?
    ;

atom:    INT
    |    ID
    |    '(' expr ')'
    ;
```

Il faut ensuite ajouter les actions correspondantes pour la construction de l'arbre. Comme on peut le voir dans le listing 2, chacun des opérateurs ajoute un nœud à l'arbre qui a pour valeur l'opération appliquée aux deux sous-arbres. Si l'expression est un identifiant, on récupère sa valeur dans la table de hachage et s'il n'est pas défini, on renvoie un message d'erreur. S'il s'agit d'un entier, on transforme la chaîne de caractère en l'entier correspondant.

Listing 2– Fichier Eval.g

```
expr returns [int value]
    : ^('+' a=expr b=expr)  { $value = a+b; }
    | ^('-' a=expr b=expr)  { $value = a-b; }
    | ^('*' a=expr b=expr)  { $value = a*b; }
    | ^('/' a=expr b=expr)  { $value = a/b; }
    | ^('^' a=expr b=expr)  { $value = (int) Math.pow(a,b); }
    | ID
      {
        Integer v = (Integer) memory.get($ID.text);
        if ( v != null ) $value = v.intValue();
        else System.err.println("undefined variable "+$ID.text);
      }
    | INT { $value = Integer.parseInt($INT.text); }
    ;
```

3 Évaluation

Le fichier d'exécution fourni permet la visualisation de l'arbre syntaxique d'une expression en entrée. On peut donc évaluer notre grammaire sur la concordance des opérateurs lors de l'évaluation, la priorité des opérations et leurs associativités. Étant donné le faible nombre d'opérateurs on peut juste vérifier à la main leur validité sans passer par un générateur qui rajouterait une couche potentielle d'erreurs. Pour la priorité des opérations il suffit de prendre une expression avec tous les opérateurs et vérifier que l'arbre est correct. Il faut ensuite permuter les opérations pour être sûr que l'ordre de lecture n'a aucun impact, puis vérifier que les parenthèses sont bien prises en compte. L'associativité se vérifie en regardant l'encapsulation des parenthèses. Pour finir quelque tests aléatoires peuvent être effectués en faisant attention aux problèmes d'overflow.

Conclusion

Antlr permet de générer facilement des analyseurs dans des langages très utilisés comme le Java, que l'on peut ensuite utiliser dans une application. Il a pour spécificité d'utiliser une analyse LL(*) qui allie expressivité et efficacité, et contribue à la recherche dans l'analyse syntaxique.