

Utiliser ANTLR Studio pour développer des grammaires ANTLR

Prashant Deva

ANTLR est un outil de génération d'analyseurs syntaxiques, d'analyseurs lexicaux et d'analyseurs d'arbre. Bien sûr vous pouvez écrire ces derniers tout seul dans votre langage préféré de programmation mais cela prendrait un temps extrêmement long et sans mentionner le nombre d'erreurs que vous feriez additionné au fait qu'il serait extrêmement difficile de comprendre la grammaire de votre analyseur syntaxiques/lexicaux.

Ainsi il est plus commode d'écrire une grammaire pour le langage que vous voulez analyser et laisser un outil de génération d'analyseur générer à votre place le code approprié. Il existe également d'autres outils tels que *lex*, *yacc*, etc qui peuvent générer des analyseurs syntaxiques/lexicaux à votre place, mais ANTLR se démarque dans la mesure où il génère un code extrêmement facile à lire et dispose d'un IDE avancée dans ANTLR Studio qui rend l'écriture de grammaires très facile.

L'écriture de grammaires ANTLR est généralement considérée comme un processus lent, pénible et difficile, seulement réalisée par de « vrai » programmeurs. et pourquoi ne le serait-il pas ? Jetez un oeil sur tout le travail que vous devriez faire « manuellement ». Tout d'abord il n'existe aucun ide approprié à ce type de travail, ainsi vous ressortez votre fidèle vieux bloc-notes, si vous uti-

lisez Windows ou emacs, si vous utilisez linux et saisissez votre grammaire. Bien sûr il n'y a pas de coloration syntaxique, d'auto-completion, configuration du contenu ou n'importe quel de ces assistants présents dans vos habituels *ide java/c++*. Ajoutez à ceci le fait que vous devez construire manuellement le fichier de grammaire chaque fois que vous commettez une erreur, qu'il n'y a absolument aucune manière appropriée de déboguer la grammaire, que vous devrez aussi coder l'analyseur/lexer dans votre langage de programmation préféré (par exemple *java/c/c++*), après cela vous obtenez tous les avantages de votre ide, bien que vous pouvez avoir à écrire quelques milliers de lignes de codes supplémentaires !

Mais tout ceci est sur le point de changer avec ANTLR Studio, un IDE pour ANTLR qui s'intègre complètement avec l'environnement de développement Eclipse. ANTLR Studio a été construit

Sur l'auteur :

L'auteur est le créateur de ANTLR Studio et le fondateur de Placid Systems.

Pour contacter l'auteur :
pdeva@placidsystems.com

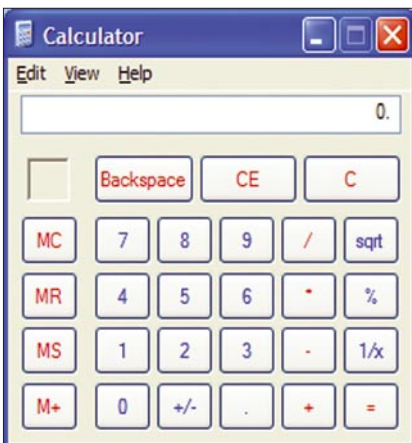


Figure 1. Calculatrice de Windows

en gardant à l'esprit la problématique du développement de grammaire en étant déconnecté de l'environnement principal de développement, ainsi il se donne beaucoup de peine pour fournir une grande expérience intégrée. Et pour le prouver nous allons construire une application simple utilisant ANTLR dans ANTLR Studio.

La calculatrice

Bien qu'elle semble très agréable au premier coup d'oeil, la calculatrice Windows possède de nombreux défauts. Premièrement, si vous trouvez écrite quelque part une expression mathématique, vous ne pouvez pas simplement la copier/coller dans la calculatrice. Vous devez recopier à la main l'expression dans son intégralité. Ainsi si c'est une expression longue, vous pourriez parfois simplement oublier à quel endroit vous en êtes, parce que la calculatrice ne vous montre pas toute l'expression, elle n'indique que le dernier résultat de l'expression que vous avez saisie jusqu'à présent. Par conséquent si vous entrez par erreur un mauvais chiffre en recopiant cette longue expression, vous ne pourrez jamais le découvrir et votre résultat sera faux sans même que vous le sachiez.

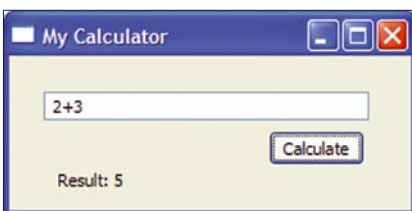


Figure 2. Notre calculatrice, d'interface simple qui vous permet de voir d'un seul coup l'ensemble de l'expression

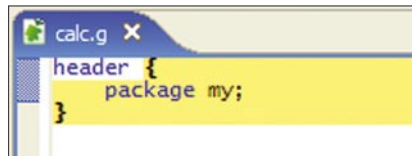


Figure 3. ANTLR Studio met automatiquement le nom du paquet lorsque vous créez un nouveau fichier de grammaire

Nous allons donc créer dans Eclipse une nouvelle calculatrice en utilisant ANTLR Studio, comme indiqué en Figure 2. Vous pouvez simplement y saisir ou copier/coller une expression mathématique entière dans la boîte de texte puis appuyer sur le bouton calculer pour voir la réponse. Cette calculatrice ne possède aucun des problèmes indiqués précédemment. Pour garder les choses simples dans ce tutoriel, nous ne permettrons que les opérateurs de multiplication et d'addition à l'intérieur des expressions.

Lorsque vous cliquez sur le bouton calculer, la calculatrice analyse l'expression de la boîte de texte et en crée un arbre. Elle parcourt alors cet arbre pour évaluer l'expression. La portion de code java de cette app est insignifiante et sans pertinence pour notre tutoriel, ainsi nous nous concentrerons uniquement sur la grammaire ANTLR, qui contient le corps du code.

L'analyseur

Dans l'explorateur de paquets, effectuer un clic droit sur le paquet où vous voulez placer la grammaire et sélectionnez *New->ANTLR Grammar*. Saisissez le nom du fichier de grammaire dans la boîte de texte qui apparaît et appuyer sur OK. Un nouveau fichier de grammaire est créé pour vous avec la section d'en-tête contenant la déclaration du paquet mise automatiquement. Saisissez l'analyseur de grammaire suivant dans l'éditeur. Notez que la caractéristique *TypeOnce feature* de l'éditeur montre automatiquement les complétions sans nécessairement appuyer sur



Figure 4. L'icône bleue de morceau de puzzle appelle l'assistant de création d'analyseur lexical

Listing 1. L'analyseur

```
class CalcParser extends Parser;
options{
    buildAST=true;
}
expr
    : mexpr (PLUS^ mexpr)*
    ;
mexpr
    : a:atom (STAR^ atom)*
    ;
atom: INT
    ;
```

[Ctrl+Space] ce qui rend la saisie de la grammaire très rapide.

Si vous connaissez la syntaxe ANTLR, ceci devrait vous sembler être une grammaire assez simple. La première ligne déclare notre analyseur. C'est juste comme une déclaration de classe en Java. En fait cela sera généré dans une classe de même nom (*CalcParser*) lorsque ANTLR compilera cette grammaire. Puis nous déclarons les options de cette règle qui indiqueront à ANTLR de créer un Arbre de Syntaxe Abstrait (Abstract Syntax Tree) ou AST utilisant l'analyseur.

Les quelques lignes suivantes définissent une règle *expr* qui contient une *mexpr* suivie optionnellement par 0 ou plusieurs marques + et une autre *mexpr*, suivie par point virgule. Le symbole * représente 0 ou plusieurs +. De même, la règle *mexpr* définit un *atom*. Un *atom* est uniquement défini pour être de type INT, ce qui représente la plupart des entiers. Nous définirons ces types plus tard dans notre analyseur lexical.

Au cas où vous vous demanderiez, le drôle de symbole ressemblant à ^ dans 2 des règles est utilisé pour la construction de l'arbre. Il indique à ANTLR que la marque doit être faite à la racine de l'arbre. Ainsi par exemple, l'arbre formé pour la règle *expr* aura PLUS à la racine, avec *mexpr* en tant qu'enfants.

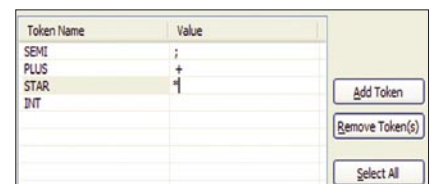


Figure 5. ANTLR Studio met automatiquement les noms des marques utilisées dans votre analyseur, ainsi vous devez juste remplir les blancs

Listing 2. L'analyseur d'arbre

```

expr returns [int r]
{
    int a,b;
    r=0;
}
: #(PLUS a=expr b=expr)
  {r = a+b;}
| #(STAR a=expr b=expr)
  {r = a*b;}
| i:INT {r = Integer.parseInt
        (i.getText());}
;

```

L'analyseur lexical

Maintenant au lieu de saisir l'analyseur lexical nous même, nous utiliserons l'assistant de création d'analyseur lexical super sympa de ANTLR Studio pour le faire à notre place. Cliquer sur l'icône bleue de puzzle *jigsaw* sur le coin supérieur droit de la barre d'outils.

Sur la première page, saisissez le nom du lexer, appelons le *CalcLexer*. Cliquer sur Next. Vous verrez que ANTLR Studio a déjà rempli les noms des marques que nous utilisons dans notre analyseur. Vous devez juste fournir ce qu'ils représentent. Par exemple saisir ; à côté de SEMI, + à côté de PLUS et * à côté de STAR. Enlever la marque INT.

Nous le traiterons dans la page suivante Cliquer sur Next. Sur la page *Add Identifieur*, cliquer sur *Add*. Maintenant saisissez INT pour le nom de la marque. Sélectionnez *One or more* et *Any Digit*. Là, nous avons défini un identificateur pour reconnaître les ints. Vous pouvez même saisir un nombre dans la boîte en dessous pour tester l'identificateur.

Figure 6. Vous ne devez plus saisir manuellement des règles complexes pour définir des identificateurs

Listing 3. Code pour appeler l'analyseur

```

try {
    StringReader reader = new
    StringReader("2+3;");
    CalcLexer lexer = new
    CalcLexer(reader);
    CalcParser parser = new
    CalcParser(lexer);
    // Analyse l'expression en entrée
    parser.expr();
    CommonAST t = (CommonAST)parser.getAST();
    // Affiche l'arbre résultant en notation LISP
    System.out.println(t.toStringTree());
    CalcTreeWalker walker =
    new CalcTreeWalker();
    // Traverse the tree created by the parser
    int r = walker.expr(t);
    return r;
}
catch (TokenStreamException e) {
    System.err.println("exception: "+e);
}
catch (RecognitionException e) {
    System.err.println("exception: "+e);
}
}

```

Cliquez sur *Next* et cocher la case *Add token to represent WhiteSpace*. Saisissez WS comme nom de la marque et sélectionnez *Ignore WhiteSpace in my language*.

Maintenant comme nous n'avons pas besoin de manipuler les commentaires et chaînes de notre langage, vous pouvez avancer et cliquer sur *Next* lors des 2 prochaines étapes.

Et notre analyseur lexical est complet, sans avoir saisi un seul mot dans notre éditeur ! Cela n'était-il pas facile ?

Figure 7. L'assistant de création d'analyseur lexical rend la manipulation des espaces dans votre langage aussi simple que de cliquer sur des cases à cocher !

L'analyseur d'arbre

Maintenant nous allons saisir notre analyseur d'arbre pour parcourir l'arbre résultant et effectuer le calcul réel sur les noeuds. Saisissez ce qui suit dans l'éditeur.

L'analyseur d'arbre ne nécessite qu'une seule règle. Il retourne un entier *r* contenant le résultat du calcul. Premièrement nous définissons 2 ints *a* & *b*. Le reste de la règle définit comment parcourir l'arbre et exécuter les calculs. Par exemple, La première alternative indique à l'analyseur d'arbre que l'arbre contient le symbole PLUS suivi de 2 expressions. Nous avons mis les résultats des deux *exprs* dans les variables *a* et *b* respectivement et mis le résultat dans *r* en les ajoutant. Notez que tout ce qui se trouve entre accolades *{}* représente du code java. ANTLR mettra tout ce qui entre ces accolades exactement comme c'est fait dans les méthodes qu'il produit pour chaque règle.

Le code Java

Et notre grammaire est complète ! Maintenant nous allons écrire le code pour appeler l'analyseur et exécuter le calcul dans le même gestionnaire du bouton *Calculer*.

Donc là, maintenant nous avons une calculatrice qui est plus agréable que la plupart des calculatrices. Essayez vous-même d'ajouter le support des opérations mathématiques. ■