SableCC : un générateur d'analyseurs lexicaux et syntaxiques en Java

Alexis Nasr
Carlos Ramisch
Manon Scholivet
Franck Dary

Compilation – L3 Informatique
Département Informatique et Interactions
Aix Marseille Université

SableCC

- SableCC est un générateur d'analyseurs syntaxiques et lexicaux.
- Il prend en entrée un fichier de spécification qui décrit le lexique et la grammaire d'un langage L.
- Il produit un programme en Java qui lit un programme P écrit en langage L et produit l'arbre de dérivation de P, si P est syntaxiquement correct.
- Le programme produit est appelé analyseur syntaxique.

Exemple

```
Package postfix;

Tokens
 number = ['0' .. '9']+;
 plus = '+';
 minus = '-';
 mult = '*';
 div = '/';
 mod = '%';
 l_par = '(';
 r_par = ')';
 blank = (' ' | 13 | 10)+;

Ignored Tokens
blank;
```

```
Productions
 expr =
  {factor} factor |
  {plus} expr plus factor |
  {minus} expr minus factor;

 factor =
  {term} term |
  {mult} factor mult term |
  {div} factor div term |
  {mod} factor mod term;


 term =
  {number} number |
  {expr} l_par expr r_par;
```

Stucture du fichier `postfix.grammar`

Deux parties principales :

- Analyseur lexical, commence par le mot clef `Tokens`
  - constitué de triplets `unité lexicale = expression régulière`
  - les `unités lexicales` sont les terminaux de la grammaire
- Analyseur syntaxique, commence par le mot clef `Productions`
  - constitué de règles de grammaires de la forme :
    `symbole non terminal = suite de symboles`
  - Lorsqu'un symbole non terminal peut se réécrire de différentes manières, chacune des alternatives est nommée :

```
expr =
 {factor} factor |
 {plus}   expr plus factor |
 {minus}  expr minus factor;
```

- L'identifiant qui suit le mot clef `Package`, indique le nom du package `Java` qui va être créé par `SableCC`.

Generation de l'analyseur

```
$ java -jar sablecc.jar postfix.grammar

-- Generating parser for postfix.grammar
Adding productions and alternative of section AST.
Verifying identifiers.
Verifying ast identifiers.
Adding empty productions and empty alternative transformation if necess
Adding productions and alternative transformation if necessary.
computing alternative symbol table identifiers.
Verifying production transform identifiers.
Verifying ast alternatives transform identifiers.
Generating token classes.
Generating production classes.
Generating alternative classes.
Generating analysis classes.
Generating utility classes.
Generating the lexer.
 State: INITIAL
 - Constructing NFA.
 - Constructing DFA.
 - resolving ACCEPT states.
Generating the parser.
```

Ce qui a été produit

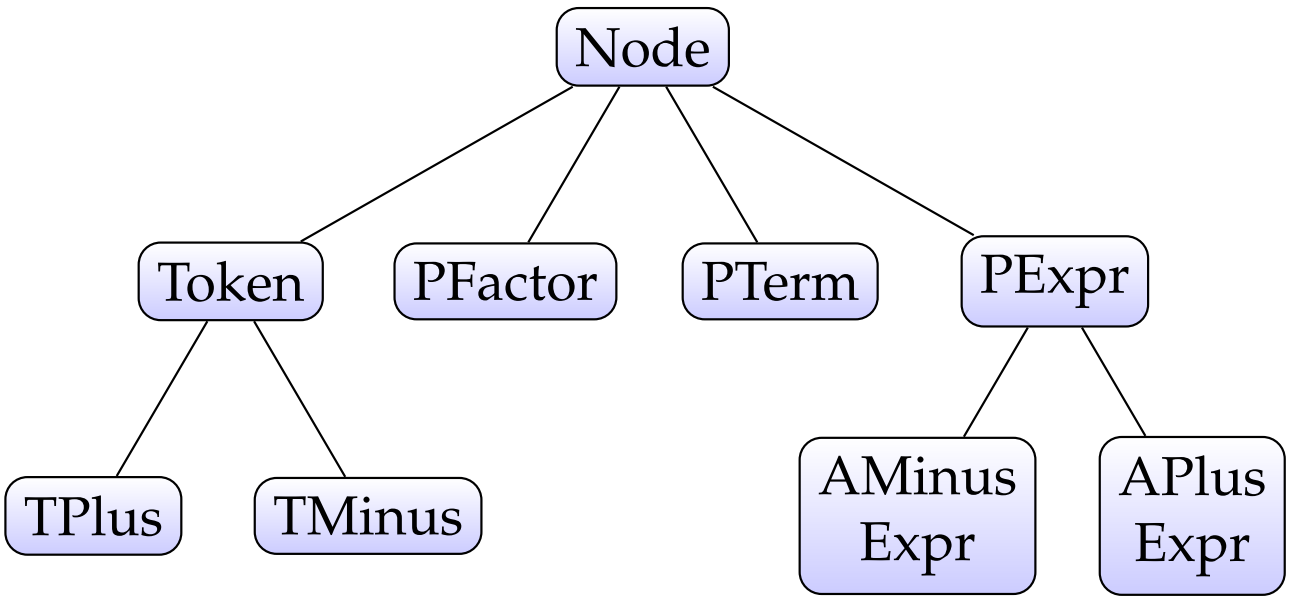
Un répertoire du nom du package, contenant lui même quatre répertoires

- `lexer` contient le code de l'analyseur lexical
- `parser` contient le code de l'analyseur syntaxique
- `node` contient les classes correspondant aux symboles et aux règles de la grammaire
- `analysis` contient des fonctions de parcours des arbres de dérivation

Dans le répertoire `node`

- **Une classe par terminal (prefixée par `T`)**
  `TBlank`, `TDiv`, `TLPar`, `TMinus`, `TMod`, `TMult`, `TNumber`, `TPlus`, `TRPar`
- **Une classe par non terminal (préfixée par `P`)**
  `PExpr`, `PFactor`, `PTerm`
- **Une classe par règle (préfixées par `A`)**
  `ADivFactor`, `AExprTerm`, `AFactorExpr`, `AMinusExpr`, `AModFactor`, `AMultFactor`, `ANumberTerm`, `APlusExpr`, `ATermFactor`
  Convention : le nom de la classe correspondant à la règle :
  `expr = {plus} expr plus factor`
  est : `APlusExpr`.
- **Deux classes abstraites**
  `Node`, `Token`
- **Deux interfaces**
  `Switchable`, `Switch`
- **Autre**
  `EOF`, `InvalidToken`, `Start`

Hiérarchie (partielle) des classes



Le répertoire `analysis`

Permet de parcourir les arbres de dérivation construits par l'analyseur à l'aide de visiteurs.

Quatre classes

- `Analysis`
- `AnalysisAdapter`
- `DepthFirstAdapter`
- `ReversedDepthFirstAdapter`

Parcours de l'arbre à l'aide d'un visiteur

- Chaque type de nœud `X` définit la méthode

```
public void apply(Switch sw){
    ((Analysis) sw).caseX(this);}
```

- L'interface `Analysis` définit les méthodes `CaseX(.)` pour tous les types `X` de nœuds

```
void caseX(X node);
```

- La classe `DepthFirstAdapter` implémente l'interface `Analysis` et effectue un parcours en profondeur de l'arbre de dérivation.

```
public void caseAPlusExpr(APlusExpr node){
  inAPlusExpr(node);
  if(node.getExpr() != null) node.getExpr().apply(this);
  if(node.getPlus() != null) node.getPlus().apply(this);
  if(node.getFactor()!= null) node.getFactor().apply(this)
  outAPlusExpr(node);
  }
```

Affichage en mode postfixé

```
class Translation extends DepthFirstAdapter
{
 public void caseTNumber(TNumber node)
 {System.out.print(node); }

 public void outAPlusExpr(APlusExpr node)
 {System.out.print(node.getPlus()); }

 public void outAMinusExpr(AMinusExpr node)
 {System.out.print(node.getMinus()); }

 public void outAMultFactor(AMultFactor node)
 {System.out.print(node.getMult()); }

 public void outADivFactor(ADivFactor node)
 {System.out.print(node.getDiv()); }

 public void outAModFactor(AModFactor node)
 {System.out.print(node.getMod()); }
}
```

Le traducteur infixe → postfixe

```
public class Compiler{
 public static void main(String[] arguments){
  try{
   System.out.println("Type an arithmetic expression:");
   // Create a Parser instance.
   Parser p =
    new Parser(
    new Lexer(
    new PushbackReader(
    new InputStreamReader(System.in), 1024)));
   // Parse the input.
   Start tree = p.parse();
   // Apply the translation.
   tree.apply(new Translation());
  }
  catch(Exception e){
   System.out.println(e.getMessage());
  }
 }
}
```

Compilation et exécution

```
$ javac Compiler.java
$ java Compiler
Type an arithmetic expression:
(45 + 36 / 2) * 3 + 5 * 2
45 36 2 / + 3 * 5 2 * +
```

TP1

- Ecriture de la grammaire du langage L sous la forme d'un fichier de spécification `SableCC`.
- Production de l'analyseur syntaxique.
- Test de l'analyseur syntaxique sur des exemples fournis.