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

Alexis Nasr
Franck Dary
Pacôme Perrotin

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

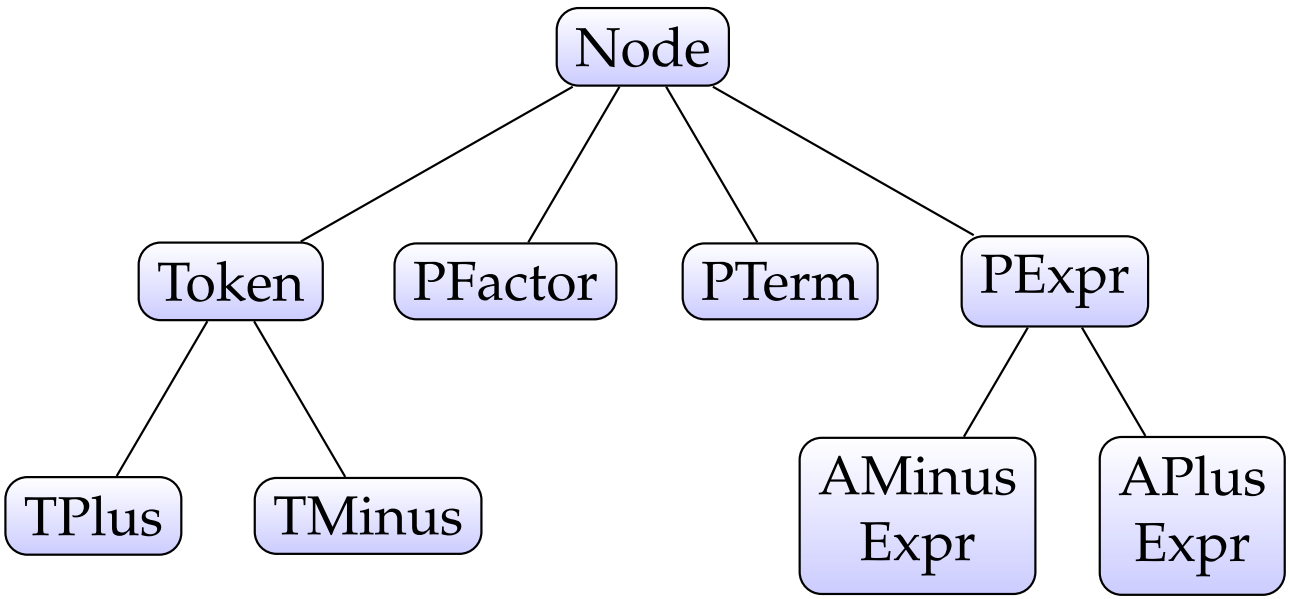
Un répertoire du nom du package, contenant lui même quatre répertoires

- `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 * +
```

Quelques détails sur la spécification de l'analyseur lexical

- La spécification de l'analyseur lexical est composé de trois parties.
  Chaque partie commence par un mot clef :
  1. `Helpers` Permet de définir des macros afin de simplifier la description des des unités lexicales.
  2. `States` Permet de définir des *états*. La reconnaissance de certaines unités lexicales peut alors être limitée à des états particuliers.
  3. `Tokens` Permet de définir les unités lexicales.

Helpers

- Un `helper` permet d'associer un nom à une expression régulière afin de l'utiliser par la suite, dans d'autres `helpers` ou dans des unités lexicales (`tokens`).
- Exemples :

```
lettre = [['a' .. 'z'] + ['A' .. 'Z']];
chiffre = ['0' .. '9'];
alpha = [lettre + ['_' + '$']];
alphanum = [lettre + chiffre];
```

Les unités lexicales (`Tokens`)

- La partie `Tokens` est le cœur de l'analyseur syntaxique, c'est là que sont définies les unités lexicales.
- Les noms de `tokens` sont des chaînes composées de caractères minuscules, de chiffres et du caractère `_`. Ils doivent commencer par une lettre.
- Exemple de définition des trois unités lexicales `ecrire`, `nombre` et `identif`

```
ecrire = 'ecrire';
nombre = chiffre+;
identif = alpha alphanum*;
```

Règle du match le plus long

- Lorsque plusieurs expression régulières reconnaîssent une partie du texte à analyser, c'est l'expression régulière qui reconnaît la suite de caractères la plus longue qui l'emporte.
- Exemple : si on définit les deux unités lexicales

  ```
  ul1 = 'ab';
  ul2 = 'abc';
  ```

  et qu'on analyse l'entrée `abcd`, c'est `ul2` qui l'emporte car elle reconnaît trois caractères alors que `ul1` n'en reconnaît que deux.

Ordre de définition des `tokens`

- Lorsque plusieurs expression régulières reconnaîssent un segment du texte à analyser de même longueur, c'est la première (dans l'ordre d'apparition dans le fichier de spécification) qui l'emporte
- Exemple (important !) : si on définit les deux unités lexicales :

```
identif = alpha alphanum*;
ecrire = 'ecrire';
```

- alors toute occurrence de `ecrire` dans le texte à analyser sera analysée comme un `identif` (ce qui n'est probablement pas ce que l'on souhaite).

Les états (`states`)

- Certaines unités lexicales ne peuvent être reconnues que lorsque l'analyseur lexical se trouve dans un état particulier.
- On définit l'ensemble des états dans la partie `States`
- Exemple :

```
States
  premier,
  second;
```

- Le premier état est l'état initial de l'analyseur.
- Chaque unité lexicale peut être associée à un ou plusieurs états.

```
{premier} ul1 = 'a';
{second}  ul2 = 'b';
{premier, second}  ul3 = 'c';
```

- Le changement d'état est provoqué par la reconnaissance de ceraines unités lexicales

```
{premier->second} ul4 = 'd';
{second->premier} ul5 = 'e';
```

Syntaxe des expressions régulières

- Les expression régulières sont composées de caractères ou d'ensembles de caractères pouvant être combinés entre eux par les opérations de :
  - concaténation `'ab'`
  - union `'a'|'b'`
  - étoile de Kleene `'a'*`
  - plus de Kleene `'a'+`
  - optionnalité `'a'?`
- Un caractère est représenté sous la forme :
  - d'un caractère entre `"` : `'a'`
  - de son code `unicode` : `13` (retour chariot)
  - de son code `unicode` en hexadécimal : `0xf`
- Un ensemble de caractères est représenté sous la forme :
  - d'un intervalle : `['a'..'z]`
  - l'union d'ensembles de caractères `['a'..'z'] + ['A'..'Z']`
  - la différence d'ensembles de caractères `[0..0xffff] - ['0'..'9']`

Quelques détails sur la spécification de l'analyseur syntaxique

- La spécification de l'analyseur syntaxique est composé de deux parties.
  Chaque partie commence par un mot clef :
  1. `Ignored tokens` Permet de spécifier certains `tokens` qui ne seront pas vus de l'analyseur syntaxique.
  2. `Productions` Permet de définir les règles de la grammaire.

Ignored tokens

- Les `Ignored tokens` permettent de simplifier les règles de grammaire.
- Exemple

```
Tokens
espaces = (' ' | 13 | 10)+;
commentaire= '#' [[0..0xffff]-[10+13]]* (10|13|10 13);

Ignored Tokens
espaces, commentaire;
```

Les règles

- Les règles se présentent sous la forme d'une partie gauche et d'une partie droite séparés par le caratère = :

  ```
  s = a s b ;
  ```

- Les éléments de la partie droite sont :
  - des non terminaux (symboles apparaissants en partie gauche d'au moins une règle),
  - des `tokens`,
- La partie gauche de la première règle est l'axiome de la grammaire.
- Tout comme les `tokens`, les symboles non terminaux sont des chaînes composées de caractères minuscules, de chiffres et du caractère _. Ils doivent commencer par une lettre.

Nommage des règles et des symboles

- Lorsqu'un non terminal peut s'écrire à l'aide de différentes règles (appelées alternatives), chaque règle doit être nommée :

```
exp4 =
{fois} exp4 fois exp5 |
{divise} exp4 divise exp5 |
{exp5} exp5 ;
```

- Lorsqu'un symbole apparaît plus d'une fois dans une partie droite de règle, chaque occurrence doit être nommée :

```
dec_fonc = id po [param]:ldv pf [varloc]:ldv bloc
;
```

- `SableCC` se sert des noms de symboles et d'alternatives dans le code qu'il génère.

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.