Génération de code

String Template

Claude Moulin

Université de Technologie de Compiègne

Printemps 2013

Sommaire

Génération

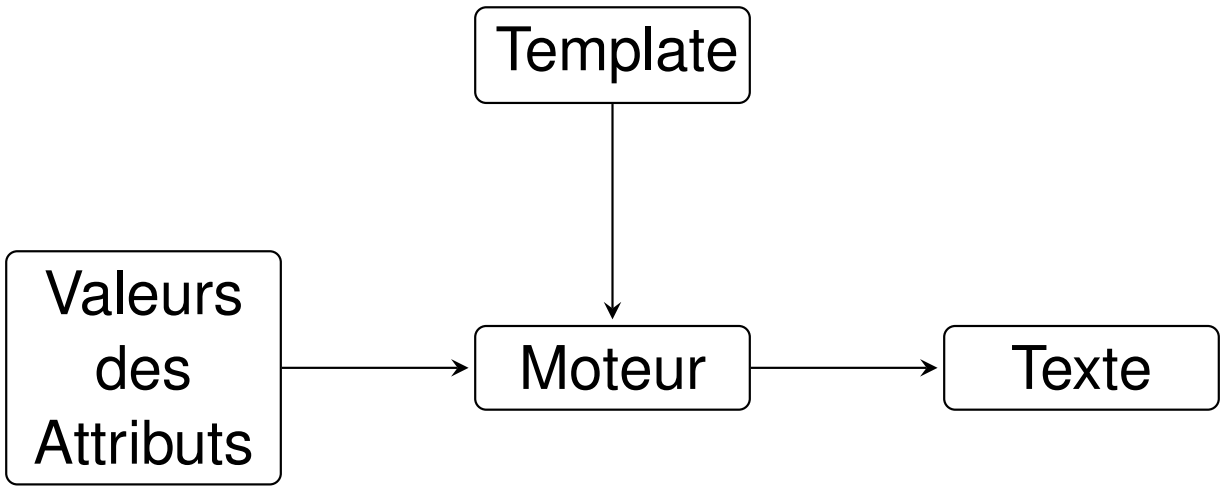
- La génération de code utilise généralement des patrons de chaînes qui permettent de :
  - structurer le fichier de sortie,
  - factoriser et paramétrer les appels générant des portions similaires,
  - séparer le générateur (la logique) et le contenu généré.

String template

- Un template est un document texte où figurent des attributs auxquels on peut attacher des valeurs. Les délimiteurs d'attribut sont ici `<` et `>`.
- Exemple de template :
  `SELECT <column> FROM <table>;`
- Attributs : `column` et `table`
- Valeurs des attributs :
  `<column>` = `name` et `<table>` = `User`
- Résultat : `SELECT name FROM User;`
- Un template peut avoir un ou plusieurs attributs.

Fonctionnement

- Un moteur de template est simplement un générateur de code qui produit du texte en utilisant des templates.



Utilisation

- `http://www.stringtemplate.org/`
  site de AntLR (ST version 4)
- `http://www.antlr.org/wiki/display/ST4/StringTemplate+4+Documentation`
- Utiliser : `antlr-3.3-complete.jar` et `ST-4.0.jar`

```
ST simple = new ST("SELECT <column> FROM <table>;");
simple.add("column", "name");
simple.add("table", "User");
System.out.println(simple.render());
```

Caractéristiques

- Groupement de templates dans un ou plusieurs fichiers.
- Utilisation d'une liste comme valeur d'attribut.
- Utilisation d'un objet comme valeur d'attribut.
- Réutilisation de template dans un autre template.
- Utilisation d'une table de valeurs.

Sommaire

Groupe de templates

- Il est intéressant de grouper différents templates dans un même fichier.

```
sql(column,table) ::= "SELECT <column> FROM <table>;"
constructor(name) ::= "public <name>() {}"
```

- On crée un groupe de templates :

```
String stfname = "filename.stg";
STGroup group = new STGroupFile(stfname);
ST st = group.getInstanceOf("sql");
st.add("column", "name");
st.add("table", "User");
```

Délimiteurs de templates

- Template écrit sur une ligne : délimiteurs de template " et "
- Template écrit sur une ligne contenant des " : délimiteurs << et >>
- Template écrit sur plusieurs lignes : délimiteurs << et >>

```
class(name,prop) ::=
<<
public class <name> {
   ...
}
>>
```

Attributs multivalués - 1

```
param(val) ::= <<x = f(<val; separator=",">)>>
```

- On indique les séparateurs : `separator=","`.
- On peut attribuer plusieurs fois une valeur.

```
STGroup group = new STGroupFile(...);
ST st = group.getInstanceOf("param");
for (int i = 0; i < 5 ; i++)
 st.add("val", 2 * i);
System.out.println(st.render());
```

- Résultat : `x = f(0,2,4,6,8)`

Attributs multivalués - 2

```
param(val) ::= <<x = f(<val; separator=",">)>>
```

- L'attribut peut être de type tableau, liste.

```
STGroup group = new STGroupFile(...);
ST st = group.getInstanceOf("param");
st.add("val", new int[]{1,10,20});
System.out.println(st.render());
```

- Résultat : `x = f(1,10,20)`

Propriétés des attributs

- Un attribut peut être un objet et un patron peut utiliser les propriétés de l'objet.

```
class(name,prop) ::=
<<
public class <name> {
  <prop.type> <prop.id>;
}
>>
```

Code

```
STGroup group = new STGroupFile(...);
ST st = group.getInstanceOf("class");
Property p = new Property("realPart", "float");
st.add("name", "Complex");
st.add("prop", p);
System.out.println(st.render());
```

- Résultat :

```
public class Complex {
  float realPart;
}
```

Réutilisation d'un template

- Un template peut utiliser un autre template pour décrire un attribut.

```
class(name,prop) ::=
<<
public class <name> {
  <prop:proptp()>
}
>>
proptp(prop) ::= <<  <prop.type> <prop.id>;  >>
```

Code

```
STGroup group = new STGroupFile(...);
ST st = group.getInstanceOf("class");
Property p = new Property("realPart", "float");
st.add("name", "Complex");
st.add("prop", p);
System.out.println(st.render());
```

- Résultat :

```
public class Complex {
 float realPart;
}
```

Liste d'objets et Réutilisation d'un template

- Un template peut utiliser un autre template et décrire les éléments d'un attribut liste.

```
class(name,prop) ::=
<<
public class <name> {
 <prop:proptp(); separator="\n">
}
>>
proptp(prop) ::= << <prop.type> <prop.id>;  >>
```

Code

```
STGroup group = new STGroupFile(...);
ST st = group.getInstanceOf("class");
Property p1 = new Property("realPart", "float");
Property p2 = new Property("imPart", "float");
st.add("name", "Complex");
st.add("prop", new Property[]{p1, p2});
System.out.println(st.render());
```

- Résultat :

```
public class Complex {
 float realPart;
 float imPart;
}
```

Création de XML

- Utiliser les autres délimiteurs d'attribut `$` et `$`.

```
xml(contact) ::=
<<
 <contact name="$contact.lastName$"
          firstname="$contact.firstName$"/>
>>

nxml(contacts) ::=
<<
<contacts>
  $contacts:xml(); separator="\n"$
</contacts>
>>
```

Code

```
STGroup group = new STGroupFile(...);
group.delimiterStartChar = '$';
group.delimiterStopChar = '$';
ST st = group.getInstanceOf("nxml");
ArrayList<Contact> l = getContacts();
st.add("contacts", l);
System.out.println(st.render());
```

Résultat

```
<contacts>
  <contact name="Tiger" firstname="SC"/>
  <contact name="Bach" firstname="Seb"/>
</contacts>
```

Utilisation d'une table de conversion

- A la place de la valeur d'un attribut, il est possible d'insérer une valeur de substitution lue dans une table.

```
proptp(prop) ::=
<<
 <prop.type> <prop.id> = <typeInitMap.(prop.type)>;
>>
typeInitMap ::= [
  "int"     : "0",
  "float"   : "0.0f",
  "boolean" : "false",
   default  : "null"
]
```

  - option : `default : key` permet de réécrire la clé comme valeur inchangée.

Code

```
STGroup group = new STGroupFile(...);
ST st = group.getInstanceOf("class");
Property p1 = new Property("realPart", "float");
Property p2 = new Property("imPart", "float");
st.add("name", "Complex");
st.add("prop", new Property[]{p1, p2});
System.out.println(st.render());
```

- Résultat :

```
public class Complex {
 private float realPart = 0.0f;
 private float imPart = 0.0f;
}
```

Sommaire

Classe simple

- On désire générer une classe ayant une ou plusieurs propriétés, les accesseurs sur ces propriétés et le constructeur correspondant.
- On considère qu'on dispose d'une classe ayant trois champs pour décrire les informations d'une propriétés :
  - le nom d'une propriété (id)
  - le nom de la propriété dont le premier caractère est en majuscule (access) ; utilisé dans l'écriture des accesseurs.
  - le type (simple) de la propriété.

Patron d'une Classe simple

```
class(name,package,prop) ::=
<<
package <package>;
public class <name> {
 <prop:proptp(); separator="\n">
 <name:defconstructor()>
 <constructor(name,prop)>
 <prop:accessor(); separator="\n">
}
>>
```

Code

```
STGroup group = new STGroupFile(...);
ST st = group.getInstanceOf("class");
Property p1 = new Property("realPart", "float");
Property p2 = new Property("imPart", "float");
st.add("package", "model");
st.add("name", "Complex");
st.add("prop", new Property[]{p1, p2});
try {
 FileWriter fw = new FileWriter("src/model/Complex.java");
 fw.write(st.render());
 fw.close();
} catch (IOException e) {
 e.printStackTrace();
}
```

Résultat

```
package model;
public class Complex {
 private float realPart = 0.0f;
 private float imPart = 0.0f;
 public Complex() {}
  public Complex(float realPart,float imPart) {
   this.realPart = realPart;
   this.imPart = imPart;
  }
  ...
  public float getImPart() {
   return imPart;
  }
  public void setImPart(float imPart) {
   this.imPart = imPart;
  }
}
```

Templates

```
proptp(prop) ::=
<<
private <prop.type> <prop.id> = <typeInitMap.(prop.type)>;
>>
accessor(prop) ::=
<<
 public <prop.type> get<prop.access>() {
  return <prop.id>;
 }
 public void set<prop.access>(<prop.type> <prop.id>) {
  this.<prop.id> = <prop.id>;
 }
>>
typeInitMap ::= [
  "int"     : "0",
  "float"   : "0.0f",
  "boolean" : "false",
   default  : "null"
]
```

Template Constructeur

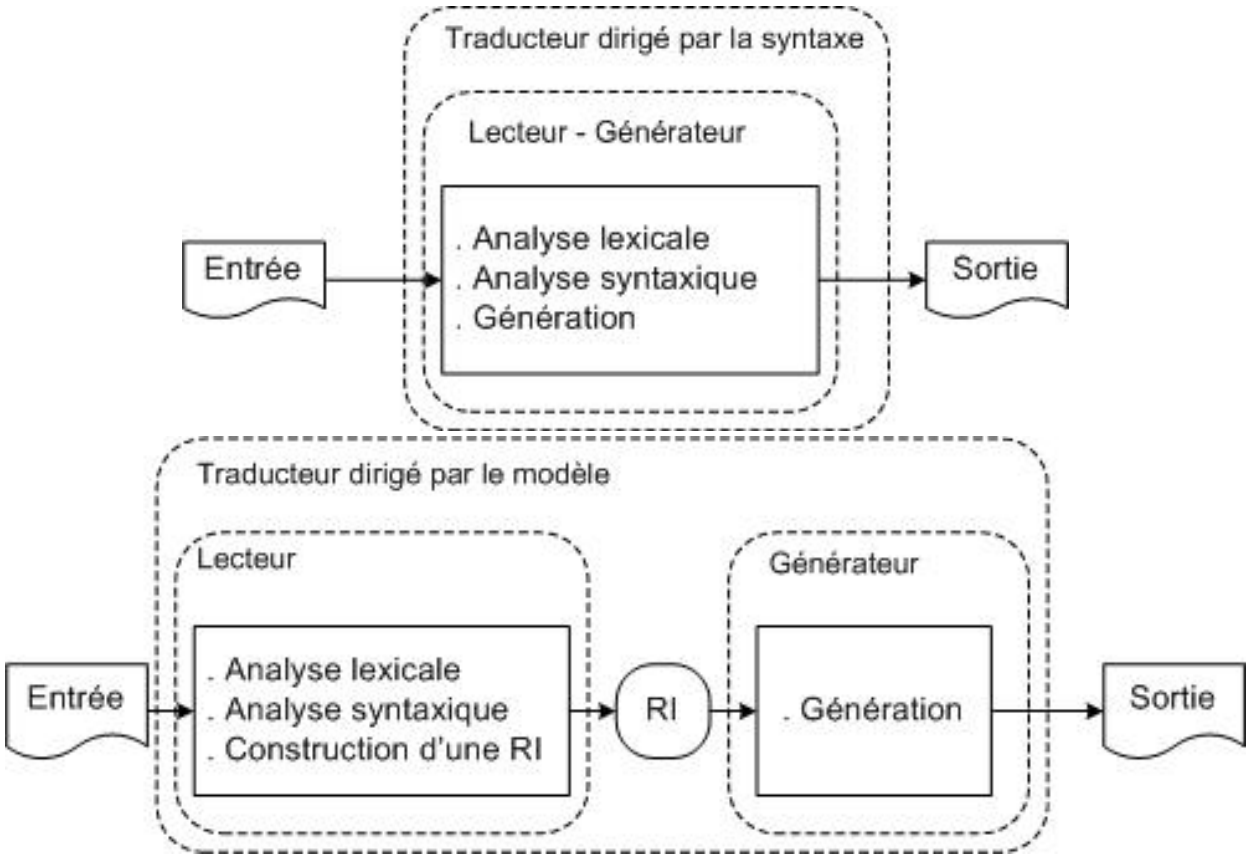
```
defconstructor(name) ::= "public <name>() {}"

constructor(name,prop) ::= <<
  public <name>(<prop:{x| <x.type> <x.id>}; separator=",">) {
    <prop:{x| this.<x.id> = <x.id>;}; separator="\n">
  }
>>
```

- `{x| <x.type> <x.id>}` est un patron anonyme qui évite d'écrire un patron séparé.

Sommaire

Situation


Traducteur dirigé par la syntaxe
Lecteur - Générateur
. Analyse lexicale
. Analyse syntaxique
. Génération
Entrée
Sortie
Traducteur dirigé par le modèle
Lecteur
. Analyse lexicale
. Analyse syntaxique
. Construction d'une RI
RI
Générateur
. Génération
Entrée
Sortie

Objectif

- Emettre en sortie une chaîne traduisant le programme source
  - pendant l'analyse syntaxique ;
  - A partir d'une représentation intermédiaire.
- Un groupe de template représente la spécification formelle du code produit.
- Les template sont plus faciles à écrire qu'une collection de création de chaîne dans les actions d'une grammaire.
- Séparer le générateur de la logique de génération de code est préférable.

Principe

- On crée les template dans un fichier externe.
- On associe un groupe de template au visiteur d'arbre.
- On insère dans le visiteur des instructions permettant de donner des valeurs aux attributs de string template.
  - Elles instancient les string template et donnent les valeurs correspondantes aux attributs.
- L'exécution du visiteur sur un arbre représentant un programme permet de générer le code en sortie dans l'attribut string template.

Exemple : source

- On désire créer une classe Java dont une méthode sera la traduction en instructions Java d'un programme Logo.

```
AV 100
TD 90
AV 100
```

Exemple : résultat

```
package logojava;
public class JavaLogo {
  Traceur traceur = new Traceur();
  public JavaLogo() {}
  public void initialiseTraceur(java.awt.Graphics g) {
    traceur.setGraphics(g);
  }
  public void run() {
     traceur.avance(100);
     traceur.td(90);
     traceur.avance(100);
  }
}
```

Analyse du résultat

```
package logojava;
public class JavaLogo {
...
  public void run() {
     traceur.avance(100);
     traceur.td(90);
     traceur.avance(100);
  }
}
```

Seule la partie suite d'instructions à l'intérieur de la méthode run est variable et dépend du programme Logo

Exemple

La règle :

```
instruction :
  'av' INT # av
| 'td' INT # td
;
```

Avec le programme : `AV 100 TD 90 AV 100`
produit en sortie après trois appels la liste d'instructions :

```
traceur.avance(100);
traceur.td(90);
traceur.avance(100);
```

Patrons instructions

Utilisation d'un groupe de patrons de chaîne externe contenant le patron `commande` ayant deux attributs `com` et `x` pour le nom de la méthode et la valeur du paramètre entier.

```
instructions(l) ::=<<
$l; separator = "\n"$
>>

commande(com,x) ::= "traceur.$com$($x$);"
```

Patrons général

Utilisation d'un patrons de chaîne externe contenant le patron `prog` ayant l'attribut `liste` la liste des instruction dans la méthode run.

```
prog(liste) ::=<<
package logojava;
public class JavaLogo {
   ...
   public void run() {
      $instructions(liste)$
   }
}
>>
```

Solution : patron

- Chaque méthode du visiteur de l'arbre de dérivation doit créer un template :
- `visitProgramme` (axiome) : rien
- `visitAv` et `visitTd` : synthétise la ligne de la commande
- `liste_instructions` : ajoute chaque instruction synthétisée à une liste destinée à l'attribut du template de la classe (prog).

Initialisation

```
public LogoST4Visitor() {
 super();
 group = new STGroupFile(TEMPLATE);
 group.delimiterStartChar = '$';
 group.delimiterStopChar = '$';
 stprogramme = group.getInstanceOf("prog");
}
```

Visiteur : av

```
public Integer visitAv(AvContext ctx) {
 ST st = group.getInstanceOf("commande");
 st.add("com","avance");
 st.add("x",ctx.INT().getText());
 String s = st.render();
 setValue(ctx, s);
 return 0;
}
```

Visiteur : Liste_instructions

```
public Integer visitListe_instructions
        (Liste_instructionsContext ctx) {
 visitChildren(ctx);
 for (InstructionContext insctx : ctx.instruction())
  stprogramme.add("liste", getValue(insctx));
 return 0;
}
```

Visiteur : retour résultat

```
public String render() {
 return stprogramme.render();
}
```

Programme (extrait)

```
FileInputStream fis =
                new FileInputStream("programs/prog.logo");
ANTLRInputStream input = new ANTLRInputStream(fis);
LogoLexer lexer = new LogoLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
LogoParser parser = new LogoParser(tokens);
ParseTree tree = parser.programme();
LogoST4Visitor visitor = new LogoST4Visitor();
visitor.visit(tree);
FileWriter fw = new FileWriter("src/logojava/JavaLogo.java");
fw.write(visitor.render());
fw.close();
```

Exemple : résultat

```
package logojava;
public class JavaLogo {
  Traceur traceur = new Traceur();
  public JavaLogo() {}
  public void initializeTraceur(java.awt.Graphics g) {
    traceur.setGraphics(g);
  }
  public void run() {
     traceur.avance(100);
     traceur.td(90);
     traceur.avance(100);
  }
}
```