

The Role Of Template Engines in Code Generation

Terence Parr
University of San Francisco

parrt@cs.usfca.edu

parrt@ANTLR.org

Introduction

- Car aerodynamics mostly trumps style today
- Similarly, the nature of generating text should dominate design decisions: use an output grammar
- Don't have output grammars, we have programs with print statements; template engines arose to encourage separation of logic/display.
- Enforcing strict separation also leads to similar grammar-like mechanism
- Conclusion: if you're generating text, you should be using something akin to StringTemplate

Outline

- Define and motivate model-view separation; give enforcement rules, show equivalence to CFGs
- Describe StringTemplate, provide example
- Relate experience building ANTLR 3.0 code generator with StringTemplate

HTML Generation

Servlet:

```
out.println("<html>");
out.println("<body>");
out.println("<h1>Servlet test</h1>");
String name = request.getParameter("name");
out.println("Hello, "+name+".");
out.println("</body>");
out.println("</html>");
```

JSP:

```
<html>
<body>
<h1>JSP test</h1>
Hello, <%=request.getParameter("name")%>.
</body>
</html>
```

Example Entanglements

- `$if(user=="parrt" && machine="yoda")$`
- `$price*.90$, $bloodPressure>130$`
- `$a=db.query("select subject from email")$`
- `$model.pageRef(getURL())$`
- `$ClassLoader.loadClass(somethingEvil)$`
- `$names[ID]$`

- `st.setAttribute("color", "Red");`

Motivation For Separation

- Encapsulation
- Clarity
- Division of labor
- Component reuse
- Single point-of-change
- Maintenance
- Interchangeable views, retargeting
- Security

Existing Engines

- **Problem:** engines don't **enforce** separation, they only **encourage** separation (Murphy, IBM keybrd, etc...)
- **Reason:** engine builders and users fear that enforcement implies fatal weakness
- **Result:** developers exploit loopholes, encoding logic in templates, thus, entangling model/view
- We can enforce separation without emasculating the power of a template engine
 - empirical evidence and theoretical support

Template Definition

- Unrestricted template:
 $t_0e_0\dots t_i e_i t_{i+1} \dots t_n e_m$
where t_i is a literal and e_i is unrestricted computationally and syntactically
- Notes:
 - unrestricted templates do not enforce separation
 - XSLT is not a template engine

Rules of Separation

1. the view cannot modify the model
2. cannot perform computations upon dependent data values
3. cannot compare dependent data values
4. cannot make type assumptions
5. data from model cannot contain display, layout information

Restricted Templates

- Restrict template to operate on read-only data values, attributes (single or multi-valued), to prevent side-effects
- e_i are attribute or template references
- Even restricted templates can generate the context-free languages
- By allowing conditional inclusion (predicates), reaches into context-sensitive languages
- XML DTDs are essentially CFGs, therefore, restricted template can generate syntax of any XML document

Equivalence to CFGs

- Attributes = terminals, templates = rules
- Can show grammar's derivation tree for any sentence maps to a nested template tree structure

Grammar

```
prog : decl func ;
decl : type ID ;
func : type ID "("
      "{"
      body
      "}"
...

```

Template

```
prog ::= "<decl()> <func()>"
decl ::= "<type> <ID> ;"
func : <<
      <type> <ID>() {
      <body()>
      }
      >>
...

```

StringTemplate

- Evolved from simple "document with holes" while building jGuru.com, after dumping JSP
- Side-effect free expressions
- No order of evaluation
- Recursion (recall output structures are nested)
- Dynamic scoping
- Lazy-evaluation
- Template inheritance/polymorphism
- Simple enough for nonprogrammers
- Strictly enforces separation of model/view
- No assignments, loops, ...

Canonical Operations

- Attribute reference:
`<type>`
- Template references (possibly recursive):
`<statementList()>`
- Apply template to multi-valued attribute:
`<decls:decl()>` or
`<decls:{<it.type> <it.name>;}>`
- Conditional include:
`<if(superClass)>extends
<superClass><endif>`

Template Groups

- Set of mutually-referential templates with formal arguments

```
group javaTemplates;  
  
method(type,name,args,body) ::= <<  
public <type> <name>( <args:arg(); separator=","> ) {  
  <body>  
}  
>>  
assign(lhs,expr) ::= "<lhs> = <expr>;"  
if(expr,stat) ::= "if (<expr>) <stat>"  
call(name,args) ::= "<name>( <args; separator=","> );"  
...
```

Template Polymorphism

- Output: "y=1;" not "x=1;" because template instance's group is subGroup

```
group sup;  
slist() ::= "<assign()>"  
assign() ::= "x=1;"
```

```
group sub;  
assign() ::= "y=1;"
```

Late bind

```
sub.setSuperGroup(sup);  
StringTemplate st = sub.getInstanceOf("slist");  
System.out.println(st.toString());
```

Group determines symbol resolution

Example: Dump Java Class

- Expected output:

```
class Dump {  
    public int i;  
    public java.lang.String name;  
    public int[] data;  
    public void main(class java.lang.String[] arg1);  
    public void foo(int arg1, float[] arg2);  
    public class java.lang.String bar();  
}
```

Dump Java Class Templates

```
group Java;

class(name,fields,methods) ::= <<
class <name> {
  <fields:field(); separator="\n">
  <methods:method()>
}
>>

field() ::= "public <type(t=it.type)> <it.name>;"

method() ::= <<
public <it.returnType> <it.name>
  (<it.parameterTypes:{<type(t=it)> arg<i>}; separator=", ">);
>>

type(t) ::= <<
<if(t.componentType)><t.componentType>[]
<else><t.name><endif>
>>
```

Dump Java Class Code

```
public class Dump {
  public int i;
  public String name;
  public int[] data;
  public static void main(String[] args) throws IOException {
    StringTemplateGroup group =
      new StringTemplateGroup(new FileReader("Java.stg"),
        AngleBracketTemplateLexer.class);
    Class c = Dump.class;
    Field[] fields = c.getFields();
    Method[] methods = c.getDeclaredMethods();
    StringTemplate classST = group.getInstanceOf("class");
    classST.setAttribute("name", c.getName());
    classST.setAttribute("fields", fields);
    classST.setAttribute("methods", methods);
    System.out.println(classST);
  }
  public void foo(int x, float[] y) {}
  public String bar() {return "";}
}
```

Dump XML Instead

```
group XML;

class(name,fields,methods) ::= <<
<class>
  <name>$name$</name>
  $fields:field()$
  $methods:method()$
</class>
>>

field() ::= <<
<field>
  <type>$type(t=it.type)$</type><name>$it.name$</name>
</field>
>>
...
```

```
<class>
  <name>Dump</name>
  <field>
    <type>int</type><name>i</name>
  </field>
  ...
</class>
```

Experience with ANTLR 3.0

- Tree walker (controller) collects data from AST (model), pushes data into templates (view)
- Decouples order of computation from order of output (this is huge)
- Enforced separation guarantees easy retargeting, no code duplication, ...
 - no code in template
 - no output strings in code generator
 - Previous code generator hopelessly entangled
- Group file format (output grammar) is great! "Executable documentation"

Sample ANTLR 3.0 Template

```
parser(name, tokens, rules, DFAs) ::= <<
class <name> extends Parser {
  <tokens:
    {public static final int <it.name>=<it.type>;}
  >
  public <name>(TokenStream input) {
    super(input);
  }

  <rules; separator="\n">

  <DFAs>
}
>>
```

Summary

- The nature of text generation and the enforcement of model-view separation dominate tool design-decisions:
 - tools should resemble output grammars
- StringTemplate is a simple template engine that evolved while building dynamic sites. It is proving exceptionally well suited to code generation tasks including ANTLR 3.0
- open-source Java, BSD license (also a C# port)
<http://www.stringtemplate.org>