A Functional Language For Generating Structured Text

Terence Parr

University of San Francisco

parrt@cs.usfca.edu

Abstract

This paper describes ST (*StringTemplate*), a domain-specific functional language for generating structured text from internal data structures that has the flavor of an output grammar. ST's feature set is driven by solving real problems encountered in complicated systems such as ANTLR version 3's retargetable code generator. Features include template group inheritance, template polymorphism, lazy evaluation, recursion, output auto-indentation, and the new notions of group interfaces and template regions. Experience shows that ST is easy to learn and satisfying to use.

ST's primary contribution is the clear identification and implementation of a tightly-bracketed solution to the problem of rendering data structures to text, as dictated by the nature of generation and the critical goal of strictly separating the generation logic from the output templates. First, the very nature of code generation suggests the use of a generational grammar. Next, enforcing separation of the generation logic from the output templates restricts the template language syntactically and semantically in such way that templates are easily shown to be equivalent to a grammar. Finally, the rules of separation, such as side-effect free expressions, coincide with the fundamentals of pure, functional programming.

1. Introduction

The growth of the web has led to the proliferation of programs that generate structured text documents. Similarly, the difficulty of writing large software systems has led to an increased interest in generative programming where one program writes another program. In both cases, programs are generating text from internal data structures and, unfortunately, many of them are unstructured blobs of generation logic interspersed with print statements. The formal systems that do exist, often referred to as template engines, still support entangling generation logic with output templates, thereby, seriously degrading their effectiveness and making it essentially impossible to build a retargetable generator.

Consider the four target language code generators for version 2 of the ANTLR recursive-descent parser generator [1]. The generators represent 39% of the total lines and are roughly 4000 lines of entangled logic and print statements for *each* language target. Building a new language target amounts to copying the entire Java file (thereby duplicating the generator logic code) and tweaking the print statements. The primary reason for this is the lack of suitable tools and formalisms. The proper formalism is that of a grammar because the output is not a sequence of random characters–the output is a sentence conforming to a particular language. Using a grammar to generate output is analogous to describing the structure of input sentences with a grammar. Rather than building a parser by hand, most programmers would use a parser generator. Similarly we need some form of *unparser generator* to generate text.

The most convenient manifestation of an output grammar is a template engine like *StringTemplate* (ST), the subject of this paper, that has a grammar-like flavor, supporting mutually-referential templates and simple embedded data value references. To see the duality relationship between templates and grammars, consider the following grammar that specifies the syntax of a class and instance variable definitions.

```
class : 'class' ID '{' decl* '}' ;
decl : 'public' TYPE ID ';' ;
```

Conversely, a group of ST templates is an inverted grammar where template and value references are escaped instead of escaping literals such as `'public'`:

```
class(ID,decl) ::= "class <ID> { <decl> }"
decl(TYPE,ID) ::= "public <TYPE> <ID>;"
```

There is a clear relationship between a template containment tree and a parse or derivation tree. Just as rule `class` invokes rule `decl`, an instance of template `class` will embed an instance of template `decl`. The only difference with a template is that an external program must create template instances and build the graph according to some generation logic rather than according to an input stream as with a grammar.

Templates are essentially exemplars of the desired output with "holes" where the programmer may stick values called *attributes* or other template instances. Compare the clarity of these exemplars with the task of imagining the emergent behavior of a series of print statements, possibly tangled up with logic.

Using the Java version of ST, the following code fragment generates "`public int i;`" using an instance of the `decl` template.

```
// load group containing templates class, decl
StringTemplateGroup templates = ... ;
StringTemplate d = templates.getInstanceOf("decl");
d.setAttribute("TYPE", "int");
d.setAttribute("ID", "i");
String code = d.toString(); // render
```

In the parlance of the *model-view-controller* (*MVC*) pattern, the templates represent the *view* and the code fragment represents both *model* (fixed `ID` and `TYPE` strings) and *controller* (code that injects attributes, implicitly building the template containment graph).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $5.00.

There are obvious benefits to a generator that has its view completely encapsulated in templates and its logic completely encapsulated in a model and controller:

1. The templates specify and document the generated code structure concisely and clearly, improving development and maintenance costs.
2. The output can be independently and radically altered simply by changing the view–all without touching or even recompiling the model.
3. The model provides a single point of change for all logic.
4. A single model-controller pair may feed multiple template groups, thus, supporting multiple target languages.
5. Templates can be reused with generators of similar applications that target the same language.

All of these benefits derive from the single important principle of *strict model-view separation*, which carries with it restrictions such as disallowing computation and logic based upon attribute values within the template and requiring side-effect free attribute expressions. Surprisingly such restricted templates are easily shown to generate the context-free languages and beyond into the context-sensitive [18]. By limiting expressions to be simple side-effect free attribute references, one may equate attribute expressions to token references and nested templates to nested rule invocations. Strict model-view separation then suggests that the general structure of a generator should be equivalent to an output grammar as does just the very idea that we are generating structured text.

This still leaves open, however, the question of template expression syntax and style. Interestingly, the model-view separation rules for code generation also coincide with the essentials of a pure functional language. Pure functional languages have side-effect free expressions and yield a result computed from the, possibly recursive, application of functions to data. ST's restrictions echo John Backus [2] sentiments in his 1978 Turing Award paper when making a plea for functional languages: "... with unrestricted freedom comes chaos."

In practice, strict model-view separation amounts to a guarantee of retargetability as all generation logic is confined to the model-controller pair and all literals are confined to the view. In light of the hideous, entangled code generators of ANTLR version 2, a primary goal during the construction of version 3 was that adding a new language target should be possible purely by adding templates. Except for a few details like character literal encodings, ANTLR version 3 has achieved this goal through the use of ST. The code generation logic is now target language agnostic and only 4000 lines (8% of the total down from 39%). The introduction of each new language target has a marginal cost of another 2000 lines, a 50% reduction over version 2. The key difference is that targets are described purely with templates not code, making it much easier and faster to build a robust target.

The very nature of code generation suggests the use of a grammar-like language. The worthy goal of strict model-view separation again suggests the use of grammars because of the duality between grammars and restricted templates. And, finally, the primary rule of separation, side-effect free expressions, is the fundamental idea behind pure, functional programming. The primary contribution of this research, therefore, is the clear identification of a tightly-bracketed solution to the problem of rendering data structures to text, as dictated by the nature of the problem and good programming practices. ST is a domain-specific language with a functional programming flavor for generating structured text that embodies this solution. Experience building ANTLR version 3 validates that ST and the approach espoused by this paper are effective for building a complex, retargetable code generator.

The remainder of the paper describes the ST language in detail (Section 2), provides an example retargetable code generator for finite automata (Section 3), and compares ST to other template engines and academic research (Section 4).

2. The ST Language

An ST "program" is a group of template definitions and a *controller* written in Java, C#, or Python that injects untyped data *attributes* as template arguments and instantiates template instances. The controller ultimately issues a render-to-text command to the top-level template instance to obtain the program output. An attribute is either a controller program object such as a string or VarSymbol object, a template instance, or sequence of attributes including other sequences. To enforce model-view separation, templates may not test nor compute with attribute values and, consequently, attributes have no need for type information. Templates may, however, know the data is structured in a particular manner such as a tree structure.

Templates are loaded into a StringTemplateGroup instance from a group file with the following general syntax:

```
group groupname;
template1(a1, a2, ..., an) ::= "..."
...
```

The collection of mutually-referential output templates serves as a kind of library of output constructs for the controller.

Templates are "compiled" to instances of type StringTemplate that act as prototypes for further instances. Each template instance contains an attribute table specific to the instance with attribute name-value pairs (the template arguments) and an abstract syntax tree (*AST*) structure shared by all instances that represents the literals and expressions. The AST is suitable for speedy interpretation of the template during program execution. The set of attributes in the attribute table is limited to the formal argument list, a_i.

A template is a function that maps an attribute or attributes to another attribute and is specified via an alternating list of output literals, t_i, and expressions, e_i, that are functions of a_i:

$$F(a_1, a_2, ..., a_m) ::= \text{“}t_0 e_0 ... t_i e_i t_{i+1} ... t_{n_t} e_{n_e}\text{”}$$

where t_i may be the empty string and e_i is restricted computationally and syntactically to enforce strict model-view separation. The e_i are distinguished from the surrounding literals via angle brackets <...>. If there are no e_i then the template is just a single literal, t_0. The invisible concatenation operator applies to any adjacent elements. Evaluating a template amounts to traversing and concatenating all t_i and e_i expression results. Evaluation is triggered by the controller via a call to the toString() method of the template.

Attribute expressions are functional forms combining canonical operations that are limited to operate on the surrounding template's attribute table or, using dynamic scoping, to operate on any enclosing template instance's attributes. All expressions are side-effect free and, thus, there are no variable assignments. Further, expressions may not affect prior computations nor the surrounding template. The four canonical attribute expression operations are:

- attribute reference <name>
- template include <supportcode()>
- conditional include
  <if(trace)>print("enter function");<endif>
- template application (i.e., *map* operation) <vars:decl()>

Expressions are evaluated *lazily* in the sense that the controller may create and assemble even highly-nested template structures as necessary without concern for the attributes they reference nor

the order in which template expressions will be rendered to text. In other words, templates may be instantiated and used by other templates long before the associated attribute arguments are available. The only restriction is that all attributes needed by all templates must be injected by the controller prior to final rendering via toString(). Such delayed evaluation is critical to decoupling the order of template computation, often dictated by the internal data structures, from the output language constructs' order specified by the target language. The model and controller must drive the order of computation rather than the template attribute reference order.

ST encourages the wholesale reuse of template groups by supporting group inheritance whereby a *subgroup* may override or augment templates from a *supergroup*. The mechanism exactly mimics the behavior of class inheritance in object-oriented programming languages. When generating code for similar languages, a programmer can factor out common templates into a supergroup. The subgroups need only specify additional or different functionality for the various targets. Template *polymorphism* ensures that the template appropriate for the associated group is instantiated. Group inheritance also proves extremely useful when a code generator must emit variations of the same code such as normal code versus code instrumented with debugging information. The main template group is not cluttered up with "if debug" conditionals and all debugging code is neatly encapsulated in a subgroup.

ST introduces a finer-grained alternative to template inheritance, dubbed *regions*, that allow a programmer to give regions (fragments) of a template a name that may be overridden in a subgroup. While regions are syntactic sugar on top of template inheritance, the improvement in simplicity and clarity over normal coarser-grained inheritance is substantial.

To promote retargetable code generators, ST supports *interface implementation à la* Java interfaces where a template group that implements an interface must implement all templates in the interface and with the proper argument lists. The interface is the published, executable documentation for building back-ends for the code generator and has proven to be an excellent way to inform programmers responsible for the various targets of changes to the requirements. The alternative is to leave all "missing template" or "invalid template attribute" errors to dynamic detection, leaving plenty of invalid expressions in untested templates.

The following sections explore each of these main concepts in more detail, explaining their significance to code generation.

2.1 Templates

Templates are defined very much like functions or methods with a name, formal untyped argument list, and a body:

$$name(a_0, a_1, ..., a_{n_a}) \texttt{ ::= "..."}$$

The body is a template represented by a double-quoted string or a double-angle-bracket <<...>> section for multi-line templates. Each template is an alternating list of output literals and attribute expressions whose results are concatenated to produce a result. In functional terms, the result is an implicit *reduce* operation over the elements using the concatenate operator. Consider a template called decl that generates simple variable definitions given access, type, and variable name attributes:

```
decl(access,type,ID) ::= "<access> <type> <ID>;"
```

Template decl has three attribute arguments, three literals, and three simple attribute reference expressions: a_0=access, a_1=type, a_2=ID, e_0=access, t_0=" ", e_1=type, t_1=" ", e_2=ID, and t_2=";".

The formal argument list, a_i, specifies the complete set of attributes that a controller may set for any instance of this template just like a function argument list. The only difference is that a controller or other template need not set every a_i attribute. For example, omitting the setAttribute() call for attribute access would result in " int i;".

Templates employ *dynamic scoping* in that expressions may reference the attributes of enclosing templates. Contrast that with the *lexical scoping* of C variants that limits parameter access to the immediately surrounding function. In the following code fragment, function f invokes g and, while parameter i is still physically available on the stack above, function g may not access i.

```
void f(int i) { g(); }// C example
void g() { int k=i; } // cannot see enclosing scope
```

Naturally, f could pass i through to g as a parameter, but when constructing templates, it is easier to directly access attributes from any enclosing template instance. This is analogous to Java methods being able to access inherited instance variables. For example, you might want all templates used to build output for a method to have access to the method name in order to mangle variable names:

```
method(type,name,decls,stats) ::= <<
<type> <name>() {
  <decls>
  <stats>
>>
decl(type,ID) ::= "<type> <name>_<ID>;"
```

Template method's name attribute is visible to template decl for use as a prefix. Here, presumably, the controller creates decl instances and injects them into a method instance.

To avoid confusion in the large, heavily-nested template tree structures common to code generators, formal arguments hide any attribute value with that name in any enclosing instance. This prevents a nested template from inadvertently using an attribute from an enclosing scope that could lead to infinite recursion, via cyclic attribute references, and other surprises. If decl had name in its formal argument list, expressions within decl would not be able to see the name attribute of method. Experience building a large dynamic web site (jGuru.com) and ANTLR version 3 with ST attests that ST's form of dynamic scoping is a big advantage and does not cause confusion or unexpected results.

ST generates exceptions at run-time if a controller or template invocation tries to set an attribute not in the template's formal argument list. Conversely, an exception is also generated when an expression references an attribute not formally defined in any template on the path to the root template. The attribute may have a null value, but it must be formally defined in an enclosing scope. Together these exceptions provide an important safeguard against attribute name mismatches, from typos or otherwise, between controller and template definitions. In practice, the exceptions reduce the number of mysterious cases where required constructs are absent from the output.

2.2 Template Expressions

Expressions embedded within templates are limited to functional forms, delimited by angle brackets <...>, comprised of four canonical operations described in the following sections.

2.2.1 Attribute references

The most common expression is a simple reference to an attribute, such as <name>, that will ultimately be converted to text via the toString() method of the object associated with name. References to missing or null attributes evaluate to the empty string. These referenced attributes may be atomic elements from the controller's application language like String or Symbol but may also be (embedded) templates. Calling toString() on a template forces it to evaluate to flat text just like an atomic element. Attributes set to null or without values result in the empty string.

Attributes are most often aggregate data types so ST allows access to an object's fields. For example, if the controller injects a Symbol object into a template as attribute s, an expression may reference s.name or any other field visible via reflection (run-time type information). ST uses method getName() if available else it tries to directly access field name. The one special case relates to dictionary attributes. The field is treated as a key and the result of the expression is the associated mapped value rather than a field of the dictionary object itself. For example, <types.int> would look up the value for string key "int" in the map identified by types, perhaps translating it to a type name in the target language.

If an attribute is multi-valued (a list or some other iteratable data structure), the attribute's value is the concatenation of calling toString() on all the elements. For example, <vars> where vars is a list with string attributes x, y, and z would evaluate to "xyz". To provide a separator between elements in a multi-valued attribute, expressions use the separator option with an arbitrary string or template expression; <vars; separator=", "> results in "x, y, z".

It is worth reinforcing here that attribute expressions reference values that may not be available until later, when the controller injects them, and attributes are always just values computed by the controller from the model and pushed into the template(s). An expression may not invoke arbitrary code in the model nor controller. The only code execution initiated by a template is the implicit invocation of toString() methods during evaluation.

2.2.2 Template include

ST encourages programmers to factor templates into small, reusable chunks, which also reduces the amount of duplicated template code. Providing a single-point-of-change for all output constructs is an important design principle. The following method template factors out a bit of support code tracing.

```
method(type,name,decls,stats) ::= <<
<type> <name>() {
  <preamble()>
  <decls>
  <stats>
}
>>
preamble() ::= <<
System.out.println("Enter method");
>>
```

Factored templates yield a nested template containment tree. For example, a constructor template, though similar to a method, should avoid duplication by referencing template instead of cutting-n-pasting the template and changing the name:

```
constructor(name,decls,stats) ::= <<
<method(...)>
>>
```

where the "..." argument is a special symbol that says, "allow invoked template method to see all of the invoking template constructor's attributes." To ensure that an unwanted type attribute did not flow through to method, attribute type could be set explicitly: <method(type="", ...)> where again "..." is literally the second argument.

2.2.3 Conditional include

Similar templates should be factored to avoid duplication, but sometimes the difference between templates depends on the presence or absence of an attribute and the programmer cannot simply carve out the common fragment. For example, a template for Java class definitions needs to emit the "extends" keyword, but only if a superclass has a value in the attributes table. Rather than have two different nearly identical versions of a class template, introducing duplication, it is better to have a single template with a conditional include that tests for the presence of a superclass:

```
class(name,sup,methods) ::= <<
class <name> <if(sup)>extends <sup><endif> {
  <methods>
}
>>
```

ST restricts conditionals to *attr-name*, testing attribute presence, and !*attr-name*, testing attribute absence. Arbitrary expressions are disallowed to ensure all logic is done outside the view. The result of complicated arbitrary computations may be pushed into the template, but the template itself may not do the computation to enforce strict separation rules. The one exception is that attributes with boolean values are treated as such to avoid confusion; early versions of ST were annoying because attributes with false values evaluated to true because they had non-null values.

Useful examples of conditionals abound in code generation; e.g., optional variable initialization values:

```
<type> <name> <if(init)>= <init><endif>;
```

and return type generation:

```
<if(type)><type><else>void<endif> <name>() {<body>}
```

2.2.4 Template application

The most important ST language construct is template application because, along with with template recursion, it obviates the need for any explicit imperative-style looping constructs such as for loops. The ability to apply a template to an attribute or the elements of a multi-valued attribute distinguishes ST's character from all other template engines more clearly than any other construct.

The simplest form is <*attr-name*:*template-name*()> and means, "apply *template-name* to each element of *attr-name* by repeatedly invoking *template-name* with its sole attribute argument set to the ith attribute element. Reduce values to a string with concatenate." A template applied to an attribute is another attribute. A template applied to a multi-valued attribute is another multi-valued attribute. For example, if attribute args is a list of Symbol objects, then <args:arg()> will apply template arg to each of those objects, yielding a new, multi-valued attribute. Template arg must be defined with a single element:

```
arg(a) ::= "<a.type> <a.name>"
```

In this case, attribute a will be an object of type Symbol. If the definition of template arg were missing a formal argument, then ST would set a default attribute, it:

```
arg() ::= "<it.type> <it.name>"
```

For template applications with very simple templates, an anonymous inline template (analogous to a lambda function) is often easier to build and more clear. The form replaces the template name with a curly-brace enclosed fragment:

```
<attr-name:{arg-name | template-fragment}>
```

Consider generating enumerated types of the form:

```
int Monday = 1;
int Tuesday = 2;
...
```

If attribute enumNames has a list of names, then the following expression will generate the appropriate integer definitions.

```
<enumNames:{e | int <e> = <i>;}>
```

where `i` is an implicitly set attribute that is the iteration number from $1..n$ for n values in an attribute. `i0` is a zero-based index.

If the enumeration values are in a second multi-valued attribute, `enumValues`, then those values may be used instead of $1..n$ by applying the fragment to the lists in parallel:

```
<enumNames,enumValues:{e,v | int <e> = <v>;}>
```

Anonymous templates may naturally contain other anonymous template fragments. The following expression accepts an attribute containing at least one declaration object with properties `type` and `vars` where `vars` is also a possibly multi-valued attribute containing variable names. Given a single declaration object corresponding to "`int x,y;`", the following nested expression yields "`int foo_x; int foo_y;`" where attribute `name` is `foo`.

```
<decls:{d | <d.vars:{v | <d.type> <name>_<v>;}>}>
```

The type name, `int`, is repeated for each variable name because it is referenced within the nested template application (iteration).

ST supports repeated function application for situations when multiple templates must be applied to an attribute. For example, if an enumeration name must be mangled in some way before its integer type is generated, the programmer may combine two or more template applications:

```
<enumNames:mangle():{e | int <e> = <i>;}>
```

or

```
<enumNames:{e|<name>_e}:{e | int <e> = <i>;}>
```

Such repeated template application supports reusability because variations of existing templates may be achieved without modifying them simply by combining different templates (though ST does not have an explicit template composition operator). HTML text generation make this more clear: `<names:bold():italics()>`, which first bolds and then italicizes a set of names without having to create a combined `boldItalics` template. `<`a`:`f`():`g`()>` is the same as the concatenation of nested function $g(f(a_i))$ results for each element a_i of attribute a.

ST also supports recursive templates to handle nested output structures naturally. For example, the following template does a preorder tree traversal, generating the `text` property for each node.

```
preorder(t) ::= "<t.text> <t.children:preorder()>"
```

The next section shows how templates may be grouped together and selectively overridden through group inheritance.

2.3 Group Inheritance

Each code generator target typically uses a separate group of templates to specify output constructs for that target language. For example, if a code generator must emit both C and Java, each target should implement a common set of templates used by the generation logic. On the other hand, what happens when each target has multiple variations? Perhaps a single C target must support both `gcc` and ANSI C or a Java target must support both 1.4 and 1.5 versions (1.5 added generics and enums, for example). How are multiple versions encoded?

Group inheritance provides an appropriate model whereby a variation on a code generation target may be defined by describing how it differs from a previously defined target. Addressing the Java 1.4 versus 1.5 issue, a `Java1_5` group could specify how to alter the main `Java` (1.4) group templates in order to use generics and enumerated types.

Consider the code generator for a parser generator that needs to define integer constants for the token types associated with token references in a grammar:

```
public static final int ID = 1;
public static final int KEY_WHILE = 2;
...
```

The following partial `Java` group factors the token type definition code to a `constants` template from the `parser` template where token types need to be generated in order to allow a *subgroup* to override the behavior.

```
group Java;

parser(name,tokens,rules) ::= <<
class <name> extends Parser {
  <constants(names=tokens, type="int")>
  <rules:rule()>
}
>>
constants(names, type) ::= <<
<names:{n | public static final <type> <n>=<i>;}>
>>
```

Java 1.5 has an explicit `enum` construct that the `Java1_5` target should take advantage of. To do so, the subgroup merely overrides template `constants`:

```
group Java1_5 : Java; // derive from Java

constants(names, type) ::= <<
public enum TokenType { <names; separator=", "> }
>>
```

Templates with the same name in a subgroup override templates in a supergroup just as in class inheritance. ST does not support overloaded templates so group inheritance does not take formal arguments into consideration.

Group inheritance would not yield its full potential without template *polymorphism*. A `parser` template instantiated via the `Java1_5` group should always look for templates in `Java1_5` rather than the `Java` supergroup even though `parser` is lexically defined within group `Java`.

When adding more output variations, a proper separation of concerns can make generating multiple targets even more complicated. For example, adding a debugging variation means adding another group of templates derived from the 1.4 and 1.5 templates to isolate all debugging code fragments in one group. Obfuscating the main templates with debugging code is not a good idea, but the resulting number of group combinations can grow very quickly. There are two primary scenarios.

In the first case, each output variation needs a special group of debugging templates. Each variation must then have a subgroup called perhaps `DbgJava` and `DbgJava1_5` that statically derive from the `Java` and `Java1_5` supergroups, respectively.

In the second case, a single template group, `Dbg`, is sufficient to alter any Java variation, but because ST only supports single inheritance, the controller must dynamically instruct the `Dbg` group to derive from either `Java` or `Java1_5`.

Without dynamic inheritance, multiple copies of `Dbg` would be required, one for each output variation. The following Java fragment demonstrates how to dynamically derive a "subgroup":

```
StringTemplateGroup java = ..., dbg = ... ;
dbg.setSuperGroup(java);
```

When originally confronted with this explosion in target variation combinations, a form of template group composition (involving repeated template application) was considered in a manner sim-

ilar to *descriptional composition* [5], but rejected to avoid introducing an extra language concept that was so similar to the existing dynamic inheritance mechanism.

2.4 Template regions

During the implementation of the first few ANTLR v3 targets, a weakness appeared in the inheritance mechanism. Inheritance proved a very blunt instrument for overriding and inserting tiny snippets of output text. First, tiny snippets had to be factored out of their natural home in a larger template and defined as full blown templates elsewhere, which reduced the clarity of the larger template and exploded the number of template definitions. Second, because target variant subgroups needed to insert little bits of code where the supergroup had none, the supergroup had to invoke and define templates with blank implementations. Subgroups overrode the blank templates, effectively inserting code into the supergroup templates. To ameliorate the situation, ST introduced the notion of a template *region* (similar to the idea of a block in Django [9]) that either marks a section of a template or leaves a "hole" that subgroups may alter.

To illustrate how and why templates mark regions for alteration by subgroups, consider ANTLR's validating semantic predicates, `{p}?`, that act like assert statements. The Java target must generate:

```
if ( !(p) ) throw new Exception("failed {p}?");
```

When debugging grammars, however, the conditional must be altered to trigger a "predicate validation" event:

```
if ( !dbgValidatePred(<p>) ) throw ...;
```

There are three approaches to encoding the two variations. First, a conditional can gate the debugging code in and out.

```
validatePred(p) ::= <<
if ( <if(debug)>!dbgValidatePred(<p>)<else>
     !(<p>)<endif> ) throw ..;
>>
```

There are two things wrong with the conditional; it makes the template hard to read and it merges the main templates with the debugging templates, a poor separation of concerns.

The second solution factors out the predicate expression:

```
validatePred(p) ::= <<
if ( !(<p:evalPredExpr()>) ) throw ...;
>>
evalPredExpr(e) ::= "<e>"
```

`evalPredExpr` is overridden in the `Dbg` subgroup to include the debugging call to `dbgValidatePred()`. The `validatePred` template is a little easier to read, but the benefit is mitigated by the cost of the additional template definition. When just looking at these two dislocated templates, `evalPredExpr` cries out to be inlined into `validatePred`, which leads to the best solution.

The final solution provides both clarity and separation of concerns. The idea is to build the template and then mark the snippets that will likely need to change:

```
validatePred(p) ::= <<
if ( !(<@eval><p><@end>) ) throw ..;
>>
```

where `<@`*r*`>...<@end>` marks a region called *r*. A `Dbg` subgroup may then override the template in a manner similar to familiar template overriding:

```
@validatePred.eval() ::= "dbgValidatePred(<p>)"
```

Regions are like *subtemplates* scoped within a template, hence, the fully-qualified name of a region is `@`*t*`.`*r*`()` where *t* is the enclosing template. In an overridden region, `@super.`*r*`()` refers to the supergroup template's original region contents.

Turning to the second weakness of normal group inheritance, how can a subgroup insert a snippet into a template defined in the supergroup? Again, normal group inheritance provides a viable but extremely heavy solution. For every insertion point, define an empty template and then reference the templates at the insertion points among the real templates. In the main ANTLR Java target group, this approach would mean adding 30 more (empty) templates to the group of 124 real templates.

The best solution is to leave a named "hole," without having to define a blank template, that subgroups may fill in by defining a template in a subgroup. For example, consider generating code for a grammar rule with a name and list of productions:

```
rule(name,productions) ::= <<
public void <name>() {
  <@preamble()>
  <productions:production()>
  <@postamble()>
}
>>
```

Not only is the structure of template `rule` still clear even with the insertion points, `preamble` and `postamble`, but superfluous templates need not be defined. The `Dbg` subgroup may, in effect, insert code into template `rule` by simply overriding the region as before with the inline regions:

```
@rule.preamble() ::= <<enterRule("<name>");>>
```

Regions support template clarity and separation of concern among groups in a fine-grained, lightweight, and unobtrusive manner. By leveraging the familiar inheritance paradigm, ST avoids introducing yet another key concept to learn. The implementation of regions also leverages the existing inheritance infrastructure for a clean, modest augmentation of the code.

2.5 Interface Implementation

The developers of the ANTLR code generation targets always have the same two questions: Initially they ask, "*What is the set of templates I have to define for my target?*" and then, during development, they ask, "*Has a change to the code generation logic forced any changes to the requirements of my template library?*"

Originally, the answer to the first question involved abstracting the list of templates and their formal arguments from the existing Java target. The answer to the second question involved using a difference tool to point out changes in the Java target from repository check-in to check-in. Without a way to formally notify target developers and to automatically catch logic-template mismatches, bugs creep in that become apparent only when the stale template definitions are exercised by the code generator. This situation is analogous to programs in dynamically typed languages like Python where method signature changes can leave landmines in unexercised code. In short, there were no good answers.

ST now supports *group interfaces* that describe a collection of template signatures, names and formal arguments, in a manner analogous to Java interfaces. Interfaces clearly identify the set of all templates that a target must define as well as the attributes they operate on. The first question regarding the required set of templates now has a good answer.

Interfaces also provide a form of type safety whereby a target is examined upon code generator startup to see that it satisfies the interface. Here is a piece of the ANTLR main target interface:

```
interface ANTLRCore;
parser(name, scopes, tokens, tokenNames, rules,
       numRules, cyclicDFAs, bitsets, ASTLabelType,
```

```
       superClass, labelType, members);
rule(ruleName, ruleDescriptor, block, emptyRule,
       description, exceptions);
/** What file extension to use; e.g., ".java" */
codeFileExtension();
```

All of the various targets then implement the interface; e.g.,

```
group Java implements ANTLRCore;
```

The code generator, which loads target templates, notifies developers of any inconsistencies immediately upon startup effectively answering the second question regarding notification of template library changes. Group interfaces provide excellent documentation, promote consistency, and reduce hidden bugs.

2.6 Lazy Evaluation

There is usually an order mismatch between convenient, efficient computation of data attributes and the order in which the results must be emitted according to the output language. The developer's choice of controller and data structures has extensive design ramifications. If the developer decides to have the templates embody both view and controller, then the order of the output constructs drives output generation. This implies that the order of attribute a_i references in the view dictates the order in which the model must compute those values, which may or may not be convenient. If the output language requires that n attributes be emitted in order $a_0..a_{n-1}$, a single forward computation dependency, $a_i = f(a_j)$ for $i < j$, represents a hazard. Each a_i computation must manually trigger computations for each attribute upon which it is dependent. A simple change in the attribute reference order in the output templates can introduce new dependencies and unforeseen side-effects that will cause bad output or even generator crashes. This approach of having the templates drive generation by triggering computations and *pulling* attributes from the model is not only dangerous but may also make the computations inconvenient and inefficient.

Decoupling the order of attribute computations from the order in which the results must be emitted is critical to avoiding dependency hazards–the controller must be separated from the view. A controller freed from the artificial ordering constraints of the output language may trigger computations in the order convenient to the internal data structures of the code generator. This choice implies that all attributes are computed *a priori* and merely *pushed* into the view for formatting. Driving attribute computation off the model is very natural, but computation results must be buffered up and tracked for later use by the view.

If the actual view (code emitter) is just a blob of print statements, the developer must build a special data structure just to hold the attributes temporarily whereas templates have built-in attribute tables where the controller can store attributes as they are created. The template then must know to delay evaluation until all attributes have been injected, effectively requiring a form of *lazy evaluation*. Because the controller computes all attributes *a priori*, however, ST can simply wait to evaluate templates until the controller explicitly invokes `toString()` on the root template instance. This invocation performs a bottom-up recursive evaluation of all templates contained in *t* followed by template *t* itself.

A simple, but real example of this problem arises when generating C code. All C functions must be declared before they are called as in the following example:

```
extern float g(int);
void f() { g(3); }
float g(int) { ... }
```

Driving the output from the view means that the list of forward references must be computed and emitted before walking the list of functions. In contrast, driving the output from the list of functions (the internal data structure) is much easier because, as functions are encountered, their names can simply be added to a list of forward declarations. ST supports a particularly satisfying solution that does not even require a separate data structure. The list of function templates can be walked twice by the output template–once for the forward declarations and once to actually generate the functions. Because the `functions` attribute is a list of function templates and has not been converted to text, attribute expressions like `<f.type>` may access the template instance's attribute table to get the function's type:

```
CFile(globals,functions) ::= <<
<globals>
<functions:{f | extern <f.type> <f.name>(<f.args>);}>
<functions>
>>
function(type,name,args) :: = "..."
```

In practice, delayed evaluation means that templates may be created and assembled as necessary without concern for the attributes they reference nor the order in which templates will be rendered to text. This convenience and safety has proven extremely valuable for complicated generators like that of ANTLR version 3.

2.7 Miscellaneous

This section introduces a number of the smaller but very interesting features of ST.

2.7.1 Auto-indentation

Properly-indented text is a very desirable generation outcome, but it is often difficult to achieve–particularly when the programmer must do this manually. ST automatically and naturally indents output by tracking the nesting level of all attribute expression evaluations and associated whitespace prefixes. For example, in the following `slist` template, all output generated from the `<statements>` expression will be indented by two spaces because the expression itself is indented.

```
slist(statements) ::= <<{
  <statements>
}
>>
```

If one of the statement attributes is itself an `slist` then those enclosed statements will be indented four spaces. The auto-indentation mechanism is actually an implementation of an output filter that programmers may override to tweak text right before it is written.

2.7.2 Maps and lists

There are situations where the generator must map one string to another such as when generating the initialization values for various types. To avoid having to put literals in the controller or model, ST supports maps directly in the group file format:

```
typeInitMap ::= [
        "int":"0",
        "float":"0.0"]
```

For example, `<typeInitMap.int>` returns "0". If the type name is an attribute not a constant like `int`, ST provides an indirect field access: `<typeInitMap.(typeName)>`.

ST provides a list construction mechanism useful for applying a single template across multiple attributes such as the following.

```
<[locals,parameters]:initialize()>
```

The familiar functional programming operations `first`, `rest`, and `last` may be used. The following template fragment generates code to sum a set of numbers, only declaring the accumulator variable `sum` once.

```
<first(numbers):{ n | int sum = <n>;}>
<rest(numbers):{ n | sum += <n>;}>
```

2.7.3 Debugging

ST provides functionality to help debug complicated template structures. First, ST (as of version 2.3) can generate template dependence graphs in Graphviz DOT [11] format. These graphs illustrate which templates contain other templates and are analogous to a function call graph diagram. Next, ST can detect when cycles exist in a tree of template instances (yielding a cyclic graph). Self-referential, hence infinitely-recursive, structures can occur through programming error and are nasty bugs to track down. Turning on "lint mode" makes ST look for cases where a template instance is being evaluated during the evaluation of itself.

2.7.4 Renderers

A simple reference to an attribute of type Integer such as <n> yields the string computed from invoking Integer's toString() method. This implicit invocation is the only code execution initiated by a template, however, the developer may register *renderer* objects (per template group or per template instance) that alter the default rendering to text. The primary application of this is web page localization for objects such as dates [19].

3. Example

This section presents an example illustrating ST at work generating both a state machine diagram in Graphviz DOT [11] format and a state machine simulator generated in Java. The two key points of interest are (*i*) that the same program is able to generate both the visual representation and the source code without modification and (*ii*) that, surprisingly, the templates themselves do no processing and simply specify how each element of the state machine is generated as text.

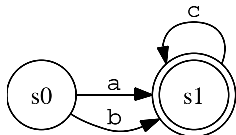

Figure 1. DFA for $(a|b)c^*$

Consider the deterministic finite automaton (DFA) shown in Figure 1, which is just a directed graph composed of two states and three labeled-edges. The image was created with the following Graphviz digraph definition.

```
digraph DFA {
  rankdir=LR;
  # define states
  node [shape = circle]; s0
  node [shape = doublecircle]; s1
  # define edges for states
  s0 -> s1 [fontname="Courier", label = "a"];
  s0 -> s1 [fontname="Courier", label = "b"];
  s1 -> s1 [fontname="Courier", label = "c"];
}
```

This DFA can easily be built in memory using simple class definitions such as State (with fields int state, boolean accept, Edge[] edges) and Edge (with fields char label and State target). The following code creates the states and links them up with edges.

```
// build (a|b)c*
State s0 = new State(0);
State s1 = new State(1);
s1.accept = true;
s0.edges = new Edge[]
    {new Edge('a',s1), new Edge('b',s1)};
s1.edges = new Edge[] {new Edge('c',s1)};
```

The controller then loads a group file, gets an instance of the template called dfa, injects a list of states, and finally renders the template to text:

```
FileReader fr = new FileReader(groupFileName);
StringTemplateGroup lib=new StringTemplateGroup(fr);
StringTemplate dfaST = lib.getInstanceOf("dfa");
dfaST.setAttribute("states", new State[] {s0,s1});
System.out.println(dfaST);
```

Each target language template group must, therefore, have at least a template called dfa that accepts a list of states. That is the only requirement placed upon the templates by the controller, but of course the templates are inescapably sensitive to the structure of the model's data. Once the controller acquires an appropriately refined and processed data structure from the model, it injects the data as an attribute into a template which can then traverse the data structure rendering it to text.

3.1 Generating DOT

Generating DOT format files using ST is straightforward. The main template, dfa, generates the overall framework and delegates the generation of states to template state. Template state in turn relies on template edges_for_state. The following group file comprises the complete set of templates necessary to generate any DFA given the data structure prepared by the controller code above.

```
group DOT;
dfa(states) ::= <<
digraph DFA {
  rankdir=LR;
  <states:state()>      <! walk state templates twice !>
  <states:edges_for_state()>
}
>>

state(s) ::= <<
node [shape = <if(s.accept)>double<endif>circle];
  s<s.state><\n>
>>

edges_for_state(src) ::= <<
<src.edges:{e |
s<src.state> -> s<e.target.state>
  [fontname="Courier", label = "<e.label>"];
}>
>>
```

3.2 Generating Java

Without modification, the controller above can generate Java code that simulates a DFA. The overall structure of the DFA simulation is a switch-on-state statement and then a switch-on-character statement for each state to compute the next state based upon edge labels. The following code fragment represents the essential state transition component of a DFA simulation.

```
loop:
while (true) {
    switch (state) {
        case 0: // state 0
            switch (c) {
                case 'a' : state = 1; consume(); break;
```

```
                case 'b' : state = 1; consume(); break;
                default :
                    error(c);
            }
            break;
        case 1: // state 1
            switch (c) {
                case 'c' : state = 1; consume(); break;
                default :
                    break loop; // accept
            }
            break;
    }
}
```

To generate this Java code instead of a DOT description the controller loads a different template group containing templates associated with generating Java. As before, the group's interface to the controller is a template called `dfa`. The structure of these Java templates is slightly different than the DOT templates in that the main template does not split apart the generation of states and the edges.

```
group Java;
dfa(states) ::= <<
while (true) {
    switch (state) {
        <states:state()>
    }
}
>>

state(s) ::= <<
case <s.state>: // state <s.state>
    switch (c) {
        <s.edges:edge()>
        default :
        <if(s.accept)>break loop;<else>error(c);<endif>
    }
    break;
>>

edge(e) ::= <<
case '<e.label>' :
    state = <e.target.state>; consume(); break;
>>
```

This example demonstrates that enforcing the independence of the controller and view does not conflict with flexibility, legibility, and expressiveness.

4. Related Work and Discussion

ST is specifically concerned with *generating* text as opposed to attribute grammar [14] derivatives and the sophisticated transformation systems such as Stratego [23], TXL [6], and DMS [3] that are concerned with *rewriting* text. These systems are declarative in nature and aspire to create language independent translation solutions that hide implementation details. Some declarative systems entangle pattern recognition, rewrite rules, and output rules, making it difficult to change targets. In practice, programmers are often uncomfortable with purely declarative "black boxes" systems because of the very fact that they do not expose internal data structures. Consequently, programmers use a hybrid approach employing a mix of imperative programming and declarative tools (parser generators, code generators, and tree walkers) to glue a system together. Each phase of the translation is exposed and programmers may choose their own data structures. There is also a very large market for text generation tools unrelated to text translation such dynamic web page generation from databases, email message generartion, and code generation from UML diagrams. ST is aimed directly at both hybrid-approach translation systems and direct data-to-text generation systems.

The idea of a template engine for text generation is not new and a few systems have been around for many years such as PTG, a component of Eli [12]. Indeed, the idea of "mail merge" for replacing addresses in form letters has been around for decades. With the explosion of dynamically-generated web sites came the proliferation of template engines, PHP and JSP being two of the earliest players. Today, dozens of engines are in use such as Jostraca [13], Velocity [24], FreeMarker [10], Tea [22], XMLC [26], Smarty [20], and Django [9]. All but XMLC are essentially Turing-complete languages or allow arbitrary calls to the model, violating model-separation in that templates can then embed "business logic" or alter the model from within the template. Django's template language's philosophy is the closest to that of ST (though it is encumbered by a web framework), but unfortunately supports arbitrary method calls to the model. XMLC, on the other hand, is so restrictive that HTML output literals must often be pushed into the template from the controller, another violation. While providing dangerous features, most engines miss the key elements, lazy evaluation and recursion being the most egregious omissions. Ironically, most of the reference manuals for template engines either espouse or claim to enforce model-view separation.

ST distinguishes itself by its strict enforcement of model-view separation, which it achieves primarily through disallowing side-effects and computations within the template. The normal imperative programming language features like assignment, loops, arithmetic expressions, and arbitrary method calls into the model are not only unnecessary, but they are very specifically what cannot be allowed if an engine is to enforce model-view separation [18]. Restricting a language so is tantamount to requiring a functional language.

ST is remarkably similar to Backus' FP functional language proposal [2] right down to having single and multi-valued attributes (what Backus calls *atoms* and *sequences*). Both FP and ST use the ":" operator for function/template application, albeit with the function and operand reversed: FP's `f:x` versus ST's `x:f()`. ST uses ":" for both apply and apply-to-all where FP used $f : x$ and $\alpha f : x$. That is, ST's `x:f():g()` means $g(f(x))$ when x is single-valued and $[g(f(x_0)), ..., g(f(x_{n-1}))]$ when x is multi-valued. ST does not have a composition operator, but ST templates would compose easily because templates operate on other templates routinely. One advantage of template composition such as `x:(f . g)()` would be a reduction in intermediate, temporary data structure generation. ST has no explicit reduce/fold operation but template elements, t_i and e_i, are implicitly reduced using concatenate to form a result. ST and FP both share conditionals and sequence construction notation (`[a,b,c]`) as well as first and remainder sequence operations.

Most template engines are built by industrial programmers because the construction of web sites is not usually within the purview of academics (dynamic page generation is the source of the recent template engine proliferation). As such, there is very little academic work published specifically on stand-alone, general purpose template engines. Most academic work focuses on the applications of code generation such as implementing domain-specific languages (DSL) [17], using multi-stage programming to enhance generality [21], automating code-reuse through "software system families" [7], interactively manipulating programs [4], generating test cases [15], and automating the assembly of software components [8]. Research work is more concerned with what to generate and how that information is computed rather than the final text generation phase *per se*. ST could be used as a component of this work.

The reader may question why we need ST when many functional languages already exist. There are two important reasons: *(i)*

ST is a domain-specific language whose syntax and semantics are tuned to text generation and *(ii)* attribute expressions must be restricted to enforce strict model-view separation. There have been several interesting functional language based pretty printing systems in Haskell [25] [16] but these systems are not meant to integrate with commonly-used programming languages such as Java and Python. Moreover, in comparison to ST, they are cumbersome and not particularly readable, though they have more features.

ST is a general purpose text generation engine with a functional language that strikes a balance between power and enforcement of model-view separation. ST supports features unavailable in other actively-used engines such as template recursion, template polymorphism, group interfaces, and lazy attribute evaluation.

5. Future

ST is currently interpreted and, although it is very fast, significant speed improvements might come from generating Java code directly and executing the compiled code. This would be very similar to the way JSP files are translated to Java code, compiled, and then executed by a web server.

A valid concern regarding ST is that it does not verify that generated text conforms to a language. Because the templates together are essentially equivalent to a grammar, it might be worth exploring whether or not there is a reasonable way to guarantee correctness or at least conformity. In principle, one could restrict template arguments to be members of a set of template names (assuming the arguments are templates):

```
body(stats in (statement|vardef)) ::= "{ <stats> }"
```

The cardinality of attributes (single-valued, zero-or-more, etc...) could also be specified and verified. Template coverage analysis could also prove useful.

6. Conclusions

ST is a domain-specific language for generating structured text from internal data structures. It distinguishes itself by strictly enforcing separation of generation logic (model/controller) and output format (view). Separation is a key component of a generator because it makes generators easier to build, understand, modify, and retarget.

The very nature of generating sentences in an output language and the enforcement of separation dictate the general form of the solution. The rules for separation restrict template expressions in such a way that their capabilities coincide with the fundamentals of a pure functional language and that of an output grammar. ST embodies this solution and focuses on conceptual integrity, robustness, regularity, and efficiency (ST is fast and its binary is 165k). ST's feature set is driven by solving real problems encountered in complicated systems such as ANTLR version 3's retargetable code generator; features include group inheritance, polymorphism, lazy evaluation, recursion, auto-indentation, and the new notions of group interfaces and regions. Experience shows that ST is easy to learn and satisfying to use.

ST is available under the BSD license at `stringtemplate.org` for Java, C#, and Python.

7. Acknowledgments

I would like to acknowledge the authors of the ST C# port, Michael Jordan and Kunle Odutola, and the authors of ST Python port, Marq Kole and Luis Leal. Thomas E. Burns co-designed many of the features in ST. Jim Idle co-designed the group interfaces. Ric Klaren co-designed template regions. John Mitchell, Loring Craymer, Monty Zukowski, Matthew Ford, and Sriram Srinivasan have provided very useful feedback and lots of ideas as have the many ST and ANTLR users.

References

[1] ANTLR. `http://www.antlr.org`

[2] John Backus. *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of programs*, Communications of the ACM Volume 21, Number 8, August 1978.

[3] I. Baxter 1992. *Design Maintenance Systems*, Communications of the ACM 35(4), April, 1992, ACM, New York.

[4] Marat Boshernitsan and Susan L. Graham. *iXj: interactive source-to-source transformations for Java*. OOPSLA Companion 2004, pp 212-213.

[5] John Boyland and Susan L. Graham, *Composing Tree Attributions*. Proc. of Principles of Programming Languages 1994; Portland, Oregon, USA.

[6] J.R. Cordy. *TXL - A Language for Programming Language Tools and Applications*, Proc. of the ACM 4th International Workshop on Language Descriptions, Tools and Applications (LDTA 2004), Barcelona, Spain, April 2004, pp. 1-27.

[7] K. Czarnecki. *Overview of Generative Software Development*. In Unconventional Programming Paradigms (UPP) 2004, Mont Saint-Michel, France, LNCS 3566, pp. 313328, 2005.

[8] Krzysztof Czarnecki and Ulrich W. Eisenecker, Components and generative programming, Proceedings of the 7th European software engineering conference. Toulouse, France. pp 2-19, 1999.

[9] Django. `http://www.djangoproject.com`

[10] FreeMarker. `http://freemarker.sourceforge.net`

[11] Graphviz `http://www.graphviz.org/`

[12] Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, William M. Waite. *Eli: a complete, flexible compiler construction system*, Communications of the ACM archive, Volume 35, Issue 2, February 1992, pp. 121-130.

[13] Jostraca. `http://www.jostraca.org/`

[14] Donald E. Knuth. *Semantics of context-free languages*, Theory of Computing Systems, Volume 2, Number 2 June 1968, pp. 127-145. Springer, New York.

[15] Bogdan Korel. *Automated software test data generation*. IEEE Transactions on Software Engineering, 16(8):870879, 1990.

[16] Daan Leijen, *PPrint, a prettier printer*, `http://www.cs.uu.nl/people/daan/download/pprint/pprint.html`

[17] Andrew Moss and Henk Muller. *Efficient Code Generation for a Domain Specific Language*. In Generative Programming and Component Engineering, pp. 47-62. Springer Verlag, September 2005.

[18] Terence Parr. *Enforcing Strict Model-View Separation in Template Engines*. In WWW2004 Conference Proceedings p. 224, May 17-20 2004, New York City.

[19] Terence Parr. *Web Application Internationalization and Localization in Action*. To appear in proceedings of The International Conference on Web Engineering, 2006. Palo Alto, California USA.

[20] Smarty. `http://smarty.php.net`

[21] Walid Taha. *A gentle introduction to multi-stage programming*. In Don Batory, Charles Consel, Christian Lengauer, and Martin Odersky, editors, Domain-specific Program Generation, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[22] Tea. `http://teatrove.sourceforge.net/tea.html`

[23] E. Visser. *Stratego: A language for program transformation based on rewriting strategies* Rewriting Techniques and Applications (RTA'01), volume 2051 of Lecture Notes in Computer Science, pp. 357-361. Springer-Verlag, May 2001

[24] Velocity. `http://jakarta.apache.org/velocity/index.html`

[25] Philip Wadler. *A prettier printer*, Unpublished manuscript, 1998.

[26] XMLC. `http://xmlc.enhydra.org`