StringTemplate : StringTemplate 3.0 Printable Documentation

This page last changed on Oct 23, 2006 by kunle_odutola@hotmail.com.

Terence Parr
University of San Francisco
parrt AT cs.usfca.edu

C# version created by Kunle Odutola
kunle UNDERSCORE odutola AT hotmail.com

Python version created by Marq Kole
marq DOT kole AT xs4all DOT nl

Contents

Related material

- DRAFT A Functional Language For Generating Structured Text
- Internationalization and Localization of Web Applications In Action
- Language Translation Using ANTLR and StringTemplate
- Intelligent Web Site Page Generation
- The Role of Template Engines in Code Generation

It is highly recommended that you read the (academically-oriented) paper, Enforcing Model-View Separation in Template Engines.

The `StringTemplates` distribution includes many unit tests that also represent a useful set of examples. The tests are defined in:

Java	`TestStringTemplate.java`
C#	`TestStringTemplate.cs`
Python	`TestStringTemplate.py`

Please see the StringTemplate 3.0 Release Notes and changes and bugs page. That page generally discusses the Java version of StringTemplate but, some of the information it contains might apply to other implementations.

StringTemplate 3.0 Release Notes

Brought to you by that maniac that brings you the ANTLR parser generator!

Terence Parr
University of San Francisco
parrt at cs dot usfca dot edu

Version 3.0, September 6, 2006

3.0 fixes lots of bugs and adds some great new features including:

- Group interfaces
- Template regions
- Automatic line wrapping
- Group loaders
- length(), strip() operators
- Map predefined operators keys and values for iterating maps.

The features were added in response to my needs building ANTLR v3's code generator and from feedback by StringTemplate users.

3.0 should be a drop-in replacement for those using ST for websites and code generation with a few minor potential incompatibilities as noted below. Also see the change list. The biggest issue is that now angle brackets are the default delimiter for StringTemplate groups.

Enhancements

- Added interfaces. See unit tests and http://www.cs.usfca.edu/~parrt/papers/ST.pdf.

```
group Java implements ANTLRCoreTarget;
rule(...) ::= "..."
...
```

  You can say "optional template(args);" also. Uses group loader to find interfaces.

- Added CommonGroupLoader so you can reference groups/interfaces now in group files etc...

```
StringTemplateGroup.registerGroupLoader(new CommonGroupLoader(dir,errorListener));
```

  The dir is a string that can be "dir1:dir2:dir3" etc... These are interpreted as relative paths to be used with CLASSPATH to locate groups. E.g., If you pass in "org/antlr/codegen/templates" and ask to load group "foo" it will try to load via classpath as "org/antlr/codegen/templates/foo". Another, PathGroupLoader, is a brain dead loader that looks only in the directory(ies) you specify in the ctor.

- Added group inheritance notation:

```
group subG : superG;
```

  Method setSuperGroup(name) now does something useful it invokes the loadGroup() method of the group loader.

- Added template regions that either marks a section of a template or leaves a "hole" that subgroups may alter. See http://www.cs.usfca.edu/~parrt/papers/ST.pdf

- added amazing line wrap facility.* Updated StringTemplateWriter to support line wrapping:

- map still iterates over values if you say <aMap> but it is now shorthand for <aMap.values>. Similarly <aMap.keys> walks over the keys. You can do stuff like: <aMap.keys:{k| <k> maps to <aMap.(k)>}>.

- added length(attribute) thanks to Kay Roepke. For now it does not work on strings; it works on attributes so length("foo") is 1 meaning 1 attribute. Nulls are counted in lists so a list of 300 nulls is length 300. If you don't want to count nulls, use
  length(strip(list)).

- added strip(attribute) that returns an iterator that skips any null values.
  strip(x)=x when x is a single-valued attribute. Added StripIterator to handle this.

- added ability to show start/stop of template with <name>...</name>.
  emitTemplateStartDebugString/emitTemplateStopDebugString and
  doNotEmitDebugStringsForTemplate(tempaltename)

- Added null=expr option on expressions. For null values in iterated attributes and single attributes that are null, use this value instead of skipping. For single valued attributes like <name; null="n/a">. It's a shorthand for

```
<if(name)><name><else>n/a<endif>
```

  For iterated values

```
<values; null="0", separator=",">
```

  you get 0 for null list values. Works for template application
  like this also:

```
<values:{v| <v>}; null="0">
```

  This does not replace empty strings like "" as they are not null.

- added build.xml ANT file (ick)

- {} and "" work as arguments now to templates

- note in doc that map strings are now templates and that <<...>> works too. default or others can have empty values (implying no value) or use "key" but not in template; it's a keyword. Also, default must be at end now (and only 1). Default value is empty as before. To return null, use "default :" at end. Can use empty values too:
  {"float":}, {"int":"0"}, ...

- added i0 which is like i except indexed from 0 not 1.

- Added StringTemplate.getDependencyGraph() to get a list of n->m edges where template n contains template m. Also made convenient getDOTForDependencyGraph(). You get a template back that lets you reset node shape, fontsize, width, height attributes. Use removeAttribute before setting so you are sure you only get one value.

- Added StringTemplate.toStructureString() to help discover nested structure.

- added toString to Aggregate so that referencing foo in isolation prints something useful when you have the following code: setAttribute("foo.{x,y}", ...).

- Added default lexer mechanism for groups so you can set once:
  public static void registerDefaultLexer(Class lexerClass)

- Improved error when property ref causes the error internally. Now shows that exception rather than invocation target exception.

- Added STG.getInstanceOf(name,attributes)

Backward incompatibilities

The following changes were worth making despite causing some backward compatibilities for some users.

- <...> is now default expression delimiter for group files.

- I had to make a more clear distinction between empty and null. Null means there is nothing there whereas empty means the value is there but renders to an empty string. IF conditionals now evaluate to empty if condition is false.

- Changed how separators are generated. Now I generate a separator for any non-null value in the list even if that value is a conditional that evaluates to false. Iterated empty values always get a separator. Note that empty is not the same thing as missing. "" is not missing and hence will get a separator.

This made the ASTExpr.write separator computation much simpler and allowed me to properly handle the new "null" option.

Bug Fixes

- allow different source of classloader:

```
InputStream is = cl.getResourceAsStream(fileName);
if ( is==null ) \{
        cl = ErrorManager.class.getClassLoader();
        is = cl.getResourceAsStream(fileName);
}
```

- fixed bug where separator did not appear in lists of stuff with lots of null entries.

- made static maps in STG synchronized, also synchronized the look up/def methods for templates in STG.

- removed reflection property lookup; too complex for value. profiling indicates it's a small cost. No thread synch issues either now.

- template evaluation for anonymous templates did not properly look up overridden templates. The

group for anonymous templates was not reset to be the current enclosing template's group (remember we can dynamically set the superGroup).

- Signature changed to use AttributeRenderer:

```
public void registerRenderer(Class attributeClassType,
                             AttributeRenderer renderer)
```

- If expr in <(expr):template()> evaluated to nothing, the template was still invoked.
- Couldn't handle List<int[]>. Nested int[] was not iteratable.
- Couldn't gen \ in a template. \<value> in a template escaped the <.
  <value> printed both slashes. Fixed.
- Couldn't use

```
\{
```

  inside of a {...} anonymous template.

- Fixed: you could not have template expressions, just simple expressions in indirect template expressions like

```
$data:("foo":a())()$}}
```

  . I decided not to allow IF expressions inside.

- now throws exception when dots are in template names or attribute names
- don't set "it" nor "attr" default attributes if there is a parameter to an anonymous block like <names:{n | ...}>
- Bug fix. If you passed in a list and then another element, it added it to the list you passed in! Now, I make a (shallow) copy of the list.
- Bug fix. If you passed an element then a list to an attr and I think it didn't flatten out properly!
- Couldn't escape in template group. \<< failed as did ...} for anonymous templates.
- When creating an aggregate list, couldn't have spaces in {...} such as "folders.{a, b}".
- Embedded templates such as {...} anonymous templates couldn't see the renderers for enclosing templates. Easy to set one renderer in root template for, say, Date and forget about it. If none found for that class in containment hierarchy, then group hierarchy is checked.
- using nativegroup to create instances now in ST.getInstanceOf. Needed so that super.foo works properly. THe new instance's group will point at creating group so polymorphism works properly.
- auto defined attribute i was not defined for <a,b: {...}> case. ooops.

- You couldn't have '=' in a string if preceded by '+' like foo+"ick=".
- Fixed bug where an expr on the first line that yields no output left a blank line.
- There was a bug in StringTemplateGroup.isDefined that it always returned true.
- If you iterated multiple empty attributes, it iterated once with null values. Bizarre. Fixed.
- Bug in polymorphism when an overridden template called its super which called another template which was also overridden; didn't call the overridden method. Fixed getInstanceOf() so that it always sets the proper group so polymorphic template lookup also starts at the right group.
- Test testLazyEvalOfSuperInApplySuperTemplateRef was wrong...The "super." prefix is (like in Java, ...) a scope override and is always evaluated relative to the template in which it was defined not the template in which it is evaluate!
- Improving error messages to be more specific when you get a parser error. I'm including more context info and hopefully the file within a group file the error occurs.
- The defaultGroup is now public so you can know StringTemplate's default group when you see it:

```
public static StringTemplateGroup defaultGroup =
            new StringTemplateGroup("defaultGroup", ".");
```

- Template polymorphism bug: wouldn't work for references in IF clauses!
- Any Map instance is now allowed as a "map" attribute not just Hashtable, HashMap.
- NullPtrExc if you registered a renderer and sent an object as an attribute of another type!
- The tree viewer didn't work; class cast problem with Hashtable vs HashMap.

Introduction

Most programs that emit source code or other text output are unstructured blobs of generation logic interspersed with print statements. The primary reason is the lack of suitable tools and formalisms. The proper formalism is that of an output grammar because you are not generating random characters--you are generating sentences in an output language. This is analogous to using a grammar to describe the structure of input sentences. Rather than building a parser by hand, most programmers will use a parser generator. Similarly, we need some form of *unparser generator* to generate text. The most convenient manifestation of the output grammar is a template engine such as `StringTemplate`.

A template engine is a simply a code generator that emits text using templates, which are really just "documents with holes" in them where you can stick values. `StringTemplate` breaks up your template into chunks of text and attribute expressions, which are by default enclosed in dollar signs $*attribute-expression*$ (to make them easy to see in HTML files). `StringTemplate` ignores everything outside of attribute expressions, treating it as just text to spit out when you call:

Java	`StringTemplate.toString()`
C#	`StringTemplate.ToString()`
Python	`StringTemplate._str_()`

For example, the following template has two chunks, a literal and a reference to attribute `name`:

```
Hello, $name$
```

Using templates in code is very easy. Here is the requisite example that prints "`Hello, World`":

Java

```
import org.antlr.stringtemplate.*;

StringTemplate hello = new
StringTemplate("Hello, $name$");
hello.SetAttribute("name", "World");
System.out.println(hello.toString());
```

C#

```
using Antlr.StringTemplate;

StringTemplate hello = new
StringTemplate("Hello, $name$");
hello.SetAttribute("name", "World");
Console.Out.WriteLine(hello.ToString());
```

Python

```
import stringtemplate

hello =
stringtemplate.StringTemplate("Hello,
$name$")
hello["name"] = "World"
print str(hello)
```

`StringTemplate` is not a "system" or "engine" or "server"; it is a library with two primary classes of interest: `StringTemplate` and `StringTemplateGroup`. You can directly create a `StringTemplate` in code, you can load a template from a file, and you can load a single file with many templates (a template group file).

Motivation And Philosophy

`StringTemplate` was born and evolved during the development of http://www.jGuru.com. The need for such dynamically-generated web pages has led to the development of numerous other template engines in an attempt to make web application development easier, improve flexibility, reduce maintenance costs, and allow parallel code and HTML development. These enticing benefits, which have driven the proliferation of template engines, **derive entirely from a single principle**: separating the specification of a page's business logic and data computations from the specification of how a page displays such information.

These template engines are in a sense are a reaction to the completely entangled specifications encouraged by JSP (Java Server Pages), ASP (Active Server Pages) and, even ASP.NET. With separate

encapsulated specifications, template engines promote component reuse, pluggable site "looks", single-points-of-change for common components, and high overall system clarity. In the code generation realm, model-view separation guarantees retargetability.

The normal imperative programming language features like setting variables, loops, arithmetic expressions, arbitrary method calls into the model, etc... are not only unnecessary, but they are very specifically what is wrong with ASP/JSP. Recall that ASP/JSP (and ASP.NET) allow arbitrary code expressions and statements, allowing programmers to incorporate computations and logic in their templates. A quick scan of template engines reveals an unfortunate truth--all but a few are Turing-complete languages just like ASP/JSP/ASP.NET. One can argue that they are worse than ASP/JSP/ASP.NET because they use languages peculiar to that template engine. Many tool builders have clearly lost sight of the original problem we were all trying to solve. We programmers often get caught up in cool implementations, but we should focus on what **should** be built not what **can** be built.

The fact that StringTemplate does not allow such things as assignments (no side-effects) should make you suspicious of engines that do allow it. The templates in ANTLR v3's code generator are vastly more complicated than the sort of templates typically used in web pages creation with other template engines yet, there hasn't been a situation where assignments were needed. If your template looks like a program, it probably is--you have totally entangled your model and view.

After examining hundreds of template files that I created over years of jGuru.com (and now in ANTLR v3) development, I found that I needed only the following four basic canonical operations (with some variations):

- attribute reference; e.g., `$phoneNumber$`
- template reference (like #include or macro expansion); e.g., `$searchbox()$`
- conditional include of subtemplate (an IF statement); e.g., `$if(title)$<title>$title$</title>$endif$`
- template application to list of attributes; e.g., `$names:bold()$`

where template references can be recursive.

Language theory supports my premise that even a minimal StringTemplate engine with only these features is very powerful--such an engine can generate the context-free languages (see Enforcing Strict Model-View Separation in Template Engines); e.g., most programming languages are context-free as are any XML pages whose form can be expressed with a DTD.

While providing all sorts of dangerous features like assignment that promote the use of computations and logic in templates, many engines miss the key elements. Certain language semantics are absolutely required for generative programming and language translation. One is *recursion*. A template engine without recursion seems unlikely to be capable of generating recursive output structures such as nested tables or nested code blocks.

Another distinctive `StringTemplate` language feature lacking in other engines is *lazy-evaluation*. `StringTemplate`'s attributes are lazily evaluated in the sense that referencing attribute "a" does not actually invoke the data lookup mechanism until the template is asked to render itself to text. Lazy evaluation is surprising useful in both the web and code generation worlds because such order decoupling allows code to set attributes when it is convenient or efficient not necessarily before a template that references those attributes is created. For example, a complicated web page may consist of many nested templates many of which reference `$userName$`, but the value of `userName` does not need to be set by the model until right before the entire page is rendered to text via `ToString()`. You can build up the

complicated page, setting attribute values in any convenient order.

`StringTemplate` implements a "poor man's" form of lazy evaluation by simply requiring that all attributes be computed *a priori*. That is, all attributes must be computed and pushed into a template before it is written to text; this is the so-called "*push method*" whereas most template engines use the "*pull method*". The pull method appears more conventional because programmers mistakenly regard templates as programs, but pulling attributes introduces *order-of-computation dependencies*. Imagine a simple web page that displays a list of names (using some mythical Java-based template engine notation):

```
<html>
<body>
<ol>
$foreach n in names$
  <li>$n$</li>
$end$
</ol>
There are $numberNames$ names.
</body>
</html>
```

Using the pull method, the reference to `names` invokes `model.getNames()`, which presumably loads a list of names from the database. The reference to `numberNames` invokes `model.getNumberNames()` which necessarily uses the internal data structure computed by `getNames()` to compute `names.size()` or whatever. Now, suppose a designer moves the `numberNames` reference to the `<title>` tag, which is **before** the reference to `names` in the `foreach` statement. The names will not yet have been loaded, yielding a null pointer exception at worst or a blank title at best. You have to anticipate these dependencies and have `getNumberNames()` invoke `getNames()` because of a change in the template.

I'm stunned that other template engine authors with whom I've spoken think this is ok. Any time I can get the computer to do something automatically for me that removes an entire class of programming errors, I'll take it!. Automatic garbage collection is the obvious analogy here.

The pull method requires that programmers do a topological sort in their minds anticipating any order that a programmer or designer could induce. To ensure attribute computation safety (i.e., avoid hidden dependency landmines), I have shown trivially in my academic paper that *pull* reduces to *push* in the worst case. With a complicated mesh of templates, you will miss a dependency, thus, creating a really nasty, difficult-to-find bug.

StringTemplate mission

When developing StringTemplate, I recalled Frederick Brook's book, "Mythical Man Month", where he identified *conceptual integrity* as a crucial product ingredient. For example, in UNIX everything is a stream. My concept, if you will, is *strict model-view separation*. My mission statement is therefore:

"StringTemplate shall be as simple, consistent, and powerful as possible without sacrificing strict model-view separation."

I ruthlessly evaluate all potential features and functionality against this standard. Over the years, however, I have made certain concessions to practicality that one could consider as infringing ever-so-slightly into potential model-view entanglement. That said, StringTemplate still seems to enforce separation while providing excellent functionality.

I let my needs dictate the language and tool feature set. The tool evolved as my needs evolved. I have done almost no feature "backtracking". Further, I have worked really hard to make this little language self-consistent and consistent with existing syntax/metaphors from other languages. There are very few special cases and attribute/template scoping rules make a lot of sense even if they are unfamiliar or strange at first glance. Everything in the language exists to solve a very real need.

StringTemplate language flavor

Just so you know, I've never been a big fan of functional languages and I laughed really hard when I realized (while writing the academic paper) that I had implemented a functional language. The nature of the problem simply dictated a particular solution. We are generating sentences in an output language so we should use something akin to a grammar. Output grammars are inconvenient so tool builders created template engines. Restricted template engines that enforce the universally-agreed-upon goal of strict model-view separation also look remarkably like output grammars as I have shown. So, the very nature of the language generation problem dictates the solution: a template engine that is restricted to support a mutually-recursive set of templates with side-effect-free and order-independent attribute references.

Defining Templates

Creating Templates With Code

Here is a simple example that creates and uses a template on the fly:

Java

```
StringTemplate query = new
StringTemplate("SELECT $column$ FROM
$table$;");
query.setAttribute("column", "name");
query.setAttribute("table", "User");
```

C#

```
StringTemplate query = new
StringTemplate("SELECT $column$ FROM
$table$;");
query.SetAttribute("column", "name");
query.SetAttribute("table", "User");
```

Python

```
import stringtemplate

  query =
stringtemplate.StringTemplate("SELECT
$column$ FROM $table$;")
  query["column"] = "name"
  query["table"] = "User"
```

where `StringTemplate` considers anything in `$...$` to be something it needs to pay attention to. By setting attributes, you are "pushing" values into the template for use when the template is printed out. The attribute values are set by referencing their names. Invoking {{toString()}} on `query` would yield

```
SELECT name FROM User;
```

You can set an attribute multiple times, which simply means that the attribute is multi-valued. For example, adding another value to the attribute named `column` as shown below makes the attribute multi-valued:

Java

```
StringTemplate query = new
StringTemplate("SELECT $column$ FROM
$table$;");
query.setAttribute("column", "name");
query.setAttribute("column", "email");
query.setAttribute("table", "User");
```

C#

```
StringTemplate query = new
StringTemplate("SELECT $column$ FROM
$table$;");
query.SetAttribute("column", "name");
query.SetAttribute("column", "email");
query.SetAttribute("table", "User");
```

Python

```
query =
stringtempatle.StringTemplate("SELECT
$column$ FROM $table$;")
  query["column"] = "name"
  query["column"] = "email"
  query["table"] = "User"
```

Invoking {{toString()}} on `query` would now yield

```
SELECT nameemail FROM User;
```

Ooops...there is no separator between the multiple values. If you want a comma, say, between the column names, then change the template to record that formatting information:

Java

```
StringTemplate query = new
StringTemplate("SELECT $column;
separator=\",\"$ FROM $table$;");
query.setAttribute("column", "name");
query.setAttribute("column", "email");
query.setAttribute("table", "User");
```

C#

```
StringTemplate query = new
StringTemplate("SELECT $column;
separator=\",\"$ FROM $table$;");
query.SetAttribute("column", "name");
query.SetAttribute("column", "email");
query.SetAttribute("table", "User");
```

Python

```
query =
stringtemplate.StringTemplate("SELECT
```

	`$column; separator=\",\"$ FROM $table$;")`
	`query["column"] = "name"`
	`query["column"] = "email"`
	`query["table"] = "User`

Note that the right-hand-side of the separator specification in this case is a string literal; therefore, we have escaped the double-quotes as the template is specified in a string. In general, the right-hand-side can be any attribute expression. Invoking {{toString()}} on `query` would now yield

```
SELECT name,email FROM User;
```

Attributes can be any object at all. `StringTemplate` calls {{toString()}} on each object as it writes the template out. The separator is not used unless the attribute is multi-valued.

Loading Templates From Files

To load a template from the disk you must use a `StringTemplateGroup` that will manage all the templates you load, caching them so you do not waste time talking to the disk for each template fetch request (you can change it to not cache; see below). You may have multiple template groups. Here is a simple example that loads the previous SQL template from a file `/tmp/theQuery.st`:

```
SELECT $column; separator=","$ FROM $table$;
```

The code below creates a `StringTemplateGroup` called `myGroup` rooted at `/tmp` so that requests for template `theQuery` forces a load of file `/tmp/theQuery.st`.

Java	`StringTemplateGroup group = new`
	`StringTemplateGroup("myGroup", "/tmp");`
	`StringTemplate query =`
	`group.getInstanceOf("theQuery");`
	`query.setAttribute("column", "name");`
	`query.setAttribute("column", "email");`
	`query.setAttribute("table", "User");`

C#	`StringTemplateGroup group = new`
	`StringTemplateGroup("myGroup", "/tmp");`
	`StringTemplate query =`
	`group.GetInstanceOf("theQuery");`
	`query.SetAttribute("column", "name");`
	`query.SetAttribute("column", "email");`
	`query.SetAttribute("table", "User");`

Python	`group =`
	`stringtemplate.StringTemplateGroup("myGroup",`
	`"/tmp")`
	`query = group.getInstanceOf("theQuery")`
	`query["column"] = "name"`
	`query["column"] = "email"`
	`query["table"] = "User"`

If you have a directory hierarchy of templates such as file `/tmp/jguru/bullet.st`, you would reference them relative to the root; in this case, you would ask for template `jguru/bullet()`.

Note

StringTemplate strips whitespace from the front and back of all loaded template files. You can add, for example, `<\n>` at the end of the file to get an extra carriage return.

Loading Templates relative to an implementation specific location

Java

Loading Templates from CLASSPATH

When deploying applications or providing a library for use by other programmers, you will not know where your templates files live specifically on the disk. You will, however, know relative to the classpath where your templates reside. For example, if your code is in package `com.mycompany.server` you might put your templates in a `templates` subdirectory of `server`. If you do not specify an absolute directory with the `StringTemplateGroup constructor`, future loads via that group will happen relative to the `CLASSPATH`. For example, to load template file `page.st` you would do the following:

```
// Look for templates in CLASSPATH as
resources
StringTemplateGroup group = new
StringTemplateGroup("mygroup");
StringTemplate st =
group.getInstanceOf("com/mycompany/server/templates/page"
```

C#

Loading Templates relative to the Assembly's Location

When deploying applications or providing a library for use by other programmers, you will not know in advance where your templates files will be located live in the file system. You will, however, often know the location of your templates relative to the where the application assembly is deployed. For example, if your code is in the an assembly named `com.mycompany.server.exe` you might put your templates in a `templates` subdirectory of the directory containing `com.mycompany.server.exe`. If you do not specify an absolute directory with the `StringTemplateGroup` constructor, future loads via that group will happen relative to the location

	of `com.mycompany.server.exe`. For example, to load template file `page.st` you would do the following:
	
	`// Look for templates relative to assembly location`
	`StringTemplateGroup group = new StringTemplateGroup("mygroup", (string)null);`
	`StringTemplate st = group.GetInstanceOf("templates/page");`

Python	**Loading Templates from sys.path**
	
	When deploying applications or providing a library for use by other programmers, you will not know where your templates files live specifically on the disk. You will, however, know relative to the `classpath` where your templates reside. For example, if your code is in package `com.mycompany.server` you might put your templates in a `templates` subdirectory of `server`. If you do not specify an absolute directory with the `StringTemplateGroup` constructor, future loads via that group will happen relative to the `sys.path`. For example, to load template file `page.st` you would do the following:
	
	`# Look for templates in CLASSPATH as resources`
	`group = stringtemplate.StringTemplateGroup("mygroup")`
	`st = group.getInstanceOf("com/mycompany/server/templates/page"`

If `page.st` references, say, `searchbox` template, it must be fully qualified as:
`<font size=2>SEARCH</font>: $com/mycompany/server/templates/page/searchbox()$`

This is inconvenient and ST may add the invoking template's path prefix automatically in the future.

Caching

By default templates are loaded from disk just once. During development, however, it is convenient to turn caching off. Also, you may want to turn off caching so that you can quickly update a running site. You can set a simple refresh interval using `StringTemplateGroup.setRefreshInterval(...)`. When the interval is reached, all templates are thrown out. Set interval to 0 to refresh constantly (no caching). Set the interval to a huge number like `Integer.MAX_INT` or `Int32.MaxValue` to have no refreshing at all.

Group Files

StringTemplate 2.0 introduced the notion of a group file that has two main attractions. First, it allows you to define lots of small templates more conveniently because they may all be defined within a single file. Second, you may specify formal template arguments that help StringTemplate detect errors (such as setting unknown attributes) and make the templates easier to read. Here is a sample group file with two templates, vardef and method, that could be used to generate C files:

```
group simple;

vardef(type,name) ::= "<type> <name>;"

method(type,name,args) ::= <<
<type> <name>(<args; separator=",">) {
  <statements; separator="\n">
}
>>
```

All groups use <...> delimiters by default. Single line templates are enclosed in double quotes while multi-line templates are enclosed in double angle-brackets. Every template must define arguments even if the formal argument list if blank.

Using templates in a group file is straightforward. The StringTemplateGroup class has a number of constructors, one of which allows you to pass in a string or file or whatever:

Java

```
String templates = "group simple;
vardef(type,name) ..."; // templates from
above
// Use the constructor that accepts a
Reader
StringTemplateGroup group = new
StringTemplateGroup(new
StringReader(templates));
StringTemplate t =
group.getInstanceOf("vardef");
t.setAttribute("type", "int");
t.setAttribute("name", "foo");
System.out.println(t);
```

C#

```
String templates = "group simple;
vardef(type,name) ..."; // templates from
above
// Use the constructor that accepts a
System.IO.TextReader
StringTemplateGroup group = new
StringTemplateGroup(new
StringReader(templates));
StringTemplate t =
group.GetInstanceOf("vardef");
t.SetAttribute("type", "int");
t.SetAttribute("name", "foo");
Console.Out.WriteLine(t);
```

Python

```
templates = "group simple;
vardef(type,name) ..."; # templates from
above
# Use the constructor that accepts a Reader
group =
```

```
stringtemplate.StringTemplateGroup(StringIO(templates))
t = group.getInstanceOf("vardef")
t["type"] = "int"
t["name"] = "foo"
print str(t)
```

The output would be: "`int foo;`".

Supergroups and interfaces

Groups may derive from other groups, thus, inheriting all of the templates from the supergroup. 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. Considering Java 1.4 versus 1.5, a `Java1_5` group could specify how to alter the main `Java` (1.4) group templates in order to use generics and enumerated types.

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 that template is lexically defined within group `Java`.

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.

The supergroup for a group may be changed dynamically using the `setSuperGroup()` method. If, however, a group must always derive from another group, use the following syntax:

```
group mygroup : supergroup;
...
```

If your group must satisfy a particular interface (see Group interfaces) and use the following syntax:

```
group mygroup implements anInterface, andAnotherInterface;
...
```

or

```
group mygroup : supergroup implements anInterface;
...
```

if the group derives from a supergroup and implements an interface.

Maps

There are situations where you need to translate a string in one language to a string in another language. For example, you might want to translate `integer` to `int` when translating Pascal to C. You could pass a `Map` or `IDictionary` (e.g. hashtable) from the model into the templates, but then you have output literals in your model! The only solution is to have `StringTemplate` support mappings. For example, here is how

ANTLR v3 knows how to initialize local variables to their default values:

```
typeInitMap ::= [
        "int":"0",
        "long":"0",
        "float":"0.0",
        "double":"0.0",
        "boolean":"false",
        "byte":"0",
        "short":"0",
        "char":"0",
        default:"null" // anything other than an atomic type
]
```

To use the map in a template, refer to it as you would an attribute For example, `<typeInitMap.int>` which returns "`0`". If your type name is an attribute not a constant like `int`, then use an indirect field access: `<typeInitMap.(typeName)>`.

Map strings are actually templates that can refer to attributes that will become visible via dynamic scoping of attributes once the map entry has been embedded within a template. This is useful for referencing things like attribute `username` from within map values. That attribute will eventually become visible when the map a value is embedded within, say, a page template.

Large strings, such as those with newlines, can be specified with the usual large template delimiters from the group file format: `<<...>>`.

The `default` and other mappings can have empty values (implying no value). if no key is matched by the map then an empty value is returned, which is the same as using "`default :`" explicitly. The keyword `key` is available if you would like to refer to the key that maps to this value. This is particularly useful if you would like to filter certain words but otherwise leave a value unchanged; use `default : key` to return the key unmolested if it is not found in the map.

Maps are defined in the group's scope and are visible if no attribute hides them. For example, if you define a formal argument called `typeInitMap` in template `foo` then `foo` cannot see the map defined in the group (though you could pass it in as another parameter). If a name is not an attribute and it's not in the group's maps table, then the super group is consulted etc... You may not redefine a map and it may not have the same name as a template in that group. The `default` value is used if you use a key as a property that doesn't exist. For example `<typeInitMap.foo>` returns "`null`". The default clause must be at the end of the map.

You'll note that the square brackets will denote *data structure* in other areas too such as `[a,b,c,...]` which makes a singe multi-valued attribute out of other attributes so you can iterate across them.

Group file format

```
group
    :   "group" ID ( ':' ID  )?  ( "implements" ID (',' ID  )* )? ';'
        ( template | mapdef )+
    ;

template
    :   (   '@' ID '.' ID
        |   ID
        )
        '(' (args)? ')' "::="
        (   STRING      // "..."
```

```
        |   BIGSTRING   // <<...>>
        )
    |   ID "::=" ID     // alias one template to another
    ;

args:   arg ( ',' arg )*
    ;

arg :   ID '=' STRING               // x="..."
    |   ID '=' ANONYMOUS_TEMPLATE   // x={...}
    ;

mapdef
    :   ID "::=" map
    ;

map :   '['
            keyValuePair (',' keyValuePair)*
            ( ',' "default" ':' keyValue )?
        ']'
    ;

keyValuePair
    :   STRING ':' keyValue
    ;

keyValue
    :   BIGSTRING
    |   STRING
    |   "key"
    |
    ;
```

Both `/* ... */` and single-line `// ...` comments are allowed outside of templates. Inside templates, you must use `<!...!>`.

An aside: All along, during my website construction days, I kept in mind that any text output follows a format and, thus, output sentences conform to a language. Consequently, a grammar should describe the output rather than a bunch of ad hoc print statements in code. This helped me formalize the study of templates because I could compare templates (output grammars) to well established ideas from formal language theory and context-free grammars. This allowed me to show, among other things, that `StringTemplate` can easily generate any document describable with an XML DTD even though it is deliberately limited. The group file format should look very much like a grammar to you.

Scoping rules and attribute look-up

See the scoping rules section for information on how formal arguments affect attribute look up.

Group files have a `.stg` file extension.

Group loaders

When group files derive from another group, StringTemplate has to know how to load that group and its supergroups. StringTemplate 2.3 introduces the {{StringTemplateGroupLoader }} interface to describe objects that know how to load groups and interfaces.

```
public interface StringTemplateGroupLoader {
    /** Load the group called groupName from somewhere.  Return null
     *  if no group is found.
     */
    public StringTemplateGroup loadGroup(String groupName);

    /** Load a group with a specified superGroup.  Groups with
     *  region definitions must know their supergroup to find templates
     *  during parsing.
     */
    public StringTemplateGroup loadGroup(String groupName,
                                         StringTemplateGroup superGroup);


    /** Load the interface called interfaceName from somewhere.  Return null
     *  if no interface is found.
     */
    public StringTemplateGroupInterface loadInterface(String interfaceName);
}
```

By default, there are two implementations: `PathGroupLoader` and `CommonGroupLoader`. `PathGroupLoader` is a simple loader that looks only in the directory(ies) you specify in the ctor (Note that you can specify the char encoding). `CommonGroupLoader`, on the other hand, is a loader that also looks in the directory(ies) you specify in the ctor, but it uses the classpath rather than absolute dirs so it can be used when the ST application is jar'd up. Use Static method:

```
StringTemplateGroup.registerGroupLoader(loader);
```

to specify a loader. For example, here is how ANTLR loads its templates:

```
/ get a group loader containing main templates dir and target subdir
String templateDirs =
    classpathTemplateRootDirectoryName+":"+
    classpathTemplateRootDirectoryName+"/"+language;
StringTemplateGroupLoader loader =
    new CommonGroupLoader(templateDirs.toString(),
                          ErrorManager.getStringTemplateErrorListener());
StringTemplateGroup.registerGroupLoader(loader);

// first load main language template
StringTemplateGroup coreTemplates =
    StringTemplateGroup.loadGroup(language);
```

In order to use the group file format inheritance specifier, `group sub : sup`, you must specify a loader.

Formal argument default values

Sometimes it is convenient to have default values for formal arguments that are used when no value is set by the model. For example, when generating a parser in Java from ANTLR, I want the super class of the generated object to be `Parser` unless the ANTLR user uses an option to set the super class to some custom class. For example, here is a partial `parser` template definition:

```
parser(name, rules, superClass="Parser") ::= ...
```

Any argument may be given a default value by following the name with an equals sign and a string or an anonymous template.

Formal argument error handling

When using a group file format to specify templates, you must specify the formal arguments for that template. If you try to set an attribute via `setAttribute` that is not specifically formally defined in that template, you will get the following exception:

Java	`NoSuchElementException`
C#	`InvalidOperationException`
Python	`NoSuchElementException`

If you reference an attribute that is not formally defined in that template or any enclosing template, you also get the same exception.

Newline handling

The first newline following the << in a template definition is ignored as it is usually used just to get the first line of text for the template at the start of a line. In other words, if you want to have a blank line at the start of your template, use:

```
foo() ::= <<

2nd line is not blank, but first is
>>
```

or

```
foo() ::= <<<\n>
same as before; newline then this line
>>
```

The last newline before the >> is also ignored and is included in the output. To add a final newline, add an extra line or <\n> before the >>:

```
foo() ::= <<
rodent

>>
```

or

```
foo() ::= <<
rodent<\n>
>>
```

The following template:

```
foo() ::= <<
```

```
rodent
>>
```

on the other hand, is identical to

```
foo() ::= "rodent"
```

Group Interfaces

To promote retargetable code generators, ST supports *interface implementation* a 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 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.

Interfaces look exactly like groups except that they don't have template implementations for the template declarations although they must have the complete parameter list. Further, a template may be defined as optional using the `optional` keyword:

```
optional headerFile(actionScope, actions, docComment, recognizer, ...);
```

Expressions

Attribute References

Named attributes

The most common thing in a template besides plain text is a simple named attribute reference such as:

```
Your email: $email$
```

The template will look up the value of `email` and insert it into the output stream when you ask the template to print itself out. If `email` has no value, then it evaluates to the empty string and nothing is printed out for that attribute expression. When working with group files, if `email` is not defined in the formal parameter list of an enclosing template, an exception is thrown.

If the attribute is multi-value such as an instance of a list, the elements are emitted without separator one after the other. If there are null values in the list, these are ignored by default. Given template `$values$` with attribute values=9,6,null,2,null then the output would be:

```
962
```

To use a separator in between those multiple values, use the `separator` option:

```
$values; separator=", "$
```

The output would be:

```
9, 6, 2
```

To emit a special value for each null element in a list, use the `null` option:

```
$values; null="-1", separator=", "$
```

Again using values=9,6,null,2,null then the output would be:

```
9, 6, -1, 2, -1
```

Property references

If a named attribute is an aggregate with a property or a simple data field, you may reference that property using *attribute*.*property*. For example:

```
Your name: $person.name$
Your email: $person.email$
```

`StringTemplate` ignores the actual object type stored in attribute `person` and simply looks for one of the following via reflection (in search order):

Java	1. A method named `getName()` 2. A method named isName() - `StringTemplate` accepts isName() if it returns a Boolean If found, a return value is obtained via reflection. The `person.email` expression is resolved in a similar manner. If the property is not accessible ala JavaBeans, `StringTemplate` attempts to find a field with the same name as the property. In the above example, `StringTemplate` would look for fields `name` and `email` without the capitalization used with JavaBeans property access methods
C#	1. a C# property (i.e. a non-indexed CLR property) named `name` 2. A method named `get_name()` 3. A method named `Getname()` 4. A method named `Isname()` 5. A method named `getname()` 6. A method named `isname()` 7. A field named `name` 8. A C# indexer (i.e. a CLR indexed property) that accepts a single string parameter – `this["name"]` If found, a return value is obtained via reflection. The `person.email` expression is resolved in a similar manner.

	As shown above, if the property is not accessible as a C# property, `StringTemplate` attempts to find a field with the same name as the property. In the above example, `StringTemplate` would look for fields `name` and `email` without the capitalization typically used with property access methods.
Python	1. A method named `getName()` 2. A method named isName() - `StringTemplate` accepts isName() if it returns a Boolean If found, a return value is obtained via reflection. The `person.email` expression is resolved in a similar manner. If the property is not accessible ala JavaBeans, `StringTemplate` attempts to find a field with the same name as the property. In the above example, `StringTemplate` would look for fields `name` and `email` without the capitalization used with JavaBeans property access methods

An exception is thrown if that property is not defined on the target object.

Because the type is ignored, you can pass in whatever existing aggregate (class) you have such as `User` or `Person`:

Java	`User u = database.lookupPerson("parrt@jguru.com"); st.setAttribute("person", u);`
C#	`User u = database.LookupPerson("parrt@jguru.com"); st.SetAttribute("person", u);`
Python	`User u = database.lookupPerson("parrt@jguru.com"); st["person"] = u`

Or, if a suitable aggregate doesn't exist, you can make a connector or "glue" object and pass that in instead:

Java

```
st.setAttribute("person", new Connector());
```

C#

```
st.SetAttribute("person", new Connector());
```

Python

```
st["person"] = Connector()
```

where `Connector` is defined as:

Java

```
public class Connector {
    public String getName() { return
"Terence"; }
    public String getEmail() { return
"parrt@jguru.com"; }
}
```

C#

```
public class Connector {
    public string Name  { get {return
"Terence";} }
    public string Email { get { return
"parrt@jguru.com";} }
}
```

Python

```
class Connector(object):
      def getName(self):
          return "Terence"

      def getEmail(self):
          return "parrt@jguru.com"
```

The ability to reference aggregrate properties saves you the trouble of having to pull out the properties with code like this:

Java

```
User u =
database.lookupPerson("parrt@jguru.com");
st.setAttribute("name", u.getName());
st.setAttribute("email", u.getEmail());
```

C#

```
User u =
database.lookupPerson("parrt@jguru.com");
st.SetAttribute("name", u.Name);
st.SetAttribute("email", u.Email);
```

Python

```
u =
database.lookupPerson("parrt@jguru.com")
st["name"[ = u.getName()
st["email"] = u.getEmail()
```

and having template:

```
Your name: $name$
Your email: $email$
```

The latter is more widely applicable and totally decoupled from code and logic; i.e., it's "better" but much less convenient. Be very careful that the property methods do not have any side-effects like updating a counter or whatever. This breaks the rule of order of evaluation independence.

Indirect property names

Sometimes the property name is itself in which case you need to use indirect property access notation:

```
$person.(propertyName)$
```

where `propertyName` is an attribute whose value is the name of a property to fetch from `person`. Using the examples from above, `propertyName` could hold the value of either `name` or `email`.

`propertyName` may actually be an expression instead of a simple attribute name.

Map key/value pair access

Java	You may pass in instances of any object that implements the `Map` interface. Rather than creating an aggregate object (though automatic aggregate creation is discussed in the next section) you can pass in a `HashMap` that has keys referencable within templates. For example,
	`StringTemplate a = new StringTemplate("$user.name$, $user.phone$");`
	`HashMap user = new HashMap();`
	`user.put("name", "Terence");`
	`user.put("phone", "none-of-your-business");`
	`a.setAttribute("user", user);`
	`String results = a.toString();`
	yields a result of "`Terence, none-of-your-business`".

C#	You may pass in instances of type `Hashtable` and `ListDictionary` but cannot pass in objects implementing the `IDictionary)} interface`

	because that would allow all sorts of wacky stuff like database access. Rather than creating an aggregate object (though automatic aggregate creation is discussed in the next section) you can pass in a {{Hashtable that has keys referencable within templates. For example,
	`StringTemplate a = new StringTemplate("$user.name$, $user.phone$");`
	`Hashtable user = new Hashtable();`
	`user.Add("name", "Terence");`
	`user.Add("phone", "none-of-your-business");`
	`a.SetAttribute("user", user);`
	`string results = a.ToString();`
	yields a result of "Terence, none-of-your-business".

Python	You may pass in instances of type `dict`. Rather than creating an aggregate object (though automatic aggregate creation is discussed in the next section) you can pass in a `dict` that has keys referencable within templates. For example,
	`a = stringtemplate.StringTemplate("$user.name$, $user.phone$")`
	`user = {}`
	`user["name"] = "Terence"`
	`user["phone"] = "none-of-your-business"`
	`a["user"] = user`
	`results = str(a)`
	yields a result of "Terence, none-of-your-business".

StringTemplate interprets Map objects to have two predefined properties: `keys` and `values` had yield a list of all keys and the list of all values, respectively. When applying a template to a map, StringTemplate iterates over the values so that <aMap> is a shorthand for <aMap.values>. Similarly <aMap.keys> walks over the keys. You can list all of the elements in a map like this:

```
<aMap.keys:{k| <k> maps to <aMap.(k)>}>.
```

Note the use of the indirect property reference `<aMap.(k)>`, which says to take the value of the `k` as the key in the lookup. Clearly without the parentheses the normal map lookup mechanism would treat `k` as a literal and try to look up `k` in the map.

Automatic aggregate creation

Creating one-off data aggregates is a pain, you have to define a new class just to associate two pieces of data. `StringTemplate` makes it easy to group data during `setAttribute()` calls. You may pass in an aggregrate attribute name to `setAttribute()` with the data to aggregate:

Java

```
StringTemplate st = new
StringTemplate("$items:{$it.last$,
$it.first$\n}$");
st.setAttribute("items.{first,last}",
"John", "Smith");
st.setAttribute("items.{first,last}",
"Baron", "Von Munchhausen");
String expecting =
        "Smith, John\n" +
        "Von Munchhausen, Baron\n";
```

C#

```
StringTemplate st = new
StringTemplate("$items:{$it.last$,
$it.first$\n}$");
st.SetAttribute("items.{first,last}",
"John", "Smith");
st.SetAttribute("items.{first,last}",
"Baron", "Von Munchhausen");
string expecting = "Smith, John\n" +
                   "Von Munchhausen,
Baron\n";
```

Python

```
st =
stringtemplate.StringTemplate("$items:{$it.last$,
$it.first$\n}$")
st.setAttribute("items.{first,last}",
"John", "Smith")
st.setAttribute("items.{first,last}",
"Baron", "Von Munchhausen")
expecting = \
   "Smith, John\n" + \
   "Von Munchhausen, Baron\n"
```

Note that the template, `st`, expects the `items` to be aggregates with properties `first` and `last`. By using attribute name

```
items.{first,last}
```

You are telling `StringTemplate` to take the following two arguments as properties `first` and `last`.

The various overloads of the `setAttribute()` method can handle from 1 to 5 arguments. The C# version uses variable-length argument list (using `params` keyword).

List construction

As of v2.2, you may combine multiple attributes into a single multi-valued attribute in a syntax similar to the group map feature. Catenate attributes by placing them in square brackets in a comma-separated list. For example,

```
$[mine,yours]$
```

creates a new multi-valued attribute (a list) with both elements - all of `mine` first then all of `yours`. This feature is handy when the model happens to group attributes differently than you need to access them in the view. This ability to rearrange attributes is consistent with model-view separation because the template cannot alter the data structure nor test its values - the template is merely looking at the data from a new perspective.

Naturally you may combine the list construction with template application:

```
$[mine,yours]:{ v | ...}$
```

Note that this is very different from

```
$mine,yours:{ x,y | ...}$
```

which iterates max(n,m) times where n and m are the lengths of `mine` and `yours`, respectively. The `[mine,yours]` version iterates n+m times.

Template References

You may reference other templates to have them included just like the C language preprocessor `#include` construct behaves. For example, if you are building a web page (`page.st`) that has a search box, you might want the search box stored in a separate template file, say, `searchbox.st`. This has two advantages:

- You can reuse the template over and over (no cut/paste)
- You can change one template and all search boxes change on the whole site.

Using method call syntax, just reference the foreign template:

```
<html>
<body>
...
$searchbox()$
...
</body>
</html>
```

The invoking code would still just create the overall page and the enclosing page template would automatically create an instance of the referenced template and insert it:

Java	
	`StringTemplateGroup group = new StringTemplateGroup("webpages", "/usr/local/site/templates");` `StringTemplate page = group.getInstanceOf("page");`

C#

```
StringTemplateGroup group = new
StringTemplateGroup("webpages",
"C:/Inetpub/wwwroot/site/templates");
StringTemplate page =
group.GetInstanceOf("page");
```

Python

```
group =
stringtemplate.StringTemplateGroup("webpages",
"/usr/local/site/templates")
page = group.getInstanceOf("page")
```

If the template you want to reference, say `searchbox`, is in a subdirectory of the `StringTemplateGroup` root directory called `misc`, then you must reference the template as: `misc/searchbox()`.

The included template may access attributes. How can you set the attribute of an included template? There are two ways: inheriting attributes and passing parameters.

Accessing Attributes Of Enclosing Template

Any included template can reference the attributes of the enclosing template instance. So if `searchbox` references an attribute called `resource`:

```
<form ...>
...
<input type=hidden name=resource value=$resource$>
...
</form>
```

you could set attribute `resource` in the enclosing template `page` object:

Java

```
StringTemplate page =
group.getInstanceOf("page");
page.setAttribute("resource", "faqs");
```

C#

```
StringTemplate page =
group.GetInstanceOf("page");
page.SetAttribute("resource", "faqs");
```

Python

```
page = group.getInstanceOf("page")
page["resource"] = "faqs"]
```

This "inheritance" (*dynamic scoping* really) of attributes feature is particularly handy for setting generally useful attributes like `siteFontTag` in the outermost `body` template and being able to reference it in any nested template in the body.

Passing Parameters To Another Template

Another, more obvious, way to set the attributes of an included template is to pass in values as parameters, making them look like C macro invocations rather than includes. The syntax looks like a set of attribute assignments:

```
<html>
<body>
...
$searchbox(resource="faqs")$
...
</body>
</html>
```

where I am setting the attribute of the included `searchbox` to be the string literal `"faqs"`.

The right-hand-side of the assignment may be any expression such as an attribute reference or even a reference to another template like this:

```
$boldMe(item=copyrightNotice())$
```

You may also use an anonymous template such as:

```
$bold(it={$firstName$ $lastName$})$
```

which first computes the template argument and then assigns it to `it`.

If you are using `StringTemplate` groups, then you have formal parameters and for those templates with a sole formal argument, you can pass just an expression instead of doing an assignment to the argument name. For example, if you do `$bold(name)$` and `bold` has one formal argument called `item`, then `item` gets the value of `name` just as if you had said {$bold(item=name)$}.

Allowing enclosing attributes to pass through

When template `x` calls template `y`, the formal arguments of `y` hide any `x` arguments of the same because the formal parameters force you to define values. This prevents surprises and makes it easy to ensure any parameter value is empty unless you specifically set it for that template. The problem is that you need to factor templates sometimes and want to refine behavior with a subclass or just invoke another shared template but invoking `y` as `<y()>` hides all of `x`'s parameters with the same name. Use `<y(...)>` syntax to indicate `y` should inherit all values even those with the same name. `<y(name="foo", ...)>` would set one arg, but the others are inherited whereas `<y(name="foo")>` only has `name` set; all other arguments of template `y` are empty. You can set manually with:

Java	`StringTemplate.setPassThroughAttributes()`
C#	`StringTemplate.SetPassThroughAttributes()`
Python	`stringtemplate.StringTemplate.setPassThroughAttributes`

Argument evaluation scope

The right-hand-side of the argument assignments are evaluated within the scope of the enclosing template whereas the left-hand-side attribute name is the name of an attribute in the target template. Template invocations like `$bold(item=item)$` actually make sense because the `item` on the right is evaluated in a different scope.

Attribute operators

StringTemplate provides a number of operators that you can apply to attributes to get a new view of that data: first, rest, last, length, strip.

Sometimes you need to treat the first or last element of multi-valued attribute differently than the others. For example, if you have a list of integers in an attribute and you need to generate code to sum those numbers, you could start like this:

```
<numbers:{ n | sum += <n>;}>
```

You need to define `sum`, however:

```
int sum = 0;
<numbers:{ n | sum += <n>;}>
```

What if `numbers` is empty though? No need to create the `sum` definition so you could do this:

```
<if(numbers)>int sum = 0;<endif>
<numbers:{ n | sum += <n>;}>
```

A more specific strategy (and one that generates slightly better code as it avoids an unnecessary initialization to 0) is the following:

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

where `first(numbers)` results in the first value of attribute `numbers` if any and `rest(numbers)` results all values in `numbers` but the first value.

The other operator available to you is `last`, which naturally results in the last value of a multi-valued attribute.

Special cases:

- operations on empty attributes yields an empty value
- `rest` of a single-valued attribute yields an empty value
- `tail` of a single-valued attribute yields the same as `first`, the attribute value

You may find it handy to use another operator sometimes: plus "string concatenate". operator. For example, you may want to compute an argument to a template using a literal and an attribute:

```
...$link(url="/faq/view?ID="+faqid, title=faqtitle)$...
```

where `faqid` and `faqtitle` are attributes you have set for the template that referenced `link`.

Terence says

I'm a little uncomfortable with this catenation operation. Please use a template instead:

```
...$link(url={/faq/view?ID=$faqid$}, title=faqtitle)$...
```

In order to emit the number of attributes in a single or multi-value attribute, use the `length` operator:

```
int data[$length(x)$] = { $x; separator=", "$ };
```

In this example, with x=5,2,9 the following would be emitted:

```
int data[3] = { 5, 2, 9 };
```

Null values are counted by `length` but you can use the `strip` operator to return a new view of your list without null values:

```
int data[$length(strip(x))] = { $x; separator=", "$ };
```

Template Application

Imagine a simple template called `bold`:

```
<b>$item$</b>
```

Just as with template `link` described above, you can reference it from a template by invoking it like a method call:

```
$bold(item=name)$
```

What if you want something bold and italicized? You could simply nest the template reference:

```
$bold(item=italics(item=name))$
```

(or `$bold(italics(name))$` if you're using group file format and have formal parameters). Template

`italics` is defined as:

```
<i>$item$</i>
```

using a different attribute with the same name, `item`; the attributes have different values just like you would expect if these template references where method calls in say Java or C# and, `item` was a local variable. Parameters and attribute references are scoped like a programming language.

Think about what you are really trying to say here. You want to say "make name italics and then make it bold", or "apply italics to the name and then apply bold." There is an "apply template" syntax that is a literal translation:

```
$name:italics():bold()$
```

where the templates are applied in the order specified from left to right. This is much more clear, particularly if you had three templates to apply:

```
$name:courierFont():italics():bold()$
```

For this syntax to work, however, the applied templates have to reference a standard attribute because you are not setting the attribute in a parameter assignment. In general for syntax *expr*:*template*(), an attribute called `it` is set to the value of *expr*. So, the definition of `bold` (and analogously `italics`), would have to be:

```
<b>$it$</b>
```

to pick up the value of `name` in our examples above.

As of 2.2 `StringTemplate`, you can avoid using `it` as a default parameter by using formal arguments. For expression `$x:y()$`, `StringTemplate` will assign the value of `x` to `it` and any sole formal argument of `y`. For example, if `y` is:

```
y(item) ::= "_$item$_"
```

then `item` would also have the value of `x`.

If the attribute to which you are applying a template is null (i.e., missing), then the application is not done as there is no work to do. Optionally, you can specify what string template should display when the attribute is null a using the `null` option:

```
$name:bold(); null="n/a"$
```

That is equivalent to the following conditional:

```
$if(name)$$name:bold()$$else$n/a$endif$
```

Applying Templates To Multi-Valued Attributes

Where template application really shines though is when an attribute is multi-valued. One of the most common web page generation issues is making lists of items either as bullet lists or table rows etc... Applying a template to a multi-valued attribute means that you want the template applied to each of the values.

Consider a list of names (i.e., you set attribute `names` multiple times) that you want in a bullet list. If you have a template called `listItem`:

```
<li>$it$</li>
```

then you can do this:

```
<ul>
$names:listItem()$
</ul>
```

and each name will appear as a bullet item. For example, if you set `names` to `"Terence"`, `"Tom"`, and `"Kunle"`, then you would see:

```
<ul>
<li>Terence</li>
<li>Tom</li>
<li>Kunle</li>
</ul>
```

in the output.

Whenever you apply a template to an attribute or multi-valued attribute, the default attribute `it` is set. Similarly, attributes `i` and `i0` (since v3.0) of type `integer` are set to the value's index number starting from `1` (`i0` starts from `0`). For example, if you wanted to make your own style of numbered list, you could reference `i` to get the index:

```
$names:numberedListItem()$
```

where template `numberedListItem` is defined as:

```
$i$. $it$<br>
```

In this case, the output would be:

```
1. Terence<br>
2. Tom<br>
3. Kunle<br>
```

If there is only one attribute value, then `i` will be 1. However, if template `numberedListItem` is defined as:

```
$i0$. $it$<br>
```

The output would be:

```
0. Terence<br>
1. Tom<br>
2. Kunle<br>
```

As when invoking templates ala "includes", a single formal argument is also set to the iterated value. For example, you could define `numberedListItem` as follows in a `StringTemplateGroup` file:

```
numberedListItem(item) ::= "$i$. $item$<br>"
```

Templates are not applied to null values in multi-valued attributes. StringTemplate behaves as if those values simply did not exist in the list. To emit a special string or template for each null value, use the `null` option:

```
$names:bold(); null="n/a"$
```

which will emit "n/a" for any null value in attribute `names`.

Applying Multiple Templates To Multi-Valued Attributes

The result of applying a template to a multi-valued attribute is another multi-valued attribute containing the results of the application. You may apply another template to the results of the first template application, which comes in handy when you need to format the elements of a list before they go into the list. For example, to bold the elements of a list do the following (given the appropriate template definitions from above):

```
$names:bold():listItem()$
```

If you actually want to apply a template to the combined (string) result of a previous template application, enclose the previous application in parenthesis. The parenthesis will force immediate evaluation of the template application, resulting in a string. For example,

```
$(names:bold()):listItem()$
```

results in a single list item full of a bunch of bolded names. Without the parenthesis, you get a list of items that are bolded.

Applying Alternating Templates To Multi-Valued Attributes

When generating lists of things, you often need to change the color or other formatting instructions depending on the list position. For example, you might want to alternate the color of the background for the elements of a list. The easiest and most natural way to specify this is with an alternating list of templates to apply to an expression of the form: $*expr*:*t1*(),*t2*(),...,*tN*()$. To make an alternating list of blue and green names, you might say:

```
$names:blueListItem(),greenListItem()$
```

where presumably `blueListItem` template is an HTML `<table>` or something that lets you change background color. `names[0]` would get `blueListItem()` applied to it, `names[1]` would get `greenListItem()`, and `names[2]` would get `blueListItem()` again, etc...

If `names` is single-valued, then `blueListItem()` is applied and that's it.

Applying Anonymous Templates

Some templates are so simple or so unlikely to be reused that it seems a waste of time making a separate template file and then referencing it. `StringTemplate` provides *anonymous subtemplates* to handle this case. The templates are anonymous in the sense that they are not named; they are directly applied in a single instance.

For example, to show a name list do the following:

```
<ul>
$names:{<li>$it$</li>}$
</ul>
```

where anything enclosed in curlies is an anonymous subtemplate if, of course, it's within an attribute expression. Note that in the subtemplate, I must enclose the `it` reference in the template expression delimiters. You have started a new template exactly like the surrounding template and you must distinguish between text and attribute expressions.

You can apply multiple templates very conveniently. Here is the bold list of names again with anonymous templates:

```
<ul>
$names:{<b>$it$</b>}:{<li>$it$</li>}$
</ul>
```

The output would look like:

```
<ul>
<li><b>Terence</b></li>
<li><b>Tom</b></li>
<li><b>Kunle</b></li>
</ul>
```

Anonymous templates work on single-valued attributes as well.

As of 2.2, you may define formal arguments on anonymous templates even if you are not using `StringTemplate` groups. This syntax is borrowed from SmallTalk though it is identical in function to `lamba` of Python. Use a comma-separated list of argument names followed by the '|' "pipe" symbol. Any single whitespace character immediately following the pipe is ignored. The following example bolds the names in a list using an argument to avoid the monotonous use of `it`:

```
<ul>
$names:{ n | <b>$n$</b>}$
</ul>
```

Clearly only one argument may be defined in this situation: the iterated value of a single list.

Anonymous template application to multiple attributes

In some cases, the model may present data to the view as separate columns of data rather than as a single list of objects, such as multi-valued attributes `names` and `phones` rather than a single `users` multi-valued attribute. As of 2.2, you may iterate over multiple attributes:

```
$names,phones:{ n,p | $n$: $p$}$
```

An error is generated if you have too many arguments for the number of attributes. Iteration proceeds while at least one of the attributes (`names` or `phones`, in this case) has values.

Indirect template references

Sometimes the name of the template you would like to include is itself a variable. So, rather than using "`<item:format()>`" you want the name of the template, `format`, to be a variable rather than a literal. Just enclose the template name in parenthesis to indicate you want the immediate value of that attribute and then add `()` like a normal template invocation and you get "`<item:(someFormat)()>`", which means "look up attribute `someFormat` and use its value as a template name; appy to `item`." This deliberately looks similar to the C function call indirection through a function pointer (e.g., "`(*fp)()`" where fp is a pointer to a function). A better way to look at it though is that the `(someFormat)` implies *immediately evaluate someFormat and use as the template name*.

Usually this "variable template" situation occurs when you have a list of items to format and each element may require a different template. Rather than have the controller code create a bunch of instances, one could consider it better to have `StringTemplate` do the creation--the controller just names what format to use.

If `StringTemplate` did not have a map definition, you could simulate its functionality. Consider generating a list of C# declarations that are initialized to 0, false, null, etc... You could define a template for `int`, `Object`, `Array`, etc... declarations and then pass in an aggregate object that has the variable declaration object and the format. In a template group file you might have:

```
group Java;

file(variables,methods) ::= <<
<variables:{ v | <v.decl:(v.format)()>}; separator="\n">
```

```
<methods>
\>>
intdecl(decl) ::= "int <decl.name> = 0;"
intarray(decl) ::= "int[] <decl.name> = null;"
```

Your code might look like:

Java

```
StringTemplateGroup group =
        new StringTemplateGroup(new
StringReader(templates),
AngleBracketTemplateLexer.class);
StringTemplate f =
group.getInstanceOf("file");
f.setAttribute("variables.{decl,format}",
new Decl("i","int"), "intdecl");
f.setAttribute("variables.{decl,format}",
new Decl("a","int-array"), "intarray");
System.out.println("f="+f);
String expecting = ""+newline+newline;
```

C#

```
StringTemplateGroup group =
        new StringTemplateGroup(new
StringReader(templates),
typeof(AngleBracketTemplateLexer));
StringTemplate f =
group.GetInstanceOf("file");
f.setAttribute("variables.{decl,format}",
new Decl("i","int"), "intdecl");
f.setAttribute("variables.{decl,format}",
new Decl("a","int-array"), "intarray");
Console.Out.WriteLine("f="+f);
string expecting = ""+newline+newline;
```

Python

```
group =
stringtemplate.StringTemplateGroup(StringIO(templates),
stringtemplate.language.AngleBracketTemplateLexer.Lexer)
f = group.getInstanceOf("file")
f.setAttribute("variables.{decl,format}",
Decl("i","int"), "intdecl")
f.setAttribute("variables.{decl,format}",
Decl("a","int-array"), "intarray")
print "f =", f
expecting = ""+os.linesep
```

For this simple unit test, the following dummy decl class is used:

Java

```
public static class Decl {
    String name;
    String type;
    public Decl(String name, String type)
{this.name=name; this.type=type;}
    public String getName() {return name;}
    public String getType() {return type;}
}
```

C#

```
public class Decl {
    string name;
    string type;
    public Decl(string name, string type)
```

	this.name=name; this.type=type;} public string Name { get {return name;} } public string Type { get {return type;} } }

Python	class Decl(object):
	
	def __init__(self, name, type_):
	self.name = name
	self.type = type_
	
	def getName(self):
	return self.name
	
	def getType(self):
	return self.type

The value of `f.ToString()` is:

```
int i = 0;
int[] a = null;
```

Missing attributes (i.e., `null` valued attributes) used as indirect template attribute generate nothing just like referencing a missing attribute.

Conditionally Included Subtemplates (IF statements)

There are many situations when you want to conditionally include some text or another template. `StringTemplate` provides simple IF-statements to let you specify conditional includes. For example, in a dynamic web page you usually want a slightly different look depending on whether or not the viewer is "logged in" or not. Without a conditional include, you would need two templates: `page_logged_in` and `page_logged_out`. You can use a single `page` definition with `if(`*expr*`)` attribute actions instead:

```
<html>
...
<body>
$if(member)$
$gutter/top_gutter_logged_in()$
$else$
$gutter/top_gutter_logged_out()$
$endif$
...
</body>
</html>
```

where template `top_gutter_logged_in` is located in the `gutter` subdirectory of my `StringTemplateGroup`.

IF actions test the presence or absence of an attribute unless the object is a `Boolean/bool`, in which case it tests the attribute for `true/false`. The only operator allowed is "not" and means either "not present" or "not true". For example, "`$if(!member)$...$endif$`".

Whitespace in conditionals issue

There is a simple, but not perfect rule: kill a single newline **after** <if>, <<, <else>, and <endif> (but for <endif> only if it's on a line by itself) . Kill newlines **before** <else> and <endif> and >>. For example,

```
a <if(foo)>big<else>small<endif> dog
```

is identical to:

```
a <if(foo)>
big
<else>
small
<endif>
 dog
```

It is very difficult to get the newline rule to work "properly" because sometimes you want newlines and sometimes you don't. I
decided to chew up as many as is reasonable and then let you explicitly say <\n> when you need to.

Functionality Summary

Syntax	Description
<*attribute*>	Evaluates to the value of *attribute*.`ToString()` if it exists else empty string.
<i>, <i0>	The iteration number indexed from one and from zero, respectively, when referenced within a template being applied to an attribute or attributes.
<*attribute.property*>	Looks for *property* of *attribute* as a property (C#), then accessor methods like `get`*Property*`()` or `is`*Property*`()`. If that fails, `StringTemplate` looks for a raw field of the *attribute* called *property*. Evaluates to the empty string if no such property is found.
<*attribute*.(*expr*)>	Indirect property lookup. Same as *attribute.property* except use the value of *expr* as the property_ name. Evaluates to the empty string if no such property is found.
<*multi-valued-attribute*>	Concatenation of `ToString()` invoked on each element. If *multi-valued-attribute* is missing his evaluates to the empty string.
<*multi-valued-attribute*; separator=*expr*>	Concatenation of `ToString()` invoked on each element separated by *expr*.
<*template*(*argument-list*)>	Include *template*. The *argument-list* is a list of attribute assignments where each assignment is of the form *arg-of-template=expr* where *expr* is

	evaluated in the context of the surrounding template not of the invoked template.
<*(expr)*(*argument-list*)>	Include *template* whose name is computed via *expr*. The *argument-list* is a list of attribute assignments where each assignment is of the form *attribute*=*expr*. Example `$(whichFormat)()$` looks up `whichFormat`'s value and uses that as template name. Can also apply an indirect template to an attribute.
<*attribute*:*template*(*argument-list*)>	Apply *template* to *attribute*. The optional *argument-list* is evaluated before application so that you can set attributes referenced within *template*. The default attribute `it` is set to the value of *attribute*. If *attribute* is multi-valued, then `it` is set to each element in turn and *template* is invoked *n* times where *n* is the number of values in *attribute*. Example: `$name:bold()` applies `bold()` to `name`'s value.
<*attribute*:*(expr)*(*argument-list*)>	Apply a template, whose name is computed from *expr*, to each value of *attribute*. Example `$data:(name)()$` looks up `name`'s value and uses that as template name to apply to `data`.
<*attribute*:*t1*(*argument-list*): ... :*tN*(*argument-list*)>	Apply multiple templates in order from left to right. The result of a template application upon a multi-valued attribute is another multi-valued attribute. The overall expression evaluates to the concatenation of all elements of the final multi-valued attribute resulting from *templateN*'s application.
<*attribute*:{*anonymous-template*}>	Apply an anonymous template to each element of *attribute*. The iterated `it` atribute is set automatically.
<*attribute*:{*argument-name* | *anonymous-template*}>	Apply an anonymous template to each element of *attribute*. Set the *argument-name* to the iterated value and also set `it`.
<*a1*,*a2*,...,*aN*:{*argument-list* | *anonymous-template*}>	Parallel list iteration. March through the values of the attributes *a1*..*aN*, setting the values to the arguments in *argument-list* in the same order. Apply the anonymous template. There is no defined `it` value unless inherited from an enclosing scope.
<*attribute*:*t1*(),*t2*(),...,*tN*()>	Apply an alternating list of templates to the elements of *attribute*. The template names may include argument lists.
<if(*attribute*)>*subtemplate* <else>*subtemplate2* <endif>	If *attribute* has a value or is a `bool` object that evaluates to `true`, include *subtemplate* else include *subtemplate2*. These conditionals may be nested.

<if(!*attribute*)>*subtemplate*<endif>	If *attribute* has no value or is a `bool` object that evaluates to `false`, include *subtemplate*. These conditionals may be nested.
<first(attr)>	The first or only element of attr. You can combine operations to say things like first(rest(names)) to get second element.
<last(attr)>	The last or only element of attr.
<rest(attr)>	All but the first element of attr. Returns nothing if $attr$ a single valued.
<strip(attr)>	Returns an iterator that skips any null values in $attr$. strip(x)=x when x is a single-valued attribute.
<length(attr)>	Return an integer indicating how many elements in length $attr$ is. Single valued attributes return 1. Strings are not special; i.e., length("foo") is 1 meaning "1 attribute". Nulls are counted in lists so a list of 300 nulls is length 300. If you don't want to count nulls, use length(strip(list)).
`\$` or `\<`	escaped delimiter prevents `$` or < from starting an attribute expression and results in that single character.
`<\ >`, `<\n>`, `<\t>`, `<\r>`	special characters: space, newline, tab, carriage return.
`<! comment !>`, `$! comment !$`	Comments, ignored by StringTemplate.

Object Rendering

The atomic element of a template is a simple object that is rendered to text by its `ToString()` method. For example, an `integer` object is converted to text as a sequence of characters representing the numeric value written out. What if you wanted commas to separate the 1000's places like `1,000,000`? What if you wanted commas and sometimes periods depending on the locale?.

Prior to 2.2, there was no means of altering the rendering of objects to text. The controller had to pull data from the model and wrap it on an object whose `ToString()` method rendered it appropriately.

As of `StringTemplate` 2.2, you may register various attribute renderers associated with object class types. Normally a single renderer will be used for a group of templates so that `Date` objects are always displayed using the appropriate `Locale`, for example. There are, however, situations where you might want a template to override the group renderers. You may register renderers with either templates or groups and groups inherit the renderers from super groups (if any).

There is a new abstraction that defines how an object is rendered to string:

Java	`class AttributeRenderer`
C#	`interface IAttributeRenderer`
Python	`class AttributeRenderer`

Here is a renderer that renders date objects tersely.

Java

```
public class DateRenderer implements
AttributeRenderer {
        public String toString(Object o) {
                SimpleDateFormat f = new
SimpleDateFormat("yyyy.MM.dd");
                return
f.format(((Calendar)o).getTime());
        }
}
...
StringTemplate st =new StringTemplate(
                "date: <created>",
AngleBracketTemplateLexer.class);
st.setAttribute("created", new
GregorianCalendar(2005, 07-1, 05));
st.registerRenderer(GregorianCalendar.class,
new DateRenderer());
String expecting = "date: 2005.07.05";
String result = st.toString();
```

C#

```
public class DateRenderer :
IAttributeRenderer
{
        public string ToString(object o) {
                DateTime dt = (DateTime) o;
                return
dt.ToString("yyyy.MM.dd");
        }
}
...
...
StringTemplate st =new
StringTemplate("date:
<created>",typeof(AngleBracketTemplateLexer));
st.SetAttribute("created", new
DateTime(2005, 07, 05, New
GregorianCalendar()));
st.registerRenderer(typeof(DateTime), new
DateRenderer());
string expecting = "date: 2005.07.05";
string result = st.ToString();
```

Python

```
import stringtemplate
import date from datetime

 class
DateRenderer(stringtemplate.AttributeRenderer):

    def str(self, o):
       return o.strftime("%Y.%m.%d")
...
...
    st = stringtemplate.StringTemplate( \
            "date: <created>", \
stringtemplate.language.AngleBracketTemplateLexer.Lexer)
    st["created"] = date(year=2005,
month=7, day=5)
    st.registerRenderer(date,
DateRenderer())
    expecting = "date: 2005.07.05"
    result = str(st)
```

In the sample code above, date objects are represented as objects of type:

Java	`Calender`
C#	`DateTime`
Python	`date`

All attributes of the date types above in template `st` are rendered using the `DateRenderer` object.

You will notice that there is no way for the template to say which renderer to use. Allowing such a mechanism would effectively imply an ability to call random code from the template. In `StringTemplate`'s scheme, only the model or controller can set the renderer. The template must still reference a simple attribute such as `<created>`. If you need the same kind of attribute displayed differently within the same template or group, you must pass in two different attribute types. This would be rare, but if you need it, you can easily still wrap an object in a renderer before sending it to the template as an attribute. For example, if you have a web site that allows editing of some descriptions, you will probably need both an escaped and unescaped version of the description. Send in the unescaped description as one attribute and send it in again wrapped in an HTML escape renderer as a different attribute.

As far as I can tell, this functionality is mostly useful in the web page generation realm rather than code generation; perhaps an opportunity will present it self though.

Template And Attribute Lookup Rules

Template lookup

When you request a named template via `StringTemplateGroup.getInstanceOf()` or within a template, there is a specific sequence used to locate the template.

If a template, *t*, references another template and *t* is not specifically associated with any group, *t* is implicitly associated with a default group whose root directory is ".", the current directory. The referenced template will be looked up in the current directory.

If a template *t* is associated with a group, but was not defined via a group file format, lookup a referenced template in the group's template table. If not there, look for it on the disk under the group's root dir. If not found, recursively look at any supergroup of the group. If not found at all, record this fact and don't look again on the disk until refresh interval.

If the template's associated group was defined via a group file, then that group is searched first. If not found, the template is looked up in any supergroup. The refresh interval is not used for group files because the group file is considered complete and enduring.

Attribute scoping rules

A `StringTemplate` is a list of chunks, text literals and attribute expressions, and an attributes table. To render a template to string, the chunks are written out in order; the expressions are evaluated only when asked to during rendering. Attributes referenced in expressions are looked up using a very specific sequence similar to an inheritance mechanism.

When you nest a template within another, such as when a `page` template references a `searchbox` template, the nested template may see any attributes of the enclosing instance or its enclosing instances. This mechanism is called *dynamic scoping*. Contrast this with *lexical scoping* used in most programming languages like C# and Java where a method may not see the variables defined in invoking methods. Dynamic scoping is very natural for templates. For example, if `page` has an attribute/value pair `font/Times` then `searchbox` could reference `$font$` when nested within a `page` instance.

Reference to attribute *a* in template *t* is resolved as follows:

1. Look in *t*'s attribute table
2. Look in *t*'s arguments
3. Look recursively up *t*'s enclosing template instance chain
4. Look recursively up *t*'s group / supergroup chain for a map

This process is recursively executed until *a* is found or there are no more enclosing template instances or super groups.

When using a group file format to specify templates, you must specify the formal arguments for that template. If you try to access an attribute that is not formally defined in that template or an enclosing template, you will get a `InvalidOperationException`.

When building code generators with `StringTemplate`, large heavily nested template tree structures are commonplace and, due to dynamic attribute scoping, a nested template could inadvertently use an attribute from an enclosing scope. This could lead to infinite recursion during rendering and other surprises. To prevent this, formal arguments on template *t* hide any attribute value with that name in any enclosing scope. Here is a test case that illustrates the point.

Java

```
String templates =
        "group test;" +newline+
        "block(stats) ::= \"{$stats$}\""
        ;
StringTemplateGroup group =
        new StringTemplateGroup(new
StringReader(templates));
StringTemplate b =
group.getInstanceOf("block");
b.setAttribute("stats",
group.getInstanceOf("block"));
String expecting ="{{}}";
```

C#

```
string templates =
        "group test;" +newline+
        "block(stats) ::= \"{$stats$}\""
        ;
StringTemplateGroup group = new
StringTemplateGroup(new
StringReader(templates));
StringTemplate b =
group.GetInstanceOf("block");
b.SetAttribute("stats",
group.GetInstanceOf("block"));
string expecting ="{{}}";
```

Python

```
templates = \
        "group test;" + os.linesep + \
        "block(stats) ::= \"{$stats$}\""
group =
stringtemplate.StringTemplateGroup(StringIO(templates))
b = group.getInstanceOf("block")
b["stats"] = group.getInstanceOf("block")
expecting ="{{}}"
```

Even though `block` has a `stats` value that refers to itself, there is no recursion because each instance of `block` hides the `stats` value from above since `stats` is a formal argument.

Sometimes self-recursive (hence infinitely recursive) structures occur through programming error and they are nasty to track down. If you turn on "lint mode", `StringTemplate` will attempt to find cases where a template instance is being evaluated during the evaluation of itself. For example, here is a test case that causes and traps infinite recursion.

Java

```
String templates =
        "group test;" +newline+
        "block(stats) ::= \"$stats$\"" +
        "ifstat(stats) ::= \"IF true then
$stats$\"\n"
        ;
StringTemplate.setLintMode(true);
StringTemplateGroup group =
        new StringTemplateGroup(new
StringReader(templates));
StringTemplate b =
group.getInstanceOf("block");
StringTemplate ifstat =
group.getInstanceOf("ifstat");
b.setAttribute("stats", ifstat); // block
has if stat
ifstat.setAttribute("stats", b); // but
make the "if" contain block
try {
    String result = b.toString();
}
catch (IllegalStateException ise) {
    ...
}
```

C#

```
string templates =
        "group test;" +newline+
        "block(stats) ::= \"$stats$\"" +
        "ifstat(stats) ::= \"IF true then
$stats$\"\n"
        ;
StringTemplate.SetLintMode(true);
StringTemplateGroup group = new
StringTemplateGroup(new
StringReader(templates));
StringTemplate b =
group.GetInstanceOf("block");
StringTemplate ifstat =
group.GetInstanceOf("ifstat");
b.SetAttribute("stats", ifstat); // block
has if stat
ifstat.SetAttribute("stats", b); // but
make the "if" contain block
try {
    string result = b.ToString();
}
catch (InvalidOperationException ise) {
    ...
```

	`}`

Python	`templates = \`
	`"group test;" + os.linesep + \`
	`"block(stats) ::= \"$stats$\"" +`
	`os.linesep + \`
	`"ifstat(stats) ::= \"IF true then`
	`$stats$\"\n"`
	
	`strigntemplate.StringTemplate.setLintMode(True)`
	`group =`
	`stringtemplate.StringTemplateGroup(StringIO(templates))`
	`b = group.getInstanceOf("block")`
	`ifstat = group.getInstanceOf("ifstat")`
	`b["stats"] = ifstat  # block has if`
	`stat`
	`ifstat["stats"] = b  # but make the`
	`"if" contain block`
	`try:`
	`  result = str(b)`
	`except IllegalStateException, ise:`

The nested template stack trace from exception object will be similar to:

```
infinite recursion to <ifstat([stats])@4> referenced in <block([stats])@3>; stack trace:
<ifstat([stats])@4>, attributes=[stats=<block()@3>]>
<block([stats])@3>, attributes=[stats=<ifstat()@4>], references=[stats]>
<ifstat([stats])@4> (start of recursive cycle)
...
```

Setting the Expression Delimiters

By default, expressions in a template are delimited by dollar signs: `$...$`. This works great for the most common case of HTML generation because the attribute expressions are clearly highlighted in the text. Sometimes, with other formats like SQL statement generation, you may want to change the template expression delimiters to avoid a conflict and to make the expressions stand out.

The start and stop strings are limited to either `$...$` or `<...>` (unless you build your own lexical analyzer to break apart templates into chunks). group file templates use `<...>` delimiters by default (in v2.2 `$...$` was the default delimiter). Templates created with the StringTemplate object constructor still use `$...$` by default.

To specify that `StringTemplate` should use a specific delimiter you must create a `StringTemplateGroup`:

Java	`StringTemplateGroup group =`
	`  new StringTemplateGroup("sqlstuff",`
	`"/tmp", AngleBracketTemplateLexer.class);`
	`StringTemplate query =`
	`  new StringTemplate(group, "SELECT`
	`<column> FROM <table>;");`
	`query.setAttribute("column", "name");`
	`query.setAttribute("table", "User");`

C#

```
StringTemplateGroup group =
     new StringTemplateGroup("sqlstuff",
"/tmp", typeof(AngleBracketTemplateLexer));
StringTemplate query = new
StringTemplate(group, "SELECT <column> FROM
<table>;");
query.SetAttribute("column", "name");
query.SetAttribute("table", "User");
```

Python

```
group =
stringtemplate.StringTemplateGroup("sqlstuff",
"/tmp", \
stringtemplate.language.AngleBracketTemplateLexer.Lexer)
query =
stringtemplate.StringTemplate(group,
"SELECT <column> FROM <table>;")
query["column"] = "name"
query["table"] = "User"
```

All templates created through the group or in anyway associated with the group will assume your the angle bracket delimiters. It's smart to be consistent across all files of similar type such as "all HTML templates use `$...$`" and "all SQL templates use `<...>`".

Template inheritance

Recall that a StringTemplateGroup is a collection of related templates such as all templates associated with the look of a web site. If you want to design a similar look for that site (such as for premium users), you don't really want to cut-n-paste the original template files for use in the new look. Changes to the original will not be propogated to the new look.

Just like you would do with a class definition, a template group may inherit templates from another group, the *supergroup*. If template *t* is not found in a group, it is looked up in the supergroup, if present. This works regardless of whether you use a group file format or load templates from the disk via a `StringTemplateGroup` object. Currently you cannot use the group file syntax to specify a supergroup. I am investigating how this should work. In the meantime, you must explicitly set the supergroup in code.

From the unit tests, here is a simple inheritance of a template, `bold`:

Java

```
StringTemplateGroup supergroup = new
StringTemplateGroup("super");
StringTemplateGroup subgroup = new
StringTemplateGroup("sub");
supergroup.defineTemplate("bold",
"<b>$it$</b>");
subgroup.setSuperGroup(supergroup);
StringTemplate st = new
StringTemplate(subgroup, "$name:bold()$");
st.setAttribute("name", "Terence");
String expecting = "<b>Terence</b>";
```

C#

```
StringTemplateGroup supergroup = new
StringTemplateGroup("super");
StringTemplateGroup subgroup = new
```

	StringTemplateGroup("sub");
	supergroup.DefineTemplate("bold",
	"it");
	subgroup.SuperGroup = supergroup;
	StringTemplate st = new
	StringTemplate(subgroup, "$name:bold()$");
	st.SetAttribute("name", "Terence");
	string expecting = "Terence";

Python	supergroup = stringtemplate.StringTemplateGroup("super") subgroup = stringtemplate.StringTemplateGroup("sub") supergroup.defineTemplate("bold", "it") subgroup.setSuperGroup(supergroup) st = stringtemplate.StringTemplate(subgroup, "$name:bold()$") st["name"] = "Terence" expecting = "Terence"

The supergroup has a bold definition but the subgroup does not. Referencing `$name:bold()$` works because subgroup looks into its supergroup if it is not found.

You may override templates:

Java	supergroup.defineTemplate("bold", "it"); subgroup.defineTemplate("bold", "it");
C#	supergroup.DefineTemplate("bold", "it"); subgroup.DefineTemplate("bold", "it");
Python	supergroup.defineTemplate("bold", "it"); subgroup.defineTemplate("bold", "it");

And you may refer to a template in a supergroup via `super.`*template*`()`:

Java	StringTemplateGroup group = new
	StringTemplateGroup(...);
	StringTemplateGroup subGroup = new
	StringTemplateGroup(...);
	subGroup.setSuperGroup(group);
	group.defineTemplate("page",
	"$font()$:text");
	group.defineTemplate("font", "Helvetica");
	subGroup.defineTemplate("font",
	"$super.font()$ and Times");
	StringTemplate st =
	subGroup.getInstanceOf("page");

C#

```
StringTemplateGroup group = new
StringTemplateGroup(...);
StringTemplateGroup subGroup = new
StringTemplateGroup(...);
subGroup.SuperGroup = group;
group.DefineTemplate("page",
"$font()$:text");
group.DefineTemplate("font", "Helvetica");
subGroup.DefineTemplate("font",
"$super.font()$ and Times");
StringTemplate st =
subGroup.GetInstanceOf("page");
```

Python

```
group =
stringtemplate.StringTemplateGroup(...)
subGroup =
stringtemplate.StringTemplateGroup(...)
subGroup.setSuperGroup(group)
group.defineTemplate("page",
"$font()$:text")
group.defineTemplate("font", "Helvetica")
subGroup.defineTemplate("font",
"$super.font()$ and Times")
st = subGroup.getInstanceOf("page")
```

The string `st.ToString()` results in "`Helvetica and Times:text`".

Just like object-oriented programming languages, `StringTemplate` has polymorphism. That is, template names are looked up dynamically relative to the invoking templates group.

> The classic demonstration of dynamic message sends, for example, would be the following example (this catches my students all the time): 

Java

```
class A {
  public void page() {bold();}
  public void bold()
{System.out.println("A.bold");}
}
class B extends A {
  public void bold()
{System.out.println("B.bold");}
}
...
A a = new B();
a.page();
```

C#

```
class A {
  public void page() {bold();}
  override public void bold()
{Console.Out.WriteLine("A.bold");}
}
class B : A {
  virtual public void bold()
{Console.Out.WriteLine("B.bold");}
}
...
...
A a = new B();
a.page();
```

This prints "B.bold" not "A.bold" because the receiver determines how to answer a message not the type of the variable. So, I have created a B object meaning that any message, such as bold(), invoked will first look in class B for bold().

Similarly, a template's group determines where it starts looking for a template. In this case, both super and sub groups define a bold template mirroring the code above. Because I create template st as a member of the subGroup and reference to bold starts looking in subGroup even though page is the template referring to bold.

Java

```
StringTemplateGroup group = new
StringTemplateGroup("super");
StringTemplateGroup subGroup = new
StringTemplateGroup("sub");
subGroup.setSuperGroup(group);
group.defineTemplate("bold",
"<b>$it$</b>");
group.defineTemplate("page",
"$name:bold()$");
subGroup.defineTemplate("bold",
"<strong>$it$</strong>");
StringTemplate st =
subGroup.getInstanceOf("page");
st.setAttribute("name", "Ter");
String expecting = "<strong>Ter</strong>";
```

C#

```
StringTemplateGroup group = new
StringTemplateGroup("super");
StringTemplateGroup subGroup = new
StringTemplateGroup("sub");
subGroup.SuperGroup = group;
group.DefineTemplate("bold",
"<b>$it$</b>");
group.DefineTemplate("page",
"$name:bold()$");
subGroup.DefineTemplate("bold",
"<strong>$it$</strong>");
StringTemplate st =
subGroup.GetInstanceOf("page");
st.SetAttribute("name", "Ter");
string expecting = "<strong>Ter</strong>";
```

Python

```
group =
stringtemplate.StringTemplateGroup("super")
subGroup =
stringtemplate.StringTemplateGroup("sub")
subGroup.setSuperGroup(group)
group.defineTemplate("bold", "<b>$it$</b>")
group.defineTemplate("page",
"$name:bold()$")
subGroup.defineTemplate("bold",
"<strong>$it$</strong>")
st = subGroup.getInstanceOf("page")
st["name"] = "Ter"
expecting = "<strong>Ter</strong>"
```

StringTemplate group maps also inherit. If an attribute reference is not found, StringTemplate looks for a map in its group with that name. If not found, the super group is checked.

Template regions

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. (Regions are similar to something in Django). While regions are syntactic sugar on top of template inheritance, the improvement in simplicity and clarity over normal coarser-grained inheritance is substantial.

Regions allow you to mark sections of a template or leave a hole in a template that you can override or define in a subgroup without having to define a separate template. For example, imagine generating code for a method with the following template:

```
group Java;
method(name,code) ::= <<
public void <name>() {
    <code>
}
>>
```

In order to get proper separation of concerns, you would like to avoid generating debugging in your main template group. You would like to have all debugging stuff encapsulated in a debugging group. You could override the entire template but then you are duplicating all of the output literal text, which breaks the "single point of change principle." Instead of using IF conditionals around the debugging code, just leave a hole that a subgroup can override:

```
group Java;
method(name,code) ::= <<
public void <name>() {
    <@preamble()>
    <code>
}
>>
```

In a subgrammar, define the region using a fully qualified name which includes the region's surrounding template name:

```
group dbg : Java;
@method.preamble() ::= <<System.out.println("enter");>>
```

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.

Consider another problem where you would like to replace a small portion of a larger template by creating a subgroup. Imagine you have a template that generates conditionals, but in debug mode you would like to track the fact that you evaluated an expression. Again, to avoid mingling debugging code with your main templates, you need to avoid "if dbg" type template expressions. Instead, mark the region within the template that might be replaced by debugging subgroup:

```
group Java;
test(expr,code) ::= "if (<@eval><expr><@end>) {<code>}"
```

where `<@`*r*`>..<@end>` marks the region called *r*. A subgroup can override this region:

```
group dbg : Java;
@test.eval() ::= "trackAndEval(<expr>)"
```

Regions may not have parameters, but because of the dynamic scoping of attributes, the overridden region may access all of the attributes of the surrounding template.

In an overridden region, `@super.r()`refers to the supergroup template's original region contents.

Auto-indentation

`StringTemplate` has auto-indentation on by default. To turn it off, use `NoIndentWriter` rather than (the default) `AutoIndentWriter`.

At the simplest level, the indentation looks like a simple column count:

```
My dogs' names
  $names; separator="\n"$
The last, unindented line
```

will yield output like:

```
My dog's names
  Fido
  Rex
  Stinky
The last, unindented line
```

where the last line gets "unindented" after displaying the list. `StringTemplate` tracks the characters to the left of the $ or < rather than the column number so that if you indent with tabs versus spaces, you'll get the same indentation in the output.

When there are nested templates, `StringTemplate` tracks the combined indentation:

```
// <user> is indented two spaces
main(user) ::= <<
Hi
\t$user:quote(); separator="\n"$
>>

quote ::= " '$it$'"
```

In this case, you would get output like:

```
Hi
\t 'Bob'
\t 'Ephram'
\t 'Mary'
```

where the combined indentation is tab plus space for the attribute references in template `quote`. Expression `$user$` is indented by 1 tab and hence any attribute generated from it (in this case the `$attr$` of `quote()`) must have
at least the tab.

Consider generating nested statement lists as in C. Any statements inside must be nested 4 spaces. Here are two templates that could take care of this:

```
function(name,body) ::= <<
void $name$() $body$
>>

slist(statements) ::= <<
{
    $statements; separator="\n"$
}>>
```

Your code would create a `function` template instance and an `slist` instance, which gets passed to the `function` template as attribute `body`. The following code:

Java

```
StringTemplate f =
group.getInstanceOf("function");
f.setAttribute("name", "foo");
StringTemplate body =
group.getInstanceOf("slist");
body.setAttribute("statements", "i=1;");
StringTemplate nestedSList =
group.getInstanceOf("slist");
nestedSList.setAttribute("statements",
"i=2;");
body.setAttribute("statements",
nestedSList);
body.setAttribute("statements", "i=3;");
f.setAttribute("body", body);
```

C#

```
StringTemplate f =
group.GetInstanceOf("function");
f.SetAttribute("name", "foo");
StringTemplate body =
group.GetInstanceOf("slist");
body.SetAttribute("statements", "i=1;");
StringTemplate nestedSList =
group.GetInstanceOf("slist");
nestedSList.SetAttribute("statements",
"i=2;");
body.SetAttribute("statements",
nestedSList);
body.SetAttribute("statements", "i=3;");
f.SetAttribute("body", body);
```

Python

```
f = group.getInstanceOf("function")
f["name"] = "foo"
body = group.getInstanceOf("slist")
body["statements"] = "i=1;"
nestedSList = group.getInstanceOf("slist")
nestedSList["statements"] = "i=2;"
body["statements"] = nestedSList
body["statements"] = "i=3;"
f["body"] = body
```

should generate something like:

```
void foo() {
    i=1;
```

```
    {
        i=2;
    }
    i=3;
}
```

Indentation can only occur at the start of a line so indentation is only tracked in front of attribute expressions following a newline.

The one exception to indentation is that naturally, `$if$` actions do not cause indentation as they do not result in any output. However, the subtemplates (THEN and ELSE clauses) will see indentations. For example, in the following template, the two subtemplates are indented by exactly 1 space each:

```
$if(foo)$
 $x$
\t\t$else
 $y$
$endif$
```

Automatic line wrapping

StringTemplate never automatically wraps lines--you must explicitly use the `wrap` option on an expression to indicate that StringTemplate should wrap lines in between expression elements. StringTemplate never breaks literals, but it can break in between a literal and an expression. the line wrapping is soft in the sense that an expression that emits text starting be for the right edge will spit out that element even if it goes past the right edge. In other words, StringTemplate does not break elements to enforce a hard right edge. It will not break line between element and separator To avoid having for example a comma appear at the left edge. You may specify the line with as an argument to `toString()` such as st.toString(72). By default, `toString()` does not wrap lines.

To illustrate the simplest form of line wrapping, consider a simple list of characters that you would like to wrap at, say, line width 3. Use the `wrap` option on the `chars` expression:

```
duh(chars) ::= "<chars; wrap>"
```

If you were to pass in `a,b,c,d,e` and used `toString(3)`, you would see

```
abc
de
```

as output. `wrap` may also take an argument but it's default is simply a
\n string.

To illustrate when you would need a non-default version for this parameter, imagine the difficult task of doing proper Fortran line wrapping. Here is a template that generates a Fortran function with a list of arguments:

```
func(args) ::= <<
       FUNCTION line( <args; separator=","> )
>>
```

```
```

Given parameters a..f as the elements of the `args` list, you would get the following output:

```
       FUNCTION line( a,b,c,d,e,f )
```

But what if you wanted to wrap lines at a width of 30? Simply use `toString(30)` and specify that the expression should wrap using newline followed by six spaces followed by the 'c' character, which can be used as the continuation character:

```
func(args) ::= <<
       FUNCTION line( <args; wrap="\n      c", separator=","> )
>>


       FUNCTION line( a,b,c,d,\n" +
      ce,f )
```

Similarly, if you want to break really long strings, use `wrap="\"+\n \""`, which emits a quote character followed by plus symbol followed by 4 spaces.

StringTemplate properly tracks newlines in the text omitted by your templates so that it can avoid emitting wrap strings right after your template has emitted a newline. StringTemplate also looks at your wrap string to find the (sole) \n character. Wrap strings are of the form A\nB and StringTemplate emits A\n first and then spits out the indentation as required by auto-indentation and then finally B. Again, imagine, the list of characters to emit, but now consider that the expression has been indented:

```
duh(chars) ::= <<
  <chars; wrap>
>>
```

With the same input a..e and `toString(4)`, you would see the following output:

```
  ab
  cd
  e
```

What if the expression is not indented with whitespace but has some text to the left? Consider dumping out an array of numbers as a Java array definition:

```
array(values) ::= <<
int[] a = { <values; wrap, separator=","> };
>>
```

With numbers

```
3,9,20,2,1,4,6,32,5,6,77,888,2,1,6,32,5,6,77,4,9,20,2,
1,4,63,9,20,2,1,4,6,32,5,6,77,6,32,5,6,77,3,9,20,2,1,
4,6,32,5,6,77,888,1,6,32,5
```

this template will emit (at width 40):

```
int[] a = { 3,9,20,2,1,4,6,32,5,6,77,888,
2,1,6,32,5,6,77,4,9,20,2,1,4,63,9,20,2,1,
4,6,32,5,6,77,6,32,5,6,77,3,9,20,2,1,4,6,
32,5,6,77,888,1,6,32,5 };
```

While correct, that is not particularly beautiful code. What you really want, is for the numbers to line up with the start of the expression; in this case under the first "3". to do this, use the `anchor` option, which means StringTemplate should line up all wrapped lines with left edge of expression when wrapping:

```
array(values) ::= <<
int[] a = { <values; wrap, anchor, separator=","> };
>>
```

Adding that option generates the following output:

```
int[] a = { 3,9,20,2,1,4,6,32,5,6,77,888,
            2,1,6,32,5,6,77,4,9,20,2,1,4,
            63,9,20,2,1,4,6,32,5,6,77,6,
            32,5,6,77,3,9,20,2,1,4,6,32,
            5,6,77,888,1,6,32,5 };
```

One final complication. Sometimes you want to anchor the left edge of all wrapped lines in a position to the left of where the expression starts. For example what if you wanted to print out three literal values first such as "1,9,2"? Because StringTemplate can only anchor at expressions simply wrap the literals and your values expression in an embedded anonymous template (enclose them with `<{...}>`) and use the anchor on that embedded template:

```
data(a) ::= <<
int[] a = { <{1,9,2,<values; wrap, separator=",">}; anchor> };
>>
```

That template yields the following output:

```
int[] a = { 1,9,2,3,9,20,2,1,4,
            6,32,5,6,77,888,2,
            1,6,32,5,6,77,4,9,
            20,2,1,4,63,9,20,2,
            1,4,6 };
```

If there is both an indentation and an anchor, StringTemplate chooses whichever is larger.

WARNING: separators and wrap values are templates and are evaluated once **before** multi-valued expressions are evaluated. You cannot change the wrap based on, for example, `<i>`.

Default values for wrap="\n", anchor="true" (any non-null value means anchor).

Output Filters

Version 2.0 introduced the notion of an `StringTemplateWriter/IStringTemplateWriter`. All text rendered from a template goes through one of these writers before being placed in the output buffer. Terence added this primarily for auto-indentation for code generation, but it also could be used to remove whitespace (as a compression) from HTML output. Most recently, in 2.3, Terence updated the interface to

support automatic line wrapping. If you don't care about indentation, you can simply subclass `AutoIndentWriter` and override `write()`/`Write()`:

Java

```
public interface StringTemplateWriter {
    public static final int NO_WRAP = -1;

    void pushIndentation(String indent);

    String popIndentation();

    void pushAnchorPoint();

    void popAnchorPoint();

    void setLineWidth(int lineWidth);

    /** Write the string and return how
many actual chars were written.
     *  With autoindentation and wrapping,
more chars than length(str)
     *  can be emitted.  No wrapping is
done.
     */
    int write(String str) throws
IOException;

    /** Same as write, but wrap lines using
the indicated string as the
     *  wrap character (such as "\n").
     */
    int write(String str, String wrap)
throws IOException;

    /** Because we might need to wrap at a
non-atomic string boundary
     *  (such as when we wrap in between
template applications
     *   <data:{v|[<v>]}; wrap>) we need to
expose the wrap string
     *  writing just like for the
separator.
     */
    public int writeWrapSeparator(String
wrap) throws IOException;

    /** Write a separator.  Same as write()
except that a \n cannot
     *  be inserted before emitting a
separator.
     */
    int writeSeparator(String str) throws
IOException;
}
```

C#

```
public interface IStringTemplateWriter
{
    void PushIndentation(string indent);

    string PopIndentation();

    void Write(string str);
}
```

Python

```
class StringTemplateWriter(object):

    def __init__(self):
        pass

    def pushIndentation(self, indent):
```

```
        raise NotImplementedError

    def popIndentation(self):
        raise NotImplementedError

    def write(self, str):
        raise NotImplementedError
```

Here is a "pass through" writer that is already defined:

Java

```
/** Just pass through the text */
public class NoIndentWriter extends
AutoIndentWriter {
    public NoIndentWriter(Writer out) {
        super(out);
    }

    public void write(String str) throws
IOException {
            out.write(str);
    }
}
```

C#

```
/** Just pass through the text */
public class NoIndentWriter :
AutoIndentWriter
{
    public NoIndentWriter(TextWriter
output) :base(output)
    {
    }

    public void Write(string str)
    {
            output.Write(str);
    }
}
```

Python

```
## Just pass through the text
#
class NoIndentWriter(AutoIndentWriter):

    def __init__(self, out):
        super(NoIndentWriter,
self).__init__(out)

    def write(self, str):
        self.out.write(str)
        return len(str)
```

Use it like this:

Java

```
StringWriter out = new StringWriter();
StringTemplateGroup group =
                new
StringTemplateGroup("test");
group.defineTemplate("bold", "<b>$x$</b>");
StringTemplate nameST = new
StringTemplate(group,
"$name:bold(x=name)$");
```

	`nameST.setAttribute("name", "Terence");`
	`// write to 'out' with no indentation`
	`nameST.write(new NoIndentWriter(out));`
	`System.out.println("output:`
	`"+out.toString());`

C#	`StringWriter output = new StringWriter();`
	`StringTemplateGroup group = new`
	`StringTemplateGroup("test");`
	`group.DefineTemplate("bold", "<b>$x$</b>");`
	`StringTemplate nameST = new`
	`StringTemplate(group,`
	`"$name:bold(x=name)$");`
	`nameST.SetAttribute("name", "Terence");`
	`// write to 'out' with no indentation`
	`nameST.Write(new NoIndentWriter(output));`
	`Console.Out.WriteLine("output:`
	`"+output.ToString());`

Python	`out = StringIO()`
	`group =`
	`stringtemplate.StringTemplateGroup("test")`
	`group.defineTemplate("bold", "<b>$x$</b>")`
	`nameST =`
	`stringtemplate.StringTemplate(group,`
	`"$name:bold(x=name)$")`
	`nameST["name"] = "Terence"`
	`# write to 'out' with no indentation`
	`nameST.write(NoIndentWriter(out))`
	`print "output:", str(out)`

Instead of using `nameST.toString()`, which calls `write` with a string write and returns its value, manually invoke `write` with your writer.

If you want to always use a particular output filter, then use

Java	`StringTemplateGroup.setStringTemplateWriter(Class` `userSpecifiedWriterClass);`
C#	`StringTemplateGroup.SetStringTemplateWriter(Type` `userSpecifiedWriterClass);`
Python	`stringtemplate.StringTemplateGroup.setStringTemplateWrite`

The `StringTemplate.toString()` method is sensitive to the group's writer class.

StringTemplate Grammars

`StringTemplate` has multiple grammars that describe templates at varying degrees of detail. At the grossest level of granularity, the `group.g` grammar accepts a list of templates with formal template arguments. Each of these templates is broken up into chunks of literal text and attribute expressions via

template.g. The default lexer uses $...$ delimiters, but the angle.bracket.template.g lexer provides <...> delimiters. Each of the attribute expression chunks is processed by action.g. It builds trees (ASTs) representing the operation indicated in the expression. These ASTs represent the "precompiled" templates, which are evaluated by the tree grammar eval.g each time a StringTemplate is rendered to string with ToString().

The grammar files are:

- group.g: read a group file full of templates
- template.g: break an individual template into chunks
- angle.bracket.template.g: <...> template lexer
- action.g: parse attribute expressions into ASTs
- eval.g: evaluate expression ASTs during ToString()

Anything outside of the StringTemplate start/stop delimiters is ignored.

A word about Strings. Strings are double-quoted with optional embedded escaped characters that are translated (escapes are not translated outside of strings; for example, text outside of attribute expressions do not get escape chars translated except \$, \< and \>).

```
<<
STRING
    :   '"' (ESC_CHAR | ~'"')* '"'
    ;
>>
```

The translated escapes are:

```
<<
ESC_CHAR
    :   '\\'
        (   'n'
        |   'r'
        |   't'
        |   'b'
        |   'f'
        |   '"'
        |   '\\'
        )
     ;
>>
```

but other escapes are allowed and ignored.

Please see the actual grammar files for the formal language specification of StringTemplate's various components.

Debugging

Debugging complex and nested StringTemplate trees can be challenging. Kay Roepke is building a graphical interface similar to ANTLRWorks for StringTemplate but until then you have a number of tools that you can use.

You can ask for the enclosing template structure with StringTemplate.getEnclosingInstanceStackString() and can get the entire structure with toStructureString() that does not print the values but shows the nested structure with the attribute names.

If for some reason StringTemplate goes into an infinite loop when you try to render a template, you probably have a circular reference in your template containment hierarchy. Turning on link mode with StringTemplate.setLintMode() will check for these cyclic references and a number of other features. This will slow down template rendering so only use this during debugging.

Added StringTemplate. getDOTForDependencyGraph() a DOT diagram showing edges from n->m where template n contains template m. It finds all direct template invocations too like <foo()> but not indirect ones like <(name)()>. This is done statically and hence StringTemplate cannot see runtime arg values on statically included templates. You get a template back that lets you reset node shape, fontsize, width, height attributes. Use removeAttribute before setting so you are sure you only get one value.

Perhaps the most potent debugging tool you have for unraveling the complex structures emitted from nested StringTemplate containment hierarchies is the use of start and stop tags that marked the beginning and end of the text generated from a particular template. Method StringTemplateGroup.emitDebugStartStopStrings() indicates whether StringTemplate should emit <templatename>...</templatename> output for templates from this group. This easily answers an important question: "what template emitted a particular piece of text in the output?" In many cases you will not want every single template to have those tags in the output. For example, in the ANTLR code generator, there is a template that indicates what the output file extension is. Clearly one does not want the file extension to have the debugging information has the code generator could not open a file with those angle brackets and so on. Here's the snippet from the code generator:

```
if ( EMIT_TEMPLATE_DELIMITERS ) {
        templates.emitDebugStartStopStrings(true);
        templates.doNotEmitDebugStringsForTemplate("codeFileExtension");
}
```

Sometimes you use or define templates improperly. Either you set an attribute that is not used or forget to set one or reference the wrong template etc... The following code snippets unable Java and C# to display template hierarchies in a tree view.

Java	
	I have made a toy visualization tool via that shows both the attributes and the way StringTemplate breaks up your template into chunks. It properly handles StringTemplate objects as attributes and other nested structures. Here is the way to launch a Swing frame to view your template:

	`StringTemplate st = ...;`
	`StringTemplateTreeView viz = new StringTemplateTreeView("sample",st);`
	`viz.setVisible(true);`

	Here is an example display: Cannot resolve external resource into attachment.

C#	
	The `StringTemplateViewer` project is a basic visualization tool that shows both the attributes and the way `StringTemplate` breaks up your template into chunks. It properly handles `StringTemplate` objects as attributes and other nested structures. Here is the way to launch a StringTemplateTreeView form to view your template:

	`StringTemplate st = ...;`
	`StringTemplateTreeView stForm = new StringTemplateTreeView("StringTemplateTreeView Example", st);`
	`Application.Run(stForm);`

	Here is a snapshot. The display is associated with the fill-a-table example below. The `StringTemplateViewer` tool for `StringTemplate` visualization is an alpha quality release. Expect all the usual problems associated with alpha quality code.

Acknowledgements

Please see http://www.antlr.org/credits.html