Added by Terence Parr, last edited by Benjamin Niemann on Nov 23, 2007 (view change)
Labels: (None)

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 stringtemplate3

hello = stringtemplate3.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.

Added by Terence Parr, last edited by Benjamin Niemann on Nov 23, 2007 (view change)
Labels: (None)

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

```
query = stringtemplate3.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 = stringtemplate3.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 = stringtemplate3.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 = stringtemplate3.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

FIXME: there was an implementation, test&document it!

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.

Java

```
StringTemplateGroup group = new StringTemplateGroup("myGroup", "/tmp");
group.setRefreshInterval(0);  // no caching
group.setRefreshInterval(Integer.MAX\_INT);  // no refreshing
```

C#

```
StringTemplateGroup group = new StringTemplateGroup("myGroup", "/tmp");
group.setRefreshInterval(0);  // no caching
group.setRefreshInterval(Int32.MaxValue);  // no refreshing
```

FIXME: Please verify if this is correct

Python

```
group = stringtemplate3.StringTemplateGroup("myGroup", "/tmp")
group.refreshInterval = 0  # no caching
group.refreshInterval = sys.maxint # no refreshing
```

Setting the expression delimiters

Added by Terence Parr, last edited by Benjamin Niemann on Nov 23, 2007 (view change)
Labels: (None)

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 = stringtemplate3.StringTemplateGroup("sqlstuff", "/tmp", lexer="angle-bracket")
query = stringtemplate3.StringTemplate("SELECT <column> FROM <table>;", group=group)
query["column"] = "name"
query["table"] = "User"
```

Python accepts either a antlr.CharScanner class (`stringtemplate3.language.DefaultTemplateLexer.Lexer`, `stringtemplate3.language.AngleBracketTemplateLexer.Lexer` or your own implementation) or the string literals `'default'` and `'angle-bracket'`. Also note the use of the keyword argument `lexer`.

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 `<...>`".

Conditionally included subtemplates

Added by Terence Parr, last edited by Terence Parr on Nov 10, 2007 (view change)
Labels: (None)

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, "$if(!member)$...$endif$".

You can also use elseif to make a chain of tests:

```
$if(x)$
...
$elseif(y)$
...
$elseif(z)$
...
$else$
...
$endif$
```

The first true expression "wins".

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.

Added by Terence Parr, last edited by Benjamin Niemann on Nov 23, 2007 (view change)
Labels: (None)

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

```
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 variable, 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 = stringtemplate3.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` that 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. Also note that the map must have keys that are Strings for indirect property referencing to work, because the key is first rendered into a string by ST and then that is used to look up the value in the map.

Difficult property names

Some property names cause parse errors because of clashes with built in keywords or because they do not match the rules for IDs as used by String Template. These difficult property names can be dealt with by quoting the property name in combination with the indirect property construct:

```
$person.("first")$       --- Build in keyword
$person.("1")$           --- non ID conforment name
```

Difficult properties names are quite likely to occur when dealing with maps. Map keys can be defined using arbitrary strings, including spaces and syntax characters used to defined templates themselves.

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 aggregate 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 = stringtemplate3.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 = stringtemplate3.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	st.passThroughAttributes = True

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 `lambda` 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 = stringtemplate3.StringTemplateGroup(file=StringIO(templates), lexer="angle-bracket")
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.

StringTemplate

Object rendering

Added by Terence Parr, last edited by Benjamin Niemann on Nov 23, 2007 (view change)
Labels: (None)

Note: You should also look at The Internationalization and Localization of Web Applications.

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 stringtemplate3
from datetime import date

class DateRenderer(stringtemplate3.AttributeRenderer):
    def toString(self, o, format=None):
        return o.strftime("%Y.%m.%d")
...
...
st = stringtemplate3.StringTemplate("date: <created>", lexer="angle-bracket")
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.

Note: In light of the new format option the following paragraph should be revised.

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.

Format Option

There are cases where the template is the only reasonable place to determine what formatting needs to be applied to an attribute. For example, when generating HTML different characters need to be escaped in an attribute value than in element content. Only the template knows where it is going to put an attribute. Another, perhaps less likely, example would be a template that is rendering Java code that has SQL statements in Java strings. Attributes within the SQL statements will need different escaping.

The format option allows the template to decided what formatting to use where but leaves the details of how the formatting is done completely in the hands of the controller.

To make use of the format option you must create a renderer that implements interface AttributeRenderer and provides an implementation for the toString method that takes a formatName String.

Java

```
public class BasicFormatRenderer implements AttributeRenderer {
    public String toString(Object o) {
        return o.toString();
    }
    public String toString(Object o, String formatName) {
        if (formatName.equals("toUpper")) {
            return o.toString().toUpperCase();
        } else if (formatName.equals("toLower")) {
            return o.toString().toLowerCase();
        } else {
            throw new IllegalArgumentException("Unsupported format name");
        }
    }
}
```

Python

```
class BasicFormatRenderer(stringtemplate3.AttributeRenderer):
    def toString(self, o, formatName=None):
        if formatName is None:
            # no formatting specified
            return str(o)

        if formatName == "toUpper":
            return str(o).upper()
        elif formatName == "toLower":
            return str(o).lower()
        else:
            raise ValueError("Unsupported format name")
```

The render is registered with a group as previously shown. The renderer can do anything it likes to format the string. The toUpper and toLower cases are examples of what can be done. It is not required that an exception is thrown if the `formatName` is not supported you could also simply return the result of `o.toString()`.

From a template you can now use any of the named formats supported by the registered renderers. For example:

```
$name;format="toUpper"$
```

The expression after the equal sign must resolve to a string that matches one of the strings that the renderer recognizes. There is no default value for the format option.

The format option can be combined with any of the other options. Format will apply to the value of the null option but not to the separator.

For example

```
$list : { [$it$] };format="toUpper",separator=" and ",null="woops"$
```

results in

```
[X] and [Y] and [WOOPS] and [Z]
```

when list contains "x", "y", null, "y" and toUpper is a supported format option of the available renderer for type String

that returns the upper case input string. Note that the value of null was upper cased but the separator " and " was not.

If you really want the separator to be formatted then you must do this

```
${$list : { [$it$] };separator=" and ",null="woops"$};format="toUpper"$
```

Added by Terence Parr, last edited by Benjamin Niemann on Nov 23, 2007 (view change)
Labels: (None)

There are 5 expression options at the moment:

- **separator**. Specify text to be emitted between multiple values emitted for a single expression. For example, given a list of names, `<names>` spits them out right next to each other. Using a separator can put a comma in between automatically: `<names; separator=", ">`. This is by far the most commonly used option. See How to construct separators?.
- **format**. Used in conjunction with the `AttributeRenderer` interface, which describes an object that knows how to format or otherwise render an object appropriately. The `toString(Object,String)` method is used when the user uses the `format` option: `$o; format="f"$`. Renderers check the formatName and apply the appropriate formatting. If the format string passed to the renderer is not recognized, then it should simply call `toString(Object)`.
  This option is very effective for locale changes and for choosing the display characteristics of an object in the template rather than encode.
  Each template may have a renderer for each object type or can default to the group's renderer or the super group's renderer if the group doesn't have one. See Object rendering#Format Option.
- **null**. Emit a special value for each null element. For example, given values=9,6,null,2,null

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

  emits:

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

  See Expressions

- **wrap**. Tell ST that it is okay to wrapped lines to get too long. The `wrap` option may also take an argument but it's default is simply a \n string. You must specify an integer width using the `toString(int)` method to get ST to actually wrap expressions modified with this option. For example, given a list of names and expression `<names; wrap>`, a call to `toString(72)` will emit the names until it surpasses 72 characters in with and then inserts a and begins emitting names again. Naturally this can be used in conjunction with the separator option. ST Never breaks in between a real element and the separator; the wrap occurs only after a separator. See Automatic line wrapping.
- **anchor**. Line up all wrapped lines with left edge of expression when wrapping. Default is anchor="true" (any non-null value means anchor). See Automatic line wrapping.

The option values are all full expressions, which can include references to templates, anonymous templates, and so on. For example here is a separator that invokes another template:

```
<ul>$name; separator=bulletSeparator(foo=" ")+" "$</ul>
```

The wrap and anchor options are implemented via the Output Filters. The others are handled during interpretation by ST. Well, the filters also are notified that a separator vs regular string is coming out to prevent newlines between real elements and separators.

Java examples

Here is an example use of the `format` option.

```
public void testRendererWithFormatAndList() throws Exception {
    StringTemplate st =new StringTemplate(
                    "The names: <names; format=\"upper\">",
                    AngleBracketTemplateLexer.class);
    st.setAttribute("names", "ter");
    st.setAttribute("names", "tom");
    st.setAttribute("names", "sriram");
    st.registerRenderer(String.class, new StringRenderer());
    String expecting = "The names: TERTOMSRIRAM";
    String result = st.toString();
    assertEquals(expecting, result);
}
```

The code registers a renderer for the String class. Without the format option, the `toString(Object)` method is used to convert strings to the emitted text. With the option, the `toString(Object, String)` method is invoked. Here is the renderer used in the example:

```
public class StringRenderer implements AttributeRenderer {
    public String toString(Object o) {
            return (String)o;
    }
    public String toString(Object o, String formatString) {
            if ( formatString.equals("upper") ) {
                    return ((String)o).toUpperCase();
            }
            return toString(o);
    }
}
```

The following code snippet is the same as the previous example except for the introduction of the separator option, which cleans up the output as you can see by the `expecting` string:

```
public void testRendererWithFormatAndSeparator() throws Exception {
    StringTemplate st =new StringTemplate(
                    "The names: <names; separator=\" and \", format=\"upper\">",
                    AngleBracketTemplateLexer.class);
    st.setAttribute("names", "ter");
    st.setAttribute("names", "tom");
    st.setAttribute("names", "sriram");
    st.registerRenderer(String.class, new StringRenderer());
    String expecting = "The names: TER and TOM and SRIRAM";
    String result = st.toString();
    assertEquals(expecting, result);
}
```

If there are null elements in the list of names, you can specify a string to replace all of the null values using the `null` option:

```
public void testRendererWithFormatAndSeparatorAndNull() throws Exception {
    StringTemplate st =new StringTemplate(
        "The names: <names; separator=\" and \", null=\"n/a\", format=\"upper\">",
        AngleBracketTemplateLexer.class);
    List names = new ArrayList();
    names.add("ter");
    names.add(null);
    names.add("sriram");
    st.setAttribute("names", names);
    st.registerRenderer(String.class, new StringRenderer());
    String expecting = "The names: TER and N/A and SRIRAM";
    String result = st.toString();
    assertEquals(expecting, result);
}
```

Python examples

If you are constructing HTML documents you have to escape plain text strings so that < or & characters appear as literal text and do not act as HTML delimiters (thus opening a wide range of possible attacks if the text originated from user input).

```
import cgi
import stringtemplate3

group = stringtemplate3.StringTemplateGroup(
    name="default", rootDir="path/to/templates/"
    )

class EscapeRenderer(stringtemplate3.AttributeRenderer):
    def toString(self, o, formatName=None):
        if formatName is None:
            # no formatting specified
            return str(o)

        if formatName == "escape":
            return cgi.escape(str(o))
        else:
            raise ValueError("Unsupported format name")

group.registerRenderer(str, EscapeRenderer())

st = group.getInstanceOf("blogEntry")
st['comment'] = database.loadComment() # an instance with username, text, url, ... attributes
```

Then you can use `$comment.text; format="escape"$` in your templates whenever an attribute is not known to be save.

Comments (Hide)

Added by Terence Parr, last edited by Benjamin Niemann on Nov 23, 2007 (view change)
Labels: (None)

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):
    NO_WRAP = -1

    def __init__(self):
        pass

    def pushIndentation(self, indent):
        raise NotImplementedError

    def popIndentation(self):
        raise NotImplementedError

    def pushAnchorPoint(self):
        raise NotImplementedError

    def popAnchorPoint(self):
        raise NotImplementedError

    def setLineWidth(self, lineWidth):
        raise NotImplementedError

    def write(self, str, wrap=None):
        raise NotImplementedError

    def writeWrapSeparator(self, wrap):
        raise NotImplementedError

    def writeSeparator(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

```
class NoIndentWriter(stringtemplate3.AutoIndentWriter):
    """Just pass through the text"""
    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 = stringtemplate3.StringTemplateGroup("test")
group.defineTemplate("bold", "<b>$x$</b>")
nameST = stringtemplate3.StringTemplate("$name:bold(x=name)$", group=group)
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.setStringTemplateWriter(userSpecifiedWriterClass)
```

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

Template and attribute lookup rules

Added by Terence Parr, last edited by Benjamin Niemann on Nov 23, 2007 (view change)
Labels: (None)

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 = stringtemplate3.StringTemplateGroup(file=StringIO(templates), lexer=' default')
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"
    )
stringtemplate3.lintMode = True
group = stringtemplate3.StringTemplateGroup(file=StringIO(templates), lexer="default")
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 stringtemplate3.language.ASTExpr.IllegalStateException, exc:
    # do something
```

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)
...
```