

This page last changed on May 12, 2006 by kunle_odutola@hotmail.com.

C# StringTemplate Documentation

Terence Parr
University of San Francisco
parrt@cs.usfca.edu
Copyright 2003-2005
<http://www.stringtemplate.org>

C# version created by

Kunle Odutola
kunle_UNDERSCORE_odutola@hotmail.com
Copyright 2005-2006
(ST# - C# StringTemplate released under BSD License)

Version 2.3b7 May 11, 2006

Please see the [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.

Note that the `TestStringTemplates.cs` file has many unit tests defined that also represent a useful set of examples. Additionally, it is highly recommended that you read the (academically-oriented) paper, [Enforcing Model-View Separation in Template Engines](#). There are some more examples including nested menu generation that will be of interest.

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 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 `StringTemplate.ToString()`. For example, the following template has two chunks, a literal and a reference to attribute `name`:

```
Hello, $name$
```

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

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

`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 C# 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 a reaction to the completely entangled specifications encouraged by ASP (Active Server Pages), JSP (Java 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.

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.

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.

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.

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 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.get_names()` (assuming the model has a C# property named `names`), which presumably loads a list of names from the database. The reference to `numberNames` invokes `model.get_numberNames()` which necessarily uses the internal data structure computed by `get_names()` to compute `names.Count` 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 `get_numberNames()` invoke `get_names()` 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.

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 C# Code

Here is a simple piece of C# that creates and uses a template on the fly:

```
StringTemplate query = new StringTemplate("SELECT $column$ FROM $table$;");
query.SetAttribute("column", "name");
query.SetAttribute("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 `query.ToString()` 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 `SetAttribute()` call as shown below makes the attribute named `column` multi-valued:

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

Invoking `query.ToString()` 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:

```
StringTemplate query = new StringTemplate("SELECT $column; separator=\"\", \"$ FROM $table$;");
query.SetAttribute("column", "name");
query.SetAttribute("column", "email");
query.SetAttribute("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 C# string. We could also have used a C# verbatim string - `@"SELECT $column; separator="", "$ FROM $table$;"`. In general, the right-hand-side can be any attribute expression. Invoking `query.ToString()` 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 C# code below creates a `StringTemplateGroup` called `myGroup` rooted at `/tmp` so that requests for template `theQuery` forces a load of file `/tmp/theQuery.st`.

```
StringTemplateGroup group = new StringTemplateGroup("myGroup", "/tmp");
StringTemplate query = group.GetInstanceOf("theQuery");
query.SetAttribute("column", "name");
query.SetAttribute("column", "email");
query.SetAttribute("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 your Assembly's Location

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, often know relative to the location of your templates relative to the where your 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");
```

StringTemplate 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 (I'm using `<...>` delimiters) 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">
}
\>>
```

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 is blank.

Using templates in a group file is straightforward. A `StringTemplateGroup` constructor accepts a `System.IO.TextReader` so you can pass in a string or file or whatever:

```
String templates = "group simple; vardef(type,name) ..."; // templates from above
StringTemplateGroup group = new StringTemplateGroup(new StringReader(templates));
StringTemplate t = group.GetInstanceOf("vardef");
t.SetAttribute("type", "int");
t.SetAttribute("name", "foo");
Console.Out.WriteLine(t);
```

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

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 a `InvalidOperationException`.

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

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 an `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)>`.

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".

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

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 file format

```
group
:   "group" ID ';' (template|mapdef)*
;

template
:   ID '(' (args)? ')' "::~=" TEMPLATE
|   ID "::~=" ID // alias one template to be another
;

args:   arg (',' arg)*
;

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

mapdef
:   ID "::~=" map
;

map :   '[' keyValuePair (',' keyValuePair)* ']'
;

keyValuePair
:   STRING ':' STRING
|   "default" ':' STRING
;
```

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 usually have a `.stg` file extension.

Functionality

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.

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):

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 accessor methods and also a field with the property name. In the above example, `StringTemplate` would look for

fields `name` and `email` without the capitalization typically used with property access methods.

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

```
User u = database.LookupPerson("parrt@jguru.com");
st.SetAttribute("person", u);
```

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

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

where `Connector` is defined as:

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

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

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

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

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".

Automatic aggregate creation

Creating one-off data aggregates in C# 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:

```
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";
```

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`.

Overloads of the `SetAttribute()` method to handle from 1 to unlimited arguments.

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 than

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

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.

Attribute operators

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

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:

- 1 You can reuse the template over and over (no cut/paste)
- 1 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 C# 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:

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

```
StringTemplate page = group.GetInstanceOf("page");
page.SetAttribute("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 `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 Expressions

When setting parameters or testing IF conditionals, you may find it handy to use the plus "string concatenate" operator. For example, when building web pages, you will find it useful to create a template called `link` and then use it to generate HTML link tags; you may want to change the way every link looks on your site and it's convenient to have one place to change things. The template might look like:

```
<a href="$url$"><b>$title$</b></a>
```

Then in a page template you might reference:

```
...$link(url="http://www.jguru.com", title="jGuru")$...
```

or you could use attributes to set the link parameters:

```
...$link(url=person.homePage, title=person.name)$...
```

Sometimes you may want to compute the URL; usually it is enough to concatenate strings:

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

where `faqid` and `faqtitle` are attributes you would have to 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$}, title=faqtitle)$...
```

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:

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

where 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 were C# method calls and `item` were 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`.

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. Another attribute `i` (of type `integer`) is also set to the value's index number starting from 1. 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.

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>"
```

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. `names0` would get `blueListItem()` applied to it, `names1` would get `greenListItem()`, and `names2` 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 curlyes 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; apply 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:

```
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");
Console.Out.WriteLine("f="+f);
String expecting = ""+newline+newline;
```

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

```
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;} }
}
```

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 `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.

Template visualization

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 `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.

%image(page.treeview.jpg)



The `StringTemplateViewer` tool for `StringTemplate` visualization is an alpha quality release. Expect all the usual problems associated with alpha quality code.



If you turn on "lint mode" via `StringTemplate.SetLintMode(true)` then you may access an attribute named `attributes`, which is a text string that recursively dumps out types, properties, etc... It does not print out their values.

Functionality Summary

| Syntax | Description |
|--|---|
| <code><attribute></code> | Evaluates to the value of <code>attribute.ToString()</code> if it exists else empty string. |
| <code><attribute.property></code> | Looks for <i>property</i> of <i>attribute</i> as a C# property, then accessor methods like <code>getProperty()</code> or <code>isProperty()</code> . If that fails, <code>StringTemplate</code> looks for a raw field of the <i>attribute</i> called <i>property</i> . Evaluates to the empty string if no such property is found. |
| <code><attribute.(expr)></code> | Indirect property lookup. Same as <i>attribute.property</i> except use the value of <i>expr</i> as the <code>property_name</code> . Evaluates to the empty string if no such property is found. |
| <code><multi-valued-attribute></code> | Concatenation of <code>ToString()</code> invoked on each element. If <i>multi-valued-attribute</i> is missing this evaluates to the empty string. |
| <code><multi-valued-attribute; separator=expr></code> | Concatenation of <code>ToString()</code> invoked on each element separated by <i>expr</i> . |
| <code><template(argument-list)></code> | Include <i>template</i> . The <i>argument-list</i> is a list of attribute assignments where each assignment is of the form <i>arg-of-template=expr</i> where <i>expr</i> is evaluated in the context of the surrounding template not of the invoked template. |
| <code><(expr)(argument-list)></code> | Include <i>template</i> whose name is computed via <i>expr</i> . The <i>argument-list</i> is a list of attribute assignments where each assignment is of the form <i>attribute=expr</i> . Example <code>\$(whichFormat())\$</code> looks up <code>whichFormat</code> 's value and uses that as template name. Can also apply an indirect template to an attribute. |
| <code><attribute:template(argument-list)></code> | Apply <i>template</i> to <i>attribute</i> . The optional <i>argument-list</i> is evaluated before application so that you can set attributes referenced within <i>template</i> . The default attribute <i>it</i> is set to the value of <i>attribute</i> . If <i>attribute</i> is multi-valued, then <i>it</i> is set to each element in turn and <i>template</i> is invoked <i>n</i> times where <i>n</i> is the number of values in <i>attribute</i> . Example: <code>\$name:bold()</code> applies <code>bold()</code> to <i>name</i> 's value. |
| <code><attribute:(expr)(argument-list)></code> | Apply a template, whose name is computed from <i>expr</i> , to each value of <i>attribute</i> . Example <code>\$data:(name())\$</code> looks up <i>name</i> 's value and uses that as template name to apply to <i>data</i> . |
| <code><attribute:t1(argument-list): ... :tN(argument-list)></code> | 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 <i>templateN</i> 's application. |
| <code><attribute:{anonymous-template}></code> | Apply an anonymous template to each element of <i>attribute</i> . The iterated <code>it</code> attribute is set automatically. |
| <code><attribute:{argument-name anonymous-template}></code> | Apply an anonymous template to each element of <i>attribute</i> . Set the <i>argument-name</i> to the iterated value and also set <code>it</code> . |
| <code><a1,a2,...,aN:{argument-list anonymous-template}></code> | Parallel list iteration. March through the values of the attributes <i>a1..aN</i> , setting the values to the arguments in <i>argument-list</i> in the same order. Apply the anonymous template. There is no defined <code>it</code> value unless inherited from an enclosing scope. |
| <code><attribute:t1(),t2(),...,tN()></code> | Apply an alternating list of templates to the elements of <i>attribute</i> . The template names may include argument lists. |
| <code><if(attribute)>subtemplate <else>subtemplate2 <endif></code> | If <i>attribute</i> has a value or is a <code>bool</code> object that evaluates to <code>true</code> , include <i>subtemplate</i> else include <i>subtemplate2</i> . These conditionals may be nested. |
| <code><if(!attribute)>subtemplate<endif></code> | If <i>attribute</i> has no value or is a <code>bool</code> object that evaluates to <code>false</code> , include <i>subtemplate</i> . These conditionals may be nested. |
| <code>\\$ or \<</code> | escaped delimiter prevents <code>\$</code> or <code><</code> from starting an attribute expression and results in that single character. |
| <code><\ >, <\n>, <\t>, <\r></code> | special characters: space, newline, tab, carriage return. |
| <code><! comment !>, \$! comment !\$</code> | Comments, ignored by <code>StringTemplate</code> . |

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 interface, `IAttributeRenderer`, that defines how an object is rendered to string. Here is a renderer that renders `DateTime` date objects tersely.

```
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();
}
```

All attributes of type `DateTime` 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 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 propagated 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 C# code.

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

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

```

supergroup.DefineTemplate("bold", "<b>$it$</b>");
subgroup.SuperGroup = supergroup;
StringTemplate st = new StringTemplate(subgroup, "$name:bold()$");
st.SetAttribute("name", "Terence");
string expecting = "<b>Terence</b>";

```

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:

```

supergroup.DefineTemplate("bold", "<b>$it$</b>");
subgroup.DefineTemplate("bold", "<strong>$it$</strong>");

```

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

```

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");

```

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 in C#, for example, would be the following example (the Java equivalent of this catches my students all the time): 😊

```

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 C# 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`.

```

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>";
```

`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 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.

```
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 = "{{}}";
```

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.

```
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 ex) {
    ...
}
```

The nested template stack trace from Exception object `ex` 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.

As of 2.0, the start and stop strings are limited to either `$...$` or `<...>` (unless you build your own lexical analyzer to break apart templates into chunks).

To use the angle bracket delimiters you must create a `StringTemplateGroup`:

```
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");
```

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 `<...>`".

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 `Int32.MaxValue` to have no refreshing at all.

Output Filters

Version 2.0 introduced the notion of an `IStringTemplateWriter`. All text rendered from a template goes through one of these writers before being placed in the output buffer. I added this primarily for auto-indentation for code generation, but it also could be used to remove whitespace (as a compression) from HTML output. If you don't care about indentation, you can simply implement `{Write()}:`

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

    string PopIndentation();
}
```

```
    void Write(string str);  
}
```

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

```
/** Just pass through the text */  
public class NoIndentWriter : AutoIndentWriter  
{  
    public NoIndentWriter(TextWriter output) :base(output)  
    {  
    }  
  
    public void Write(string str)  
    {  
        output.Write(str);  
    }  
}
```

Use it like this:

```
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());
```

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

```
StringTemplateGroup.SetStringTemplateWriter(Type userSpecifiedWriterClass);
```

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

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 C# code:

```
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);
```


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  
  $y$  
$endif$
```

Examples

You should look at `Antlr.StringTemplate.Tests.TestStringTemplate` in the `StringTemplateTests` project, which contains many tests.

Fill-a-Table Example

The manner in which a template engine handles filling an HTML table with data often provides good insight into its programming and design strategy. It illustrates the interaction of the model and view via the controller. Using `StringTemplate`, the view may not access the model directly; rather the view is the passive recipient of data from the model.

First, imagine we have objects of type `User` that we will pull from a simulated database:

```
public class User {  
    string name;  
    int age;  
    public User(string name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public string Name { get { return name; } }  
    public int Age { get { return age; } }  
}
```

Our database is just a static list:

```
static User[] users = new User[] {
```

```

    new User("Boris", 39),
    new User("Natasha", 31),
    new User("Jorge", 25),
    new User("Vladimir", 28)
};

```

Here is my simple overall page design template, `page.st`:

```

<html>
<head>
<title>${title$}</title>
</head>
<body>
<h1>${title$}</h1>

$body$

</body>
</html>

```

The body attribute of `page.st` will be set to the following template `users.inline.st` by my web server infrastructure (part of the controller):

```

<table border=1>
$users:{
  <tr>
    <td>${it.name$}</td><td>${it.age$}</td>
  </tr>
}$
</table>

```

Again, it is the default attribute passed to a template when you apply that template to an attribute or attributes. `it.name` gets the `name` property, if it exists, from the `it` object. That is, `StringTemplate` uses reflection to call the `Name` property, then the `get_Name()` method then others including the `getName()` method on the incoming object. By using reflection, I avoid a type dependence between model and view.

Now, imagine the server and templates are set up to format data. My page definition is part of the controller that pulls data from the model (the database) and pushes into the view (the template). That is all the page definition should do--interpret the data and set some attributes in the view. The view only formats data and does no interpretation.

```

public class UserListPage : SamplePage {
    /** This "controller" pulls from "model" and pushes to "view" */
    public void generateBody(StringTemplate bodyST) {
        User[] list = users; // normally pull from database
        // filter list if you want here (not in template)
        bodyST.SetAttribute("users", list);
    }

    public string getTitle() { return "User List"; }
}

```

Notice that the controller and model have no HTML in them at all and that the template has no code with side-effects or logic that can break the model-view separation. If you wanted to only see users with age < 30, you would filter the list in `{{generateBody()}}` rather than alter your template. The template only displays information once the controller pulls the right data from the model.

Pushing factorization further, you could make a `row.st` component in order to reuse the table row HTML:

```
<tr>
  <td>${it.name}</td><td>${it.age}</td>
</tr>
```

Then the user list template reduces to the more readable:

```
<table border=1>
$users:row()$
</table>
```

Naturally, you could go one step further and make another component for the entire table (putting it in file `table.st`):

```
<table border=1>
$elements:row()$
</table>
```

then the body template would simply be:

```
$table(elements=users)$
```

Internationalization and localization

StringTemplate provides a simple and effective method for localizing web pages. The goal is to alter a page based upon the locale; that is, page strings or other content must change depending on a locale. This article not only illustrates how to make a pages change text depending on locale, it shows how the same site may easily have two different *skins* (site "looks").

This technique works well in practice for real sites. Schoolloop.com is a case in point. Click on the link that says "*en espanol*" to flip the site into Spanish mode. The exact same templates are used; all strings are pulled from a series of resource bundles. There is no duplication of pages to change the strings.

Multiple skins

First let's look at multiple skins in order to show how templates are loaded for this example.

Multiple site looks are organized into their own directories. A `StringTemplateGroup` object rooted at that directory will load templates directly from there. In my example, I made two skins, blue and red. Here is how the group is loaded:

```
string skin="blue";
// get a template group rooted at appropriate skin
// we are using a path relative to our application assembly's location
string absoluteSkinRootDirectoryName = Path.Combine(new
DirectoryInfo(AppDomain.CurrentDomain.BaseDirectory).Parent.FullName, skin);
StringTemplateGroup templates = new StringTemplateGroup("test", absoluteSkinRootDirectoryName);
```

Then, when you ask for an instance of a page, it pulls from whichever directory `skin` is set to:

```
StringTemplate page1ST = templates.GetInstanceOf("page1");
```

Here is page 1 in the blue skin:

```
<html>
<title>${strings.page1_title}</title>
<body>
<font color=blue>
<p>${strings.intro$

<p>${strings.page1_content$
</font>
</body>
</html>
```

and here is page 2:

```
<html>
<title>${strings.page2_title}</title>
<body>
<font color=blue>
<p>${strings.page2_content$
</font>
</body>
</html>
```

For the red here is page 1:

```
<html>
<title>${strings.page1_title}</title>
<body>
<font color=red>
<h1>${strings.page1_title}</h1>

<p>${strings.intro$
<hr>
<p>${strings.page1_content$
</font>
</body>
</html>
```

and page 2:

```
<html>
<title>${strings.page2_title}</title>
<body>
<font color=red>
<h1>${strings.page2_title}</h1>
<hr>
<p>${strings.page2_content$
</font>
</body>
</html>
```

The thing to note is that there is no text, just formatting in these page templates. Those strings from the `strings` attribute are used for all text that could change per locale.

Localizing template strings

Once the code knows how to load templates, the locale must dictate which strings are displayed. The templates clearly have attribute references pulling from an object instance labelled `strings` (it could be a hash table). For this example, I'm assuming that I have a few different strings and that I'm storing them in resource files called `Content.Strings.resx`. The language specific alternatives are named `Content.Strings[language-code].resx` where **language-code** is the two-letter language code. Here are the name/value entries stored in `Content.Strings.resx`:

```
intro=Welcome to my test page for internationalization with StringTemplate
page1_title=Page 1 testing I18N
page2_title=Page 2 testing I18N
page1_content=This is page 1's simple page content
page2_content=This is page 2's simple page content
```

and here are the entries in the `Content.Strings.fr.resx` file:

```
intro=Bienvenue sur la page de test d'internationalisation avec StringTemplate
page1_title=Page 1 test de I18N
page2_title=Page 2 test de I18N
page1_content=Le contenu de la page 1
page2_content=Le contenu de la page 2
```

To load these per the current locale is pretty easy:

```
// allow them to override language from argument on command line
String language = defaultLanguage;
if ( args.length>0 ) {
    language = args[0];
}
Thread.CurrentThread.CurrentUICulture = new CultureInfo(language);
ResourceManager resMgr = new ResourceManager("ST.Examples.i18n.Content.Strings",
    typeof(ResourceWrapper).Assembly);
ResourceWrapper strings = new ResourceWrapper(resMgr);
```

The `strings` properties object is just a wrapper around the `ResourceManager` so you can directly pass to `StringTemplate` templates as an attribute:

Here's are the relevant bits of the wrapper class:

```
class ResourceWrapper
{
    ResourceManager mgr;
    public ResourceWrapper(ResourceManager mgr) { this.mgr = mgr; }

    public string intro          { get { return mgr.GetString("intro"); } }
    .....
    .....
    public string page2_content  { get { return mgr.GetString("page2_content"); } }
}
```

And here's how this is passed directly to the `StringTemplate`:

```
StringTemplate page1ST = templates.GetInstanceOf("page1");
page1ST.SetAttribute("strings", strings);
```

To generate the page, just say:

```
Console.Out.WriteLine(page1ST);
```

The output will be (for the en locale):

```
<html>
<title>Page 1 testing I18N</title>
<body>
<font color=blue>
<p>Welcome to my test page for internationalization with
StringTemplate

<p>This is page 1's simple page content
</font>
</body>
</html>
```

If I change the locale to fr then without changes templates, the following is generated:

```
<html>
<title>Page 1 test de I18N</title>
<body>
<font color=blue>
<p>

<p>Le contenu de la page 1
</font>
</body>
</html>
```

Source and compilation

Here is the C# code, different strings, and different skins:

- [VS.NET project file](#)
- [Test.cs](#)
- [English strings resource file](#)
- [French strings resource file](#)
- [The entire project in a zip archive](#)

You naturally need ANTLR too; [get it here](#).

You can compile this example like this:

```
nant -t:net-1.1
```

Summary

In summary, the key element to demonstrate here is that you do not have to duplicate all of your templates to change what they say. You can leave the formatting alone and, with a simple hashtable pulled from a data file, push in the proper strings per the locale. I also took the opportunity to show off just how easy it is to make multiple site skins.

StringTemplate Grammar

`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.

A Common Syntax Question

Why have a template like:

<<

Check this faq entry: `$link(url="/faq/view?ID="+id, title="A FAQ")$`.

>>


instead of


<<

Check this faq entry: `$link(url="/faq/view?ID="+id, title="A FAQ")$`.

>>

using `id` instead of `id` for that attribute reference? The simplest answer is that you are already within the scope of an attribute expression between the `$...$` and hence `StringTemplate` knows that `id` must be a reference to an attribute by the grammar alone. The template delimiters mark the beginning and end of what `StringTemplate` cares about.

Another way to look at it is the following. Surely syntax  makes the most sense:

 `SELECT $col+blort$ FROM $table$`

(ii) `SELECT $$col+$blort$$ FROM $table$`

(iii) `SELECT $col+$blort$ FROM $table$`

Syntax (ii) is rather verbose and redundant wouldn't you say? Syntax (iii) doesn't even work because the `+` is outside the realm of `StringTemplate` delimiters.

Acknowledgements

Tom Burns (CEO jGuru.com) co-designed `StringTemplate` and listened to me think out loud incessantly. I would also like to thank Monty Zukowski for planting the `template` "meme" in my head back in the mid 1990's. Loring Craymer and Monty both helped me hone these ideas for use in source-to-source language translation. Matthew Ford has done a huge amount of thinking about `StringTemplate` and has submitted numerous suggestions and patches. Anthony Casalena at <http://www.squarespace.com> has been a big help beta-testing `StringTemplate`. Thanks to the users who have submitted suggestions, bugs, sample code, etc...