StringTemplate 4 Wiki Home

Welcome to the StringTemplate 4 wiki. This wiki complements and supports the StringTemplate website. Inside you will find the StringTemplate documentation, tutorials and a FAQ. StringTemplate (ST) v4 is a complete rewrite of ST v3, again released under the BSD license. v4 is about twice as fast and uses less memory than v3.

Download source or binary

We you find what you need to help learn about StringTemplate. If not, tell us about it on the mailing list.

Bug reports can be found at StringTemplate 4's JIRA

Unknown macro: 'twitter'

parrt blog posts

Blog Posts

Matching parse tree patterns, paths created by Terence Parr [Administrator] Terence Parr Sep 01, 2013

The real story on null vs empty created by Terence Parr [Administrator] Terence Parr Dec 27, 2012

Tree rewriting in ANTLR v4 created by Terence Parr [Administrator] Dec 08, 2012 Terence Parr

ANTLR and Shroedinger's Tokens created by Terence Parr [Administrator] Terence Parr Jun 08, 2012

problem: antlr not generating .java files for .g file created by Unknown User (10msitfkarim) ANTLR 3 May 09, 2012

Recently Updated

Motivation and philosophy
Mar 11, 2014 • updated by Terence Parr [Administrator] • view change

Model adaptors
Feb 02, 2014 • updated by Mike Cargal • view change

Motivation and philosophy
Nov 09, 2013 • updated by Anonymous • view change

Group file syntax
Jul 10, 2013 • updated by Terence Parr [Administrator] • view change

4.0.7 Release Notes
Jan 05, 2013 • created by Terence Parr [Administrator]

StringTemplate 4 Documentation
Dec 27, 2012 • updated by Terence Parr [Administrator] • view change

Templates
Dec 27, 2012 • updated by Terence Parr [Administrator] • view change

ST4
Nov 25, 2012 • attached by Terence Parr [Administrator]

4.0.6 Release Notes

Sep 26, 2012 • created by Terence Parr [Administrator]

Introduction
Aug 06, 2012 • updated by Terence Parr [Administrator] • view change

Altering property lookup for Scala
Jun 20, 2012 • updated by Terence Parr [Administrator] • view change

FAQ - Object models
Jun 19, 2012 • created by Terence Parr [Administrator]

Differences between v3 and v4
May 22, 2012 • commented by Anonymous

Differences between v3 and v4
May 21, 2012 • commented by Terence Parr [Administrator]

Model adaptors
Apr 23, 2012 • updated by Anonymous • view change

Wiki Contents

ST credit block

Java reference implementation Terence Parr University of San Francisco `parrt AT cs.usfca.edu`	Python port Benjamin Niemann `pink AT odahoda DOT de`	C# port Sam Harwell	Objective-C Alan Condit

Release Notes

4.0 Release Notes

BIG THANKS to Sam Harwell and, again, to Udo Borkowski for debugging help and suggestions. Sam is doing the C# implementation. Benjamin Niemann is doing the Python port. Alan Condit is doing Objective-C.

4.0 – March 27, 2011

- ST.add() returns self now so we can chain. t.add("x", 1).add("y", "hi");
- import from files in jar didn't work.
- removed field tokens from STGroupString
- improved imports lookup
- made fields of events public final.
- augmented debug event toString and fixed start/stop issue with eval events. renamed fields to be more clear.
- Added IndentEvent for dbg
- ^(INDENT expr-sub-tree) is now ^(INDENTED_EXPR INDENT expr-sub-tree) with changes to grammars. More consistent with subtree root being operator. STViz now highlights indentation properly in template pane.
- Altered CodeGenerator.g to pass AST node for indentation not just string. This way we get INDENT operations into the sourceMap for indent debug events.

4.0b5 – March 6, 2011

- true/false were only allowed as default args; now allowed as template arg expressions in templates. Works as dictionary value too.
- couldn't have anonymous templates inside a region.
- parentheses were a bit weird in conditions. Now, conditions cannot use parentheses to mean "early evaluation" except as obj.(propName)
- nativeGroup of all implicit templates was STGroup.defaultGroup.
- removed all writes of the enclosingInstance at evaluation time; fixed issue for STViz.
- comments on line by themselves don't emit \n to output
- STViz tried to highlight AST pane even when we switched ASTs
- combined load_str, write into write_str single op. minor optimizations too. Seems a tiny bit faster per benchmarks.
- Added ST.VERSION auto-updated by ANT.
- added STGroupString
- Added support for this in group.g:

```
oldStyleHeader // ignore but lets us use this parser in AW for both v3 and v4
  :   'group' ID ( ':' ID )? ( 'implements' ID (',' ID)\* )? ';'
  ;
```

- IndexOutOfBounds Exception when using "cap" format on empty string
- @t.() ::= "" caused NPE
- Region redefinition caused NPE. "<@r>a<@end><@r()>"
- STViz couldn't see first subtemplate when computing template range in output.
- Was incorrectly computing filename to load template .st files from group dir.
- Listener was not notified upon "no such template" in group dir.
- Redid how ST found imported files, dir, etc.. Can now import a template file even. Can be absolute path or relative path. If relative, it looks in dir of .stg file with import then CLASSPATH.
- The listener of import groups is now set to that of group that imports them.
- Regions behave like <if> tags now. Indent respected if <@r>...<@end> on indented single line. Indent/newlines ignored after those tags if on separate lines.

4.0b4 – February 5, 2011

BIG THANKS to Udo Borkowski for his help debugging these betas and his suggestions.

- added write to file methods

- had infinite loop for expr: "<t()$"
- the default file encoding is now UTF-8
- early eval <(x) > using new STWriter derived from type of current STWriter. e.g., AutoIndentWriter.
- didn't detect nonterminated comment. <!bad comment>
- added two literals "true" and "false" to the template argument syntax; e.g., stat(name,x=true,y=false) ::= "..."
- it was treating "..." default arg as a template not string.
- throws STException now upon not finding group file or group dir instead of sending err to listener.
- default args couldn't have subtemplates t(x,y={<x:{s|<s>}>}) ::= "..."
- Added a new benchmark from Oliver Zeigermann. Discovered 60% of time is spent using reflection invoke() for properties.

4.0b3 – January 28, 2011

- exception in lexer blew out of parsing
- missing '}' in {...} caused infinite loop
- NPE in storeArgs if empty arg list
- removed debugging prints.
- x={<(...)>} default arg was hardcoded to <...> not $..$ or whatever.
- The grammar needed to match and ignore an optional INDENT before region @end
- when redefining a region (template) the newline before the >> was kept.
- WS not ignored in front of STRING token in expressions.
- closing STViz doesn't exit vm now.
- throws exception if registering renderer or model adaptor for primitive

4.0b2 – January 22, 2011

- Order of static init issue; an error mgr was null.
- Fixed some unit testing the Windows friendly
- Fix bug in triple if-elseif-elseif-elseif; added unit tests
- bug where I did not say current_ip when calling exec() from writeObject
- Updated README to include install information

4.0b1 – January 14, 2011

4.0.1 Release Notes

This primarily a bug fix release, but also includes some new features. This is the release that allowed me to incorporate StringTemplate v4 into ANTLR v3.

Download ST v4 here

New feature list

- <% ... %> templates that ignore indentation and newlines.
- Improved StringTemplate Inspector GUI
- "..." pass-through operator sets argument x of an included template to be the value of x in the enclosing template.
- Added STGroup.iterateAcrossValues static boolean for v3 compatibility v3 iterates across values not keys like v4. But to convert ANTLR templates, it's too hard to find without static typing in templates.
- improve some error messages
- Added <aggr.{prop1, prop2}> notation for injecting attributes (see Introduction#inject_aggr) in ST.add(). E.g.,

```
st.add("users.{name,id}", "Ter", "parrt");
```

April 10, 2011

- Added STNoSuchAttributeException to distinguish from no such property.
- Pass through didn't handle case properly of empty or nonexistent attributes. We only pass through nonempty values. Further, it makes no sense to set values for parameter x if x has no definition above. That is the same as having no value. This is required to get default attributes to work with passthru in all cases. Added unit tests.
- Oops. That is not quite correct. If no value exists or the attribute itself does not exist, we must set the parameter to null ifthere is no default parameter. otherwise the interpreter will complain about a missing argument value.
- Was missing alreadyLoaded = true; in STGroupString.

- Bug fix for <@super.r()> exprs; it tried to doubly-define region.
  Code generator didn't generate mangled name either; fixed and then updated
  http://www.antlr.org/wiki/display/ST4/Template+to+Bytecode+mapping
- Improved error msg when referencing implicit attributes like 'i'.

April 9, 2011

- Sam pointed out that regions didn't work with <% %>
- Cleaned up STViz, the StringTemplate Inspector GUI.
- STViz expands template attribute view when you click in output
- STViz template and attribute tree views now grouped.
- default arguments were not evaluated in context of invoked template;
  couldn't see other args.
- Added '...' pass through arg back in. Only allowed with named arg lists
  or as sole arg. Not allowed in <(name)()> indirect includes.
  Inserts new passthru bytecode to set any unset args.
- AST pane not updated upon new ST selection
- Scroll output pane when ST selected
- Added STGroup.iterateAcrossValues static boolean for v3 compatibility
  v3 iterates across values not keys like v4. But to convert ANTLR templates,
  it's too hard to find without static typing in templates.
- Cleaned up ST.locals[] creation; centralized in Interpreter.
- MapModelAdaptor made copy of STs unnecessarily and w/o copying locals[]

April 8, 2011

- Added scope tree showing all inherited attributes (dynamic scoping) in lower
  left "attributes" pane.
- Undid some bugs I introduced concerning selecting templates. Tried to
  move the cursor in output window which triggered multiple / wrong update
  events.

April 6, 2011

- Ignore indentation in <% .. %> templates but keep white space between elements
- Added better error msg for internal errors during template evaluation.
  context [outputFile parser genericParser rule ruleBlockSingleAlt alt element
  ruleRefAndListLabel ruleRef] 1:1 internal error caused by: java.lang.NullPointerException
  at org.stringtemplate.v4.ST.rawSetAttribute(ST.java:294)
  at org.stringtemplate.v4.Interpreter.storeArgs(Interpreter.java:576)
  at org.stringtemplate.v4.Interpreter.super_new(Interpreter.java:495)
  ...

April 4, 2011

- in TestSubtemplates.java there was an extra (bad) import
- fixed a couple of unit tests that failed.
- removed .class files from depot
- added t() ::= <% ... %> template that ignores all newlines inside template.
  This allows arbitrary formatting within a template that does not result
  in new lines in the output. This is useful when you have a really complicated
  template with IFs and such that needs to generate output all on the same line.
  Currently, this can be quite challenging. There's no way to read a huge
  template on one line.

April 3, 2011

- ST.getAttribute() only looks in that template now. Can't look up
  since it doesn't know what interp is executing. It's just to get
  an attr out of a template now. Moved dynamically scope getAttribute()
  to Interpreter.
- STRuntimeMessage takes an Interpreter interp arg now.
- ST dropped some weight. No need for enclosingInstance ptr now. That is
  properly done in interpreter as a stack of scopes. Now, there is no
  side-effect whatsoever in ST instances for execution. THREAD SAFE eval now.
- Interpreter interp added to ModelAdaptor. BREAKING CHANGE IF YOU'VE BUILT
  a model adaptor (rare)
- Internal clean up so stack of template evaluation scopes has debug info.
  Required changes across lots of files. Started referring to scopes
  so entire path to root is available and with debugging info if debug on
  for that interpreter. Extracted InstanceScope from inner class.

March 30 - April 2, 2011

- ST.inspect() now returns an STViz, which has all the goodies and gives

you access to the GUI stuff.
- added aggr.{prop1, prop2} for ST.add(). Too useful. (Was in v3). Use ST.addAggr("aggr.{p1,p2}", a1, a2);
- refactored to fold DebugST into ST; adds one object ptr to every ST instance but worth reduction in complexity. "new ST(...)" calls didn't work (not DebugST objects) in inspector. ST.inspect() for any ST now.
- Fixed bug in STViz. Didn't highlight entire output when you click topmost template.
- STGroup.debug no longer there nor static. It's an instance var of Interpreter. ST.inspect() tells interp to debug. STGroup.trackCreationEvents says to record where in code an ST was created and where code added attributes.
- Gutted tree model for STViz, refactored debugging/event tracking code.
- creation events had wrong location (launch of interp location); only tracks now for externally/injected created templates.

March 29, 2011

- Fixed bug where escaped quotes in template defs were not unescaped for use by compiler.

4.0.2 Release Notes

This is a bug fix release. Download ST v4 here

4.0.2 – May 3, 2011

- Backing out change from 4/17; don't want Serializable implementation.
- Improved error msg for out of order required parameters (after optional ones)

April 26, 2011

- rest() stripped nulls, which it shouldn't. Was inconsistent with trunc(), etc...

April 21, 2011

- Made STGroup.iterateAcrossValues an instance variable not static. That needed a change to convertAnythingIteratableToIterator, etc.. to non-static.
- Removed STDump as unneeded; Use STViz.

April 17, 2011

- Added implements Serializable to ST, STGroup.

April 16, 2011

- Made compatible with Java 1.5; removed Arrays.copyOf() ref.
- Updated ANT build to ref ant lib dir not /usr/local/lib/

April 11, 2011

- Dictionaries weren't inherited. Added unit test.

4.0.3 Release Notes

June 21, 2011

This is a bug fix release, many done by Udo Borkowski. Download ST v4 here

- Major overhaul of template names:
  - '/' allowed as starting ID letter like </a/b()>
  - getInstanceOf names must be fully qualified. If you don't put / on front, one is added for you.
  - template refs in expr are relative to location of surrounding template unless prefixed with /. In that case they are relative to root of group.
  - import statement no longer allows fully qualified file name.
  - Changed all unit tests to use fully qualified names and see results that way.
  - *Also note that import statement no longer interprets fully qualified path to location on disk. A fully qualified path is now interpreted as relative to group root to be consistent.*
- {} wasn't allowed as a template
- STGroup.unload() calls unload() on each group in the imports list instead of clearing the list. (Thanks to Sam...wait, did Udo already try this?)
- STRuntimeMessage got NPE upon ST.impl == null
- ctor ST() is protected; not for users. bad users!

- Removed warning (access static member through instance)
- Fixed and added tests
- Fixed test case for <\n> to handle different line.separator sizes
- BUG: On Windows wrapped lines are separated with \r\r\n
- made tests run on Windows and non-US locales
- STGroupDir.load(String name) no longer checks for (parent) group file when name specifies no parent (no '/')
- unload in STGroup now also unloads the import relationships
- Fixed test testRendererWithPredefinedFormat2 to also work in non-PDT timezones
- Fixed tests testArg1, testArg2 in TestGroupSyntaxErrors
- Fixed "URI is not hierarchical" issue when STGroupFile is imported from jar file
- Added getTemplateNames to STGroup
- passthru() didn't watch for empty formal args
- fixed bug raising a NullPointerException when a formalArg's default value has a syntax error.
  Example: main(a={(<"")>}) ::= ""
- STGroupFile.getName() returns group name also for imported groups (was null before).

4.0.4 Release Notes

July 18, 2011

This is a bug fix release with a new "delimiters" group file feature. Download ST v4 here

New feature:

- Added delimiters "<", ">" notation to group file.

Bug fixes:

- added "get import list" method.
- added methods to allow deep vs shallow setting of renderers; interp always
  asks native group defining template for the renderer.
- STGroup.getInstanceOf was not auto-adding "/" to front if none was present
- Updated javadoc on getAttributeRenderer()
- Updated javadoc on unload/importTemplates in STGroup() (ub)
- Added test case for unload of groups specified in group file imports. (ub)
- STGroup.unload() now removes imports that were specified in the group
  file, but only calls unload() on templates that were explicitly added
  in the program. Resolves both Sam's and Udo's concerns. 🙂
- subtemplates {...} in subdirectories didn't work.

4.0.5 Release Notes

February 8, 2012

This is a bug fix release with a new STRawGroupDir class. Download ST v4 here

New feature:

- Added STRawGroupDir that expects pure templates in .st files not template defs
  with headers as is usually the case. So use $name$ not

  ```
  foo(name) ::= "$name"
  ```

  This makes it much easier to use raw HTML files, for example, with ST.

Bug fixes:

- STLexer.tokens was missing from src build
- synchronized object model adaptor
- import is now considered illegal in group files embedded in STGroupDirs; they lead to trouble and make no sense.
- Fixed Misc.newline issue in test code (Udo)

4.0.6 Release Notes

September 26, 2012

This is a bug fix release. (Download ST v4 here)

- STRawGroupDir problem and ST("template") issue. When there are no formal args for template t and you map t across some values, t implicitly gets arg "it". E.g., "$names:bold()$" and bold as "it".
- Fixed https://github.com/antlr/stringtemplate4/issues/5

- Made fields in the error messages public

4.0.7 Release Notes

Improvements

- Improved error location reporting
- Allow [] as a dictionary value, resolves antlr/stringtemplate4#33
- Several STViz updates:
    - Highlight template subexpressions and literal text responsible for output
    - Gray out hidden (inherited+aliased) attributes
    - Highlight user-instanced templates in bold

Bug fixes

- Escapes: >\> means >> inside of <<...>>.
- Escapes: \>> means >> inside of <<...>> unless at end like <<...\>>>>.
  In that case, use <%..>>%> instead.
- Added warning about: "Missing newline after newline escape <\\>"
- %\> is the escape to avoid end of string
- Fix issues with bytecode to source mapping
- Fix several STViz bugs
- Fix several unit tests
- Explicit InstanceScope tracking in the interpreter
- throw exceptionWhen the attribute name is no to be consistent with the
- other check for '.' in the name.
- Allow [] as a default value for formal arguments (fixes antlr/stringtemplate4#20)
- Add method STViz.waitForClose()
- Specify -Dtest.interactive to have STViz tests leave the window open for the user
- Don't cache the STNoSuchPropertyException and STNoSuchAttributeException instances.
- Add ErrorType.NO_SUCH_ATTRIBUTE_PASS_THROUGH, reported on <foo(...)> where foo contains a parameter with no default value and no matching attribute exists in the surrounding scope.
- Improved message when reporting ErrorType.NO_SUCH_PROPERTY.
- DateRenderer and StringRenderer now use the provided locale (fixes antlr/stringtemplate4#11)
- Fixes for handling of arrays (fixes antlr/stringtemplate4#12 and other unreported issues)
- Try to load template file (.st) if group file (.stg) failed with IOException (fixes antlr/stringtemplate4#14)
- Add STGroup.GROUP_FILE_EXTENSION and STGroup.TEMPLATE_FILE_EXTENSION
- Updated documentation, code cleanup

StringTemplate 4 Documentation

See Release Notes and Bugs. Looking for StringTemplate 3 Documentation? v3 API?

Installation

- Java
- C#
- Python
- Objective-C

Introductory material

- Introduction
- Ports: Java (reference implementation), C#, Python, Objective-C
- StringTemplate cheat sheet
- Motivation and philosophy

Groups

- Group file syntax
- Group inheritance
- Template regions

Templates

- Literals
- Expressions
- Template includes
- Expression options
- Conditionals

- Anonymous templates
- Map operations
- Functions
- Lazy evaluation
- Missing and null attribute evaluation

Whitespace and formatting

- Auto-indentation
- Automatic line wrapping

Customizing StringTemplate behavior

- Error listeners
- Renderers
- Model adaptors

Debugging

- StringTemplate Inspector GUI

Implementation

- Template to Bytecode mapping

Differences between v3 and v4

Speed and memory

ST v4 seems to be about 2x faster than v3 when there's no memory pressure / GC thrashing. With memory constraints, ST v4 is much faster because it seems to use less memory. This data comes from upgrading ANTLR to use ST v4.

Migrating to v4 from v3

In general, migrating to v4 is straightforward. The biggest difference is that the names of the types have changed (I got tired of typing StringTemplateGroup all the way out 😀) as well as the package. Instead of base package org.stringtemplate, we use org.stringtemplate.v4.

- StringTemplate -> ST
- StringTemplateGroup -> STGroup, STGroupFile, or STGroupDir. References to `new StringTemplateGroup(directory)` should become `new STGroupDir(directory)`. References to `new StringTemplateGroup(new FileReader("file.stg"))` should become `new STGroupFile("file.stg")`.
- We pass in delimiter start and stop characters to the constructor of groups rather than passing in AngleBracketTemplateLexer.class. The default delimiters are angle brackets `<...>` instead of dollar signs `$...$`.
- setAttribute() -> add(). This makes it clear that we are adding rather than resetting attribute values. The semantics of the same except that we can add null values.
- There is no group header in group files but they're ignored for backward compatibility. Instead, use (possibly multiple) `import "filename.stg"` or `import "dirname"` or `import "template.st"`.
- All explicit and implicit references to ST.toString() become calls to render(). It was difficult to debug templates because the debugger keeps calling toString to display objects. Also, it's better to be explicit that we are rendering a template string. There are a lot of implicit conversions of templates to strings. For example,

  ```
  System.out.println(st);
  ```

  should be

  ```
  System.out.println(st.render());
  ```

- All templates require formal arguments unless you use constructor `new ST("a template")`. So, a template file looks like

```
bracket(x) ::= "[<x>]"
```

instead of just `[<x>]`.

- The syntax inside templates is the same except that there is no need for `...` argument that said ``pass through outer arguments.''
- There is no implicit `it` or `attr` attribute during generation now. convert `<names:{<it>}>` to `<names:{n | <n>}>`. If you use `<names:foo()>` and foo looks like:

```
foo() ::= "<it>"
```

then convert it

```
foo(x) ::= "<x>"
```

ST will automatically pass in iterated value as the first argument (as it would do in v3).

- The functionality of existing templates should be the same except that iterating across a Map is on the keys not values for v4. In v3, it iterated across the values. We had to use <aMap.keys:{...}> to iterate across the keys. This is the only change that could be painful because of the dynamic typing nature of ST. There's no static tool to find these occurrences.
- The only other thing is that you might notice some differences in the whitespace handling.

New Functionality

- STViz template visualizer; setting STGroup.debug creates DebugST objects not ST and tracks which Java code sets which attributes and/or creates templates. You need to set the debug flag in order to use the visualizer.
- multiple template group inheritance using `import`
- Added ModelAdaptors, an object that knows how to convert property references to appropriate actions on a model object. Given <a.foo>, we look up foo via the M adaptor if "a instanceof(M)".
- For <if(...)> conditionals, you can use && || conditional operators instead of nested IF statements
- `strlen`, `trim`, and `reverse` functions for lists
- predefined DateRenderer and StringRenderer (understands formats upper, lower, cap, url-encode, xml-encode) objects.
- v4 has <%...%> template definitions that are like <<..>> except that they ignore all new lines. This is very convenient for complicated templates that need to be broken up for easy viewing but that shouldn't have newlines in the output.
- Added ST.format methods for use in general Java code.

```
int[] num =
    new int[] {3,9,20,2,1,4,6,32,5,6,77,888,2,1,6,32,5,6,77,
        4,9,20,2,1,4,63,9,20,2,1,4,6,32,5,6,77,6,32,5,6,77,
        3,9,20,2,1,4,6,32,5,6,77,888,1,6,32,5};
String t =
    ST.format(30, "int <%1>[] = { <%2; wrap, anchor, separator=\", \"> };", "a",
num);
System.out.println(t);
```

Yields:

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

- \n in a template becomes \r\n or \n in output depending on platform.
  We can always use \n in templates. To force \r\n on unix, though, we can still
  create a writer:

```
st.write(new AutoIndentWriter(sw,"\r\n")); // force \r\n as newline
```

- renderers know locale optionally; only one method in renderer now with all args
- The default file encoding for templates and template groups is UTF-8.

Differences

Major

- template arguments can be named or passed by position at the call site as we would in Java. For example, given a template t with two arguments, we can use<t("a","b")> to pass two strings instead of <t(first="a", last="b")>. In v3, you would get into weird situations where you would have to do <t(a=a, b=b)>. now we can say <t(a,b)>.
- Map iteration is on keys not values for v4
- no "it" iteration value
- groups:
  - Unify groups; groups are just template files cat'd together
  - STGroup not StringTemplateGroup, STGroupDir, STGroupFile
  - no group header in group files but multiple import "filename.stg" or import "dirname"
- render() not toString()
- v4 does not support template interfaces

Minor

- We can't pass arguments to the templates in map operations like this: `<names:foo("hi")>`.
- default args can see other args: `t(x,y={<x:{s|<s><z>}>},z="foo") ::= <<...>>`
- allows hasXXX property method in addition to the getXXX and isXXX.
- <a.b> for missing b is no longer an error
- templates do not have listeners; only groups have listeners
- renderers are per group and the group the interpreter feeds off of native group. Renderers in v4 now operate on class subtrees instead of exact types. Renderer associated with type t works for object o if

```
t.isAssignableFrom(o.getClass()) // would assignment t = o work?
```

  So it works if o is subclass or implements t.
- Can only have 65535 char in any one template (instr operands are shorts and write instruction refs back into source template pattern).
- allows <{...}> so we can add strip/filter/format option to ignore WS
- reset() doesn't exist anymore. Getting an instance of a template provides a "resetted" one.
- missing vs empty; null in list and false-IF are like missing; empty is "" or <else><endif> clause
- null values for attr allowed; st.add("name", null). same as missing.
  it's added to list if we add other values afterwards. same
  now as sending in list of null

- arg "..." is a string (noninterpreted) and {...} is an anon template that allows full nesting etc...
- can't do separator=names:{..} (no map ops in option expr)
  but can do optionname={...}
- i, i0 don't propagate in dyn scoping
- setting attr i or i0 has no effect; they go 1..n 0..n-1 anyway
- <...> by default
- ".." means string literal, {...} and <<...>> are templates; e.g., in map defs
- change doc to say dict not map for dicts in group files.
- for dictionaries: render any key to a string for lookup; m.(key)
- trim just one WS char from start of multi-line templates, not all whitespace; none at end; no longer needed since we use formal args in .st files now. "..." don't get anything stripped.
  indentation of IF stuff is ignored as is newline on end.

```
[
    <if(x)>
    foo
    <else>
    bar
    <endif>
]
```

  it did two indents in STv3. result now is

```
[
    foo
]
```

- <multi-char> not allowed. do <\r><\n> instead. no <\r>. <\n> does locale specific output
- groups have no name other than dir name or group file name
- \n stripped if no output on a line and literal or <...> including IF

EMPHATICALLY SAME:

- null elements get ignored. null list elements do not get counted for <i> and <i0> indexes.
- trim whitespace at start/end of templates; just too hard to get rid of whitespace later.

thread safety

assume thread safe modifying access to templates, adaptors, attributes, imports, and renderer access on STGroup/ST (they are synchronized).

Template to Bytecode mapping

Expressions

expression	bytecode
<expr>	*expr* write
<(expr)>	*expr* tostr write
<expr; o1=e1, o2=e2>	*expr* options *e1* store_option o1-option-index *e2* store_option o2-option-index write_opt
text	load_str string-pool-index

true	true
false	false
a	load_local attribute-index ; if a is template argument
i	load_local attribute-index
i0	load_local attribute-index
a	load_attr a-string-pool-index
a.b	load_attr a ; from now now, a means its string index load_prop b
a.(b)	load_attr a load_attr b load_prop_ind
t()	new t,0 ; string pool index of t
super.r()	super_new region_t_r,0 ; region r in template t
t(e1,e2,e3)	*e1* *e2* *e3* new t,3
t(...)	args passthru t new_box_args t
t(a1=e1,a2=e2,a3=e3)	args *e1* store_arg a1 *e2* store_arg a2 *e3* store_arg a3 new_box_args t
t(a1=e1,a2=e2,...)	args *e1* store_arg a1 *e2* store_arg a2 passthru t new_box_args t
(expr)(args)	*expr* tostr *args* new_ind num-args
a:t()	load_attr a null new t,1 map
a:t(x)	load_attr a null x new t,2 map
a:t(),u()	load_attr a null new t,1 null new u,1 rot_map 2

a,b:t()	load_attr a load_attr b null null new t,2 zip_map 2
first(expr)	*expr* first ; predefined function
[a,b,c]	list *a* add *b* add *c* add

Anonymous templates

expression	bytecode
{t}	new _subN,0
a:{x | ...}	load_attr a null new _subN, 1 map
a,b:{x,y | ...}	load_attr a load_attr b null null new _subN,2 zip_map 2

If statements

upon if, create 'end' label.
upon else, create 'else' label.

statement	bytecode
<if(a)>t<endif>	load_attr a brf end *t* write end:
<if(a)>t<else>u<endif>	load_attr a brf else *t* write br end else: *u* write end:

<if(a)>t<elseif(b)>u<else>v<endif>	load_attr a brf lab1 *t* write br end lab1: load_attr b brf lab2 *u* write br end lab2: *v* write end:
<if(!a)>t<endif>	load_attr a not brf end *t* write end:
a||b	load_attr a load_attr b or
a&&b	load_attr a load_attr b and

Auto-indentation

expr	bytecode
<expr>\n	*expr* write newline
\n\t<expr>	newline indent "\t" *expr* write dedent

Size limitations

I use unsigned shorts not ints for the bytecode operands and addresses. This limits size of templates but not the output size. In single template, you can have only 64k of

- attributes
- unique property name refs
- unique template name refs
- options (there are only about 5 now)
- lists or template names in a map/iteration operation
- bytecodes (short addressed)
- chunks of text outside of expressions. effectively same thing as saying can have at most 64k / n expressions where n is avg size of bytecode to emit an expression. E.g., 3 bytes to write a chunk of text.

StringTemplate Inspector GUI

assumes ST 4.0.1

One of the frustrations with ST v3 was that it was sometimes hard to debug the templates. For v4, everything is a lot cleaner and I can present for you a nice GUI that tells you everything about a template and any nested templates. There are some key bits of information you need when debugging a template:

- What template fragment emitted this output?
- What did this template fragment emit?
- What are the attributes associated with this template instance?
- What attributes are visible to a particular template?

- Where in the source code did I set an attribute?
- Where in the source code did I create this template?
- How is ST interpreting the syntax of my template?

The GUI frame answers these questions and provides extra information including the generated bytecodes compiled from the original template as well as an execution trace through the interpreter.

How to launch the inspector

To use the GUI inspector, you don't need to do anything except call `ST.inspect()` on the root of your template hierarchy. If you also want to track template creation events and attribute injection events, turn on the static `STGroup.trackCreationEvents` flag. The interpretation of templates is totally side effect free so you can render and inspect templates as you wish and in any order.

```
STGroup.trackCreationEvents = true;
STGroup group = new STGroupFile("t.stg");
ST st = group.getInstanceOf("test");
st.add("attrname", value);
...
st.inspect();
```

Inspector window components

Here's basic view when you start up:

- tree of templates is the upper left pane
- attributes of the templates stack (path from selected template to the root of the total anarchy) are in the lower left pane
- error messages from the compiler or interpreter go in the bottommost pane
- the output is in the upper right pane
- one of three things is in the lower right pane:
    - the template source code and the AST built by the ST compiler
    - the byte codes compiled from that source
    - the execution trace of the byte code interpreter.

Clicking template in hierarchy shows related output

Clicking on one of the templates in the upper left quadrant resets the display so that the output from that template instance is highlighted in the output pane. The stack of templates in the lower left pane also changes. Opening up the template names in that pane shows the attributes injected into that template.

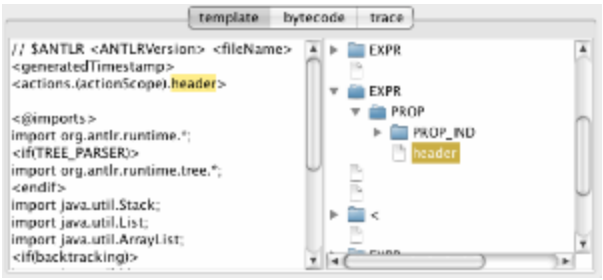
Corresponding Java source code locations

Also notice that the location in the source code where the template was created to show and also the place in the source where the attribute was added is shown. The main outputFile template was created at location CodeGenerator.java line 296. In the attributes pane you can see that rewriteMode was injected at line 343, and so on. If there is no location, that means that the template was automatically created by StringTemplate internally or the attribute was not set.

Clicking in output window shows which template emitted that text

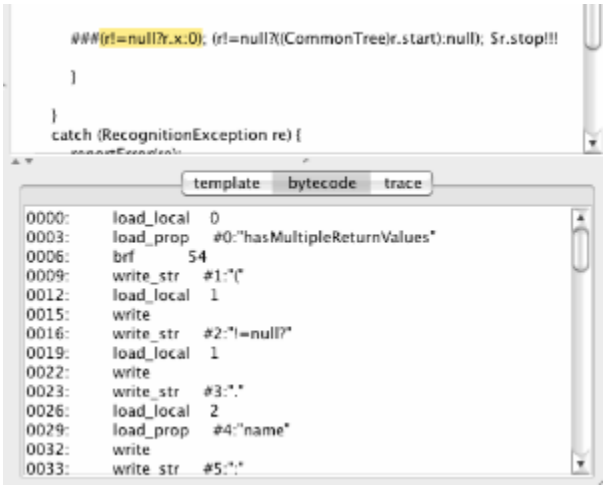
AST window shows how ST compiler interpreted template source

Sometimes it's helpful to figure out how ST interprets the syntax of your source code. When looking at the template source, the right subpane next to it shows the AST. If you click on an element in the AST, it highlights the place in the source code for which it was created.



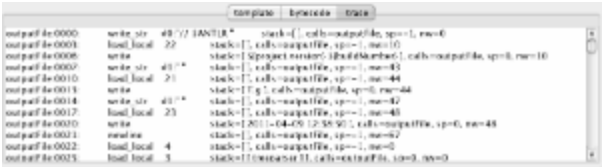
Displaying compiled ST bytecodes

If you'd like to see what the bytecode looks like, you can click on that pane.



Displaying bytecode interpreter trace

Finally, you can also see the interpreter in action (mostly of educational value). The trace shows the stack of the byte code interpreter, the call stack, and the number of output characters.



Dynamic scoping of attributes

When you reference an attribute, say, x within your template, ST tries to resolve x within the injected attributes of that template. If you can't find it, ST looks up the enclosing template change towards the root of the overall template hierarchy. This dynamic scoping allows deeply nested templates to refer to attributes injected into the root template. You might, for example, want to know the overall file name in some deeply nested template. Or, you might want to know the surrounding method name when you're generating code in nested template for the various statements.

Sometimes, though, it can be hard figuring out why an attribute as the wrong value. You can look at the lower left pane for the stack of templates to see the scope context. Simply start at the bottom and look upwards until you find the first definition of your attribute. Here's an example where the output shows "CommonTree" because the ruleLabelPropertyRef_start references attribute labelType:

```
(<scope>!=null?((<labelType>)<scope>.start):null)
```

labelType happens to be a default attribute of a template above in the hierarchy that references <ASTLabelType>. The ASTLabelType value is set as an argument of genericParser template way up the hierarchy, almost at the root.

Template regions

Template regions

ST introduces a finer-grained alternative to template inheritance, dubbed *regions*. (Regions are similar to a feature in Django). This feature allows a programmer to mark a location or series of lines in a template, and give it a name. A subgroup which inherits this template can provide replacement code to override just the named region. This avoids having to override the supergroup's template with a whole replacement template, when just a small addition or replacement is needed. While regions are syntactic sugar on top of template inheritance, the improvement in simplicity and clarity over normal coarser-grained inheritance is substantial.

Regions and subdirectories
Don't use regions in subdirectories; it's a bit broken and I'm having trouble nailing down exact semantics. Subject to change. So, if you put a template t in subdir S for fully-qualified name /S/t under a STGroupDir then don't use regions in t. Regions at root level are fine. Note I mean subdir not subgroup/inheritance stuff. That works.

Add text at a location

For example, in a code-generation scenario, imagine using the following template called `method` to produce the text for a method:

Java.stg

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

Suppose that you also want the option for the method template to place debugging statements into the generated method code. (To be clear about this example: this would be debugging code in the generated Java method, not code to debug the template processing itself.)

You could start placing debug text into the existing template, making it optional using the conditionally-included subtemplates feature, placing <if(...)> etc around the debugging lines. But that clutters up the templates of the Java group considerably, and also fails to achieve proper separation of concerns.

Instead you would like to have all debugging stuff encapsulated in a separate template group which focuses on debugging. In that template group, you *could* create an overriding template for method by copying and pasting the entire existing method template and inserting your additions. But then you are duplicating all of that output literal text, which breaks the "single point of change principle."

Instead just leave a hole in the main method template that a subgroup can override, here a location marked with `<@preamble()>`:

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

In a template subgroup focusing on debugging (group dbg), define the region using a fully qualified name which includes the region's surrounding template name, `@method.preamble()`, and supply the replacement text:

Dbg.stg

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

Regions are like *subtemplates* scoped within a template, hence, the fully-qualified name of a region is `@`*t*`.`*r*`()` where *t* is the enclosing template and r is the region name.

Replace a region of existing template text

Consider another problem where you would like, in a template subgroup, to replace a small portion of a large inherited template. Imagine you have a template that generates conditional statements in the output language, but you would also like to be able to generate a debug version of these statements which track the fact that an expression was evaluated.

(To be clear about this example, ths template's purpose is to produce "if" statements in the output language, here Java. That "if" is unrelated to the issue of using template <if(...)> expressions, which we are discussing how to avoid.)

Again, to avoid mingling debug version code with your main templates, you want to avoid "if dbg" type template expressions. Instead, mark the region within the template that might be replaced by an inheriting subgroup focusing on debugging. Here the code is marked with the pair of markers `<@eval>...<@end>`:

Java.stg

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

where `<@r>..<@end>` marks the region called *r*. Now a template subgroup can override (replace) this region:

Dbg.stg

```
import "Java.stg"
@test.eval() ::= "trackAndEval(<expr>)"
```

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

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

(I'm guessing this is trying to say: Within the replacement template text, ie: right-hand-side, you can use the symbol @super.r() to insert the original region contents. Also guessing that "super" is a keyword, and should not be replaced, while "r" *should* be replaced with the actual region name. Pretty sure this needs to be enclosed in expression delimiters, not just bare. -- GW)

Using StringTemplate with Java

Installation

All you need to do is get the StringTemplate jar into your CLASSPATH as well as its dependent ANTLR jar. Download the following and put into your favorite lib directory such as /usr/local/lib on UNIX:

- antlr-complete.jar
- StringTemplate download page. Get ST-4.X.jar

Add to your CLASSPATH. On UNIX that looks like

```
$ export
CLASSPATH="/usr/local/lib/antlr-3.3-complete.jar:/usr/local/lib/ST-4.X.jar:$CLASSPATH"
```

Java will now see all the libraries necessary to execute ST stuff. Also, check out the latest StringTemplate build. It has both the source and the binary Java jar.

Hello world

Here's a simple, complete program to test your installation.

Hello.java

```
import org.stringtemplate.v4.*;

public class Hello {
    public static void main(String[] args) {
        ST hello = new ST("Hello, <name>");
        hello.add("name", "World");
        System.out.println(hello.render());
    }
}
```

Here's how to compile and run it from the command line:

```
/tmp $ javac Hello.java
/tmp $ java Hello
Hello, World
```

Loading template groups

Group files

To load a group file, use the `STGroupFile` subclass of `STGroup`:

load file name

```
STGroup g = new STGroupFile("test.stg");
```

This tells StringTemplate to look in the current directory for test.stg. If not found, `STGroupFile` looks in the `CLASSPATH`. You can also use a relative path. The following looks for subdirectory `templates` in the current directory or, if not found, in a directory of the CLASSPATH.

load relative file name

```
STGroup g = new STGroupFile("templates/test.stg");
```

You can also use a fully qualified name:

load fully qualified file name

```
STGroup g = new STGroupFile("/usr/local/share/templates/test.stg");
```

Group directories

Group files, described above, are like directories of templates packed together into a single file (like text-based jars). To load templates stored within a directory as separate .st files, use `STGroupDir` instances:

load relative directory of templates

```
STGroup g = new STGroupDir("templates");
```

If `templates` is not found in the current directory, StringTemplate looks in the CLASSPATH. Or, you can specify the exact fully qualified name:

load fully qualified directory of templates

```
STGroup g = new STGroupDir("/usr/local/share/templates");
```

Group strings

For small groups, it sometimes makes sense to use a string within Java code:

```
String g =
 "a(x) ::= <<foo>>\n"+
 "b() ::= <<bar>>\n";
STGroup group = new STGroupString(g);
ST st = group.getInstanceOf("a");
String expected = "foo";
String result = st.render();
assertEquals(expected, result);
```

URL/URI/Path quagmire

Make sure to pass either a valid file name as a string or a valid URL object. File/dir names are relative like "foo.stg", "foo", "org/foo/templates/main.stg", or "org/foo/templates" OR they are absolute like "/tmp/foo". This is incorrect:

```
STGroup modelSTG = new STGroupFile(url.getPath());
```

because it yields a file path to a jar and then inside:

```
file:/somedirectory/AJARFILE.jar!/foo/main.stg
```

This isn't a valid file system identifier. To use URL stuff, pass in a URL object not a string

See Converting between URLs and Filesystem Paths for more information.

API documentation

Java API

Automatic line wrapping

Automatic line wrapping

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

That said, if there's a newline in the literal to emit, it will wrap at the newline.

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

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

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

```
abc
de
```

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

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

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

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

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

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

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

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

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

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

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

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

```
ab
  cd
  e
```

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

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

With numbers

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

this template will emit (at width 40):

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

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

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

Adding that option generates the following output:

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

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

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

That template yields the following output:

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

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

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

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

Using StringTemplate with CSharp

Installation

Hello world

API documentation

StringTemplate cheat sheet

Expression elements

See Template expressions

Syntax	Description
<*attribute*>	Evaluates to the string value of *attribute* (toString(), ToString(), *_str_*()) if it exists else empty string.
<i>, <i0>	The iteration number indexed from one and from zero, respectively, when referenced within a template being applied to an attribute or attributes. Is only visible to the template being applied to the iterator values.
<*attribute*.*property*>	Looks for *property* of *attribute* as a property (C#), then accessor methods like `get`*Property*`()` or `is`*Property*() or `has`*Property*(). If that fails, `StringTemplate` looks for a raw field of the *attribute* called *property*. Evaluates to the empty string if no such property is found.
<*attribute*.(*expr*)>	Indirect property lookup. Same as *attribute*.*property* except use the value of *expr* as the property_ name. Evaluates to the empty string if no such property is found.
<*multi-valued-attribute*>	Concatenation of string values of the elements. If *multi-valued-attribute* is missing, it evaluates to the empty string.
<*multi-valued-attribute*; separator=*expr*>	Concatenation element string values separated by *expr*.
<[*mine*, *yours*]>	Creates a new multi-valued attribute (a list) with elements of `mine` first then all of `yours`.
<*template*(*argument-list*)>	Include *template*. The *argument-list* is a list of attribute expressions or attribute assignments where each assignment is of the form *arg-of-template=expr*. *expr* is evaluated in the context of the surrounding template not of the invoked template. Example, `bold(name)` or `bold(item=name)` of `item` is an argument of template `bold`. The sole argument or the final argument, if argument assignments syntax is used, can be the "pass through" argument "..."

<*(expr)*(*argument-list*)>	Include *template* whose name is computed via *expr*. The *argument-list* is a list of attribute expressions or attribute assignments where each assignment is of the form *attribute=expr*. Example `<(whichFormat)()>` looks up `whichFormat`'s value and uses that as template name. Can also apply an indirect template to an attribute.
<*attribute*:*template*(*argument-list*)>	Apply *template* to *attribute* with optional *argument-list*. Example: `<name:bold()>` applies `bold()` to `name`'s value. The first argument of the template gets the iterated value. The template is not applied to null values.
<*attribute*:*(expr)*(*argument-list*)>	Apply a template, whose name is computed from *expr*, to each value of *attribute*. Example `<data:(name)()>` looks up `name`'s value and uses that as template name to apply to `data`.
<*attribute*:*t1*(*argument-list*): ... :*tN*(*argument-list*)>	Apply multiple templates in order from left to right. The result of a template application upon a multi-valued attribute is another multi-valued attribute. The overall expression evaluates to the concatenation of all elements of the final multi-valued attribute resulting from *templateN*'s application.
<*attribute*:{x | *anonymous-template*}>	Apply an anonymous template to each element of *attribute*. The iterated value is set to argument x. The anonymous template references <x> to access the iterator value.
<*a1*,*a2*,...,*aN*:{*argument-list* | *anonymous-template*}>	Parallel list iteration. March through the values of the attributes *a1*..*aN*, setting the values to the arguments in *argument-list* in the same order. Apply the anonymous template.
<*attribute*:*t1*(),*t2*(),...,*tN*()>	Apply an alternating list of templates to the elements of *attribute*. The template names may include argument lists.
`\<` or `\>`	escaped delimiter prevents < or > from starting an attribute expression and results in that single character.
`<\ >`, `<\n>`, `<\t>`, `<\r>`	special character(s): space, newline, tab, carriage return. Can have multiple in single <...> expression.
`<\uXXXX>`	Unicode character(s). Can have multiple in single <...> expression.
`<\\>`	Ignore the immediately following newline char. Allows you to put a newline in the template to better format it without actually inserting a newline into the output
`<! comment !>`	Comments, ignored by StringTemplate.

Functions

Syntax	Description
<first(attr)>	The first or only element of attr. You can combine operations to say things like first(rest(names)) to get second element.
<length(attr)>	Return the length of a mult-valued attribute or 1 if it is single attribute. If attribute is null return 0. Strings are not special; i.e., length("foo") is 1 meaning "1 attribute". Nulls are counted in lists so a list of 300 nulls is length 300. If you don't want to count nulls, use length(strip(list)).
<strlen(attr)>	Return the length of a string attribute; runtime error if not string.
<last(attr)>	The last or only element of attr.
<rest(attr)>	All but the first element of attr. Returns nothing if <attr> is single valued.
<reverse(attr)>	Return a list with the same elements as v but in reverse order. null values are NOT stripped out. use reverse(strip(v)) to do that.
<trunc(attr)>	returns all elements but last element
<strip(attr)>	Return a new list w/o null values.
<trim(attr)>	Trim whitespace from back/front of a string; runtime error if not string.

Statements

See Templates#conditionals

Syntax	Description
<if(*attribute*)>*subtemplate* <else>*subtemplate2* <endif>	If *attribute* has a value or is a boolean object that evaluates to `true`, include *subtemplate* •else include *subtemplate 2*. These conditionals may be nested.

<if(*x*)>*subtemplate* <elseif(*y*)>*subtemplate2* <elseif(*z*)>*subtemplate3* <else>*subtemplate4* <endif>	First attribute that has a value or is a boolean object that evaluates to `true`, include that subtemplate. These conditionals may be nested.
<if(!*attribute*)>*subtemplate*<endif>	If *attribute* has no value or is a `bool` object that evaluates to `false`, include *subtemplate*. These conditionals may be nested.

Conditional expressions can include "or" and "and" operations as well as parentheses. E.g.,

```
<if((!a||b)&&!(c||d))>broken<else>works<endif>
```

Groups

See Group file syntax

```
t1(arg1,arg2,...,argN) ::= "template1" // single-line template
// multi line template
t2(args) ::= <<
template2
>>
// multi line template that ignores indentation and newlines
t2(args) ::= <%
template3
%>
```

To import other templates, use the import statement:

```
import "directory"
import "template file.st"
import "group file.stg"
```

The paths can be absolute, but should probably be relative to the class path or the directory of the template that imports them.

Reserved words

Don't use these as attribute names or template names:

true, false, import, default, key, group, implements, first, last, rest, trunc, strip, trim, length, strlen, reverse, if, else, elseif, endif, delimiters.

Using StringTemplate with Python

Installation

Hello world

API documentation

Expression options

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.
- **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 new line 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.

Using StringTemplate with Objective-C

Installation

Hello world

API documentation

Auto-indentation

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

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

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

StringTemplate performs auto indentation as the text gets emitted during rendering using class `AutoIndentWriter`, which is an implementation of a generic `STWriter` (interface, protocol, or nothing 😊 depending on the port implementation language).

To turn off auto indentation, tell StringTemplate to use `NoIndentWriter` by invoking the `write(`*`writer`*`)` method instead of the usual render method:

Java

```
StringWriter sw = new StringWriter();
NoIndentWriter w = new NoIndentWriter(sw);
st.write(w); // same as render() except with a different writer
String result = sw.toString();
```

C#

```
(TBD)
```

Python

```
(TBD)
```

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 called *attributes*. An attribute is either a program object such as a string or `VarSymbol` object, a template instance, or sequence of attributes including other sequences. Template engines are domain-specific languages for generating structured text. `StringTemplate` breaks up your template into chunks of text and attribute expressions, which are by default enclosed in angle brackets *<attribute-expression>* (but you can use whatever single character start and stop delimiters you want). `StringTemplate` ignores everything outside of attribute expressions, treating it as just text to spit out. To evaluate a template and generate text, we "render" it with a method call:

Java	`ST.render()`
C#	`ST.Render()`
Python	`ST.render()`

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.stringtemplate.v4.*;
...
ST hello = new ST("Hello, <name>");
hello.add("name", "World");
System.out.println(hello.render());
```

C#

```
using Antlr4.StringTemplate;

Template hello = new Template("Hello, <name>");
hello.Add("name", "World");
Console.Out.WriteLine(hello.Render());
```

Python

```
import stringtemplate4

hello = stringtemplate4.ST("Hello, <name>")
hello["name"] = "World"
print str(hello.render())
```

MVC Pattern
In the parlance of the model-view-controller (MVC) pattern, templates represent the *view* and the code fragment represents both model (the `name` string) and *controller* (that pulls from the model and injects attributes into the view).

`StringTemplate` is not a "system" or "engine" or "server"; It is designed to be embedded inside other applications and is distributed as a small library with no external dependencies except ANTLR (used for parsing the StringTemplate template language).

Groups of templates

The primary classes of interest are `ST`, `STGroupDir`, and `STGroupFile`. You can directly create a template in code, you can load templates from a directory, and you can load a file containing a collection templates (a template group file). Group files behave like zips or jars of template directories.

For example, let's assume we have two templates `decl` and `init` in directory `/tmp`:

/tmp/decl.st

```
decl(type, name, value) ::= "<type> <name><init(value)>;"
```

/tmp/init.st

```
init(v) ::= "<if(v)> = <v><endif>"
```

We can access those templates by creating a STGroupDir object. We then ask for an instance with `getInstanceOf()` and inject attributes with `add()`:

Java

```
STGroup group = new STGroupDir("/tmp");
ST st = group.getInstanceOf("decl");
st.add("type", "int");
st.add("name", "x");
st.add("value", 0);
String result = st.render(); // yields "int x = 0;"
```

C#

```
(untested)
using Antlr4.StringTemplate;


TemplateGroup group = new TemplateGroupDirectory("/tmp");
Template st = group.GetInstanceOf("decl");
st.Add("type", "int");
st.Add("name", "x");
st.Add("value", 0);
String result = st.Render(); // yields "int x = 0;"
```

Python

```
import stringtemplate4
(TBD)
```

If you would like to keep just the template text and not the formal template definition around the template text, you can use STRawGroupDir. Then, decl.st would hold just the following:

/tmp/decl.st

```
<type> <name><init(value)>;
```

That makes it easier for graphics designers and HTML people to work with template files, although the formal parameter definitions are okay for people using these to generate source code as they will usually be programmers not graphics people.

This example demonstrates some key syntax and features. Template definitions look very similar to function definitions except that the bodies are strings. Template decl takes three arguments by reference as only two of them directly. Instead of expanding `value` immediately, invokes/includes template `init` instead with an argument of `value`. Alternatively, Template `init` could take no arguments. It would still see attribute `value` though because of *dynamic scoping*. That essentially means that a template can reference the attributes of any invoking template.

Note, that to get the spacing correct, there is no space between expression `<name>` and `<init()>`. If we do not inject a declaration initialization (attribute `value`), we don't want to space between the name and the ';'. Template `init` emits `" = <v>"` only if the controller code injects a value, which we do here (0). In this case, we have injected two strings and one integer, but we can send in any object we want; more on that below.

Sometimes it's more convenient to collect templates together into a single unit called the group file. For example, we can collected the template .st files into a single .stg group file:

/tmp/test.stg

```
decl(type, name, value) ::= "<type> <name><init(value)>;"
init(v) ::= "<if(v)> = <v><endif>"
```

To pull templates from this file instead of a directory, all we have to do is change our constructor to use `STGroupFile`:

Java

```
STGroup group = new STGroupFile("/tmp/test.stg");
ST st = group.getInstanceOf("decl");
st.add("type", "int");
st.add("name", "x");
st.add("value", 0);
String result = st.render(); // yields "int x = 0;"
```

C#

```
(untested)using Antlr4.StringTemplate;

TemplateGroup group = new TemplateGroupFile("/tmp/test.stg");
Template st = group.GetInstanceOf("decl");
st.Add("type", "int");
st.Add("name", "x");
st.Add("value", 0);
string result = st.Render(); // yields "int x = 0;"
```

Python

```
import stringtemplate4
(TBD)
```

Accessing properties of model objects

Template expressions can access the properties of objects injected from the model. For example, consider the following `User` object.

Java

```
public static class User {
    public int id; // template can directly access via u.id
    private String name; // template can't access this
    public User(int id, String name) { this.id = id; this.name = name; }
    public boolean isManager() { return true; } // u.manager
    public boolean hasParkingSpot() { return true; } // u.parkingSpot
    public String getName() { return name; } // u.name
    public String toString() { return id+":"+name; } // u
}
```

C#

```
 (TBD)
```

Python

```
(TBD)
```

We can inject instances of `User` just like predefined objects like strings and can refer to properties using the '.' dot property access operator. StringTemplate interprets `o.p` by looking for property p within object o. The lookup rules differ slightly between language ports, but in general they follow the old JavaBeans naming convention. StringTemplate looks for methods `getP()`, `isP()`, `hasP()` first. If it fails to find one of those methods, it looks for a field called p. In the following example, we access properties id and name. Also note that the template uses $...$ delimiters, which makes more sense since we are generating HTML.

Java

```
ST st = new ST("<b>$u.id$</b>: $u.name$", '$', '$');
st.add("u", new User(999, "parrt"));
String result = st.render(); // "<b>999</b>: parrt"
```

C#

```
(TBD)
```

Python

```
(TBD)
```

Property reference `u.id` evaluates to the field of the injected User object whereas `u.name` evaluates to the "getter" for the field `name`.

StringTemplate renders all injected attributes and any reference properties to text using the string conversion method natural for the implementation language; e.g., `toString()` in Java, `ToString()` in C#, and `str{_}{}_()` in Python. In this case, a reference to u would yield "999:parrt".

Injecting data aggregate attributes

Being able to pass in objects and access their fields is very convenient but often we don't have a handy object to inject. 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 add() calls. You may pass in an aggregrate attribute name to add() with the data to aggregate. The syntax of the attribute name describes the properties. For example `"a.{p1,p2,p3}"` describes an attribute called a that has three properties p1, p2, p3. Here's an example:

Java

```
ST st = new ST("<items:{it|<it.id>: <it.lastName>, <it.firstName>\n}>");
st.addAggr("items.{ firstName ,lastName, id }", "Ter", "Parr", 99); // add() uses
varargs
st.addAggr("items.{firstName, lastName ,id}", "Tom", "Burns", 34);
String expecting =
        "99: Parr, Ter"+newline +
        "34: Burns, Tom"+newline;
```

C#

```
(TBD)
```

Python

```
(TBD)
```

Applying templates to attributes

Let's look more closely at how StringTemplate renders attributes. It does not distinguish between single and multi-valued attributes. For example, if we add attribute `name` with value "parrt" to template "<name>", it renders to "parrt". If we call `add()` twice, adding values "parrt" and "tombu" to `name`, it renders to "parrttombu". In other words, multi-valued attributes render to the concatenation of the string values of the elements. To insert a separator, we can use the separator option: "<name; separator=\", \">". Without changing the attributes we inject, the output for that template is "parrt, tombu". To alter the output emitted for each element, we need to iterate across them.

StringTemplate has no "foreach" statement. Instead, we apply templates to attributes. For example, to surround each name with square brackets, we can define a bracket template and apply it to the names:

```
test(name) ::= "<name:bracket()>" // apply bracket template to each name
bracket(x) ::= "[<x>]"            // surround parameter with square brackets
```

Injecting our list of names as attribute `name` into template `test` yields "[parrt][tombu]". Combining with the separator operator yields "[parrt], [tombu]":

```
test(name) ::= "<name:bracket(); separator=\", \">"
bracket(x) ::= "[<x>]"
```

StringTemplate is dynamically typed in the sense that it doesn't care about the types of the elements except when we access properties. For example, we could pass in a list of User objects, User(999,"parrt") and User(1000,"tombu"), and the templates would work without alteration. StringTemplate would use the "to string" evaluation function appropriate for the implementation language to evaluate <x>. The output we'd get is "[999:parrt], [1000:tombu]". StringTemplate sets the first parameter of the template it's applying to the iterated value (x in this case).

Sometimes creating a separate template definition is too much effort for a one-off template or a really small one. In those cases, we can use *anonymous templates* (or *subtemplates*). Anonymous templates are templates without a name enclosed in curly braces. They can have arguments, though, just like a regular template. For example, we can redo the above example as follows.

```
test(name) ::= "<name:{x | [<x>]}; separator=\", \">"
```

Anonymous template {x | [<x>]} is the in-lined version of bracket(). Argument names are separated from the template with the '|' pipe operator.

StringTemplate will iterate across any object that it can reasonably interpret as a collection of elements such as arrays, lists, dictionaries and, in statically typed ports, objects satisfying iterable or enumeration interfaces.

General use of StringTemplate for formatting

The ST.format method is great for general use in general code.

Java

```
int[] num =
    new int[] {3,9,20,2,1,4,6,32,5,6,77,888,2,1,6,32,5,6,77,
        4,9,20,2,1,4,63,9,20,2,1,4,6,32,5,6,77,6,32,5,6,77,
        3,9,20,2,1,4,6,32,5,6,77,888,1,6,32,5};
String t =
    ST.format(30, "int <%1>[] = { <%2; wrap, anchor, separator=\", \"> };", "a", num);
System.out.println(t);
```

C#

```
(TBD)
```

Python

```
(TBD)
```

Yields:

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

Templates

Templates are essentially exemplars of the desired output with "holes" where the programmer may stick untyped values called attributes or other template instances. To enforce model-view separation, templates may not test nor compute with attribute values and, consequently, attributes have no need for type information. Templates may, however, know the data is structured in a particular manner such as a tree structure.

A template is a sequence of text and expression elements, optionally interspersed with comments. At the coarsest level, the basic elements are:

text
<*expr*>
<! *comment* !>

Escape delimiters with a backslash character: \< or \>.

Template expression delimiters
This documentation uses <...> to delimit expressions, but you can use any single start and stop character. For HTML, $..$ is a much better choice obviously. You can set the delimiters as you create templates or template groups.

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

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

Expression literals

StringTemplate has the following literals.

Syntax	Discussion
true	Boolean true value
false	Boolean false value
char	*char* *space* | \n | \r | \t | \uXXXX
\\	Ignore the immediately following newline char. Allows you to put a newline in the template to better format it without actually inserting a newline into the output
"*string*"	A string of output characters
{*template*}	An anonymous subtemplate
{*args* | *template*}	An anonymous subtemplate with arguments
[ ]	An an empty list.

[*expr1*, *expr2*, ..., *exprN*]	A list with *N* values. It behaves like an array or list injected from controller code.

For more on list semantics, see The real story on null vs empty.

Attribute expressions

Syntax	Discussion
a	Look up *a* and converts it to a string using the appropriate implementation language conversion function such as toString(), ToString(), o *_str()*. *StringTemplate uses dynamic scoping and so it looks up the enclosing template chain searching for _a*. If not defined locally as a formal argument, StringTemplate looks at the template that invoked the current template, and so on. Evaluates to the empty string if *a* does not exist.
(*expr*)	Evaluate *expr* to a string. This is useful for computing the name of the template or property.
expr.p	Get property p of *expr*. *expr* is typically just an attribute; e.g., in expression `<user.name>`, *expr* is `user` and p is `name`. Looks for p as a property (C#), then accessor methods getProperty() or isProperty() or hasProperty(). If that fails, StringTemplate looks for a raw field of the attribute called p. Evaluates to the empty string if no such property is found. See also Introduction#properties
expr1.(*expr2*)	Evaluate *expr2* to a string and use that as the name of a property, reducing this case to the previous.

Template include

Syntax:

expr(*args*)

where

args
args ...
args *expr1*, *expr2*, __, *exprN*
args a1=*expr1*, a2=*expr2*, __, aN=*exprN*
args a1=*expr1*, a2=*expr2*, __, aN=*exprN*, ...

Templates have optional arguments. These arguments can be a list of expressions, as in most languages, or a list of assignments to the named arguments of the target template. The list of assignment style is less efficient but useful for templates that take many arguments.

Normally, formal argument definitions hide any attributes visible above in the enclosing template chain. If you replace the argument list with an ellipsis, "...", the included template can see all of the attributes from above. If you use a named argument assignments, the last argument can also be this "pass through" operator. In this case, it sets any remaining arguments. In general, the pass through operator sets argument x of the included template to be whatever x is in the enclosing template.

Discussion

To include another template, just reference that template like a function call with any arguments you need. Note that *expr* is either a template name or a parenthesized expression that evaluates to the name of a template. E.g., (templateName)(). It is an error if there is a mismatch between the number of values passed into t and t's formal arguments.

Applying (mapping) templates across attributes

Syntax

Case 1 *expr* : t(*args*)
Case 2 *expr* : t1(*args*), t2(*args*), ..., tN(*args*)
Case 3 *expr1*, ..., *exprN* : t(*args*)

Templates t1..tN can be template names or anonymous subtemplates that define the appropriate number of arguments:

{a1, a2, ..., aN | ...}

Discussion

Case 1. StringTemplate applies template t to each element of the *expr*. It assigns the array value to the first attribute in the formal argument list of t. Any extra *args* passed in, are assigned to the other formal arguments. (It is an error if there is a mismatch between the number of values

passed into t and t's formal arguments.)

Consider a list of variable names, `names`, that we want to surround with parentheses and assume we have a template called `parens`:

```
parens(x) ::= "(<x>)"
```

Expression `names:parens()` invokes `parens` once for every element of `names`. For example,

```
["a", "b", "c"]:parens()
```

yields `(a)(b)(c)`. Formal argument `x` is assigned the iterated value at each invocation of `parens`. In contrast,

```
parens(["a", "b", "c"])
```

yields `(abc)`.

Note that, for single valued attributes, template application is like an alternate include syntax. For example, `"a":parens()` is the same as `parens("a")`.

Case 2. StringTemplate does a round robin walk through the templates. This arose for the case with HTML tables where you want to alternate the color between rows. E.g., 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 names2 would get blueListItem again, etc...

Case 3. At each iteration, StringTemplate pulls a single element from each of the expressions and passes them to template t or the anonymous template. Iteration proceeds while at least one of the attributes has values. Is an example that applies an anonymous template to two lists. Number of expressions must match the number of formal arguments in the template.

```
<names,phones:{ n,p | <n>: <p>}>
```

This is like the Python zip function that creates a list of tuples from multiple lists.

Conditionals

Syntax

<if(*boolexpr1*)>*subtemplate*
<elseif(*boolexpr2*)>*subtemplate2*
...
<elseif(*boolexprN*)>*subtemplateN*
<else>*defaultsubtemplate*
<endif>

where

boolexpr *boolexpr* || *boolexpr*
boolexpr *boolexpr* && *boolexpr*
boolexpr !*boolexpr*
boolexpr *expr*

boolexpr is generally a simple attribute reference *a*, property reference *a.p*, or !*booleanexpr*. Lowest to highest precedence order: || then && then !.

Discussion

The conditional expressions test of the presence or absence of an attribute. Strict separation of model and view requires that expressions **cannot** test attribute values such as `name=="parrt"`. If you do not set an attribute or pass in a null-valued attribute, that attribute evaluates to false. StringTemplate also returns false for empty lists and maps as well a "empty" iterators such as 0-length lists (see Interpreter.testAttributeTrue()). All other attributes evaluate to true with the exception of Boolean objects. Boolean objects evaluate to their object value. Strictly speaking, this is a violation of separation, but it's just too weird to have Boolean false objects evaluate to true just because they are non-null.

Boolean expressions can use "or" and "and" operators though, again, I feel that it's a violation of model view separation. I decided to yield to lobbying efforts because we can already simulate these operators with nested conditionals. Use at your peril. 

Anonymous subtemplates

Syntax

{ *template* }
{*arg1*, ..., *argN* | *template* }

Discussion

Anonymous subtemplates are typically used for small or one-off templates that you need to apply to an attribute. They are literally templates without names and can have arguments used by the template expressions. For example, the following snippet converts a list of variable names to a list of integer variable definitions.

```
<vars:{v | int <v>;}>
```

If a subtemplate is applied to an attribute, it also has two hidden arguments: `i` and `i0`, which are the 1-base and 0-based iteration indexes, respectively. For example, the following expression converts a list of variable names to assignments: "a=1;b=2;".

```
<["a","b"]:{v | <v>=<i>;}>
```

Anonymous subtemplates are also useful to pass snippets to other templates. Given template:

```
method(name,body,cleanup) ::= <<
void <name>() {
  <body>
  <cleanup>
}
>>
```

we could invoke it with subtemplates:

```
<method(name="f", body={x=1;}, cleanup={printf("leaving <name>");})>
```

The output is

```
void f() {
  x=1;
  printf("leaving f");
}
```

Because of dynamic scoping, the snippet can see attribute `name` to fill in the string of the printf.

Functions

StringTemplate has a number of side-effect free built-in functions that operate on attributes. Each function takes a single attribute and returns a single value. For the complete list of functions, see StringTemplate cheat sheet#functions.

Lazy evaluation

There is usually an order mismatch between convenient, efficient computation of data attributes and the order in which the results must be emitted according to the output language. The developer's choice of controller and data structures has extensive design ramications. If the developer decides to have the templates embody both view and controller, then the order of the output constructs drives output generation. This implies that the order of attribute ai references in the view dictates the order in which the model must compute those values, which may or may not be convenient. If the output language requires that n attributes be emitted in order a0..an1, a single forward computation dependency, ai = f(aj) for i < j, represents a hazard. Each ai computation must manually trigger computations for each attribute upon which it is dependent. A simple change in the attribute reference order in the output templates can introduce new dependencies and unforeseen side-effects that will cause bad output or even generator crashes. This approach of having the templates drive generation by triggering computations and pulling attributes from the model is not only dangerous but may also make the computations inconvenient and inefcient.

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

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

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

Formal template syntax

```
template : element* ;

element
    : INDENT? COMMENT NEWLINE
    | INDENT singleElement
    | singleElement
    | compoundElement
    ;

singleElement
    : exprTag
    | TEXT
    | NEWLINE
    | COMMENT
    ;

compoundElement : ifstat | region ;

exprTag : LDELIM expr ( ';' exprOptions )? RDELIM  ;

region
    : INDENT? LDELIM '@' ID RDELIM
      template
      INDENT? LDELIM '@end' RDELIM NEWLINE?
    ;

subtemplate : '{' ( ID ( ','  ID )* '|' )? template INDENT? '}' ;

ifstat
    : INDENT? LDELIM 'if' '(' conditional ')' RDELIM
```

```
      template
      ( INDENT? LDELIM 'elseif' '(' conditional ')' RDELIM template )*
      ( INDENT? LDELIM 'else' RDELIM template )?
      INDENT? LDELIM 'endif' RDELIM NEWLINE?
    ;

conditional : andConditional ( '||' andConditional )* ;

andConditional : notConditional ( '&&' notConditional )* ;

notConditional
    : '!' notConditional
    | memberExpr
    ;

notConditionalExpr : ID ( '.'  ID  |  '.' '(' mapExpr ')'  )* ;

exprOptions : option ( ',' option )*  ;

option : ID ( '=' exprNoComma )? ;

exprNoComma : memberExpr ( ':' mapTemplateRef )? ;

expr : mapExpr ;

mapExpr
    : memberExpr ( ( ',' memberExpr )+  ':' mapTemplateRef )?
      ( ':'  mapTemplateRef ( ',' mapTemplateRef )*  )*
    ;

mapTemplateRef
    : ID '(' args ')'
    | subtemplate
    | '(' mapExpr ')' '(' argExprList? ')'
    ;

memberExpr : includeExpr ( '.' ID  |  '.' '(' mapExpr ')'  )* ;

includeExpr
    : ID '(' expr? ')'
    | 'super' '.' ID '(' args ')'
    | ID '(' args ')'
    | '@' 'super' '.' ID '('  ')'
    | '@' ID '('  ')'
    | primary
    ;

primary
    : ID
    | STRING
    | 'true'
    | 'false'
    | subtemplate
    | list
    | '(' conditional ')'
    | '(' expr ')' ( '(' ( argExprList )? ')' )?
    ;

args: argExprList
```

```
    | namedArg ( ',' namedArg )*
    |
    ;

argExprList : arg ( ',' arg )* ;

arg : exprNoComma ;

namedArg : ID '=' arg ;

list: '[' ']'
    | '[' listElement ( ',' listElement )* ']'
    ;

listElement
    : exprNoComma
```

```
    |
    ;
```

Motivation and philosophy

Why should I use StringTemplate?

StringTemplate ensures the separation of the specification of business logic and computation required to generate structured text from the specification of how that text is presented (i.e. its formatting).

For web developers, this translates to ensuring that a web page's business and persistence logic is separated from the the page's presentation.

For developers involved in code generation, this translates to ensuring generation and persistence logic is separated from output code structure. This separation directly supports the development of retargetable code generators.

Other benefits of model-view separation:

1. Encourages the development of templates that are reusable in similar applications
2. Unentangled templates serves as clear documentation of the generated code structure
3. The templates (and hence the output or view) can be changed independently. Using more than one set of such templates is the basic mechanism for retargetable code generation
4. Nonprogrammers can modify output structure because it's not threaded with code

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 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** b e 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 surprisingly 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.

Motivation for v4

Because of the weird license on ANTLR v2 (it's not BSD), I needed to remove StringTemplate's dependency on that library. That means I had to upgrade the grammars in StringTemplate to use ANTLR v3. I decided to completely reimplement StringTemplate for a v4 release because the old

version is a complete mess on the inside since it grew organically.

Backward compatibility was the goal not the rule. I fixed the little quirks to make a very clean definition. It is also be a hell of a lot faster because I use the byte code interpreter instead of a tree-based interpreter.

If you're interested in interpreters, you might want to take a look at the compiler and byte code interpreter I built for this version. It's very simple and clean:

https://github.com/antlr/stringtemplate4

Group inheritance

It's a common problem to need to generate a website with multiple skins or a code generator that must generate multiple variations of the same language such as Java 1.4 and 1.5 (1.5 added enums, for example). In either case, we want to design a variation as it differs from an existing code generation target. The easiest way to do this is to create a group of templates that override templates from a "supergroup" using the `import` statement. Templates in the current group override templates inherited from the imported group.

Let's look at the problem of generating enumerated types. In Java 1.4, we simulate enums with something like this:

Java 1.4

```
public static final int MyEnum_A = 1;
public static final int MyEnum_B = 2;
```

whereas Java 1.5 lets us do this:

Java 1.5

```
public enum MyEnum { A, B }
```

The goal is to keep the model and controller the same and to use a different group of templates according to the output we need. Here is what a piece of a Java 1.4 code generation template group file:

Java1_4.stg

```
class(name, members) ::= <<
class <name> {
    <members>
}
>>

constants(typename, names) ::= "<names:{n | <constant(n,i)>}; separator={<\n>}>"

constant(n) ::= "public static final int <typename>_<n>=<i>;"
```

Instead of copying and altering the entire group for Java 1.5, we can import the 1.4 group and alter just the part that changes, template `constant`s:

Java1_5.stg

```
import "Java1_4.stg"

/** Override constants from Java1_4.stg */
constants(typename, names) ::= <<
public enum <typename> { <names; separator=", "> }
>>
```

Group Java1_5 the inherited template `classs` will write template `constants`. The following sample code creates group objects referring to both template group files, creates and filled in instances of template `class`, and finally prints out the rendered text.

Java

```
public void test(String[] args) {
    STGroup java1_4 = new STGroupFile("/tmp/Java1_4.stg");
    STGroup java1_5 = new STGroupFile("/tmp/Java1_5.stg");
    System.out.println( getCode(java1_4) );
    System.out.println( getCode(java1_5) );
}
public String getCode(STGroup java) {
    ST cl = java.getInstanceOf("class"); // create class
    cl.add("name", "T");
    ST consts = java.getInstanceOf("constants");
    consts.add("typename", "MyEnum");
    consts.add("names", new String[] {"A","B"});
    cl.add("members", consts); // add constants as a member
    return cl.render();
}
```

C#

```
(TBD)
```

Python

```
(TBD)
```

Here is the output we get:

StringTemplate output

```
class T {
    public static final MyEnum_A=1;
    public static final MyEnum_B=2;
}
class T {
    public enum MyEnum { A, B }
}
```

Simply by switching template group pointers in the test function, we have generated different code--only the templates changed.

Template polymorphism

In object-oriented programming languages, the type of the receiving object dictates which overridden method to call and response to method call `o.f()`. Similarly, StringTemplate looks up templates according to the group of the template doing the invocation. Template *polymorphism* is at work.

Imagine we're generating an HTML page using templates from group file site.stg (using $...$ delimiters) that invokes a `searchbox` template defined within the same group file:

site.stg

```
page(content) ::= <<
<html>
<body>
$searchbox()$
$content$
</body>
</html>
>>

searchbox() ::= "<form method=get action=/search>...</form>"
```

To create an instance of `page`, inject some test content, and print it out, we can use the following controller code.

Java

```
STGroup g = new STGroupFile("site.stg", '$', '$');
ST page = g.getInstanceOf("page");
page.add("content", "a test page");
System.out.println(page.render());
```

C#

```
(TBD)
```

Python

```
(TBD)
```

We get the following output.

StringTemplate output

```
<html>
<body>
<form method=get action=/search>...</form>
a test page
</body>
</html>
```

Now, let's define a subgroup that overrides searchbox so that it generates nothing.

bland.stg

```
import "site.stg"
searchbox() ::= ""
```

We can use identical code except for changing the source of the templates:

```
STGroup g = new STGroupFile("bland.stg", '$', '$');
...
```

That yields the following output

StringTemplate output

```
<html>
<body>
a test page
</body>
</html>
```

Template page uses the overwritten version of the search box because we created that the page template via the subgroup. A template instantiated via the bland group should always start looking for templates in bland rather than the site supergroup even though searchbox is physically dened within site.stg.

Dynamic inheritance

When dealing with lots of output language variations, a proper separation of concerns can make generating multiple targets very complicated. For example, adding a debugging variation to our Java templates from above means adding another group of templates derived from both the 1.4 and 1.5 templates. The goal is to isolate all debugging code fragments in one group instead of cramming it all into the main templates. The problem is that we need a new debugging variation for each of the 1.4 and 1.5 templates--the import statement takes a literal string. We would need a new debugging group file to refer to different Java group file variations. In other words, DbgJava1_4.stg would have `import "Java1_4.stg"` and DbgJava1_5.stg would have `import "Java1_5.stg"`.

Obviously the number of variations can explode. To deal with this problem, StringTemplate allows you to dynamically import/inherit templates using the controller instead of an import statement in the group file (and is the only way to do it when using directories of templates instead of group files). Here's some code that assumes there are no imports in the group files. To create group dbg_java1_5, it imports java1_5 group, which itself imports java1_4.

Java

```
STGroup java1_4 = new STGroupFile("/tmp/Java1_4.stg");
STGroup java1_5 = new STGroupFile("/tmp/Java1_5.stg");
java1_5.imporTemplates(java1_4); // import "Java1_5.stg"
STGroup dbg_java1_4 = new STGroupFile("/tmp/Dbg.stg");
STGroup dbg_java1_5 = new STGroupFile("/tmp/Dbg.stg");
dbg_java1_4.importTemplates(java1_4); // import "Java1_4.stg"
dbg_java1_5.importTemplates(java1_5); // import "Java1_5.stg"
```

C#

```
(TBD)
```

Python

```
(TBD)
```

Inheritance and subdirectories

There are 2 kinds of people using StringTemplate: web type folks and code generation type folks. The web folks usually have directory trees full of templates and code generation people tend to have group files. Everything is fine with respect to inheritance as long as the two don't mix. It's just too complicated thinking about inheritance hierarchies as well as directory hierarchies as well as polymorphism all at the same time. So, I've made this illegal by throwing an unsupported operation exception if you try to do an import in a group filed its nested within an outer STGroupDir.

Remember, for each group there should only be one STGroup object. So, imagine we have group file foo.stg and template a.st in a directory called /tmp and we create a group object to handle that stuff:

Java

```
STGroup dir = STGroupDir("/tmp");
dir.getInstanceOf("a"); // no problem; looks in "/tmp/a.st"
dir.getInstanceOf("/foo/b"); // no problem if foo.stg has b() template
```

So far so good. Now, what you cannot do is have foo.stg import something because it is nested within dir:

```
import "bar.stg"  // causes unsupported operation exception
b() ::= "..."
```

If I did,

```
STGroup g = new STGroupFile("/tmp/foo.stg");
```

then there is no problem. Difference is that you don't want to mix inheritance with subdirectories and a group file within a STGroupDir acts like a subdirectory.

Group file syntax

Group files are collections of templates and dictionaries and have `.stg` file suffixes. Group files can also import templates or groups. The basic format looks like:

```
delimiters
imports
dictionaries
templates
```

Setting template expression delimiters

Use `delimiters` keyword to set delimiters per group file:

```
delimiters "%", "%"
t(x) ::= "%x%"
```

<...> are the default.

The legal delimiters are pretty limited. Avoid using characters that can be part of an expression like (...) and [...].

Import statements

You can import a single template file, a group file, or a directory of templates. File suffixes within the string operand indicates which:

/tmp/main.stg

```
import "/tmp/test.st"   // import a single template
import "/tmp/test.stg"  // import a group of templates from a file
import "/tmp/test"      // import a directory of templates
```

Instead of providing fully qualified path names, it's more flexible to specify relative path names. For example, if we import just the file name, StringTemplate looks for the files in the directory of the referring file (`/tmp`, in this case):

/tmp/main.stg

```
import "test.st"   // import a single template
import "test.stg"  // import a group of templates from a file
import "test"      // import a directory of templates
```

In the Java reference implementation, StringTemplate also looks for files and directories in the CLASSPATH. Using relative paths is particularly important if we want to load templates from jar files. Java cannot find files specified with absolute path names within jar files. Please refer to Using StringTemplate with Java.

Templates with the same name override templates from imported groups just like method overriding and class inheritance. See Group inheritance .

Dictionaries

There are situations where you need to translate a string in one language to a string in another language. For example, you might want to

translate `integer` to `int` when translating Pascal to C. You could pass a `Map` or `IDictionary` (e.g. hashtable) from the model into the templates, but then you have output literals in your model!& The `StringTemplate` solution is to support a dictionary feature. For example, here is a dictionary that maps Java type names to their default initialization 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 dictionary in a template, refer to it as you would an attribute. `<typeInitMap.int>` returns `"0"` from the map. If your type name is an attribute not a constant like `int`, then use an indirect property access: `<typeInitMap.(typeName)>`.

Dictionary strings can also be templates that can refer to attributes that will become visible via dynamic scoping of attributes once the dictionary value has been embedded within a template.

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

The `default` and other mappings cannot have empty values. They have empty values by default. If no key is matched by the map then an empty value is returned. The keyword `key` is available if you would like to refer to the key that maps to this value. This is particularly useful if you would like to filter certain words but otherwise leave a value unchanged; use `default : key` to return the key unmolested if it is not found in the map.

Dictionaries 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 any imported groups are consulted. You may not redefine a dictionary and it may not have the same name as a template in that group. The default clause must be at the end of the map.

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

Template definitions

Template definitions look like function definitions with untyped arguments:

```
templateName(arg1, arg2, ..., argN) ::= "single-line template"
```

or

```
templateName(arg1, arg2, ..., argN) ::= <<
multi-line template
>>
```

or

```
templateName(arg1, arg2, ..., argN) ::= <%
multi-line template that ignores indentation and newlines
%>
```

Default values

You can give default values to arguments as well (to the consecutive final n arguments). For example, here is a template that defines classes to extend Object if the invoking template does not set or the controller code does not inject attribute `sup`:

```
class(name,members,sup="Object") ::= "class <name> extends <sup> { <members> }"
```

Region definitions

Regions are small chunks of template code extracted from a larger surrounding template. To define or override a region, prefix the template

definition with the surrounding template name:

```
@surroundingTemplate.templateName(arg1, arg2, ..., argN) ::= "single-line template"
```

See Template regions for more details.

Aliases

To alias a template to another, use notation

```
aliasName ::= templateName
```

This is useful when controller code refers to two different templates, but we want to implement both in the same way without cutting and pasting.

Formal grammar

```
group
    :   delimiters? ('import' STRING)* ( template | dict )+
    ;

delimiters
    : 'delimiters' STRING ',' STRING
    ;

template
    :   (   '@' ID '.' ID '(' ')'
        |   ID            '(' formalArgs ')'
        )
        '::'=
        (   STRING            // "..."
        |   BIGSTRING         // <<...>>
        |   BIGSTRING_NO_NL   // <%...%>
        )
    |   ID "::=" ID     // alias one template to another
    ;

formalArgs
    :   formalArg ( ',' formalArg )* ( ',' formalArgWithDefaultValue )*
    |   formalArgWithDefaultValue ( ',' formalArgWithDefaultValue )*
    |
    ;

formalArg : ID ;

formalArgWithDefaultValue
    :   ID ( '=' STRING | '=' ANONYMOUS_TEMPLATE | '=' 'true' | '=' 'false' )
    ;

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

dict: ID '::=' '[' pairs ']' ;

pairs
   :   keyValuePair ( ',' keyValuePair )* ( ',' defaultValuePair )?
   |   defaultValuePair
   ;

keyValuePair : STRING ':' keyValue ;

keyValue
    :   BIGSTRING
    |   ANONYMOUS_TEMPLATE
    |   STRING
    |   TRUE
    |   FALSE
    |   'key'
    ;

defaultValuePair : 'default' ':' keyValue ;
```

Error listeners

To get notified when StringTemplate detects a problem during compilation of templates or, at runtime, when interpreting templates, provide StringTemplate with an error listener. The default listener sends messages to standard error/output, which are generally not what you want in a larger application. Here are the listener definitions in the various ports:

Java

```
public interface STErrorListener {
    public void compileTimeError(STMessage msg);
    public void runTimeError(STMessage msg);
    public void IOError(STMessage msg);
    public void internalError(STMessage msg);
}
```

C#

```
(TBD)
```

Python

```
(TBD)
```

The STMessage instances include information such as the ErrorType and any arguments. Evaluating the message to a string, as appropriate for the port language, yields a suitable message or you can pull it apart yourself.

You can specify a listener per group or per execution of the interpreter. To catch compile errors, make sure to set the listener before you trigger an action that processes the group file or loads templates:

listener per group

```
STGroup g = ...;
g.setListener(myListener);
g.getInstance("foo");
...
```

If you want to track interpretation errors with a particular listener, use the appropriate ST.write() method:

listener per rendering

```
STGroup g = ...;
ST st = g.getInstance("foo");
st.write(myWriter, myListener);
```

Imported groups automatically use the listener of the importing group.

Model adaptors

StringTemplate lets you access the properties of injected attributes, but only if they follow the JavaBeans naming pattern ("getters") or are publicly visible fields. This works well if you control the attribute class definitions, but falls apart for some models. Some models, though, don't follow to get her method naming convention and so template expressions cannot access properties. To get around this, we need a model adaptor that makes external models look like the kind StringTemplate needs. If object o is of type *T*, we register a model adaptor object, *a*, for *T* that converts property references on o. Given <o.foo>, StringTemplate will ask *a* to get the value of property foo. As with renderers, *a* is a suitable adaptor if "o is instance of *a*'s associated type". For the statically typed language ports, here are the interfaces:

Java

```
public interface ModelAdaptor {
    public Object getProperty(Interpreter interpreter, ST self, Object o, Object
property, String propertyName)
        throws STNoSuchPropertyException;
}
```

C#

```
(TBD)
```

Python

```
(TBD)
```

Property name type
Property names are usually strings but they don't have to be. For example, if o is a dictionary, the property could be of any key type. The string value of the property name is always passed to the renderer by StringTemplate.

Example

Java

```
class UserAdaptor implements ModelAdaptor {
    public Object getProperty(Interpreter interpreter, ST self, Object o, Object
property, String propertyName)
        throws STNoSuchPropertyException
    {
        if ( propertyName.equals("id") ) return ((User)o).id;
        if ( propertyName.equals("name") ) return ((User)o).theName();
        throw new STNoSuchPropertyException(null, "User."+propertyName);
    }
}

public static class User {
    private int id; // ST can't see; it's private
    private String name;
    public User(int id, String name) { this.id = id; this.name = name; }
    public String theName() { return name; } // doesn't follow naming conventions
}

String template = "foo(x) ::= \"<x.id>: <x.name>\"\n";
STGroup g = new STGroupString(template);
g.registerModelAdaptor(User.class, new UserAdaptor());
ST st = g.getInstanceOf("foo");
st.add("x", new User(100, "parrt"));
String expecting = "100: parrt";
String result = st.render();
```

C#

```
(TBD)
```

Python

```
(TBD)
```

Inheriting from ObjectModelAdaptor
You can inherit your ModelAdaptor from ObjectModelAdaptor to leverage its ability to handle "normal" attributes. You can choose to override the results of any given property or to handle properties that would not normally be handled by the default ObjectModelAdaptor.

Example

Java

```
class UserAdaptor extends ObjectModelAdaptor {
    public Object getProperty(Interpreter interpreter, ST self, Object o, Object
property, String propertyName)
        throws STNoSuchPropertyException
    {
  // intercept handling of "name" property and capitalize first character
        if ( propertyName.equals("name") ) return
((User)o).name.substring(0,1).toUpperCase()+((User)o).name.substring(1);
  // respond to "description" property by composing desired result
  if ( propertyName.equals("description") ) return "User object with id:" +
((User)o).id;
  // let "id" be handled by ObjectModelAdaptor
        return super.getProperty(interpreter,self,o,property,propertyName);
    }
}

public static class User {
    public int id; // ST can see this and we'll let ObjectModelAdaptor handle it
    public String name;  // ST can see this, but we'll override to capitalize
    public User(int id, String name) { this.id = id; this.name = name; }
}

String template = "foo(x) ::= \"<x.id>: <x.name> (<x.description>)\"\n";
STGroup g = new STGroupString(template);
g.registerModelAdaptor(User.class, new UserAdaptor());
ST st = g.getInstanceOf("foo");
st.add("x", new User(100, "parrt"));
String expecting = "100: Parrt (User object with id:100)";
String result = st.render();
```

C#

```
(TBD)
```

Python

```
(TBD)
```

Renderers

The atomic element of a template is a simple attribute (object) that is rendered to text by its the appropriate string evaluation method for the port's language (toString, ToString, *_str_*, ...). For example, an integer object is converted to text as a sequence of characters representing the numeric value . What if we want commas to separate the 1000's places like `1,000,000`? What if we want commas and sometimes periods depending on the locale? For more, see The Internationalization and Localization of Web Applications.

StringTemplate lets you register objects that know how to format or otherwise render attributes to text appropriately. There is one registered renderer per type per group. In the statically defined port languages like Java and C#, we use an interface to describe these renderers:

```
public interface AttributeRenderer {
    public String toString(Object o, String formatString, Locale locale);
}
```

To render expression <*e*>, StringTemplate looks for a renderer associated with the object type of *e*, say, *t*. If *t* is associated with a registered renderer, *r*, it is suitable and StringTemplate invokes the renderer method

In Java:

Expression syntax	**How interpreter invokes renderer *r***
<*e*>	*r*.toString(*e*, null, *locale*)
<*e*; format="*f*">	*r*.toString(*e*, "*f*", *locale*)

In C#:

Expression syntax	**How interpreter invokes renderer *r***
<*e*>	*r*.ToString(*e*, null, *locale*)
<*e*; format="*f*">	*r*.ToString(*e*, "*f*", *locale*)

StringTemplate supplies either the default locale, or whatever locale was set by the programmer. If the format string passed to the renderer is not recognized then the renderer should simply call the usual string evaluation method.

To register a renderer, we tell the group to associate an object type with a renderer object. Here's an example that tells StringTemplate to render numbers with an instance of `NumberRenderer` using the Polish locale:

```
String template =
    "foo(x,y) ::= << <x; format=\"%,d\"> <y; format=\"%,2.3f\"> >>\n";
STGroup g = new STGroupString(template);
g.registerRenderer(Number.class, new NumberRenderer());
ST st = group.getInstanceOf("foo");
st.add("x", -2100);
st.add("y", 3.14159);
String result = st.render(new Locale("pl"));
// resulted is " -2 100 3,142 " since Polish uses ' ' for ',' and ',' for '.'
```

StringTemplate matches the types of expressions with the renderers using the "is instance of" relationship. As in this example, we registered a renderer for numbers and StringTemplate used it for subclasses such as integers and floating-point numbers. Here's the renderer definition:

```
/** Works with Byte, Short, Integer, Long, and BigInteger as well as
 *  Float, Double, and BigDecimal.  You pass in a format string suitable
 *  for Formatter object:
 *
 *  http://java.sun.com/j2se/1.5.0/docs/api/java/util/Formatter.html
 *
 *  For example, "%10d" emits a number as a decimal int padding to 10 char.
 *  This can even do long to date conversions using the format string.
 */
public class NumberRenderer implements AttributeRenderer {
    public String toString(Object o, String formatString, Locale locale) {
        // o will be instanceof Number
        if ( formatString==null ) return o.toString();
        Formatter f = new Formatter(locale);
        f.format(formatString, o);
        return f.toString();
    }
}
```

StringTemplate comes with three predefined renderers: DateRenderer, StringRenderer, and NumberRenderer.

StringTemplate FAQ

FAQ - Object models

Altering property lookup for Scala

Sriram Srinivasan contributed this Scala code that lets you directly access scala properties. It's a model adapter that adds a lookup for a method called "foo()" when the property name is "foo". The problem is using classes in scala with the extremely convenient case class:

case class Person(name: String, salary: Double)

This creates a class of the form

class Person { public String name(), public String name(String)}

etc.

Sriram could make scala to generate beans style accessors. For that he would have to do the much uglier:

case class Person(@BeanProperty name: String, @BeanProperty salary: Double)

Anyway, the following code shows how to create an adapter that works better with Scala.

```
import org.stringtemplate.v4._
import org.stringtemplate.v4.misc._
import scala.collection.JavaConversions._
class ScalaObjectAdaptor extends ObjectModelAdaptor {
 @throws(classOf[STNoSuchPropertyException])
 override
 def getProperty(interp: Interpreter, self:ST, o:Object, property:Object,
propertyName:String): Object = {
   var value: Object = null
   val c = o.getClass
   if ( property==null ) {
     return throwNoSuchProperty(c.getName() + "." + propertyName)
   }
   // Look in cache for Member first
   var member = classAndPropertyToMemberCache.get(c, propertyName)
   if ( member == null ) {
```

```
     member = Misc.getMethod(c, propertyName)
   }
   if (member == null) {
     return toJava(super.getProperty(interp, self, o, property, propertyName))
   }
   try {
     member match {
       case m: java.lang.reflect.Method => toJava(Misc.invokeMethod(m, o, value))
       case f: java.lang.reflect.Field => toJava(f.get(o))
     }
   }
   catch {
     case _ => throwNoSuchProperty(c.getName() + "." + propertyName)
   }
 }
 // recursively convert scala collections (and nested collections) to nested java
collections
 def toJava(o: Object): Object = {
   o match {
     case l:List[_] => {
     var lo = l.asInstanceOf[List[Object]]
     // Convert to a Scala list of java objects, then make into an array
     lo.map (elem => toJava(elem)).toArray
     }
     case m:Map[_,_] => {
     // convert map values to java values.
     var om = m mapValues (v => toJava(v.asInstanceOf[Object]))
     // return map as a java hash map
     mapAsJavaMap(om)
     }
     case s:Set[_] => s.asInstanceOf[Set[Object]].map(v => toJava(v)).toArray
     case _ => o // no change
   }
 }
}

/// Test driver.
object STest {
   var template = """
                  |fld(f) ::= "<f.name> .... <f.ty>"
                  |il(i) ::= "'<i>'"
                  |prop(m) ::= "<m.name> @ <m.city>"
                  |clss(cls) ::= <<
                  |structure <cls.clsname> {
                  |  <cls.flds:fld();separator="\n">
                  |  dict[foo] = <cls.dict.foo>
                  |  dict[bar] = <cls.dict.bar>
                  |  <cls.intlist:il(); separator=" ">
                  |  Props: {
                  |    <cls.listmap:prop(); separator="\n">
                  |  }
                  |}
                  |>>""".stripMargin
 def main(args: Array[String]) = {
    var stg = new STGroupString(template)
    stg.registerModelAdaptor(classOf[Object], new ScalaObjectAdaptor())
   case class Fld(name: String, ty: String)
   case class Clss(clsname: String, flds: Set[Fld], dict: Map[String, Int], intlist:
List[Int], listmap: List[Map[String, String]])
```

```
   var flds = Set(new Fld("x", "Int"), new Fld("y", "Float"))
   var dict = Map("foo" -> 1, "bar" -> 2)
   var intlist = List(1,2,3,4,5)
   var listmap = List(Map("name" -> "ter", "city" -> "sf"), Map("name" -> "sriram",
"city" -> "berkeley"))
   var st = stg.getInstanceOf("clss")
   st.add("cls", new Clss("Foo", flds, dict, intlist, listmap))
   println(st.render())
   // Should print
   /*structure Foo {
        x .... Int
        y .... Float
        dict[foo] = 1
        dict[bar] = 2
        '1' '2' '3' '4' '5'
        Props: {
          ter @ sf
          sriram @ berkeley
        }
```

```
    }*/
  }
}
```

ST v4 TODO list

Bigger items

Smallish items

- testEmbeddedSubtemplate fails; it counts newlines in subtemplate in <% %>
- add <^> to ignore all indents?
- check for cyclic template invocation; lint mode
- should adaptors get imported from super group?
- make sure id()::=... same as id.st
- allow render look up import groups? allow per template?
- remove 64k limit on char indexes and bytecode branches? i see (short) around.
- make sure that the GUI can see templates created from maps in group files.
- can gui handle template with empty elements like <if()><endif>? Interval(a,b) might make this hard.