Umber Catalan 0.3.5 User Guide

Chapter 1 - Introduction

Catalan is a data transformation service that specializes in processing input data into arbitrary output formats. XML documents control the conversion process, listing step by step which transformations to invoke, and how to arrange content into a PDF document.

Catalan acts as a bridge between XML and many different custom data formats, including ASCII text, HTML, JavaBeans, PDF, and even other XML formats. The Catalan Transform XML document drives the conversion process, modifying the data in small steps as it passes through the chain of operations, until it emerges in the final format. Catalan provides a comprehensive toolkit of standard operations, but also makes it easy to plug in new operations at the client level. The Catalan PDF engine provides a flexible layout and rendering system able to convert any XML format into PDF documents.

The Bellows library manages the low-level input data, providing tools for converting XML data into its native generic Datum objects, and for accessing nested Datum content through a query language similar to XPath. Catalan makes extensive use of the Bellows Traversal class and NodeProcessor interface.

Chapter 2 - The Transformer

The Transformer class is the front end to all Catalan data processing. Each Transformer object holds a Transform XML specification which declares a set of transform operations to perform on input data. The Transformer can run the data through the entire set of its transforms in order, or it can pick and choose individual transforms by name. This chapter describes how to invoke the Transformer through Java code, and also through the custom Ant Task.

Section 2.1 - Java Interface

The entire Transformer API consists of a constructor and three method calls; most of the work goes into creating the Transform XML documents to drive the process.

```
public Transformer (Datum transform)
public List processAll (List nodes)
public List process (String[] ids, List nodes)
public String[] getProcessorIds ()
```

The constructor accepts the Transform XML specification in Bellows Datum format. Your loading code might look something like this:

```
Reader reader = new FileReader ("transform.xml");
SAXLoader loader = new SAXLoader ();
Datum xml = loader.load (reader);

Transformer transformer = new Transformer (xml);
```

The Transform XML typically takes one of two forms: a self-contained set of operations meant to be run as a single unit, and a non-sequential library of operations that can be run in arbitrary order. The first case uses the processAll() method:

```
List input = getInputNodes ();  // or whatever...
List results = transformer.processAll (input);
```

This sends the entire List of input data nodes through each transform operation. If the loaded Transform XML contained three operations, the processAll() method would send the entire List of input nodes through the first transform, then send the results of that transform through the second transform, and finally send the cumulative results of the previous two transforms through the third operation. The Transformer then returns the fully processed node list from processAll().

The other processing method takes a list of operation identifiers. Each operation will always have a unique id generated by the Transformer, although the identifier can be declared explicitly in the Transform XML, with an 'id' attribute on the operation. The getProcessorIds() method returns an array of all ids in the Transformer. These ids can be used to select which operations to run with the process() method. The example below runs the first three operations, then repeats the second operation twice:

```
// Assume at least three operations exist.
String[] ids = transformer.getProcessorIds ();
String[] ops = { ids[0], ids[1], ids[2], ids[1], ids[1] };

List input = getInputNodes ();  // or whatever...
List results = transformer.process (ops, input);
```

The Transformer also provides a main() method to make it easier to invoke it from the command line. The main() method requires at least two parameters. The first parameter must be a valid Transform XML document. The second and later

parameters are all loaded into the input node queue and run through the Transform XML from the first parameter.

The main() method currently only supports the processAll() method, which means any Transform XML specification it invokes will execute all its transforms from start to end, in order. Later versions might support the process() method too.

The typical way to use the command-line Transformer is to store your Transform XML and any input data inside files (although Transformer can also load content across the network if you use URLs) and send the changes to output files. When main() finishes, any nodes still in the queue will be noted in the log file then discarded.

The following command will run the Transformer with the Transform XML in the file "transform.xml", then load the contents of the file "input.txt", replace all instances of the text "before" with "after", and write the results to the file "output.txt". We pass the three file names as parameters, and let the Transform XML load, manipulate, and save the data.

```
java -cp $CLASSPATH org.writersforge.catalan.transform.Transformer
  transform.xml output.txt input.txt
```

The Transform XML to handle this might look something like the example below. We'll learn more about this XML format in the next chapter.

```
<transform>
  <group start="1">
    <import format="text"/>
    <replace newtext="after" oldtext="before"/>
  </group>
  <export/>
</transform>
```

Incidentally, since the <group> element processes all nodes after the first, you could add as many input files to the end of the command line as you wanted. All of them would be loaded, processed, appended together, and written to the single output file.

```
java -cp $CLASSPATH org.writersforge.catalan.transform.Transformer
  transform.xml output.txt input1.txt input2.txt ...
```

Section 2.2 - Ant Task

Catalan supplies a simple Ant task wrapper around the Transformer class. The <catalan> task can only load text input data which you specify in nested <input> XML elements, although using the <import> operation you can load more complex data from files. The <catalan> task only supports the Transformer.processAll() method, so the Transform XML you feed it will always execute from top to bottom, ignoring any "id" parameters.

To use the <catalan> task in your Ant script, you must first declare it with the Ant <taskdef> task. It also depends on the Bellows and Apache Commons Logging libraries, so you must include those in the <taskdef> classpath along with the Catalan jar file:

```
<taskdef classname="org.writersforge.catalan.util.TransformTask" name="catalan">
  <classpath>
    <pathelement location="lib/commons-logging-1.0.2.jar"/>
    <pathelement location="lib/bellows-0.2.0.jar"/>
    <pathelement location="lib/catalan-0.1.0.jar"/>
  </classpath>
</taskdef>
```

Immediately after the <taskdef>, you can start using the <catalan> task. The "transform" attribute holds the path to the Transform XML file. The following example would load the contents of the file "xml/my-xml2pdf.xml" as Transform XML and perform all of its operations in sequence on the two input data nodes, "data/input.xml" and "data/output.pdf". It is up to the Transform XML to invoke <import> and <export> to load the input.xml file and save the transformed results to the output.pdf file.

```
<catalan transform="xml/my-xml2pdf.xml">
  <input>data/input.xml</input>
  <input>data/output.pdf</input>
</catalan>
```

Each <catalan> task may only include one Transform XML file, but can hold an unlimited number of <input> data nodes. The <input> elements can only contain regular text, and cannot be nested XML.

To make it easier to bulk process files, the <catalan> task supports embedded <fileset> elements, using the standard Ant syntax. Each file it finds will be converted into an <input> element. Thus, if the "stuff" directory contains the files file1.txt, file2.txt, file3.xml, and file4.txt, the task invocation

```
<catalan transform="xml/my-xml2pdf.xml">
  <input>data/input.xml</input>
  <fileset dir="stuff">
    <include name="*.txt"/>
  </fileset>
  <input>data/output.pdf</input>
</catalan>
```

is exactly equivalent to the task below. Only the files ending in ".txt" are included. If any files are later added to the "stuff" directory, they will automatically be included the next time the task is run, without requiring the <catalan> task to be updated.

```
<catalan transform="xml/my-xml2pdf.xml">
  <input>data/input.xml</input>
  <input>stuff/file1.txt</input>
  <input>stuff/file2.txt</input>
  <input>stuff/file4.txt</input>
  <input>data/output.pdf</input>
</catalan>
```

Chapter 3 - The Transform XML Specification

The Transform XML specification is a high level set of Catalan transformation operations. Each operation has a unique XML element name and a set of attributes and child elements to customize its behavior. This chapter examines the XML content of each operation and the resulting effects on the input data as it passes through the transform.

Section 3.1 - Format

The Transform XML document is always contained in a <transform> root element. Each transform operation is an immediate child of the <transform> element, and will receive a unique processor id. Technically, not all operations have to reside at the top level. As we'll see later, the <group> operation embeds other operations so they can be called as a single unit.

Here's a sample transform that changes all occurrences of the String "to" to the String "from", then concatenates all input nodes into a single long String. We'll learn more about the syntax of these operations in later sections.

```
<transform>
  <replace id="op1" newtext="from" oldtext="to"/>
  <concat id="op2"/>
</transform>
```

The following code initializes a Transformer with the above Transform XML specification and runs some input data through it.

```
String xml = "<transform>" +
  "<replace id='op1' oldtext='to' newtext='from'/>" +
  "<concat id='op2'/>" +
  "</transform>";

Reader reader = new StringReader (xml);
SAXLoader loader = new SAXLoader ();
Datum spec = loader.load (reader);
Transformer transformer = new Transformer (spec);

// Load input data.
List input = new LinkedList ();
input.add ("to");
input.add ("button");
input.add ("away");
input.add (new Integer (123));

List results = transformer.processAll (input);
```

The <replace> transform will convert the input data into the nodes "from", "butfromn", "away", and the Integer 123 respectively. The <concat> operation append all of those into the String "frombutfromnaway123", which becomes the single List node in the results variable.

If we wanted to skip the <replace> operation, we could use the process() method instead of processAll():

```
String[] ops = { "op2" };
List results = transformer.process (ops, input);
```

The results variable would instead contain the String "tobuttonaway123".

Section 3.2 - Import and Export

The Transformer does not really care how the data content gets into the input

data queue. If you're initializing it directly with the Java Transformer API, you can take care of loading your data and placing it into the List of input data nodes however you like.

However, an easier approach is to let the Transformer do the work for you. Simply place the file name or a URL to the data content in the queue and run the <import> operation on it. You must specify a 'format' attribute to tell <import> what data type it should try to load the data into. A format of "text" loads the contents into a single String object; a format of "binary" loads the contents into a byte[] array object; and a format of "xml" loads the contents into a Datum tree.

```
<import format="text"/>
```

By default, the <import> operator will replace the file node with the contents of the loaded data. An optional boolean 'keep-orig' parameter tells the operation to keep the file node in the data queue. Thus, given a file of "input.txt" with the contents "Input is great!", the plain import operator above would transform the List of input data ["input.txt"] into the List ["Input is great!"]. Conversely, the operator below with the keep-orig parameter would result in the data queue of ["input.txt", "Input is great!"].

```
<import format="text" keep-orig="yes"/>
```

The final <import> attribute only affects the importing of XML data. The "keep-whitespace" attribute forces the XML loader to keep track of all whitespace in the original XML document. By default, <import> only loads whitespace content when it is in the same element as other PCDATA content; whitespace indentation and line feeds inside element-only content are ignored. This extra whitespace can significantly bloat the resulting Datum tree, and the extra PCDATA nodes can displace any hardcoded element index references you have in your Bellows query paths. Thus, you should only import with strict whitespace when you really need to.

```
<import format="xml" keep-whitespace="yes"/>
```

When you are done with your transformations, you will probably want to store the results in a file somewhere. The <export> operation does this for you. The simple form below uses the first node as the output file, and appends the remaining nodes to the resulting file.

```
<export/>
```

For example, with the input data ["output.txt", "line1 ", "line2 ", "line3 "], the <export> operator will save the contents "line1 line2 line3 " to the file "output.txt" and remove all four nodes from the data queue. If the 'keep-orig' parameter is true, the operator would keep all four nodes in the data queue.

```
<export keep-orig="yes"/>
```

The export operator also lets you explicitly declare an output file in the Transform XML. In this case, the operator sends all data nodes to the output file, and does not consume the first node for a file name. The output example above would result in the file "output2.txt" containing "output.txtline1 line2 line3 " when processed with the operator below:

```
<export file="output2.txt"/>
```

By default, each invocation of <export> will delete the previous contents of the target file. To append content to the end of an existing file without overwriting previous content, use the "append" attribute:

```
<export append="yes"/>
```

Although the command-line processor only allows for String arguments for the input and output files, the import and export processors also accept File, URL, Reader/Writer, and InputStream/OutputStream objects as well. If you are constructing the input node list in Java code, you can pass in whatever is most convenient. For instance, the code below would work the same as the first "output.txt" example:

```
Writer writer = new FileWriter ("output.txt");
List input = new LinkedList ();
input.add (writer);
input.add ("line1 ");
input.add ("line2 ");
input.add ("line3 ");
```

Section 3.3 - Groups

The <group> operator does not perform any special data processing of its own, but rather serves as a container for gathering collections of operations together into a single unit. Groups can be nested to arbitrary depths. As simple aggregators, <group> operators work on all types of data content.

```
<group>
  <replace newtext="new" oldtext="old"/>
  <group>
    <concat/>
    <normalize/>
  </group>
</group>
```

The <group> operator can also function as a node filter, for operations that you only want to affect certain nodes. The simplest way of selecting nodes is by the index position in the node list. You specify a list of one or more indices to process with the "range" attribute. You can specify single nodes by giving their zero-based index number; you can specify a range by giving two index numbers with a hyphen in the middle; and finally, you can join more than one index or range by commas.

To demonstrate how the "range" parameter affects the filtering, we'll pass a set of input data nodes through different <group> operations: ["zero", "one", "two", "three", 4, 5, "six"]. The node set is a combination of String nodes (in quotes) and Integer nodes (without quotes).

In the first case, the range index is a 2. Nothing much happens, since the "two" is simploy concatenated to itself. The output nodes are the same as the input nodes.

```
<group range="2">
  <concat/>
</group>
```

The next case demonstrates an open-ended range, "2-". The selection starts at index two and continues to the last node. The <concat> operator splices all selected nodes into a single String. In this case, the processed node set becomes ["zero", "one", "twothree45six"]. The <group> selected all but the first two nodes.

```
<group range="2-">
  <concat/>
</group>
```

In the next case, we set the end index to 4. This selects all nodes up to and including the node at index 4. The processed node set is ["zeroonetwothree4", 5, "six"].

```
<group range="-4">
  <concat/>
</group>
```

By specifying both start and end, we can take a slice out of the middle. This example results in the node set ["zero", "one", "twothree4", 5, "six"], after selecting nodes 2 through 4.

```
<group range="2-4">
  <concat/>
</group>
```

Finally, you can pick and choose whichever nodes you want by separating each block with a comma. The range below selects nodes 1, 3, and 4. Notice that the contents are inserted in place of the first replaced node, leaving a gap between nodes 2 and 5: ["zero", "onethree4", "two", 5, "six"]

```
<group range="1,3-4">
  <concat/>
</group>
```

You can also filter nodes by which Java class or interface each node implements. Only nodes of the given class are affected. All nodes that don't match the filter are appended to the end, after processing. The example below concatenates all String nodes, and shuffles the two Integer nodes to the end, leading to ["zeroonetwothreesix", 4, 5].

```
<group class="java.lang.String">
  <concat/>
</group>
```

The <group> operator also supports regular expression filters with the "regexp" attribute. This filter compares a stringified copy of each node to the expression in the 'regexp' attribute. Like the "class" filter, the "regexp" filter shuffles unmatched nodes to the end of the list. In this example, the expression selects all nodes starting with a "t" or a "4". Note that it finds the Integer, 4. The results of this operation are ["twothree4", "zero", "one", 5, "six"].

```
<group regexp="[t4].*">
  <concat/>
</group>
```

Finally, you can combine all of the <group> parameters in the same operation. The start and end range is applied first, and the class and regexp filters are applied to each of the selected nodes. This combination does result in some behavior that might seem strange at first, but does actually make sense. The <group> operator places all nodes rejected by the filter(s) at the end of the selected index range, not at the end of the entire node list. The nodes outside the range are not even considered by the filters.

In the example below, the nodes "zero" and "six" fall outside the range, and thus are not affected by any of the embedded operators (in this case, <concat>). Nodes 2 through 5 are sent through the class and regexp filters. The class filter rejects 4 and 5, and the regexp filter rejects "one" and 5. It only takes one rejection to reject a node, so all three of those nodes are shuffled to the end of the selected range. The "two" and "three" nodes pass both filters, and are concatenated together and placed at the beginning of the selected range, resulting in this processed node list: ["zero", "twothree", "one", 4, 5, "six"]

```
<group class="java.lang.String" range="1-5" regexp="[t4].*">
  <concat/>
</group>
```

After the nodes are selected and processed, they must be inserted back into the output node list somehow. The default action is to insert the processed nodes into the output list at the lowest index of all selected nodes. Thus, given the the input list ["old0", "old1", "old2", "old3", "old4", "old5"] and the text replace operation which renames all occurrences of "old" to "new":

```
<group range="1,3-4">
  <replace newtext="new" oldtext="old"/>
</group>
```

by default, the output list would be ["old0", "new1", "new3", "new4", "old2", "old5"]. The "old2" node, which falls in the middle of the selection but is not actually processed, is shuffled to the end, immediately after the re-inserted nodes.

The "action" attribute provides a means to change this default behavior. The default action is "insert". The operation below is exactly the same as the example above:

```
<group action="insert" range="1,3-4">
```

```
  <replace newtext="new" oldtext="old"/>
</group>
```

The "append" action places the processed nodes at the end of the output list; the operation above with an append action would create the output list of ["old0", "old2", "old5", "new1", "new3", "new4"]. The "prepend" action would create an output of ["new1", "new3", "new4", "old0", "old2", "old5"].

If you don't care about the results of your processed nodes (perhaps your processors had some other side effect and you want to ignore their result nodes), you can use the "delete" action to replace the processed nodes with nothing, for example ["old0", "old2", "old5"]. If you want your original list back like it was before the <group> operation, regardless of which nodes you selected, you can use the "revert" action. A revert will insulate your node queue from any changes the <group> operation makes, even in other nested <group> nodes.

Section 3.4 - Text Replacement

Transform XML supports two operations for simple text search and replace: <replace> and <lookup>. The <replace> transform Replaces all occurrences in the input data of the 'oldtext' attribute with the value of the 'newtext' attribute. The example below replaces all hyphen characters with "X" characters. So input data with three text nodes ["a--ple", "------", "e-ample"] is transformed into these three output nodes: ["aXXple", "XXXXXX", "eXample"].

```
<replace newtext="X" oldtext="-"/>
```

The <replace> operation has an optional "count" attribute which limits the number of text replacements it makes per input node. The count resets for each new node. The example below would convert the input data ["a--ple", "------", "e-ample"] to ["aXXple", "XX----", "eXample"].

```
<replace count="2" newtext="X" oldtext="-"/>
```

The <lookup> operation performs variable replacements in the style of UNIX shell script variables. By default, variables take the form of "${variable}". The extra markup is part of the variable, and is removed when the variable is replaced. For example, the text "The ${version} version of ${product}" would become "The 0.1.3 version of Catalan" when run through the transform below.

```
<lookup>
  <var name="product" text="Catalan"/>
```

```
  <var name="version" text="0.1.3"/>
</lookup>
```

The optional "start-token" and "end-token" attributes can be used to define alternate markup for the variables. For example, the text "The [version] version of [product]" would become "The 0.1.3 version of Catalan" when run through the transform below.

```
<lookup end-token="]" start-token="[">
  <var name="product" text="Catalan"/>
  <var name="version" text="0.1.3"/>
</lookup>
```

Section 3.5 - Splitting and Joining

Transform XML also has transform operations for splitting apart and joining together text data nodes. The <tokenize> operation chops up input data anywhere it finds one of the given tokens. The example below would convert the input data ["one,two::three;", ":four:"] into ["one", "two", "three", "four"] by splitting on all commas, semicolons, and colons.

```
<tokenize>
  <token>,</token>
  <token>;</token>
  <token>:</token>
</tokenize>
```

The optional "include-delimiters" attribute tells the tokenizer to include the tokens in the output alongside the other data. The example below would convert the input data ["one,two::three;", ":four:"] into ["one", ",", "two", ":", ":", "three", ";", ":", "four", ":"]. The tokenizer includes each token as a separate output data node, even when two or more of them appear consecutively.

```
<tokenize include-delimiters="yes">
  <token>,</token>
  <token>;</token>
  <token>:</token>
</tokenize>
```

The opposite transform to the tokenizer is the <concat> operation, which combines the specified input data nodes into a single String output node. The concatenator converts any non-String data into String data with the

String.valueOf() method before splicing it all together. Thus, output from a simple <concat> transform will always be a single node with String data.

For example, given the mixed String, Integer, and Datum input data of ["one", new Integer(2), <three/>, <four><five/></four>], the transform below would result in literal String output of ["one2<three/> <four> <five/> </four> "]. The extra whitespace is a by-product of the XML to String conversion.

```
<concat/>
```

The <concat> operator has an optional "count" attribute to limit the number of input nodes it affects. The concatenator converts input data nodes into String form until it reaches the specified node count or runs out of input data. Any unprocessed nodes are passed to the output untouched. For example, given the input data in the previous example, the following example would create output of ["one2<three/> ", <four><five/></four>]. The first three input data nodes are concatenated and the fourth node (the second Datum tree) is passed through as a non-Stringified Datum tree.

```
<concat count="3"/>
```

The "separator" attribute is another optional <concat> property. The separator is inserted between all nodes during the concatenation process. By default, the separator is a blank string, and does not affect the output. Given the operation

```
<concat separator="; "/>
```

and the input data ["one", "two", three"], the output data would be a single String node: "one; two; three". The separator is not inserted at the head or tail of the string, only in the middle.

Section 3.6 - Whitespace Normalization

The <normalize> operation converts all consecutive spans of whitespace into single space characters. By default, the space (" "), tab (" "), and return (" " and " ") characters are considered to be whitespace. The string " white space   " becomes " white space ".

```
<normalize/>
```

The normalizer also works with custom whitespace, resolving all spans of custom tokens into single custom output characters. If any <token> elements are defined, the default whitespace tokens no longer apply. You can also change the

token the whitespace resolves to, with the "resolver" attribute. In the example below, all consecutive spans of space and tab characters will resolve to the '#' character. Thus, the string " white space " would become "#white#space# #".

```
<normalize resolver="#">
  <token/>
  <token> </token>
</normalize>
```

For even more sophisticated normalization, the normalizer offers support for custom exclusion areas, declared with <exclude> elements inside the <normalize> operator. Each exclusion area can be delimited by the recurrence of a single token, with the "delim" attribute, or with different start and end tokens, using the "start-delim" and "end-delim" attributes.

The transform below looks for exclusion areas toggled on and off by the "|" character, and also exclusion areas that start with "[" and end with "]". Outside of the exclusion areas, it collapses all consecutive spans of its lone token character, "-", into single "X" characters. Thus, it would convert the text "--one--|---two--|--three----[--four-five---]--six" into "XoneX|---two--|XthreeX[--four-five---]Xsix".

```
<normalize resolver="X">
  <token>-</token>
  <exclude delim="|"/>
  <exclude end-delim="]" start-delim="["/>
</normalize>
```

Section 3.7 - Fixed-Length ASCII

To facilitate the parsing and creation of fixed-length ASCII or binary data, Transform XML offers the <to-ascii> and <from-ascii> operations.

The <to-ascii> processor packs simple Java objects into a packed ASCII data string according to the field specification implemented by the AsciiFieldManager helper class. It does its best to convert the input data objects into the field types in the spec. Any input data that doesn't fit in the spec are passed through, untouched. Input data too long to fit into its field will be clipped, which unfortunately may result in data loss. No input data nodes are consumed while processing padding fields.

The ASCII field specification is a series of field type and size declarations. Each field contains a mandatory numerical length and type, and an optional array count. The field length determines how many ASCII characters of data correspond to a single element of that field data. The array count determines how many consecutive element fields exist in that slice of the ASCII data. Whitespace in the field spec is completely ignored.

The field type must be one of six categories:

Type	Description	Java Type
x	padding	N/A
b	byte	java.lang.Byte
c	character	java.lang.Character
i	integer	java.lang.Integer
s	string	java.lang.String
f	float	java.lang.Double

Non-array field specs are simply the field length and the type. Thus, the field spec "4s 3i" declares seven characters of ASCII data; the first four characters make up a String object and the final three characters are converted into an Integer object. Thus, the ASCII data "1234567" would yield a String value of "1234" and an Integer value of 567. This association works both ways, so conversely a String field of "nope" and an Integer of 34 would pack into ASCII data "nope34 ".

Array field specs provide an easy way to load large chunks of uniformly sized data fields into a single Java array object. To declare an array spec, append the array size to the field's spec, surrounded by square brackets. For example, a field spec of "2s 2i[4]" declares one two-character String and four two-character Integer objects. With this field spec, the ASCII data "0123456789" would unpack into a String of "01" and an Integer[] array of [23, 45, 67, 89].

The <to-ascii> operator looks for this field spec in the "spec" attribute. For example, given the input data [12345, "one", "two", "three", "four"], the processor and field spec below would result in output data of ["1234 onetwothr ", "four"]. The default padding for 'x' fields is the space character; the "four" node does not fit into the field spec, so it is just passed through to the output.

```
<to-ascii spec="4i 2x 3s[3] 3x"/>
```

The optional padding attribute lets you change the default padding. The padding string is repeated across all padding fields. Given the input data from the previous example, the transform below would produce output data of ["1234ABonetwothrCDA", "four"].

```
<to-ascii padding="ABCD" spec="4i 2x 3s[3] 3x"/>
```

The <from-ascii> operator goes the opposite direction, exploding one or more ASCII fragments into Java Objects and arrays. The operator processes each input node separately, applying the entire field spec to each node. For example, given the input data ["1234--onetwothrxxx"] and the transform below, the output data would be [1234, ["one", "two", "thr"]]. The padding "--" and "xxx" are completely ignored when going from ASCII to Object. Thus, the input data ["1234..onetwothrABC"] would result in exactly the same output data.

```
<from-ascii spec="4i 2x 3s[3] 3x"/>
```

Section 3.8 - Objects and XML

Transform XML provides two operations for converting between Datum XML trees and raw Java objects: <build-xml> for transforming objects into a Datum tree, and <explode-xml> for splitting a Datum tree into Java objects.

The <build-xml> processor channels input data nodes into an XML structure, based on a push/pop stack of formatting directives. The <start-element/> element pulls the next input data node, converts it to a string, and uses that as the element name. The <end-element/> directive closes the current element. It's possible to nest elements to arbitrary depths. The <attribute/> element pulls the next two input nodes, using the first for the attribute name and the second for the attribute value. Finally, the <pcdata/> element appends the current input node to the PCDATA content of the current element.

```
<build-xml>
  <start-element/>
  <pcdata/>
  <attribute/>
  <attribute/>
  <end-element/>
</build-xml>
```

Given the transform above, the input data ["one", "two", "three", "four", "five", "six"] processed by the below transform would create a Datum tree

corresponding to the XML:

```
<one five="six" three="four">two</one>
```

The <pcdata> directive grabs the "two" node for element content, and each of the <attribute> directives grab a pair of input nodes. If the input data contains more nodes than the <build-xml> transform uses, the extra nodes will be copied directly to the output, after the Datum tree.

By default, the <build-xml> transform pulls all of its non-markup content from the input data nodes. However, it is possible to override that content with static text inside the transform. The element name can be set with the 'name' attribute; the attribute content can be set with the 'name' and 'value' attributes; and PCDATA content can be set by simply including it as PCDATA in the <pcdata> element. Statically set values do not consume input data. For example:

```
<build-xml>
  <start-element name="staticroot"/>
  <pcdata/>
  <attribute name="attr1"/>
  <attribute name="attr2" value="staticval"/>
  <pcdata>--</pcdata>
  <pcdata/>
  <end-element/>
</build-xml>
```

Input data of ["one", "two", "three", "four", "five", "six"] processed with the above transform would result in the XML content

```
<staticroot attr1="two" attr2="staticval">one--three</staticroot>
```

followed by output data of ["four", "five", "six"]. The unused input nodes are passed straight through to the output.

The <explode-xml> processor performs the inverse operation and decomposes Datum XML trees into component Java objects. The <query> child elements select which parts of the XML document to operate on; each <explode-xml> processor can contain more than one query, and queries can be nested inside of each other. Nested queries act upon the set of Datum objects selected by the parent query, with a relative path.

Inside the query, commands select the content to place in the output. The <property> command looks up the named XML attribute in all selected elements.

The <type> command places the current element name in the output. The <datum> command copies the Datum object itself into the output. The <int>, <string>, and <float> commands place static data nodes into the output. Those commands result in Integer, String, and Double objects, respectively.

```
<explode-xml>
  <query path="root/child">
    <property name="prop1"/>
    <property name="prop2"/>
    <int value="1"/>
    <string value="two"/>
    <float value="3.3"/>
    <query path="child/*[@use=yes]">
      <type/>
      <property name="id"/>
    </query>
  </query>
  <query path="root/child[2]">
    <datum/>
  </query>
</explode-xml>
```

Section 3.9 - JavaBeans and XML

Another helpful feature of Transform XML is the ease at which it translates between live object data in JavaBeans and Datum XML content. It provides many high level style hints for generating various styles of XML. The mapping between XML and JavaBean is fairly direct and clean, and does not match the Java XML Peristence specification used by Jdk1.4's XMLEncoder class and Bellow's BeanLoader and BeanUnloader classes.

The <xml-to-bean> operation maps an XML document into a JavaBean instance. It consumes two input nodes: the JavaBean class, as either a String or a Class instance; and a Datum tree. The converter will do its best to recursively load the XML data into the JavaBean, matching element and attribute names to JavaBean properties. It supports many different naming styles for both elements and attributes, e.g., "my-bean", "my_bean", "MY-BEAN", "MY_BEAN", "MyBean", and "myBean". It initially looks for primitive JavaBean properties as child element PCDATA content. If the property does not exist as a child element, the operator will search for the same property as an attribute of the parent element.

```
<xml-to-bean/>
```

The loader will ignore all content that does not map to JavaBean properties. Because of this, as long as the property names don't overlap, the same XML document can polymorphically be loaded into more than one type of JavaBean instance.

Conversely, the <bean-to-xml> operator generates an XML Datum tree from a JavaBean instance, converting JavaBean properties into XML-style element names, e.g., "my-bean", "my-bean-property". By default, it creates all properties as nested child elements.

```
<bean-to-xml/>
```

The optional "collapse" attribute tells the <bean-to-xml> operator to store all of its primitive JavaBean properties as attributes instead of elements. It Still creates child elements for complex properties like nested JavaBeans. Thus, when the "collapse" attribute is absent or set to "false", all JavaBean properties will be stored in child elements; if "collapse" is set to "true", the XML will contain a mix of attributes and elements.

```
<bean-to-xml collapse="true"/>
```

By default, the <bean-to-xml> operator converts JavaBean property names from the form "myBeanProperty" to hyphen-separated lowercase element and attribute names, e.g., "my-bean-property". The optional "naming-style" changes this naming style to use alternate conventions. The examples below represent the styles "my-bean", "my-bean", "my_bean", "MY-BEAN", "MY_BEAN", "MyBean", and "myBean", respectively.

```
<bean-to-xml naming-style="default"/>
<bean-to-xml naming-style="lower-hyphen"/>
<bean-to-xml naming-style="lower-underscore"/>
<bean-to-xml naming-style="upper-hyphen"/>
<bean-to-xml naming-style="upper-underscore"/>
<bean-to-xml naming-style="case-delim"/>
<bean-to-xml naming-style="javabean"/>
```

Section 3.10 - Converting Text to XML

One of the disadvantages to using XSLT is that you can only use it on XML documents. Catalan addresses this issue by keeping the processing architecture and API agnostic to the type of input and output data it will allow. It's up to

the processor in each transform stage to decide which objects to place back in the queue. This flexibility makes it possible to create operators which can convert between XML and other data types. In this section we will discuss an easy way to convert text a document with simple formatting constraints into a full-fledged XML document.

The official text-to-XML conversion operator is <text-to-xml>. It uses a simple XML specification to divide the text content into easily accessible parts. The operator has one optional "convert" parameter which determines whether or not to convert XML data into Datum content. The "convert" parameter defaults to "no", which loads XML content in your source ASCII document as PCDATA content.

```
<text-to-xml/>
<text-to-xml convert="yes"/>
```

The operator breaks the text document into chapters and sections according to single-line titles with different underline characters in the following line.
The "~" underline should only appear at the top of the document, and denotes the top-level <document> title; the "=" character declares the beginning of a <chapter> element; and the "-" underline character declares a new <section>. The text of the title is stored in a "title" attribute of its corresponding XML element. The titles can be flush left, indented, or centered, but must be surrounded by blank lines.

Free-flowing text must always be aligned to the far left, i.e., not indented.
Java, XML, and other example text should all be indented at least one space. The example below shows how the operator works. (Note that since this user guide is itself processed by <text-to-xml>, the title examples must be "protected" from XML-izing, by prefixing each line of the sample text document with a "|" character.)

INPUT:

```
|         Document Title
|         ~~~~~~~~~~~~~~
|
|          Chapter Title
|          =============
|
| This is the intro.
|
| Section 1 Title
```

```
| ---------------
|
| This is line one of the first paragraph.
| This is line two of the first paragraph.
|
|   This is example text.
|
| This is the second paragraph.
|
|   <root>
|     <child/>
|     <child/>
|   </root>
|
| Section 2 Title
| ---------------
|
| This is the second section.
```

OUTPUT:

```
<document title="Document Title">
  <chapter title="Chapter Title">
    <section>
      <text>This is the intro.</text>
    </section>
    <section title="Section 1 Title">
      <text>This is line one of the first paragraph.
  This is line two of the first paragraph.</text>
      <example>This is example text.</example>
      <text>This is the second paragraph.</text>
      <xml>
        <root>
          <child/>
          <child/>
        </root>
      </xml>
    </section>
    <section title="Section 2 Title">
      <text>This is the second section.</text>
    </section>
```

```
  </chapter>
</document>
```

The "convert" parameter will determine whether the contents of any generated <xml> elements are loaded as String data in a PCDATA node, or as a Datum tree inside the <xml> node.

Section 3.11 - HTML Markup

HTML is very similar to XML, and because of this, any HTML viewer will fail to display elements it does not support. If your viewable HTML data includes any XML content, you must alter the XML markup so the HTML viewer will recognize it as data, not markup. This typically amounts to converting content like this:

```
<root>
  <child/>
</root>
```

by expanding all "<" characters into the special literal XML entity "<", like this:

```
&lt;root>
  &lt;child/>
&lt;/root>
```

The content no longer looks like HTML elements, so the HTML viewer will display it. The <xml-to-html> operator performs this for you. If the input data is a String, it will perform a text search/replace on all "<" characters and return the result.

```
<xml-to-html/>
```

If the input data is a PCDATA Datum node, it will change the PCDATA contents of the Datum node and return the same Datum node. If the input data is a Datum tree, but the root node is not PCDATA, the operator will convert the tree into String data, run the search/replace on it, then pack it into a new PCDATA node. This helps keep the data type consistent: If the input data is a String, the output will be a String; if the input data is a Datum, the output will be a Datum.

As a pure convenience, Transform XML has another operator for prettying up java code with HTML markup. It marks up Java identifiers in bold dark blue, string literals and char data in red, numbers in blue, and comments in green. It works

on String data and directly on PCDATA Datum nodes.

```
<java-to-html/>
```

Section 3.12 - XML Manipulation

In addition to a JavaBean-XML mapping, Transform XML provides an extensive set of operations for XML-XML mappings, for in-place modifications of Datum XML content. All of these operators use the prefix "xform" in their operator name:

```
<xform-insert/>
<xform-delete/>
<xform-copy/>
<xform-move/>
<xform-rename/>
<xform-wrap/>
<xform-inline/>
<xform-to-element/>
<xform-to-attribute/>
<xform-style/>
```

The first five operators perform simple structural changes, without modifying the data content itself. They can alter the location or name of both elements and attributes, and can make additional copies of existing content.

The <xform-insert> operator evaluates the Bellows query in the "select" attribute (or defaults to the current node if "select" is omitted), then creates a copy of its static inlined element content for each of the matched query nodes in the target XML document. The example below would place a copy of the full <newContent> element, including any attributes, into all <child> elements immediately inside the base <root> element of the input tree. As with all xform processors, all non-Datum content is passed through to the output untouched.

```
<xform-insert select="root/child">
  <newContent>
    <subContent1/>
    <subContent2 prop="value"/>
  </newContent>
</xform-insert>
```

The insert operation can also serve as a templating mechanism when its "expand" attribute is set to "yes" or "true". In this case, the processor will

recursively search the static insert contents for variables of the format "${query}" in PCDATA (not in attributes). Wherever it finds an expandable query, it will run the query against the current root node of the input XML document -- not the selected target nodes -- and insert a copy of that data in place of the query.

For example, assume the query "root/data/stuff" results in two empty <stuff/> elements. The insert operation

```
<xform-insert expand="yes" select="root/child">
  <newContent>${root/data/stuff}</newContent>
</xform-insert>
```

would place the XML content below into each target <child> node:

```
<newContent>
  <stuff/>
  <stuff/>
</newContent>
```

The expandable queries can be nested as deeply in the static content as you like; the processor will resolve all of them it finds.

The insert operator can also insert attribute content by specifying the name of the attribute to create in the "attribute" attribute and the attribute contents in the "value" attribute. The transform will create static attributes in all matched query nodes. This example would create a 'newProp' attribute with the value of 'newValue' on all selected <child> elements.

```
<xform-insert attribute="newProp" select="root/child" value="newValue"/>
```

The <xform-delete> operator completely removes all selected elements or attributes from the XML document. When given with just a "select" attribute, it will delete element contents. This example removes all <child> elements inside the <root> element.

```
<xform-delete select="root/child"/>
```

With an additional "attribute" attribute, the transform deletes the named attribute from all selected elements. The transform below removes the 'origProp' attribute from all selected <child> elements.

```
<xform-delete attribute="origProp" select="root/child"/>
```

The <xform-copy> operator creates a new copy of the selected elements or attributes at each element that the 'dest' query selects. This example would make copies of all selected <child> elements and put them into each selected <to> element. If more than one destination node is selected, the processor will make multiple copies of the same source elements. The transform does not alter the original content, e.g., the 'root/child' nodes below.

```
<xform-copy dest="root/to" select="root/child"/>
```

To copy attributes instead of elements, simply add an "attribute" attribute. If more than one attribute is copied into the same destination node, the second and later attributes are mangled to keep the attribute names unique, by appending numbers to the duplicated attributes. Thus, if the example below matches three 'origProp' attributes in the selected <child> elements, the processor will create the attributes 'origProp', 'origProp2', and 'origProp3' in each destination element.

```
<xform-copy attribute="origProp" dest="root/to" select="root/from/child"/>
```

The <xform-move> operator moves element or attribute content to other parts of the XML tree. It behaves exactly like the copy processor, except it deletes all the source nodes. If the destination selects more than one node, the source nodes will be copied separately to each destination node.

```
<xform-move dest="root/to" select="root/from/child"/>
```

As with the other structural operators, add the "attribute" attribute to affect attributes instead of elements. In this case, the move operator will move attributes to other elements in the XML tree. It behaves exactly like the attributes copy processor, except it deletes all the source attributes. If the destination selects more than one attribute, the attributes will be copied separately to each destination node, with any necessary attribute name mangling.

```
<xform-move attribute="origProp" dest="root/to" select="root/from/child"/>
```

The copy and move operations both append their content to the end of any pre-existing content in the target elements. The "position" attribute provides a mechanism to place the copied or moved content into the beginning or middle of the target elements. All new content is wedged into the child array at the specified position. Thus, given a copy operation that selects two <child> nodes, and a destination <to> node that looks like this:

```
<to>
```

```
  <old0/>
  <old1/>
  <old2/>
</to>
```

and the following copy operation with a position index of 1:

```
<xform-copy dest="root/to" position="1" select="root/child"/>
```

the resulting <to> node would look like this:

```
<to>
  <old0/>
  <child/>
  <child/>
  <old1/>
  <old2/>
</to>
```

The <xform-rename> operator renames all selected elements to the new name. The example below would rename all selected <child> elements to <newChild>.

```
<xform-rename new-name="newChild" select="root/child"/>
```

Adding "attribute" causes the operation to rename attributes in all selected elements. The example below would rename all 'oldProp' attributes in the selected <child> elements to 'newProp'.

```
<xform-rename attribute="oldProp" new-name="newProp" select="root/child"/>
```

The <xform-wrap> operator is a specialized combination of the insert and move operations that wraps the selected elements with a newly created wrapper element. In the example below, the processor would place all selected <child> elements into <child-wrap> elements, without losing their place in the <root> element. Thus, after the transform, the same <child> elements could be selected with a query of 'root/child-wrap/child'.

```
<xform-wrap select="root/child" wrapper="child-wrap"/>
```

In the example below, the transform wraps each immediate <child> descendent of <root>, but ignores the unselected <other> node, and the nested <child> node.

INPUT:

```
<root>
  <child/>
  <child/>
  <other/>
  <child>
    <child/>
  </child>
</root>
```

OUTPUT:

```
<root>
  <child-wrap>
    <child/>
  </child-wrap>
  <child-wrap>
    <child/>
  </child-wrap>
  <other/>
  <child-wrap>
    <child>
      <child/>
    </child>
  </child-wrap>
</root>
```

The <xform-inline> operation performs the reverse of the wrap operation. It decreases element nesting, removing all selected elements without deleting the child content of those elements. Essentially, this is a non-recursive delete. All inlined content is inserted in place; if an inlined element has more than one child, all children will be inserted into the parent where the former inlined element was. This may offset the index counts of later elements. All attributes in the inlined elements are lost. Thus, the following transform would convert the input data into the sample output data below:

```
<xform-inline select="root/child"/>
```

INPUT:

```
<root>
  <child>
    <grandchild1/>
```

```
      <grandchild2/>
    </child>
    <other/>
    <child>
      <grandchild3/>
    </child>
  </root>
```

OUTPUT:

```
  <root>
    <grandchild1/>
    <grandchild2/>
    <other/>
    <grandchild3/>
  </root>
```

The <xform-to-element> operation converts selected attributes into PCDATA elements. For each of the selected nodes, the processor will move the requested attribute into a child element, placing the content into PCDATA inside the element. In the example below, an element "<child prop='value'/>" would become "<child><prop>value</prop></child>". Elements without the attribute will not be altered.

```
  <xform-to-element attribute="prop" select="root/child"/>
```

The optional "new-name" attribute can specify an element name other than the attribute name. The version of the transform below would result in "<child><myprop>value</myprop></child>" when run against the example above:

```
  <xform-to-element attribute="prop" new-name="myprop" select="root/child"/>
```

The <xform-to-element> operation also supports the "position" attribute in the same manner as <xform-copy> and <xform-move>.

The <xform-to-attribute> operation is not an exact inverse of the to-element operation. The to-element operator maps exactly one attribute to exactly one element, whereas the to-attribute operator maps potentially more than one element to a single parent attribute, depending on how many child elements it finds in any given parent element. It extracts all PCDATA from all selected elements and appends it together into a single string, then assigns it to the named attribute. The entire content of all nodes becomes one attribute, and any attributes in the

selected nodes are lost. The transform below demonstrates this.

```
<xform-to-attribute attribute="prop" select="root/child"/>
```

INPUT:

```
<root>
  <child child-prop="child-prop-value">CHILD1</child>
  <notChild>NOT-CHILD</notChild>
  <child>CHILD2</child>
</root>
```

OUTPUT:

```
<root prop="CHILD1CHILD2">
  <notChild>NOT-CHILD</notChild>
</root>
```

The final xform operator, <xform-style>, recursively converts the selected elements and all their attributes into the requested naming style. It uses the same styles as the <bean-to-xml> transform above. An optional select parameter specifies which branches to convert; if the select query is omitted, the processor will convert the entire tree. The style operator is the only xform operator with an optional "select" attribute.

This transform will convert the style for all "root/child" elements and attributes and all their children into the JavaBean naming convention, regardless of what naming style they had before.

```
<xform-style new-style="javabean" select="root/child"/>
```

In addition to the styles supported by the <bean-to-xml> operator, the <xform-style> operator also responds to the styles "collapsed" and "expanded". These two new styles correspond to the <bean-to-xml> "collapse" attribute. The "collapsed" style will recursively convert all elements which contain only PCDATA into attributes of the same name in the parent element. Elements which contain other element content will not be converted.

```
<xform-style new-style="collapsed" select="root/child"/>
```

The "expanded" style performs the reverse, recursively converting the attributes of all selected nodes into PCDATA elements.

```
<xform-style new-style="expanded" select="root/child"/>
```

Section 3.13 - Auto-Numbering

Auto-numbering is another convenience feature in Transform XML. The <xform-tag> operation recursively counts and assigns section numbers to selected elements in a Datum tree. Embedded <path> elements select each level of numbering with any legal Bellows query.

The example below works on the root <document> node, numbering each <chapter> element, then separately numbering all <section> elements inside each selected <chapter>, prepended with the parent chapter's number. By default, the tag is stored in a "number" attribute on each tagged element, and the nested numbers are delimited by a "." character.

```
<xform-tag select="document">
  <path>document/chapter</path>
  <path>chapter/section</path>
</xform-tag>
```

The example above would perform the following transformation:

INPUT:

```
<document>
  <chapter>
    <section/>
  </chapter>
  <chapter>
    <section/>
    <section/>
    <section/>
  </chapter>
</document>
```

OUTPUT:

```
<document>
  <chapter number="1">
    <section number="1.1"/>
  </chapter>
  <chapter number="2">
    <section number="2.1"/>
```

```
    <section number="2.2"/>
    <section number="2.3"/>
  </chapter>
</document>
```

The <xform-tag> lets you override the name of the generated attribute, the index delimiter, and the number to start counting at:

```
<xform-tag attribute="sectnum" select="root/document" separator="-" start-index="2">
  <path>document/chapter</path>
  <path>chapter/section</path>
</xform-tag>
```

This alternate example would produce the output below when run against the same input data:

OUTPUT:

```
<document>
  <chapter sectnum="2">
    <section sectnum="2-1"/>
  </chapter>
  <chapter sectnum="3">
    <section sectnum="3-1"/>
    <section sectnum="3-2"/>
    <section sectnum="3-3"/>
  </chapter>
</document>
```

Section 3.14 - Selecting XML Nodes

The <xml-select> operator is a container similar to the <group> operator, except that it runs its embedded operations on elements inside a Datum tree. For each input Datum node, it runs the Bellows query in the optional "select" attribute (defaulting to the entire tree if the "select" attribute is absent) to determine which XML content it should process. It than passes the selected Datum content to each of its embedded operators.

Like the <group> operator, <xml-select> operators can be nested to arbitrary depths. The "select" attribute always acts on the nodes selected from the parent operator. Because of this, the select statements of nested operators will seem to overlap a bit.

In the example below, the outer <xml-select> operator passes each of its <child> sub-elements to its two nested operators: the first <xform-rename> operator, and the inner <xml-select> operator. The inner <xml-select> runs the "child/grandchild" query separately on each <child> element, and passes each <grandchild> to the second <xform-rename> operator. This makes it possible to iterate through an XML tree, performing different operations on different branches of the XML content.

```
<xml-select select="root/child">
  <xform-rename new-name="new-child"/>
  <xml-select select="child/grandchild">
    <xform-rename new-name="new-grandchild"/>
  </xml-select>
</xml-select>
```

Section 3.15 - Velocity Template Processing

To facilitate code generation and more generic templating, Catalan provides an easy to use wrapper around the Jakarta Velocity template engine. The <velocity> operation is flexible about which parameters should be statically defined and which should come from the input queue. This includes the template itself, as well as the parameters fed to it. In general, if a parameter is defined in <velocity>, that value will be used, but if it's missing, the processor will pull it from the input queue as needed.

The simplest form uses a Velocity template declared inside the operation and a single key value, "node". See the official Jakarta Velocity docs for a full description of how the templates work.

```
<velocity>
  <template>($node)</template>
  <key name="node"/>
</velocity>
```

This operation will send each input node through the template, passing the node to the Velocity engine as the template parameter "$node". Velocity will then resolve all the variable references inside the template that it can, and pass the results to the output. For example, given the input node list ["one", "two"], the template above would create an output list of ["(one)", "(two)"].

The <velocity> operation will pull as many input data nodes as it needs to fill

in the missing key values. If the template had two keys, it would use up two input nodes for each template pass.

```
<velocity>
  <template>($node1 $node2)</template>
  <key name="node1"/>
  <key name="node2"/>
</velocity>
```

If the input list ["one", "two"] was passed through the operation above, the op would only produce a single output node, ["(one two)"]. A list of ["one", "two", "three", "four"] would produce an output of ["(one two)", "(three four)"]. The op requires a value for every key/value pair. If it doesn't have enough input nodes to fully load all keys, it will fill in the missing nodes with blanks. Thus, the input list ["one", "two", "three"] would produce an output of ["(one two)", "(three)"].

It's also possible to pull the Velocity template from the input queue. To do this, simply leave off the <template> element. The <velocity> operation will grab the first node in the input queue and use that as the template. It will use the same template for more than one pass, if there are enough input nodes.

```
<velocity>
  <key name="node1"/>
  <key name="node2"/>
</velocity>
```

Given this, the input queue of ["($node1 $node2)", "one", "two", "three", "four"] would load the template from the first node, and iterate twice over the value pairs in the rest of the queue, giving an output of ["(one two)", "(three four)"].

You can also pass key names in through the input list. The op below would consume three input nodes per pass. The first goes to the node1 value, the second to the second key name, and the third to the second key value.

```
<velocity>
  <key name="node1"/>
  <key/>
</velocity>
```

With this template, an input of ["($node1 $xxx)", "one", "xxx", "two"] will

result in an output of ["(one two)"]. This is particularly useful for dynamically generated templates.

The <key> parameters are freshly pulled from the input list each time the template is run. However, sometimes it is useful to set key parameters once, and share them across all template passes. The <global> element does this, acting just like a <key> except it is only assigned once, when the <velocity> operation is initially loaded.

```
<velocity>
  <template>($global $local)</template>
  <global name="global"/>
  <key name="local"/>
</velocity>
```

Given the op above, the first input node goes to the $global key, and the rest go to the $local key, one node per template pass. The input list ["one", "two", "three"] would result in an output list of ["(one two)", "(one three)"], with the node "one" reused each pass as the $global parameter. When the <template> element is not specified, the <velocity> operation will always grab that first, followed by any <global> parameters, leaving the rest for template iterations.

The <velocity> operation also has special support for handling XML documents. By default, a Bellows Datum XML object passed to the op will behave like a normal Java object which you can invoke methods on inside the template, just like any Velocity parameter. However, this does not make it easy, or even possible, to use the Bellows query engine. Also, if you are generating Java source code from an XML source document, there is often no way to convert XML content into valid Java identifiers, especially when hyphens are present.

Catalan provides the DatumAdapter wrapper class to make all of this easier. If you set the "wrap-datum" attribute to "true" or "yes", the <velocity> op will wrap every Datum it finds in a DatumAdapter object, even those returned by Bellows queries.

```
<velocity wrap-datum="yes">
  <key name="xmlroot"/>
</velocity>
```

In general, the DatumAdapter provides a more Velocity-friendly API for each Datum XML node. You can always get a reference to the original Datum object with the getRoot() method.

```
public Datum getRoot ()
```

For example, consider the XML document below loaded by the previous <velocity> operation:

```
<my-root id="the-root">
  <child id="child1"/>
  <child id="child2"/>
  <child id="child3"/>
</my-root>
```

When the XML content is loaded as a Datum tree and passed to the template, it can be accessed through the "$xmlroot" parameter. The Velocity template "$xmlroot.getRoot().getType()" would resolve to the root element's name, "my-root". Furthermore, Velocity allows parameterless JavaBean property accessors to be expressed in a shorthand form, as just the property name. Thus, the template above could also be expressed as "$xmlroot.Root.Type".

You can call other methods against the raw Datum object from the template. For example, this template resolves to "the-root":

```
$xmlroot.Root.getProperty("id")
```

Unfortunately, when you grab a Datum attribute value directly, it comes in whatever format it occurs in the XML document. The root element above is "my-root". If you wanted to use that to generate a Java identifier, for example a class name, the source code would not compile because the hyphen is not a valid character for an identifier. You would end up with a class of "my-root", not the legal hyphenless "MyRoot" form.

DatumAdapter makes use of the Bellows helper class, PropertyName, for easy translations between various property naming styles. PropertyName supports the API below; the getPropertyName() method returns the original property name and the other methods return the same property name in alternate formats. See the Bellows API documentation for more info.

```
public String getPropertyName ()
public String getCaseDelimitedName ()
public String getBeanPropertyName ()
public String getLowerHyphenatedName ()
public String getLowerUnderscoredName ()
public String getUpperHyphenatedName ()
```

```
public String getUpperUnderscoredName ()
```

The DatumAdapter wrapper offers alternatives to the Datum.getType() and Datum.getProperty() methods which each wrap the normal return value in a PropertyName object:

```
public PropertyName getType ()
public PropertyName getProperty (String property)
```

To solve the earlier class naming problem, you might use a template of "$xmlroot.Type.CaseDelimitedName". This would return the "my-root" element name as a PropertyName object, which you could then convert to "MyRoot" with getCaseDelimitedName(). If you wanted to express it as a constant field, "MY_ROOT", you could use a template of "$xmlroot.Type.UpperUnderscoredName".

The DatumAdapter also gives you some convenience methods for extracting PCDATA content from a Datum node:

```
public String getPcdata ()
public String getPcdata (boolean collapseWhiteSpace)
public String getSafePcdata ()
```

The first getPcdata() method grabs all PCDATA from the current element and any of its children, leaving all whitespace intact. This is just a wrapper around the DatumBrowser.extractPcdata() method in Bellows. The second getPcdata() method allows you to specify whether you want the whitespace collapsed or not.

The final method, getSafePcdata(), returns a quotable version of the PCDATA, suitable for including in quoted strings in Java. It strips out all line breaks and extra whitespace and escapes all raw quote (") characters with backslashes. For example, this template assigns the PCDATA content to the Java reference "text":

```
String text = "$xmlroot.SafePcdata";
```

If the non-safe PCDATA accessor was used, the following XML would result in an uncompilable Java class:

```
<my-root>This text
   is multiline
   with unescaped &quot; characters</my-root>
```

The generated content would look like this, an obvious error:

```
String text = "This text
 is multiline
 with unescaped " characters";
```

The getSafePcdata() version looks like this instead:

```
String text = "This text is multiline with unescaped " characters";
```

Another critical Bellows feature that DatumAdapter wraps is the Bellows query engine. This lets you select and filter XML data inside a Bellows tree with a simple text query string. For example, the query "root/child" would select all <child> elements who are immediate children of the top-level <root> element. The query "root/child[@id=child1]" would select only those <child> elements that have an "id" attribute set to the value "child1".

The query() method in DatumAdapter runs a query against the wrapped Datum object and returns the results. Those results are also wrapped in DatumAdapter objects so you don't lose the extra wrapper functionality for queried nodes.

```
public DatumAdapter[] query (String query)
```

DatumAdapter provides two other convenience query methods to make templating with XML content even easier. The queryPcdata() method runs a query, then extracts the PCDATA from each returned node. The queryProperties() method runs a query then looks up the requested attribute on each returned Datum object, wrapped inside a PropertyName object.

```
public String[] queryPcdata (String query)
public PropertyName[] queryProperties (String query, String attribute)
```

To demonstrate the simplicity of templating from XML content, consider the following example.

```
<my-root name="the-root">
  <child name="child-prop1"/>
  <child name="child-prop2"/>
  <child name="child-prop3"/>
</my-root>
```

Suppose we wanted to create JavaBean accessor methods for each child element. We could run the query "my-root/child" to get an array of the <child> elements, then use the #foreach Velocity command to iterate through them, creating a chunk of code for each one. Assuming the XML content was sitting in the input node as a

Datum tree, we could invoke <velocity> like this:

```
<velocity wrap-datum="yes">
  <key name="xmlroot"/>
</velocity>
```

The template for the entire Java class file might look like this:

```
public class $xmlroot.Type.CaseDelimitedName
{
#foreach ($child in $xmlroot.query("my-root/child"))
    private String _$child.getProperty("id").BeanPropertyName;
#end

#foreach ($child in $xmlroot.query("my-root/child"))
#set ( $prop = $child.getProperty("id") )
    public String get$prop.CaseDelimitedName ()
    {
        return _$prop.BeanPropertyName;
    }

public void set$prop.CaseDelimitedName (String $prop.BeanPropertyName)
{
    _$prop.BeanPropertyName = $prop.BeanPropertyName;
}

#end
```

The generated Java source would look like this:

```
public class MyRoot
{
    private String _childProp1;
    private String _childProp2;
    private String _childProp3;

public String getChildProp1 ()
{
    return _childProp1;
}

public void setChildProp1 (String childProp1)
{
```

```
        _childProp1 = childProp1;
    }

    public String getChildProp2 ()
    {
        return _childProp2;
    }

    public void setChildProp2 (String childProp2)
    {
        _childProp2 = childProp2;
    }

    public String getChildProp3 ()
    {
        return _childProp3;
    }

    public void setChildProp3 (String childProp3)
    {
        _childProp3 = childProp3;
    }
```

Section 3.16 - SQL Parsing

Catalan offers a simple custom operation <sql-bean> for converting SQL declarations into Java objects. Currently only "CREATE TABLE" SQL is supported. Sending DDL through <sql-bean> will result in one instance of an SqlTable object for each TABLE in the DDL. The SqlTable object lets you extract fields, SQL types, Java types, key fields, and more, through a simple JavaBean API. See the Catalan SqlTable API documentation for more details.

```
<sql-bean/>
```

By default, the <sql-bean> operation returns the basic Java types as Object wrappers (e.g., Integer, Float). If you want it to convert SQL types into primitive types whenever possible (e.g., int, float), call it with the "use-primitives" attribute set to "yes" or "true":

```
<sql-bean use-primitives="yes"/>
```

The SqlTable object is particularly useful in combination with the Velocity processor, for converting DDLs into other formats like XML or Java source code.

Section 3.17 - PDF Generation

XML content residing in the data processing pipeline can be organized and converted into a PDF file using the <pdfgen> operation. It attempts to convert Datum trees into a binary PDF file in the form of a byte[] array. The specification for mapping XML to PDF is rather complicated, and is the sole topic of the final two chapters in this manual.

The basic format of the <pdfgen> operation is:

```
<pdfgen>
  <layout>
    <style-map/>
    <region-map/>
    <document/>
  </layout>
</pdfgen>
```

See later chapters for details on how to populate this operation.

Section 3.18 - Debugging

Although the <debug> operator does not affect the nodes it processes in any way, it exhibits a useful side effect that may end up saving you many hours of fiddling with a debugger. The <debug> operator sends stringified versions of each node it hits to the logger (which might be stdout, log4j, or the Jdk1.4 logger, among others).

In its simplest form, <debug> logs each node to the "debug" channel:

```
<debug/>
```

The "log-level" attribute lets you specify a different logging channel; this attribute can have the values: "trace", "debug", "info", "warn", "error", and "fatal". The exact behavior of each channel will depend on which logging backend you are using.

```
<debug log-level="info"/>
```

To distinguish between multiple calls to <debug>, you can decorate the log message with one or both of the optional "prefix" and "postfix" attributes:

```
<debug postfix="]" prefix="AFTER TEXT REPLACE ["/>
```

If the simple String node "data contents" were passed through the default <debug/> operation, it would log the same String, "data contents". However, with the pre/postfix example above, the debugger would log the String "AFTER TEXT REPLACE [data contents]".

To send a subset of input data nodes to the log, you can wrap the <debug> operator inside a <group> operator, like this:

```
<group class="java.lang.String">
  <debug/>
</group>
```

This operation would log only data nodes which were String objects, passing over any Integer, Datum, or other class types without logging them.

The <debug> operator can also be used to send static log messages during a transform. Whenever you set the "message" attribute, the debug logger will write out the static log text exactly once, and will not log node data. The "log-info" attribute works in this case, but the "prefix" and "postfix" attributes are ignored, since they are static text too.

Common uses for static logging are to add logging checkpoints to the transform process, and to delimit node dumps. The example below logs the string "BEFORE", then logs each node, then logs the string "AFTER":

```
<debug log-info="trace" message="BEFORE"/>
<debug log-info="trace"/>
<debug log-info="trace" message="AFTER"/>
```

Section 3.19 - Custom Transforms

Sometimes you need to perform a transformation not easily covered by the standard Transform XML specification. Catalan provides an easy way to add your custom processors to the Transform XML specification, or to create your own Transform XML mapping from the ground up.

The core Transform XML mapping, from XML to Java processor classes, resides in the ProcessorRegistry class. The registry builds its default mapping from a simple XML document in the default-registry.xml file in the Catalan jar file. The file looks like this (for clarity, only the first three entries are duplicated here):

```
<registry>
  <bean-to-xml class="org.writersforge.catalan.transform.BeanToXml"/>
  <build-xml class="org.writersforge.catalan.transform.ObjectToDatum"/>
  <concat class="org.writersforge.catalan.transform.Concatenator"/>
    ...
</registry>
```

Each entry in the registry maps an XML element to a Java class. For example, the first entry above tells the registry to invoke the BeanToXml processor class every time it sees a <bean-to-xml> operation in the Transform XML.

The default ProcessorRegistry constructor loads the default-registry.xml mapping:

```
public ProcessorRegistry()
```

You can create an entire mapping from scratch, for example to rename the bundled Catalan processors or to create a mapping from your own custom processors. Load your XML mapping into a Datum tree and pass it to the alternate constructor:

```
public ProcessorRegistry (Datum baseMap)
```

Once the initial registry exists, you can add further layers of mappings with the two "add" methods:

```
public void addOverride (Datum map)
public void addFallback (Datum map)
```

Both of these methods add <registry> mappings just like the original one. If none of the new mappings contain duplicate elements of any of the old mappings, the two methods are identical. All mappings are merged into a single registry. However, if a conflict arises between XML element names, the addOverride() method will replace the existing mappings with its own; the unobtrusive addFallback() method will discard the new ones in favor of the old ones.

The ProcessorRegistry class also contains two other methods for performing mapping operations, but they are primarily used internally by the Transformer class:

```
public Class lookup (String opName)
public NodeProcessor createProcessor (Datum xmlOp)
```

Let's say you have a custom processor that you want to register, so you can use it in alongside the standard Transform XML. First, create the registry, then

load the registry XML into it with the addOverride() method:

```
String xml =
  "<registry>" +
    "<my-proc class='org.mystuff.MyProcessor'/>" +
  "</registry>";
Datum myMap = DatumReader.fromXml(xml);

ProcessorRegistry registry = new ProcessorRegistry();
registry.addOverride(myMap);
```

The registry now contains the full default mapping, plus your custom <my-proc> mapping. The next step is to hook up the Transformer to the registry. You must do this when you initially create the Transformer object; the Transformer only reads the registry when it is first instantiated, so if you make changes in the registry object after creating your Transformer, those changes will be ignored.

The constructors for Transformer look like this:

```
public Transformer (Datum transform)
public Transformer (Datum transform, ProcessorRegistry registry)
```

The first one loads the Transform XML using the default registry with no custom additions. To add custom mappings, you must use the second constructor:

```
Transformer transformer = new Transformer (transformXml, registry);
```

All mapped classes in the registry must somehow extend the BaseNodeProcessor class. Any classes that do not properly extend this base class will be ignored by the ProcessorRegistry, and thus the Transformer. All processor classes used by a single Transform XML document are created and initialized when the Transformer object is created. Each Transform XML operation is passed in entirety to one of the processor's constructors; thus, the processor can use any XML attributes and child elements in the transform operation to initialize itself.

The ProcessorRegistry is responsible for creating each processor from its corresponding fragment of Transform XML. It dynamically searches for constructors in the following order:

```
1) public MyProc (Datum, ProcessorRegistry)
2) public MyProc (Datum)
3) public MyProc ()
```

If the processor has a constructor with a Datum and ProcessorRegistry parameter, it will pass the Transform XML into the first parameter and itself into the second parameter. This constructor is primarily used for the <group> and <xml-select> operations, to provide a callback path to the registry for instantiating embedded operations.

Most processors will implement one of the other constructors. The second constructor takes only the Transform XML fragment. If a processor doesn't need any initialization data, it can simply rely on the default constructor.