

# Analyze with XSLT, Part 1: Analyze non-XML data with XSLT

Create string parsing routines to convert documents into XML elements

Skill Level: Introductory

[Chuck White](mailto:chuck@tumeric.net) ([chuck@tumeric.net](mailto:chuck@tumeric.net))  
XSLT consultant and Web engineer  
Freelance Developer

16 Dec 2003

This tutorial explores how to create string parsing routines in XSLT so that you can tokenize straight, non-XML text, thus turning that text into a series of XML elements. Specifically, this tutorial examines how to convert such documents as weblogs and Web configuration files into XML for improved readability and programmatic access.

## Section 1. Introduction

### Should I take this tutorial?

This tutorial is the first of multi-part series that details the benefits of using XSL Transformations (XSLT) through the example of a fictional research team known as MindMap. The tutorial explores how to create string parsing routines in XSLT so that you can tokenize straight, non-XML text, thus turning that text into a series of XML elements. Specifically, this tutorial examines how to convert documents such as weblogs and Web configuration files into XML for improved readability and programmatic access.

Developers should have a basic understanding of XML in general (you can get a

basic grounding in XML itself through the [Introduction to XML](http://www.ibm.com/developerworks/edu/x-dw-xmlintro-i.html) tutorial, <http://www.ibm.com/developerworks/edu/x-dw-xmlintro-i.html>), and be familiar with basic XSLT tasks. The more knowledge you have of XSLT, the better, but you should be able to follow the tutorial with a less than intermediate-level knowledge of XSLT. If you don't know XSLT at all, try the [Create multi-purpose Web content with XSLT](http://www.ibm.com/developerworks/edu/x-dw-xwebxslt-i.html) tutorial first (<http://www.ibm.com/developerworks/edu/x-dw-xwebxslt-i.html>).

In addition, you'll certainly find the last section on regular expressions using XSLT 2.0 easier to follow if you have some background in regular expressions, but this isn't a prerequisite for successfully completing the tutorial.

## What is this tutorial about?

Welcome to the MindMap Research Team's explorations into advanced XSLT. XSLT 1.0 is much more powerful than a lot of people realize, and developers often use JavaScript to do things XSLT can do effectively. The MindMap Research Team uses XSLT as part of its analytical toolbox for interpreting data related to cognitive processes. The goal is to maximize the team's flexibility by leveraging the ability to change algorithms and methods without having to recompile applications. The MindMap team contracts with universities to provide a variety of highly customized software-based analytical tools, and as such it deals with several aspects of the analytical process, from data acquisition to visualization to tying the data in to other resources. This tutorial examines how to turn a basic text file into XML and manipulate its data. Future tutorials will examine more complex scenarios, including detailed analysis, conversion of data to SVG, and binding data to Web services.

To turn a text file into XML, the MindMap team considered two alternatives. First they considered Perl, but because they preferred a pure XML solution they chose XSLT. They were using XSLT anyway to do the analysis that you'll be learning about in future tutorials in this series. This tutorial demonstrates how, like the MindMap team, you can roll your own **tokenizer**. A tokenizer is simply a software routine that lets you create XML elements out of non-XML documents. (For example, you can create a tokenizer using Perl.) In this tutorial, you'll use XSLT to accomplish the following:

- Pass log or configuration data into an XML document
- Convert a comma-delimited text file to XML elements
- Build string search routines using XPath 1.0 functions
- Parse string data to display specific substring-based information
- Turn your string search routines and string parsers into re-usable templates

- Find and use generic string templates, as well as extensions to EXSLT that enable you to perform calculations on nodes and process strings with less code than basic XSLT
- Use regular expressions through XPath 2.0

As an added bonus, I've included some general [XSLT tricks of the trade](#) in the tutorial summary section.

## Tools

Make sure you install and test the following tools before beginning the tutorial:

- An **XML parser** (<http://xml.coverpages.org/publicSW.html#xmlToolsTOC>) and an **XSLT processor** (<http://www.xml.com/pub/a/2001/03/28/xsltmark/results.html>). This tutorial assumes you have and know how to use each of these.
- A **text editor** that supports **UTF-8** (<http://www.cl.cam.ac.uk/~mgk25/unicode.html>) or an **XML editor** ([http://www.garshol.priv.no/download/xmltools/cat\\_ix.html#SC\\_XMLEditors](http://www.garshol.priv.no/download/xmltools/cat_ix.html#SC_XMLEditors)).
- For XSLT 2.0 examples, **Saxon** (<http://saxon.sourceforge.net/>), an **XSLT 2.0-compliant processor** (<http://www.w3.org/TR/xslt20/>).
- Optionally, a **Perl interpreter** for running the initial example (<http://ftp.linux.cz/pub/perl/ports/>).
- Finally, you'll want to download [dataxsltsourcecode.zip](#) to obtain the files referred to throughout this tutorial. You should download them before starting the tutorial so that you can follow along more easily, because you'll often encounter code fragments taken from longer files and it helps to view those fragments from the context of the overall file.

---

## Section 2. Passing log data into an XML document

### The problem space

The research that the MindMap team does requires a lot of machines, so the team needs to keep track of all the servers on its network. Currently, the team keeps all of the machine information in a text file. This text file serves as a configuration file. The

file is hard to read and difficult to program against, so the team has decided to port the file to XML.

You'll probably find that this kind of need arises often in your own endeavors. For example, at times when you need to tokenize (convert to XML elements) a comma-delimited list. Of course, you can use Excel to convert a comma-delimited list to XML, but then you're stuck with a lot of proprietary Microsoft namespaces. OpenOffice (see [Resources](#)) is a little better because its spreadsheet application at least converts comma-delimited text using open source-based standards, but it still uses lots of namespaces. Here, you'll see how to tokenize a long string of text -- or, in other words, take that string of text and create XML elements out of it based on a set of patterns.

## XSLT requires XML

To create a transform using an XSLT processor, you must work with XML data. This is because an XSLT processor expects well-formed XML. Text files, of course, aren't generally well-formed XML (although curmudgeons will probably e-mail me and say that XML files are text files, but you know what I mean). Therefore, this project -- turning straight textual data into XML -- seems doomed from the start. It isn't. You just need to give the project a little nudge. To do that, you simply need to find a way to wrap the text within at least one XML element. In this case, I'll use Perl, since it's so common in open source environments. If you don't have Perl installed on your machine, simply use a text editor to manually wrap the text within one root element named `WebConfig`.

## Creating a root element by hand

For the purposes of this tutorial, it doesn't matter how you create your XML file. The easiest way to get started is to take the `WebConfig.txt` file, (see [Tools](#) to download the file), and wrap it in a `WebConfig` element. This means you need to put a `WebConfig` tag at the very top of the file, and a `WebConfig` closing tag at the very end, as shown below (note also the XML declaration at the top of the document):

```
      <?xml version="1.0"?>
<WebConfig>
physical
FirstHost(sandbox_acct_app, sandbox_acct_app, sandbox_funkyApps1, ResignalCount=1,
delay=0, regional=0, hostaddress=sandbox.cust.qa.tumeric.net, port=1809)
...
</WebConfig>
```

The actual text file is considerably longer than this, but you get the idea.

## Creating a root element in a production environment

In a production environment, the process described in [Creating a root element by hand](#) may or may not work for you. If you have a lot of files to process, it obviously won't. If you just have one or two, it's not a big deal to hand edit a text file. But what if you have hundreds of files? Maybe you need to process all your weblogs using techniques that are similar to what I use in this tutorial. In that case, you'll need to come up with a way to run your files in a batch. One easy, open source method for this is to use Perl. (You may have other methods in mind; for example, Java technology has tokenizing libraries available.)

```
use strict;
# Open the webconfig.txt file for input
open(CSV_FILE, "webconfig.txt") ||
die "Can't open file: $!";

# Open the webconfig.xml file for output
open(XML_FILE, ">webconfig.xml") ||
die "Can't open file: $!";

# Print the initial XML header and the root element
print XML_FILE "<?xml version=\"1.0\"?>\n";
print XML_FILE "<WebConfig>\n";

while(<WebConfig >) {
    my @xml = <CSV_FILE>;
    print XML_FILE @xml;
}

# Close the root element
print XML_FILE "</WebConfig>";
# Close all open files
close CSV_FILE;
close XML_FILE;
```

Run this script by following the instructions in your Perl command line interpreter. These all vary depending on your interpreter (the basic difference will lie in the parameters you send to your interpreter).

---

## Section 3. Converting a comma-delimited text file to XML elements

### Tokenizing comma delimiters

As I mentioned earlier, tokenizing is the process of creating XML elements out of

something -- in this case, that something is a comma-delimited text file. Any kind of delimiter can be used to tokenize, and you're not limited to one kind of delimiter, which gives you a lot of flexibility in how your transform looks.

If you think about it, delimiters are perfectly suited to XML documents. If you focus on the text that lives in between the delimiters, you'll see that you already have XML elements -- without the markup and without the element names. All you have to do is provide the names. For MindMap to create a more understandable host configurations system, they want to match each line of text in a `configshost.txt` file (available as part of the sample code download from [Tools](#)) with a corresponding element that indicates the machine described on that line of text and that machine's properties. For this reason, you're ultimately going to call your main element's `machine` elements. But I'm going to cheat a little here (for reasons that will become obvious in a moment) and call this first batch of elements `value` elements. I'll refine this process as the tutorial moves along.

## Breaking apart the text based on delimiters

Remember in the previous section how I mentioned that delimiters actually create elements for you? The first step in the process is to name your delimiter. Which delimiter you choose will help you determine how your XML is set up. For example, you could simply create an element for every group of characters between commas. This kind of choice would result in a very granular document, so you would store each group of characters that live between commas in a `value` element. You'll see how to do that in the next panel, [The `xsl:param` element](#).

## The `xsl:param` element

The key to managing just about any serious string processing in XSLT is the `xsl:param` element and its kissing cousin, the `xsl:with-param` element. `xsl:param` is used to possibly store an existing value (I say possibly because it may not contain an initial value), while `xsl:with-param` is used to pass a new value into the parameter as the XSLT processor wends its way through the document tree. That way, you can say to the processor, "The first time I speak to you, I want you to give me the value of all the characters that exist before the first comma you find. Then, I want all the values after the comma. Keep doing this until the end of the document."

Suppose you have a template named `parseCommas` which contains an `xsl:param` element named `entities`. Forget for a moment what you want to do with the template. You know this much for sure: You need to process the entire document, so it's pretty safe to say you should start with this template:

```
<xsl:template match="WebConfig">
  <data>
    <xsl:call-template name="parseCommas">
      <xsl:with-param name="entities" select="." />
    </xsl:call-template>
  </data>
</xsl:template>
```

This tells the processor that you'll be processing the `parseCommas` template with the `entities` parameter and that the `entities` parameter's value will be the entire `WebConfig` element.

## Creating the comma-delimiter template

Keeping in mind what I said in [The `xsl:param` element](#) about giving the processor very specific instructions, the next step is to tell the processor that you want all of the data before the first comma. Of course, that alone is easy to do, but you'll need to do this in such a way that as the processor walks the document tree, a new value set is passed through the `parseComma` template. To start, create a shell template:

```
<xsl:template name="parseCommas">
  <xsl:param name="entities" />
</xsl:template>
```

Next, create a variable to store substrings that exist before a comma:

```
...
<xsl:template name="parseCommas">
  <xsl:param name="entities" />
  <xsl:variable name="data" select="substring-before($entities,',')" />
</xsl:template>
...
```

Next, the MindMap team will make the template recursive.

## Making the `parseCommas` template recursive

A recursive template basically just calls itself, like functions do in other languages. To avoid a stack overflow in the processor, you need a termination statement. This tells the template when to stop calling itself; if you don't provide the termination statement the template keeps calling itself over and over again, forever, like an infinite loop in other languages, hence the stack overflow. So, within your `parseCommas` template, you'll actually call the `parseCommas` template. Crazy, huh? Eventually, if you build the template correctly, there will be no more characters in your test string, or, more specifically, in the parameter you named at the outset when you passed the entire document to the `entities` parameter. This will dictate

your termination statement. That's what I love about XSLT -- when you want to tell the processor something, such as "when there is actually some character data in a named string, do something," you almost speak English to it:

```
<xsl:template name="parseCommas">
  <xsl:param name="entities" />
  <xsl:variable name="data" select="substring-before($entities,',')" />
  <xsl:choose>
    <xsl:when test="string-length($data) > 0">
      <value>
        <xsl:value-of select="$data" />
      </value>
    ..
  </xsl:when>
```

See how you've used one of XPath's trusty string functions to test for a string length? You'll often see this in XSLT string processing, so hold onto it. In fact, XPath offers a whole bevy of nice string functions that you'll encounter frequently in XSLT string processing.

## Finishing your comma parsing template

So now, your template looks like this:

```
<xsl:template name="parseCommas">
  <xsl:param name="entities" />
  <xsl:variable name="data"
    select="substring-before($entities,',')" />
  <xsl:choose>
```

In [Making the parseCommas template recursive](#), I mentioned that a recursive template requires a termination statement. This is shown below:

```
    <xsl:when test="string-length($data) > 0">
      <value>
        <xsl:value-of select="$data" />
      </value>
```

As the processor walks through the tree, each step passes the next group of characters:

```
    <xsl:call-template name="parseCommas">
      <xsl:with-param
        name="entities"
        select="substring-after($entities,',')" />
    </xsl:call-template>
  </xsl:when>
  <xsl:otherwise>
```

Note here that `otherwise` means if there is text but no commas after it, you still want to process it. From this point on you only have one chunk left to do:

```

    <value>
      <xsl:value-of select="$entities" />
    </value>
  </xsl:otherwise>
</xsl:choose>
</xsl:template>

```

You can see here how I've instructed the processor to simply jump from one comma to the next, and told it to process each chunk of characters in between each comma.

## Calling the template and viewing the results

At the beginning of the tutorial, I showed you how to call the template. Since it was called within the `WebConfig` template, you will need to also process that template. In XSLT, you can't get much easier than this:

```

<xsl:template match="/">
  <xsl:apply-templates select="WebConfig" />
</xsl:template>

```

The results can be seen in the file named `StepOneResult.htm`. This file, along with the complete XML file, is available as part of the source code package in [Tools](#). Here's a part of it:

```

<data>
<row>
<value>
physical
FirstHost(sandbox_acct_app</value>
<value> sandbox_acct_app</value>
<value> sandbox_funkyApps1</value>
<value> ResignalCount=1</value>
<value> delay=0</value>
<value> regional=0</value>
<value> hostaddress=sandbox.cust.qa.tumeric.net</value>
<value> port=1809)
ArchiveHost(sandbox_arc_recent_app</value>
<value> sandbox_arc_recent_app</value>
<value> sandbox_funkyApps1</value>
<value> ResignalCount=1</value>
<value> delay=0</value>
<value> regional=0</value>
<value> hostaddress=sandbox.cust.qa.tumeric.net</value>
<value> port=1809)
OrigArchiveHost(sandbox_d_legacy_app</value>
<value> ScubaHostPhysical)
<!-- results truncated -->
</value>

```

```
</row>
</data>
```

Do you see anything wrong with these results? Actually, nothing is really wrong with them. The results are exactly what you called for in the routines. However, these routines weren't quite enough to get the job done. You know how to parse comma-delimited lists, but there's more work to be done. You can download the initial XSLT document -- use the source file in [Tools](#) or create your own.

---

## Section 4. Building string search routines using XPath 1.0 functions

### Searching for strings

To cook up some really good string parsing, you'll need to learn to do a few things. One of these is searching for strings. This is actually very easy at a basic level. XPath defines a function called `contains()`. Whenever you need to find a specific string, you simply name what you're looking for in the `contains()` function. The function takes two arguments: The first is the string you wish to search (usually you'll simply name a node), and the second is the string you're searching for. So if you want to find out if the `SearchFoo` element contains the string `Foo`, you would do this:

```
contains(SearchFoo, 'Foo')
```

### Replacing strings

Here's the `contains()` function in action. The next template is similar to the `parseCommas` template; the fundamental difference is that in this template you're searching for a specific string of text and doing something with it. In this case, you're replacing it. Using the carriage returns as delimiters, you'll literally replace those carriage returns with XML elements using the following named template, which will be called from elsewhere in your code (or from some other code):

```
<xsl:template name="br-replace">
  <xsl:param name="text"/>
  <xsl:variable name="cr" select="'&#xa;'"/>
  <xsl:choose>
```

If the value of the `$text` parameter contains a carriage return (obtained through the `contains()` function), the XSLT processor will return the substring of `$text` before the carriage return:

```
<xsl:template name="br-replace">
...
<xsl:when test="contains($text,$cr)"><machine><xsl:value-of
select="normalize-space(substring-before($text,$cr))"/></machine>
...
  <xsl:call-template name="br-replace">
    <xsl:with-param name="text" select="substring-after($text,$cr)"/>
  </xsl:call-template>
</xsl:when>
```

## Using the `normalize-space()` function

It's a good idea to use the `normalize-space()` function when processing strings of text, especially when your output is XML, to avoid things like carriage returns appearing when you don't want them to.

In the previous code example, you probably noticed that the `normalize-space()` function was used to deal with some whitespace issues. In fact, you may have noticed in that one white space appeared in front of several elements' first character in the first template (the `parseCommas` template). The `normalize-space()` function collapses white space so that no gaps are between the characters named in the argument: `normalize-space('someString')`.

## Creating separate elements for each machine

You may want to test for other kinds of line breaks, depending on the systems you are working with. Here, I've created `machine` elements that surround the `value-of` element. As in the earlier example, the named template is processed recursively:

```
...
  <xsl:call-template name="br-replace">
    <xsl:with-param name="text" select="substring-after($text,$cr)"/>
  </xsl:call-template>
</xsl:when>
<xsl:otherwise>
  <xsl:value-of select="$text"/>
</xsl:otherwise>
</xsl:choose>
</xsl:template>
```

The results of this transform, in highly truncated form, look something like this:

```
<data>
```

```
<machine>FirstHost(sandbox_acct_app, sandbox_acct_app, sandbox_funkyApps1,
ResignalCount=1, delay=0, regional=0, hostaddress=sandbox.cust.qa.tumeric.net,
port=1809)</machine>
<machine>ArchiveHost(sandbox_arc_recent_app, sandbox_arc_recent_app,
sandbox_funkyApps1, ResignalCount=1, delay=0, regional=0,
hostaddress=sandbox.cust.qa.tumeric.net, port=1809)</machine>
<machine>OrigArchiveHost(sandbox_d_legacy_app, sandbox_d_legacy_app,
sandbox_funkyApps1, ResignalCount=1, delay=0, regional=0,
hostaddress=sandbox.cust.qa.tumeric.net, port=1809)</machine>
...
```

Click on the `StepTwoResult.htm` file (part of the download package available in [Tools](#)) to see the results of this transform. The entire XSLT file is also available as part of this download.

## The tale of the unhappy client

Unfortunately, the MindMap team is still not pleased. The problem is that although each machine has notation by way of the `machine` element, all the other information for each machine is within that element as well, leaving you with one long string. Although this kind of element is not very helpful at this point, it's a start:

```
<machine>ArchiveHost(sandbox_arc_recent_app, sandbox_arc_recent_app,
sandbox_funkyApps1, ResignalCount=1, delay=0, regional=0,
hostaddress=sandbox.cust.qa.tumeric.net, port=1809)</machine>
```

You can probably figure out that the machine name is `ArchiveHost`, but that's about it. In the next section, [Parsing string data to display specific substring-based information](#), you'll find out how to break down the data some more.

---

## Section 5. Parsing string data to display specific substring-based information

### Using substring functions to extract text

In much the same way that you can walk up and down an XML tree when processing templates, you can also walk along a string, back and forth, continually cutting it down until it eventually runs out of characters. This can come in handy because, as you've seen, your result data so far has not been particularly satisfying. One of the keys to accomplishing this string walkathon is by using your old friend, the `xsl:with-param` element, which allows you to select more than one delimiter

for tokenizing:

```
<xsl:template name="tokenize">
  <xsl:param name="input" select="'" />
  <xsl:param name="delimiters" select="'" />
  <xsl:call-template name="delimiter">
    <xsl:with-param name="string" select="$input" />
    <xsl:with-param name="delimiters"
      select="$delimiters" />
  </xsl:call-template>
</xsl:template>
```

The `delimiters` parameter picks two delimiters that will be used for tokenizing elements into `value` elements.

## Using delimiters to tokenize

By chopping up the `delimiters` parameter, you can selectively choose when to tokenize. You can do this from within the shell of a brand new `delimiter` template by declaring a variable named (strangely enough) `delimiter`:

```
<xsl:template name="delimiter">
  <xsl:param name="string" />
  <xsl:param name="delimiters" />
  <xsl:variable name="delimiter"
    select="substring($delimiters, 1, 1)" />
</xsl:template>
```

This starts you off by selecting only the first delimiter, the "(" character. The `substring()` function is your tool here. In case you're not familiar with the `substring()` function, it's an XPath function that selects a portion of a string (hence the name *substring*). It takes two mandatory arguments and one optional argument:

- The string from which to extract the substring (mandatory)
- The starting point (mandatory)
- The string length, or how many characters your new substring should consist of (optional)

So, for example `substring("phrase", 1, 1)` calculated for the string "phrase" would return the `p` character, and `substring("phrase", 2, 3)` would return the characters `hra`.

Now that you've named your delimiter in a string and chopped it up once (to give you the "(" character), you'll create a cascading parameter set to move on to the other part of the string (the ")" character). First, you'll need to do two things:

- Figure out a way to keep track of the mess
- Determine your initial input string

## A tip for keeping track of parameters

Before you go too far with this -- and because keeping track of parameters can be a challenge -- you may find it useful to have a tool for managing them. One way to follow a parameter's journey through a node tree is to use an `xsl:copy-of` element at the point in the XSLT document where you wish to know the value of something. Simply create a literal result element and give it a sensible name -- `trace` seems like a pretty good one to me -- then, place a `copy-of` statement within the `trace` element:

```
<xsl:if test="not(starts-with($string, $delimiter))">
  <xsl:call-template name="delimiter">
    <xsl:with-param name="string"
      select="substring-before($string, $delimiter)" />
    <xsl:with-param name="delimiters"
      select="substring($delimiters, 2)" />
  </xsl:call-template>
  <trace>
    <xsl:copy-of select="$delimiters" />
  </trace>
</xsl:if>
```

This will give you output similar to the following:

```
<MachineName> port=1809</MachineName>
<trace></trace>
<MachineName>
ArchiveHost</MachineName>
<trace>( )</trace>
```

## Getting your initial value from an input string

At first, the `input` parameter consists of all the text in the file. This is established a bit earlier in the stylesheet, under the template defined for the `WebConfig` element:

```
<xsl:template match="WebConfig">
  <data>
    <xsl:call-template name="tokenize">
      <xsl:with-param name="input" select="text()" />
    </xsl:call-template>
  </data>
</xsl:template>
```

The `input` parameter starts life out as the entire text node of the `WebConfig`

element. This might seem initially confusing, because if you look at the `tokenize` template, from which the `delimiter` template is called, it appears that the `string` parameter is given an initial value that consists of an empty string. And, indeed, it has:

```
<xsl:template name="tokenize">
  <xsl:param name="input" select="" />
  ...
```

However, the `tokenizer` template is itself called by the `WebConfig` template, which passes the entire node into the `input` parameter:

```
<xsl:template match="WebConfig">
  <data>
    <xsl:call-template name="tokenize">
      <xsl:with-param name="input" select="." />
    </xsl:call-template>
  </data>
</xsl:template>
```

Keep in mind that in XSLT, parameter values don't actually change, even though they seem to. What is actually happening is that as the XSLT processor walks the tree, the **result tree fragment** that's generated by a specific XSLT instruction changes. This is an important distinction, because you don't actually assign values to variables or parameters in XSLT, you merely express what portion of the tree they represent at any given moment in the XSLT processor's journey through the XML document tree.

So the trick in passing parameter values around is to remember the starting point of the XSLT processor when a parameter is initially declared. Keeping track of a parameter's position in a node tree as the XSLT processor does its thing can be a bit of a trick -- one that is best mastered through practice.

## Passing the input string

As the initial input string is passed along through the set of cascading parameters, it continues to get shorter as each strand between each pair of parentheses is stripped out. You'll get to that next, but first, check out the power of the `contains()` function by doing a test for strings containing "Host", then give those strings an element named `MachineName`:

```
<xsl:choose>
  <xsl:when test="not($delimiter)">
    <xsl:if test="contains($string, 'Host')">
      <MachineName>
        <xsl:value-of select="normalize-space($string)" />
      </MachineName>
    </xsl:if>
  </xsl:choose>
```

```
...
```

This will identify each machine. Now you're ready for the set of cascading parameters.

## Recalling the parseComments template

If the input string does not contain any of the characters in the "( )" sequence of characters, then it's safe to process the `parseComments` template, which you may recall from [Passing the input string](#) and which appears in this portion of the tutorial as well:

```
...
  <xsl:call-template name="parseCommas">
    <xsl:with-param name="entities" select="$string" />
  </xsl:call-template>
</xsl:when>
```

After all, it is a useful tool -- the problem was that the original input string needed some pre-processing, which you've now done.

## Applying a trace using a variable

Here is a good opportunity to try the little trick I mentioned earlier -- you know, the one about keeping track of things with a variable. This is a variation on the same theme:

```
<xsl:when test="contains($string, $delimiter)">
  <xsl:variable name="test" select="$string" />
  ...
```

If you don't have a debugging tool, you'll need to take the extra step of writing out a `value-of` statement. This will give you the position of the processor's **cursor** (to borrow a term from the database world) in the node tree as it iterates through the text during the cascade. This will also help you troubleshoot problems as they arise. The `when` statement above simply wants to know if the test string contains one of the two delimiters. It will, of course, since the string consists of all the text in the `WebConfig` node.

## Using cascading parameter values

Now the real cascading action begins, as first the processor tests to see if the string value at this point starts with either the "(" or the ")". If not, the delimiter template is

called using the substring before the first instance of one of the delimiters.

```

...
<xsl:if test="not(starts-with($string, $delimiter))">
  <xsl:call-template name="delimiter">
    <xsl:with-param name="string"
      select="substring-before($string, $delimiter)" />

    <xsl:with-param name="delimiters"
      select="substring($delimiters, 2)" />
  </xsl:call-template>
</xsl:if>
<xsl:call-template name="delimiter">
  <xsl:with-param name="string"
    select="substring-after($string, $delimiter)" />

  <xsl:with-param name="delimiters" select="$delimiters" />
</xsl:call-template>
</xsl:when>
...

  <xsl:otherwise>
    <xsl:call-template name="delimiter">
      <xsl:with-param name="string" select="$string" />

      <xsl:with-param name="delimiters"
        select="substring($delimiters, 2)" />
    </xsl:call-template>
  </xsl:otherwise>
</xsl:choose>
</xsl:template>

```

## Taking a look at your result tree

Speaking of output, take a look at how the output of your original transformation has improved (see [Creating separate elements for each machine](#) to view the original). Most of the file has been removed to save space, but you can download the file in its entirety from [Tools](#).

```

<data>
<MachineName>FirstHost</MachineName>
<value>FirstHost</value>
<value>sandbox_acct_app</value>
<value>sandbox_acct_app</value>
<value>sandbox_funkyApps1</value>
<!-- more elements here -->
</data>

```

You can also download the same file containing a result from using the `copy-of` statement and the `trace` element from .

## Analyzing the data

Now that you've created a new XML file, you're ready to analyze the data. It's much

easier to deal with and analyze as an XML file than it was as a text file: Not only do you have programmatic access to it, but it's also easier to search for element names than strands of text. However, I've known a few Perl developers who may disagree with this because of Perl's powerful regular expressions support -- something that is being introduced in XSLT 2.0, and that I'll look at briefly towards the end of this tutorial.

So now, you'll want to save your results file as an XML file. Call it `StepThreeResult.xml`.

## Writing a simple query for initial analysis

From there it's easy to analyze results, because all you're doing is applying basic XSLT principles. For example, if you want to know how many `resignal` counts there are, you can simply write a template using an XPath query:

```
<xsl:template match="/">
  <xsl:apply-templates select="data" />
</xsl:template>
<xsl:template match="data">
  Resignal Count Values: <xsl:value-of
    select="count(value[text()='ResignalCount=1'])" />
</xsl:template>
```

This stylesheet (available as part of the download from [Tools](#)) will return the following value:

```
Resignal Count Values: 40
```

---

## Section 6. Making your string search routines and string parsers re-usable templates

### Creating a generic template

Making search routines and string parsing templates reusable is a crucial part of using your resources wisely. No matter what size your organization is, having a library of well-documented, re-usable templates is a great time-saver.

In fact, the MindMap team loved the `br-replace` template (which you saw earlier in the tutorial) so much that they dropped it into their directory of generic XSLT

templates and now use it nearly every day. Generic templates act as function libraries and should be made in as generic a way as possible so they can be reused. Consider the template below -- you don't define any of the matching nodes in this initial template; a good generic template should be able to stand on its own, callable by any file. This isn't always possible, but it's always desirable. You should be able to use this template with any string of text that has breaks in it:

```
<xsl:template name="links">
  <xsl:param name="String" />
  <xsl:choose>
    <xsl:when test="contains($String,',')">
      <a href="#id{substring-before($String,',')}">
        <xsl:value-of
          select="substring-before($String,',')" />
      </a>
      <xsl:text></xsl:text>
      <xsl:call-template name="links">
        <xsl:with-param name="String"
          select="substring-after($String,',')" />
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <a href="#id{$String}">
        <xsl:value-of select="$String" />
      </a>
      <xsl:text></xsl:text>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

## Calling a generic template

If you're entering text into a text editor, name the file `links.xsl` or download it from the source package, available in [Tools](#). Then, create a new XSLT file named `call-links.xsl`, and enter the following code:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" />
  <xsl:include href="links.xsl" />
  <xsl:template match="/">
    <html>
      <head>
        <title>Links</title>
      </head>
      <xsl:apply-templates select="WebConfig" />
    </html>
  </xsl:template>

  <xsl:template match="WebConfig">
    <body>
      <xsl:call-template name="links">
        <xsl:with-param name="String" select="." />
      </xsl:call-template>
    </body>
  </xsl:template>
  ...
```

Note that not all the code is included here and as the code stands it is not yet well formed. You'll need to either finish it yourself or download the file. As you can see, only the XSLT document doing the calling contains any path information. You can try this on just about any XML document with lots of text. You don't need to change the named template, just the parameters on the calling template.

---

## Section 7. Finding and using generic templates

### Introducing EXSLT

You can do a Web search to find generic templates. MindMap team members generally type in the functionality they're looking for, then the term "generic template" (in quotes), then "XSLT". For example, one of the MindMap team members who wants a generic string replacement template might type the following into Google:

```
string replace "generic template" XSLT
```

This has resulted in a growing library of reusable templates for the MindMap team. (See [Resources](#) for more EXSLT links.)

EXSLT consists of a group of extensions to the XSLT language. These extensions have been developed by a group of dedicated XSLT enthusiasts. The underlying architecture consists of generic templates, but there is one caveat: They're often not portable in the way other XSLT templates are because they sometimes (but not always) rely on proprietary extensions and sometimes JavaScript.

An EXSLT function's purpose can range from handling regular expressions to manipulating node sets on the fly (both of which are common wish list items for XSLT developers, and both will be standard features in XSLT 2.0). Which extensions you use generally depend on which XSLT processor you are using. They work *if* the processor has built-in support for them, which is where the dependency lies. You can't use an extension built for Saxon on MSXML processors, and vice versa, so EXSLT is an attempt to standardize extensions. However, you need to check the EXSLT site (in [Resources](#)) to determine if the processor you're using supports a specific extension. Luckily, the site is replete with sample generic templates and examples on how to use them.

### Using an EXSLT function

The first step in using an EXSLT function is to go to [www.exslt.org](http://www.exslt.org) and look for the extension you want. For example, that Web site has some very handy string manipulation functions, and even a regular expression function. Some of them are also less stable than others. In preparing for this article, I found a number of them that simply didn't work, but the site is pretty reliable about indicating whether or not specific functions can be relied on.

The main trick in using an EXSLT function is to find out what that function actually is. I refer to EXSLT functions as either a process defined in a named template, which you then call with a parameter value, or as an actual function. Take a look, for example at this call to the EXSLT `math:math_max` function, which works with the Saxon and Xalan processors (but not MSXML).

```
<?xml version="1.0" encoding="utf-8"?>
<stylesheet xmlns="http://www.w3.org/1999/XSL/Transform"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:func="http://exslt.org/functions"
xmlns:math="http://exslt.org/math" version="1.0"
extension-element-prefixes="math"
math:doc="http://www.exslt.org/math">
<xsl:output method="html" />
<import href="math.max.function.xsl"/>
<import href="math.max.template.xsl"/>
<xsl:template match="/">
<MaxMachine>
<xsl:call-template name="math:max">
<xsl:with-param name="nodes"
select="MachineName/value" />
</xsl:call-template>
</MaxMachine>
</xsl:template>
</stylesheet>
```

The use of the namespaces is an absolute requirement because they tell the processor to expect an extension function defined within the EXSLT library. The `nodes` parameter assumes an element named `values` that contains numbers. If you have string characters, you'll need to parse those out of the `value` element first. The result of applying the template will be the maximum value of all the `value` elements.

One advantage of generic templates like this is that you just need to know how to call the parameters and where to put your template within the scope of the rest of your document.

**Note:** Generally, if the site says a specific function is not considered stable, they really mean it.

Many of the templates in the EXSLT library have since found their way, in simplified form, into XSLT 2.0. And as XSLT 2.0 approaches, the MindMap team is nearly breathless with anticipation, because they'll be able to use one of the most powerful string parsing mechanisms available: regular expressions.

---

## Section 8. Using regular expressions through XSLT 2.0

### How XSLT 2.0 will change your world

XSLT 2.0 is nearing release, and when it arrives and receives substantial processor support, your life will be a lot easier. The MindMap team has been getting a jump on things by looking over the early drafts of the XSLT 2.0 specification (see [Resources](#)).

XSLT 2.0 will take the hassle out of quite a few things that the MindMap team has found somewhat frustrating with XSLT 1.0, particularly string management. XSLT 2.0 provides support for regular expressions, which are a pattern-based method of searching for specific strings of text found in other languages with powerful string manipulation capabilities like Perl.

However, you shouldn't use XSLT 2.0 in a production environment quite yet. It's hard to say when the specification will be finalized, and it could still change substantially. The current working draft is in Last Call, which means this is the last opportunity for public comment, but this has been a slow-moving specification, so final delivery might not happen before the second half of 2004. After that, it will take some time for the language to mature and for XSLT processors to work out the bugs (and I expect plenty of them because the architecture of the language experienced some core changes, especially with regards to data typing).

So what do you need to experiment with XSLT 2.0 today? Saxon, an XSLT processor developed by Michael Kay, provides an immediate opportunity for testing basic features of XSLT 2.0 (see [Resources](#) for the latest version.)

### Using regular expressions

The `replace` template that you saw earlier in the tutorial was a fairly simple one, but extremely complex string replacement routines require extremely complex templates. One of the requirements for XSLT 2.0 was to ease the pain in that area. In addition, reality has surfaced: Sites with huge traffic flows should avoid heavy XSLT processing anyway (and let a database do the work whenever it can), because XSLT has proven to be expensive on heavily trafficked sites. Whether XSLT 2.0 will change that remains to be seen, but clearly this statement is much easier to build than the template required in XSLT 1.0:

```
<p>
```

```
<xsl:value-of select="replace(., '(Host)', 'Machine')"/>
</p>
```

The `replace()` function is new in XSLT 2.0, and takes three arguments:

- The input string you want to base your search on
- A regular expression to use for singling out the specific characters you want to replace
- The text with which you wish to replace the characters in the second argument

---

## Section 9. Summary and XSLT tricks of the trade

### Summary: The MindMap team has only just begun

Actually, that's not true. The MindMap team really has gone deep into the processing power of XSLT, but you've only just begun to get a peek at what they're up to. For them, doing some string manipulation tasks was just the beginning. It was this encounter with XSLT that convinced them of the language's power, so it's only fitting that you start by taking a look at their earliest endeavors. Starting with a routine host configuration text file that consists of some data about the machines on their network, they've been able to convert it into an XML file. This gives them programmatic access to each host machine on different layers, depending on their needs.

You can do the same thing. If you're running a large Web site, for example, you can manage builds by writing configuration specs for specific branches of a source control system that refer to specific elements representing different machines.

In this tutorial, you've had just a cursory glimpse at how you can use XML as an analysis tool. You saw how the MindMap team set up a text file for analysis, and you were given an idea of what might come next in terms of examining the resulting XML's elements. In the next tutorial, you'll see how the MindMap team extends this concept much further as you explore XSLT as an analysis tool.

Next, I'll offer a couple of bonus [XSLT tricks of the trade](#).

## XSLT tricks of the trade

Remember that straight text-only files cannot be processed by an XSLT processor. This is because an XSLT processor works with an XML parser, which first looks at your input to be sure it is well-formed XML. The other thing to remember about your input string is that it may contain characters like < and >, or &. Don't wrap them in CDATA elements, a mistake I see frequently. (Okay, it's not really a mistake, it's just not good practice.) Get those things converted somehow. A good Perl interpreter is a handy thing to have, and if you're working on small-scale projects you can always use Emacs or a similar text processing program.

You can get pretty wild and crazy with parameters. In fact, you can come awfully close to mimicking the *changing* of variable values that many XSLT newbies automatically seek out when first experimenting with XSLT by cascading parameter values among recursive template calls. By changing the value of the string as it passes through each parameter (and let's be clear here: The string value is not actually passing through by a variable; the XSLT processor is simply reading a different part of the source tree depending on when a template is called -- XSLT is completely side-effect-free), you can manipulate strings and output what you want.

## Downloads

Description	Name	Size	Download method
Sample code	x-dataxsltsourcecode.zip	9KB	<a href="#">HTTP</a>

[Information about download methods](#)

## Resources

- Check out the rest of this *developerWorks* tutorial series on XSLT for more information on advanced uses for XSL transformations:
  - ["XSLT as an analysis tool"](#) (February 2004)
  - ["Layer XSLT stylesheets"](#) (March 2004)
  - ["Create 3D representations with XSLT and SVG"](#) (April 2004)
  - ["Tie in data with Web services and XSL Transformations"](#) (August 2004)
- Don't know XSLT? Better try the [IBM Tutorial on beginning XSLT](#) (<http://www.ibm.com/developerworks/edu/x-dw-xwebxslt-i.html>) (*developerWorks* March 2003).
- Get [information- on UTF-8 and Unicode](#), two important pieces of the XML puzzle (<http://www.cl.cam.ac.uk/~mgk25/unicode.html>).
- To work with XML, you need an XML parser. Check out this list of [XML parsers](#) (<http://xml.coverpages.org/publicSW.html#xmlToolsTOC>).
- It isn't possible to process XSLT documents without an XSLT processor. Browse this list for some of the more common [XSLT Processors](#) (<http://www.xml.com/pub/a/2001/03/28/xsltmark/results.html>).
- Check out [Mastering XSLT](#) (<http://www.amazon.com/exec/obidos/tg/detail/-/0782140947/103-5714425-8867848?v=glance>) by this author.
- Read about [EXSLT](#) (<http://www.exslt.org>), a community initiative to provide extensions to XSLT. While you're at it, check out Uche Ogbuji's article "[EXSLT by example](#)" (<http://www.ibm.com/developerworks/library/x-exslt.html>), which uses practical examples to introduce and demonstrate some useful EXSLT functions (*developerWorks*, February 2003).
- Learn more about [OpenOffice](#) (<http://www.openoffice.org/>), an office suite that runs on all major platforms and provides access to all functionality and data through open-component based APIs and an XML-based file format. Uche Ogbuji also discusses OpenOffice in his [Thinking XML](#) column (<http://www.ibm.com/developerworks/xml/library/x-think15/>) (*developerWorks*, January 2003).
- You can opt to run the initial example in this tutorial using a [Perl interpreter](#) (<http://ftp.linux.cz/pub/perl/ports/>).
- You don't have to pay gobs of money for an XML editor. Try this list of [free XML editors](#) ([http://www.garshol.priv.no/download/xmltools/cat\\_ix.html#SC\\_XMLEditors](http://www.garshol.priv.no/download/xmltools/cat_ix.html#SC_XMLEditors)).

- Learn more about regular expressions in Mark-Jason Dominus' article "[How Regexes Work](http://perl.plover.com/Regex/article.html)" which demonstrates how to write a regular expression package from scratch (<http://perl.plover.com/Regex/article.html>).
- Don't know XML? Well, here's the [XML specification](http://www.w3.org/XML/) (<http://www.w3.org/XML/>). You can also check out Doug Tidwell's "[Introduction to XML](http://www.ibm.com/developerworks/edu/x-dw-xmlintro-i.html)" tutorial (<http://www.ibm.com/developerworks/edu/x-dw-xmlintro-i.html>) here on *developerWorks* (August 2002).
- You should always keep a bookmark to the current specifications. Here's the [XSLT 1.0 spec](http://www.w3.org/TR/xslt) (<http://www.w3.org/TR/xslt>), and you can view the [current status of XSLT 2.0](http://www.w3.org/TR/xslt20/) (<http://www.w3.org/TR/xslt20/>).
- Visit this earlier article by the author for an early "[Peek Into the Future of XSLT 2.0](http://www.devx.com/xml/Article/11147)" (<http://www.devx.com/xml/Article/11147>). Note that some of the technical stuff may have changed.
- How Michael Kay manages to get as much done as he does is beyond me. He not only is the editor of the upcoming XSLT 2.0 specification, but cranks out a pretty good XSLT processor, [Saxon XSLT editor](http://saxon.sourceforge.net/) (<http://saxon.sourceforge.net/>), which you'll need if you want to develop XSLT 2.0 applications before the final spec is released.
- The Apache Group, developers of the famous Apache Web Server, has an extensive XML development program. Visit the [Apache XML Project](http://xml.apache.org/) (<http://xml.apache.org/>), which includes some very interesting XSLT involving multiple documents using a process they call "piping".
- Take a look at this [list of Perl interpreters](http://ftp.linux.cz/pub/perl/ports/) (<http://ftp.linux.cz/pub/perl/ports/>).
- If you're just desperate to try grouping, or simply need to use grouping to complete a project, visit the section of the XSL FAQ hosted by Dave Pawson on [Muenchian techniques](http://www.dpawson.co.uk/xsl/sect2/muench.html#d7739e13) (<http://www.dpawson.co.uk/xsl/sect2/muench.html#d7739e13>).
- Learn how XSL transformations can be used with WebSphere Application Server in "[XSL Transform Basics with WebSphere Application Server Version 4.0x](http://www.ibm.com/developerworks/websphere/techjournal/0204_sundman/sundman.html)" by Joel Sundman ([http://www.ibm.com/developerworks/websphere/techjournal/0204\\_sundman/sundman.html](http://www.ibm.com/developerworks/websphere/techjournal/0204_sundman/sundman.html)) (*developerWorks*, April 2002).
- [Browse for books](#) on these and other technical topics.
- Finally, find out how you can become an [IBM Certified Developer in XML and related technologies](#).

## About the author

## Chuck White

Chuck White, a [Studio B](#) author, has been working with XML since before its official inception in February, 1998. He was co-author of [Mastering XML Premium Edition](#) (with Linda Burman and the W3C's XML Activity Lead, Liam Quin) and author of [Mastering XSLT](#), both from Sybex Books. His latest books are [Developing Killer Web Apps with Dreamweaver MX & C#](#) (also for Sybex Books) and *HTML, XHTML, and CSS Bible, 3rd Edition* for Wiley, for which he is co-author with Steve Schafer. Chuck is currently working with the XSL Team at eBay as a project consultant and Web engineer.