

icoya ***OpenContent***

**Fast and easy
management of
active websites**

Zope User Guide

- ↘ **Structured Text**
- ↘ **Zope Page Templates**
- ↘ **METAL Expressions**

German translation provided by:**struktur AG**

Junghansstraße 5
D-70469 Stuttgart
Germany

E-Mail: info@struktur.de

www.struktur.de
www.strukturag.com

www.icoya.de
www.icoya.org
www.icoya.net
www.icoya.com

COPYRIGHT

The copyright to each Open Publication is owned by its author(s) or designee.

The Open Publication works may be reproduced and distributed in whole or in part, in any medium physical or electronic, provided that the terms of this license are adhered to, and that this license or an incorporation of it by reference (with any options elected by the author(s) and/or publisher) is displayed in the reproduction.

Proper form for an incorporation by reference is as follows:

This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at <http://www.opencontent.org/openpub/>).

The reference must be immediately followed with any options elected by the author(s) and/or publisher of the document (see section VI).

Commercial redistribution of Open Publication-licensed material is permitted.

Any publication in standard (paper) book form shall require the citation of the original publisher and author. The publisher and author's names shall appear on all outer surfaces of the book. On all outer surfaces of the book the original publisher's name shall be as large as the title of the work and cited as possessive with respect to the title.

Table of Contents

1	Help on Structured Text	
1.1	Help on Structured Text	.4
1.1.1	Structured Text Basics	.4
1.1.2	Using Indentation	.4
1.1.3	Lists and Items	.5
1.1.4	Example Code	.5
1.1.5	Hyperlinks	.6
1.1.6	Advanced Usage	.6
1.1.7	Conclusion	.6
1.1.8	Resources	.6
1.2	Structured Text	.7
1.2.1	Introduction	.7
1.2.2	Structured Text Manipulation	.7
1.2.3	Including Structured Text in DTML	.8
1.2.4	Other Resources	.8
2	Page Templates	
2.1	Zope Page Templates: Getting Started	.9
2.1.1	Introduction to Page Templates	.9
2.1.2	Why Yet Another Template Language?	.9
2.1.3	Applying The Principles	.9
2.1.4	Creating a Page Template	.9
2.1.5	Simple Expressions	.10
2.1.6	Inserting Text	.10
2.1.7	Repeating Structures	.10
2.1.8	Conditional Elements	.11
2.1.9	Defining Variables	.11
2.1.10	Changing Attributes	.11
2.1.11	Conclusion	.12
2.2	Zope Page Templates: Advanced Usage	.12
2.2.1	Mixing and Matching Statements	.12
2.2.2	Statements with Multiple Parts	.12
2.2.3	String Expressions	.13
2.2.4	Nocall Path Expressions	.13
2.2.5	Python Expressions	.13
2.2.6	Other Builtin Variables	.13
2.2.7	Alternate Paths	.13
2.2.8	Dummy Elements	.13
2.2.9	Inserting Structure	.14
2.2.10	Basic Python Expressions	.14
2.2.11	Getting at Zope Objects	.14
2.2.12	Using Scripts	.15
2.2.13	Calling DTML	.15
2.2.14	Python Modules	.15
2.3	TALES Specification Version 1.3	.16
2.3.1	TAL Specification Version 1.4	.17
2.4	Frequently Asked Questions about Zope Page Templates	.20
3	Metal Expressions	
3.1	METAL for Beginners	.22
3.2	A beginners guide to METAL	.22
3.2.1	What ist METAL?	.22
3.2.2	Getting Started with Macros	.22
3.2.3	What I use Macros for	.23
3.2.4	Links and Other Resources	.23
3.2.5	METAL Specification Version 1.0	.23
	Open Publication License	.25

1.1 Help on Structured Text

By Paul Everitt

Engineers spend a lot of time communicating, primarily by email but also in documentation. However, writing by engineers is complicated by a simple fact: the world consumes writing largely in presentation formats such as HTML and PDF.

In theory this should be no problem, as we would all march happily off and write in "DocBook" (or perhaps LaTeX), the supposed lingua franca of documentation. However most tools don't support DocBook (or LaTeX) very well, and even if tools were mature, most engineers would reject them.

Why? Engineers spend most of their time communicating in plain text. Their tools (vi and Emacs) are oriented toward text. The vast majority of words they communicate are in email. Finally, what little documentation you can squeeze out of engineers is in the form of "docstrings" in source code.

Wouldn't it be nice if there was a non-tag, text-oriented system for engineers to express semantic meaning? This is the problem *Structured Text* tackles. With Structured Text, format-independent writing becomes extremely convenient and natural, once a few rules are learned. Furthermore, Structured Text can be extended to cover advanced and custom uses.

To get a quick idea of what Structured Text does, the following words in Structured Text:

Sometimes the *best* approach to complexity is simplicity. A good structured text system is:

- o Convenient
- o Rich

is rendered into the following HTML:

```
<p>
Sometimes the <em>best</em> approach to complexity
is simplicity. A good structured text system is:
</p>

<ul>
<li><p>Convenient</p></li>
<li><p>Rich</p></li>
</ul>
```

and the following [DocBook](http://www.nwalsh.com/docbook/) (<http://www.nwalsh.com/docbook/>) XML:

```
<para>
Sometimes the <emphasis>best</emphasis> approach to
complexity is simplicity. A good structured text system
is:
</para>

<itemizedlist>
<listitem><para>Convenient</para></listitem>
<listitem><para>Rich</para></listitem>
</itemizedlist>
```

In fact, the text of this article is written in Structured Text. In this article, we'll look at the basics of Structured Text, organizing large text into sections, advanced formatting, and meta-data issues.

1.1.1 Structured Text Basics

Let's plunge into structured text and look at the basics by correlating it to ideas in HTML.

The most basic idea in Structured Text is a paragraph. The following snippet of:

This is the first paragraph.

This is the second paragraph.

...is converted to the following in HTML:

```
<p>This is the first paragraph.</p>
```

```
<p>This is the second paragraph.</p>
```

That is, white space matters in Structured Text. This is a very intuitive idea. For instance, in email paragraphs are separated by white space.

To introduce emphasis, Structured Text uses another text convention: asterisks. Note the following snippet:

This is the *first* paragraph.

This is the **second** paragraph.

In HTML, this snippet introduces the em tag and the strong tag:

```
<p>This is the <em>first</em> paragraph.</p>
```

```
<p>This is the <strong>second</strong> paragraph.</p>
```

Again, this is a common pattern in email. Several other common patterns are supported, such as referring to a piece of jargon:

When you see 'STX', you know this is shorthand for 'Structured Text'.

The HTML output is as follows:

```
<p>When you see <code>STX</code>, you know this is
shorthand for
<code>Structured Text</code>.</p>
```

1.1.2 Using Indentation

The preceding section focused on text conventions that convey a semantic meaning. This semantic meaning, when processed by Structured Text, produces certain HTML tags.

In Structured Text, indentation is also very important in conveying semantic meaning. The most basic is the idea from HTML of headings.

In the following snippet, indentation is used to convey an outline-like structure:

Using Indentation

The preceding section focused on text conventions that convey a semantic meaning. This semantic meaning, when processed by Structured Text, produces certain HTML tags.

This produces the following HTML:

```
<h1>Using Indentation</h1>

<p>The preceding section focused on text conventions that convey a semantic meaning. This semantic meaning, when processed by Structured Text, produces certain HTML tags.</p>
```

That is, the indentation conveyed a semantic meaning. The paragraph was subordinate to the heading, and the relationship is thus expressed in HTML. In fact, outline relationship can be continued:

```
Using Indentation

The preceding section focused on text conventions that convey a semantic meaning. This semantic meaning, when processed by Structured Text, produces certain HTML tags.
```

Basics of Indentation

In this section we will investigate the basics of indentation...

Hyperlinks

This produces the following HTML:

```
<h1>Using Indentation</h1>

<p>The preceding section focused on text conventions that convey a semantic meaning. This semantic meaning, when processed by Structured Text, produces certain HTML tags.</p>

<h2>Basics of Indentation</h2>

<p>In this section we will investigate the basics of indentation...</p>

<h2>Hyperlinks</h2>
```

1.1.3 Lists and Items

Lists are also supported in Structured Text, including unordered, ordered, and descriptive lists. The convention unordered lists is a common pattern in text-based communication:

HTML has three kinds of lists:

- o Unordered lists
- o Ordered lists
- o Descriptive lists

Structured Text allows you to use the symbols *, o, and - to connote list items. The above example produces this HTML:

```
<p>HTML has three kinds of lists:</p>

<ul>

<li><p>Unordered lists</p></li>

<li><p>Ordered lists</p></li>
```

```
<li><p>Descriptive lists</p></li>

</ul>
```

The Structured Text conventions for ordered lists is shown below:

HTML has three kinds of lists:

1. Unordered lists
2. Ordered lists
3. Descriptive lists

This produces:

```
<p>HTML has three kinds of lists:</p>

<ol>

<li><p>Unordered lists</p></li>

<li><p>Ordered lists</p></li>

<li><p>Descriptive lists</p></li>

</ol>
```

Descriptive lists are also easily accommodated using double dashes:

- Unordered Lists -- Generally includes a series of bullets when viewed in HTML.
- Ordered Lists -- HTML viewers convert the list items into a numbered series.
- Descriptive Lists -- Usually used for definitional lists such as glossaries.

This becomes the following HTML:

```
<dl><dt>Unordered Lists</dt><dd><p>Generally includes a series of bullets when viewed in HTML.</p></dd>

<dt> Ordered Lists</dt><dd><p>HTML viewers convert the list items into a numbered series.</p></dd>

<dt> Descriptive Lists</dt><dd><p>Usually used for definitional lists such as glossaries.</p></dd>

</dl>
```

1.1.4 Example Code

As mentioned above, Structured Text authors can use an easy convention to get the monotype semantics of the CODE tag from HTML. For instance:

When you see the dialog box, hit the 'Ok' button.

...is rendered into the following HTML:

```
<p>When you see the dialog box, hit the <code>Ok</code> button.</p>
```

However, sometimes you want long passages of code. For instance, what if you wanted to document a Python function in the middle of an article discussing Python? You can indicate a code block by ending a paragraph with `::`, and indenting the following paragraph(s). For instance, this Structured Text snippet:

```
In our next Python example, we convert human years to
dog years::
def dog_years(age):
    """Convert an age to dog years"""
    return age*7
```

...would be converted to the following HTML:

```
<p>In our next Python example, we convert human years
to dog years:</p>
```

```
<pre>
def dog_years(age):
    """Convert an age to dog years"""
    return age*7
</pre>
```

The convention of combining `::` at the end of a paragraph-ending sentence and indenting a block does more than apply CODE semantics. It also escapes the indented block. That is how the Structured Text and HTML snippets in this article are left alone, rather than being rendered.

For example, the less than, greater than, and ampersand symbols in this code block are escaped:

Here's an HTML example::

```
<html>
<p>This is a page about dogs & cats.</p>
</html>
```

...to produce this HTML:

```
<p>Here's an HTML example:</p>
<pre>
<html>
<p>This is a page about dogs & cats.</p>
</html>
</pre>
```

1.1.5 Hyperlinks

In the previous sections we focused on ways to get certain presentation semantics in HTML by using common text conventions.

But the web isn't just HTML. Linking words and phrases to other information and including images are equally important. Fortunately Structured Text supports conventions for hyperlinks and image tags.

Let's start with a simple hyperlink. If we have a Structured Text paragraph discussing Python:

```
For more information on Python, please visit the "Python
website": http://www.python.org/.
```

This becomes:

```
<p>For more information on Python, please visit the <a
href="http://www.python.org/">Python website</a>.
```

The convention is fairly simple:

- The text of the reference is enclosed in quotes.
- The second quotation mark is followed by a colon and a URL.
- The URL can be followed by punctuation.

This basic convention has a number of variations. For instance, relative URLs are possible, as are mailto URLs.

(Note: in the above example, there should not be a space between the last quote and the colon. This is due to a bug in the version of structured text currently running on Zope.org. This bug has been fixed in more recent versions of Zope.)

1.1.6 Advanced Usage

There are more obscure extensions to Structured Text to handle cross references, tables, images, and more.

One of the great things about structured text is that if you don't like its rules it's fairly easy to extend. This is made possible by the recent rewriting of Structured Text sometimes referred to as "Structured Text NG". For example, you could create a LaTeX outputter, or you could change structured text to recognize a different syntax for hyperlinks.

Structured Text is available in Zope [and is integrated into the Zope [Content Management Framework](http://cmf.zope.org) (<http://cmf.zope.org>)] but you can also use it outside of Zope. To use Structured Text in Zope just create a document or file containing structured text, then call it like so:

```
<dtml-var my_document fmt=structured-text>
```

This will give you the HTML representation of `my_document`. The [Zope Book](http://www.zope.org/Members/michel/ZB) (<http://www.zope.org/Members/michel/ZB>) is an example of a Project that uses Structured Text outside of Zope. The book was written in Structured Text with some modifications to support figure handling, and the publisher's in-house markup format. Python scripts parse the input and create output in HTML and PDF.

Structured Text use is also used in Python doc strings. A number of Python documentation extraction tools support Structured Text. Currently work is under way on the Python [doc-sig](http://www.python.org/sigs/doc-sig) (<http://www.python.org/sigs/doc-sig>) to develop docstring conventions, and a docstring processing system.

1.1.7 Conclusion

Structured Text gives you an easy way to express yourself in plain text. The Structured Text implementation allows you to tailor the syntax and output. Structured Text is integrated into Zope and is also usable outside Zope.

1.1.8 Resources

[Structured Text Wiki](http://www.zope.org/Members/jim/StructuredTextWiki/FrontPage)

(<http://www.zope.org/Members/jim/StructuredTextWiki/FrontPage>)
- discusses structured text and STXNG.

[reStructuredText](http://structuredtext.sourceforge.net/)

(<http://structuredtext.sourceforge.net/>)

- A Structured Text alternative being developed as a Python docstring standard.

1.2 Structured Text

Created by [millejoh](http://www.zope.org/Members/millejoh) (<http://www.zope.org/Members/millejoh>).
 Last modified on 2000/10/16.
 See also <http://www.zope.org/Members/millejoh/structuredText>

1.2.1 Introduction

Structured Text, being a highly Cool Thing, is a tool that Should Be Known To All. It is excellent for quickly prototyping Web pages for our favorite playground: Zope.

However I find myself unable to always remember all of the functionality of the structured text document, ergo the raison d'être for this HOWTO.

This is almost just a copy and paste of documentation I found in the source code. When I first put this document together I was unaware that I didn't know how to actually include structured text into my DTML methods and documents. For those of you facing similar quandries I have included a short example at the end of this document on how to include structured text in DTML documents. Don't worry- it is really quite simple once you know the magical incantations. Enjoy!

Suggestions of any sort are always [appreciated](mailto:jmiller1@uop.com) (<mailto:jmiller1@uop.com>).

1.2.2 Structured Text Manipulation

Parse a structured text string into a form that can be used with structured formats, like html.

Structured text is text that uses indentation and simple symbology to indicate the structure of a document.

A structured string consists of a sequence of paragraphs separated by one or more blank lines. Each paragraph has a level which is defined as the minimum indentation of the paragraph. A paragraph is a sub-paragraph of another paragraph if the other paragraph is the last preceding paragraph that has a lower level.

Special symbology is used to indicate special constructs:

- ✎ A single-line paragraph whose immediately succeeding paragraphs are lower level is treated as a header.
- ✎ A paragraph that begins with a -, *, or o is treated as an unordered list (bullet) element.
- ✎ A paragraph that begins with a sequence of digits followed by a white-space character is treated as an ordered list element.
- ✎ A paragraph that begins with a sequence of sequences, where each sequence is a sequence of digits or a sequence of letters followed by a period, is treated as an ordered list element.
- ✎ A paragraph with a first line that contains some text, followed by some white-space and -- is treated as a descriptive list element. The leading text is treated as the element title.
- ✎ Sub-paragraphs of a paragraph that ends in the word example or the word examples, or :: is treated as example code and is output as is.
- ✎ Text enclosed single quotes (with white-space to the left of the first quote and whitespace or punctuation to the right of the second quote) is treated as example code.

✎ Text surrounded by * characters (with white-space to the left of the first * and whitespace or punctuation to the right of the second *) is emphasized.

✎ Text surrounded by ** characters (with white-space to the left of the first ** and whitespace or punctuation to the right of the second **) is made strong.

✎ Text surrounded by _ underscore characters (with whitespace to the left and whitespace or punctuation to the right) is made underlined.

✎ Text enclosed by double quotes followed by a colon, a URL, and concluded by punctuation plus white space, or just white space, is treated as a hyper link. For example:

```
"Zope":http://www.zope.org/ is ...
```

Is interpreted as

```
<a href="http://www.zope.org/">Zope</a> is ...
```

Note: This works for relative as well as absolute URLs.

✎ Text enclosed by double quotes followed by a comma, one or more spaces, an absolute URL and concluded by punctuation plus white space, or just white space, is treated as a hyper link. For example:

```
"mail me", mailto:amos@digicool.com.
```

Is interpreted as

```
<a href="mailto:amos@digicool.com">mail me</a>."
```

✎ Text enclosed in brackets which consists only of letters, digits, underscores and dashes is treated as hyper links within the document. For example:

As demonstrated by Smith [a12] this technique is quite effective. [r1]

```
(http://www.zope.org/Members/millejoh/structuredText/#r1)
```

Is interpreted as ...

```
by Smith <a href="#a12">[12]</a> this ...
```

Together with the next rule this allows easy coding of references or end notes.

✎ Text enclosed in brackets which is preceded by the start of a line, two periods and a space is treated as a named link. For example:

```
.. [a12] "Effective Techniques" Smith, Joe
```

Is interpreted as [12]

```
"Effective Techniques" .... [r2]
```

```
(http://www.zope.org/Members/millejoh/structuredText/#r2)
```

Together with the previous rule this allows easy coding of references or end notes.

[r1] According to the HTML 4.0 [specification](http://www.w3.org/TR/REC-html40/types.html#type-id) (<http://www.w3.org/TR/REC-html40/types.html#type-id>) identifiers must start with a letter so using references like [12] is a bit of a no-no, though it does work (at least in all the browsers I've tried). The StructuredText Python module should probably be updated to do something nice like prepend ref whenever it sees only numbers in a link.

[r2] The use of name here is deprecated, id should be used instead. Not there is much you as the document author can do about this since it's StructuredText that is generating the HTML (and it probably should be modified to do the right thing) but at least when somebody comes to nitpick you can say "Yes, I know. Everybody knows about this, now go away and let me get some work done before I have to show just how strong my Kung Fu really is."

1.2.3 Including Structured Text in DTML

I had the hardest time actually getting my structured text to render into HTML until I learned the following incantation:

```
<dtml-var stx_doc_name fmt=structured-text>
```

Without the `fmt=structured-text` part you will just get the raw text, (something like [this](http://www.zope.org/Members/millejoh/structuredText/raw_text) (`http://www.zope.org/Members/millejoh/structuredText/raw_text`)) most likely *not* what you want.

1.2.4 Other Resources

It is well worth your time to check out Tres Seaver's [Structured Text Document](http://www.zope.org/Members/tseaver/STX_Document) (`http://www.zope.org/Members/tseaver/STX_Document`) product. Conveniently provided is a ZClass one can use to wrap one's STX content. Get it, use it, and be happy.

There are a few minor caveats to its use that should have absolutely no bearing on 99% of what one would want to use this product for. Nonetheless, do check the release notes for those details before you use the product (but you were going to do that regardless of what I said, right?).

2.1 Zope Page Templates: Getting Started

2.1.1 Introduction to Page Templates

by Evan Simpson

Page Templates are a web page generation tool. They help programmers and designers collaborate in producing dynamic web pages for Zope web applications. Designers can use them to maintain pages without having to abandon their tools, while preserving the work required to embed those pages in an application. In this article, we'll go through the basics of Page Templates and how you can use them in your web site to create dynamic web pages easily.

The goal of Page Templates is natural workflow. A designer will use a WYSIWYG HTML editor to create a template, then a programmer will edit it to make it part of an application. If required, the designer can load the template *back* into his editor and make further changes to its structure and appearance. By taking reasonable steps to preserve the changes made by the programmer, he will not disrupt the application. Page Templates aim at this goal by adopting three principles:

1. Play nicely with editing tools.
2. What you see is very similar to what you get.
3. Keep code out of templates, except for structural logic.

A Page Template is like a model of the pages that it will generate. In particular, it is a valid HTML page. Since HTML is highly structured, and WYSIWYG editors carefully preserve this structure, there are strict limits on the ways in which the programmer can change a page and still respect the first principle.

Page Templates are for programmers and designers who need to work together to create dynamic web pages. If you create and edit all of your web pages with a text editor, you might not care for Page Templates. Then again, they can be simpler to use and understand than the alternative, DTML.

2.1.2 Why Yet Another Template Language?

There are plenty of template systems out there, some of them quite popular, such as ASP, JSP, and PHP. Since the beginning, Zope has come with a template language called DTML. Why invent another?

First, none of these template systems are aimed at HTML designers. Once a page has been converted into a template, it is invalid HTML, making it difficult to work with outside of the application. Each of them violates the first or second principle of Zope Page Templates to one degree or another. Programmers should not "hijack" the work of the designers and turn HTML into software. XMLC, part of the Enhydra project, shares our goal, but requires the programmer to write substantial amounts of Java support code for each template. Second, all of these systems suffer from failure to separate presentation, logic, and content (data). Their violations of the third principle decrease the scalability of content management and website development efforts that use these systems.

2.1.3 Applying The Principles

Page Templates can be downloaded and installed in your Zope by going to the Page Template [download page](#)

(<http://www.zope.org/Members/4am/ZPT>). There, you will find detailed instructions on how to download and install the Zope Page Templates product that you will need to try the examples in this article.

Page Templates use the Template Attribute Language (TAL). TAL consists of special tag attributes. For example, a dynamic page title might look like this:

```
<title tal:content="here/title">Page Title</title>
```

The `tal:content` attribute is a TAL statement. Since it has an XML namespace (the `tal:` part) most editing tools will not complain that they don't understand it, and will not remove it. It will not change the structure or appearance of the template when loaded into a WYSIWYG editor or a web browser. The name `content` indicates that it will set the content of the title tag, and the value `"here/title"` is an expression providing the text to insert into the tag.

To the HTML designer using a WYSIWYG tool, this is perfectly valid HTML, and shows up in their editor looking like a title should look like. The designer, not caring about the application details of TAL, only sees a *mockup* of the dynamic template, complete with dummy values like "Page Title" for the title of the document.

When this template is saved in Zope and viewed by a user, Zope turns this static content into dynamic content and replaces "Page Title" with whatever "here/title" resolves to. In this case, "here/title" resolves to the title of the object to which the template is applied. This substitution is done dynamically, when the template is viewed.

This example also demonstrates the second principle. When you view the template in an editor, the title text will act as a placeholder for the dynamic title text. The template provides an example of how generated documents will look.

There are template commands for replacing entire tags, their contents, or just some of their attributes. You can repeat a tag several times or omit it entirely. You can join parts of several templates together, and specify simple error handling. All of these capabilities are used to generate document structures. You **can't** create subroutines or classes, write loops or multi-way tests, or easily express complex algorithms. For these tasks, you should use Python.

The template language is deliberately not as powerful and general-purpose as it could be. It is meant to be used inside of a framework (such as Zope) in which other objects handle business logic and tasks unrelated to page layout.

For instance, template language would be useful for rendering an invoice page, generating one row for each line item, and inserting the description, quantity, price, and so on into the text for each row. It would not be used to create the invoice record in a database or to interact with a credit card processing facility.

2.1.4 Creating a Page Template

If you design pages, you will probably use FTP or WebDAV instead of the Zope Management Interface (ZMI) to create and edit Page Templates. See your Zope administrator for instructions. For the very small examples in this article, it is much easier to use the ZMI. For more information on using FTP or WebDAV with Zope, see [The Zope Book](#) (<http://www.zope.org/Members/michel/ZB/>) or Jeffrey Shell's recent [WebDAV article](#) (<http://www.zope.org/Documentation/Articles/WebDAV>)

If you are a Zope administrator or a programmer, you will pro-

bably use the ZMI at least occasionally. You may also use emacs, cadaver, or some other client. See the [Zope Book](http://www.zope.org/Members/michel/ZB/) (<http://www.zope.org/Members/michel/ZB/>) for instructions on setting up Zope to work with various clients.

Use your web browser to log into the Zope management interface as you normally would with Zope. Choose a Folder (the root is fine) and pick "Page Template" from the drop-down add list. Type "simple_page" in the add form's Id field, then push the "Add and Edit" button.

You should now see the main editing page for the new Page Template. The title is blank, the content-type is text/html, and the default template text is in the editing area.

Now you will create a very simple dynamic page. Type the words "a Simple Page" in the Title field. Then, edit the template text to look like this:

```
This is <b tal:replace="template/title">the Title</b>.
```

Now push the "Save Changes" button. The edit page should show a message confirming that your changes have been saved. If some text starting with <-- Page Template Diagnostics is added to the template, then check to make sure you typed the example correctly and save it again. You don't need to erase the error comment; Once the error is corrected it will go away.

Click on the Test tab. You should see a mostly blank page with "This is a Simple Page." at the top.

Back up, then click on the "Browse HTML source" link under the content-type field. This will show you the *unrendered* source of the template. You should see "This is **the Title**." Back up again, so that you are ready to edit the example further.

2.1.5 Simple Expressions

The text "template/title" in your simple Page Template is a *path expression*. This is the most commonly used of the expression types defined by the TAL Expression Syntax (TALES). It fetches the title property of the template. Here are some other common path expressions:

- 📄 request/URL: The URL of the current web request.
- 📄 user/getUserName: The authenticated user's login name.
- 📄 container/objectIds: A list of Ids of the objects in the same Folder as the template.

Every path starts with a variable name. If the variable contains the value you want, you stop there. Otherwise, you add a slash (/) and the name of a sub-object or property. You may need to work your way through several sub-objects to get to the value you're looking for.

There is a small built in set of variables, such as request and user, that will be listed and described later. You will also learn how to define your own variables.

2.1.6 Inserting Text

In your "simple_page" template, you used the tal:replace statement on a bold tag. When you tested it, it replaced the entire tag with the title of the template. When you browsed the source, you saw the template text in bold. We used a bold tag in order to highlight the difference.

In order to place dynamic text inside of other text, you typically use tal:replace on a span tag. Add the following lines to your example:

```
<br>
The URL is <span
tal:replace="request/URL">URL</span>.
```

The span tag is structural, not visual, so this looks like "The URL is URL." when you view the source in an editor or browser. When you view the rendered version, it may look something like:

```
<br>
The URL is http://localhost:8080/simple_page.
```

Remember to take care when editing not to destroy the span or place formatting tags such as b or font inside of it, since they would also be replaced.

If you want to insert text into a tag but leave the tag itself alone, you use tal:content. To set the title of your example page to the template's title property, add the following lines above the other text:

```
<head>
<title tal:content="template/title">The Title</title>
</head>
```

If you open the "Test" tab in a new window, the window's title will be "a Simple Page".

2.1.7 Repeating Structures

Now you will add some context to your page, in the form of a list of the objects that are in the same Folder. You will make a table that has a numbered row for each object, and columns for the id, meta-type, and title. Add these lines to the bottom of your example template:

```
<table border="1" width="100%">
<tr>
<th>#</th><th>Id</th><th>MetaType</th>
<th><th>Title</th>
</tr>
<tr tal:repeat="item container/objectValues">
<td tal:content="repeat/item/number">#</td>
<td tal:content="item/id">Id</td>
<td tal:content="item/meta_type">Meta-Type</td>
<td tal:content="item/title">Title</td>
</tr>
</table>
```

The tal:repeat statement on the table row means "repeat this row for each item in my container's list of object values".

The repeat statement puts the objects from the list into the item variable one at a time, and makes a copy of the row using that variable. The value of "item/id" in each row is the Id of the object for that row.

You can use any name you like for the "item" variable, as long as it starts with a letter and contains only letters, numbers, and underscores (_). It only exists in the <tr> tag; If you tried to use it above or below that tag you would get an error.

You also use the tal:repeat variable name to get information about the current repetition. By placing it after the builtin variable repeat in a path, you can access the repetition count from zero (index), from one (number), from "A" (Letter), and in

several other ways. So, the expression `repeat/item/number` is 1 in the first row, 2 in the second row, and so on.

Since one `tal:repeat` loop can be placed inside of another, more than one can be active at the same time. This is why you must write `repeat/item/number` instead of just `repeat/number`. You must specify which loop your interested in by including the loop name.

2.1.8 Conditional Elements

View the template, and you'll notice that the table is very dull looking. Let's improve it by shading alternate rows. Copy the second row of the table, then edit it so that it looks like this:

```
<table border="1" width="100%">
  <tr>
    <th>#</th><th>Id</th><th>Meta-Type
  </th><th>Title</th>
</tr>
<tbody tal:repeat="item container/objectValues">
  <tr bgcolor="#EEEEEE"
  tal:condition="repeat/item/even">
    <td tal:content="repeat/item/number">#</td>
    <td tal:content="item/id">Id</td>
    <td tal:content="item/meta_type">Meta-Type</td>
    <td tal:content="item/title">Title</td>
  </tr>
  <tr tal:condition="repeat/item/odd">
    <td tal:content="repeat/item/number">#</td>
    <td tal:content="item/id">Id</td>
    <td tal:content="item/meta_type">Meta-Type</td>
    <td tal:content="item/title">Title</td>
  </tr>
</tbody>
</table>
```

The `tal:repeat` has not changed, you have just moved it onto the new `tbody` tag. This is a standard HTML tag meant to group together the body rows of a table, which is how you are using it. There are two rows in the body, with identical columns, and one has a grey background.

View the template's source, and you see both rows. If you had not added the `tal:condition` statements to the rows, then the template would generate both rows for every item, which is not what you want. The `tal:condition` statement on the first row ensures that it is only included on even-indexed repetitions, while the second row's condition only lets it appear in odd-indexed repetitions.

A `tal:condition` statement does nothing if its expression has a true value, but removes the entire statement tag, including its contents, if the value is false. The odd and even properties of `repeat/item` are either zero or one. The number zero, a blank string, an empty list, and the builtin variable `nothing` are all false values. Nearly every other value is true, including non-zero numbers, and strings with anything in them (even spaces!).

2.1.9 Defining Variables

Your template will always show at least one row, since the template itself is one of the objects listed. In other circumstances, you might want to account for the possibility that the

table will be empty. Suppose you want to simply omit the entire table in this case. You can do this by adding a `tal:condition` to the table:

```
<table border="1" width="100%"
  tal:condition="container/objectValues">
```

Now, when there are no objects, no part of the table will be included in the output. When there are objects, though, the expression `"container/objectValues"` will be evaluated twice, which is mildly inefficient. Also, if you wanted to change the expression, you would have to change it in both places.

To avoid these problems, you can define a variable to hold the list, and then use it in both the `tal:condition` and the `tal:repeat`. Change the first few lines of the table to look like this:

```
<table border="1" width="100%"
  tal:define="items container/objectValues"
  tal:condition="items">
  <tr>
    <th>#</th><th>Id</th><th>Meta Type
  </th><th>Title</th>
</tr>
<tbody tal:repeat="item items">
```

The `tal:define` statement creates the variable `items`, and you can use it anywhere in the table tag. Notice also how you can have two TAL attributes on the same table tag. You can, in fact, have as many as you want. In this case, they are evaluated in order. The first assigns the variable `items` and the second uses `items` in a condition to see whether or not it is false (in this case, an empty sequence) or true.

Now, suppose that instead of simply leaving the table out when there are no items, you want to show a message. To do this, you place the following above the table:

```
<h4 tal:condition="not:container/objectValues">There
Are No Items</h4>
```

You can't use your `items` variable here, because it isn't defined yet. If you move the definition to the `h4` tag, you can't use it in the table tag any more, because it becomes a local variable of the `h4` tag. You could place the definition on some tag that enclosed both the `h4` and the table, but there is a simpler solution. By placing the keyword `global` in front of the variable name, you can make the definition last from the `h4` tag to the bottom of the template:

```
<h4 tal:define="global items container/objectValues"
  tal:condition="not:items">There Are No Items</h4>
<table border="1" width="100%"
  tal:condition="items">
```

The `not:` in the first `tal:condition` is an expression type prefix that can be placed in front of any expression. If the expression is true, `not:` is false, and vice versa.

2.1.10 Changing Attributes

Most, if not all, of the objects listed by your template have an `icon` property, that contains the path to the icon for that kind of object. In order to show this icon in the meta-type column, you will need to insert this path into the `src` attribute of an `img` tag, by editing the meta-type column in both rows to look like this:

```
<td>
  <span tal:replace="item/meta_type">Meta-Type</span>
</td>
```

The `tal:attributes` statement replaces the `src` attribute of the image with the value of `item/icon`. The value of `src` in the template acts as a placeholder, so that the image is not broken, and is the correct size.

Since the `tal:content` attribute on the table cell would have replaced the entire contents of the cell, including the image, with the meta-type text, it had to be removed. Instead, you insert the meta-type inline in the same fashion as the URL at the top of the page.

2.1.11 Conclusion

This introduction to Page Templates shows you how you can create and edit simple templates, and how you can use the TAL and TALES languages to create dynamic content without munging your HTML or breaking your fancy editing tools. As you can see, Page Templates are designed to make both the HTML designer and the web application programmer work together seamlessly without being at odds with each others requirements. In the next article in this series, we'll explore some of the more advanced features of page templates.

2.2 Zope Page Templates: Advanced Usage

by Evan Simpson

In our first [article](http://www.zope.org/Documentation/Articles/ZPT1) (<http://www.zope.org/Documentation/Articles/ZPT1>) we described how to use page templates and their simple language, TAL, to create dynamic presentation. In this article, we'll look at some more advanced features of TAL. In the second half of this article, we'll look at the TAL Expression Syntax, or *TALES*.

2.2.1 Mixing and Matching Statements

As you have seen in the example template, you can put more than one TAL statement on the same tag. There are three limits you should be aware of, however.

1. Only one of each kind of statement can be used on a single tag. Since HTML does not allow multiple attributes with the same name, you can't have two `tal:define` on the same tag.
2. Both of `tal:content` and `tal:replace` cannot be used on the same tag, since their functions conflict.
3. The order in which you write TAL attributes on a tag does not affect the order in which they execute. No matter how you arrange them, the TAL statements on a tag always execute in the following order: define, condition, repeat, content / replace, attributes.

To get around these limits, you can add another tag and split up the statements between the tags. If there is no obvious tag type that would fit, use `span` or `div`.

For example, if you want to define a variable for each repetition of a paragraph, you can't place the `tal:define` on the same tag as the `tal:repeat`, since the definition would happen before all of the repetitions. Instead, you would write either of the following:

```
<div tal:repeat="p phrases">
  <p tal:define="n repeat/p/number">
    Phrase number <span tal:replace="n">1</span> is
    "<span tal:replace="p">Phrase</span>".</p>
</div>
```

```
<p tal:repeat="p phrases">
  <span tal:define="n repeat/p/number">
    Phrase number <span tal:replace="n">1</span> is
    "<span tal:replace="p">Phrase</span>".</span>
</p>
```

2.2.2 Statements with Multiple Parts

If you need to set multiple attributes on a tag, you can't do it by placing multiple `tal:attributes` statements on the tag, and splitting them across tags is useless.

Both the `tal:attributes` and `tal:define` statements can have multiple parts in a single statement. You separate the parts with semicolons (;), so any semicolon appearing in an expression in one of these statements must be escaped by doubling it (;;). Here is an example of setting both the `src` and `alt` attributes of an image:

```

```

Here is a mixture of variable definitions:

```
<span tal:define="global logo here/logo.gif; ids
  here/objectIds">
```

2.2.3 String Expressions

String expressions allow you to easily mix path expressions with text. All of the text after the leading string: is taken and searched for path expressions. Each path expression must be preceded by a dollar sign (\$). If it has more than one part, or needs to be separated from the text that follows it, it must be surrounded by braces ({}). Since the text is inside of an attribute value, you can only include a double quote by using the entity syntax """. Since dollar signs are used to signal path expressions, a literal dollar sign must be written as two dollar signs (\$\$). For example:

```
"string:Just text."
"string:© $year, by Me."
"string:Three ${vegetable}s, please."
"string:Your name is ${user.getUserName}!"
```

2.2.4 Nocall Path Expressions

An ordinary path expression tries to render the object that it fetches. This means that if the object is a function, Script, Method, or some other kind of executable thing, then expression will evaluate to the result of calling the object. This is usually what you want, but not always. For example, if you want to put a DTML Document into a variable so that you can refer to its properties, you can't use a normal path expression because it will render the Document into a string. If you put the nocall: expression type prefix in front of a path, it prevents the rendering and simply gives you the object. For example:

```
<span tal:define="doc nocall:here/aDoc"
  tal:content="string:${doc/id}: ${doc/title}">
  Id: Title</span>
```

This expression type is also valuable when you want to define a variable to hold a function or class from a module, for use in a Python expression.

2.2.5 Python Expressions

A Python expression starts with python:, followed by an expression written in the Python language. See the section on writing Python expressions for more information.

2.2.6 Other Builtin Variables

You have already seen some examples of the builtin variables template, user, repeat, and request. Here is a complete list of the other builtin variables and their uses:

- ❏ *nothing*: a false value, similar to a blank string, that you can use in tal:replace or tal:content to erase a tag or its contents. If you set an attribute to nothing, the attribute is removed from the tag (or not inserted), unlike a blank string.

- ❏ *default*: a special value that doesn't change anything when used in tal:replace, tal:content, or tal:attributes. It leaves the template text in place.
- ❏ *options*: the keyword arguments, if any, that were passed to the template.
- ❏ *attrs*: a dictionary of attributes of the current tag in the template. The keys are the attributes names, and the values are the original values of the attributes in the template.
- ❏ *root*: the root Zope object. Use this to get Zope objects from fixed locations, no matter where your template is placed or called.
- ❏ *here*: the object on which the template is being called. This is often the same as the *container*, but can be different if you are using acquisition. Use this to get Zope objects that you expect to find in different places depending on how the template is called.
- ❏ *container*: the container (usually a Folder) in which the template is kept. Use this to get Zope objects from locations relative to the template's permanent home.
- ❏ *modules*: the collection of Python modules available to templates. See the section on writing Python expressions.

2.2.7 Alternate Paths

The path template/title is guaranteed to exist every time the template is used, although it may be a blank string. Some paths, such as request/form/x, may not exist during some renderings of the template. This normally causes an error when the path is evaluated.

When a path doesn't exist, you often have a fallback path or value that you would like to use instead. For instance, if request/form/x doesn't exist, you might want to use here/x instead. You can do this by listing the paths in order of preference, separated by vertical bar characters (|):

```
<h4 tal:content="request/form/x | here/x">Header</h4>
```

Two variables that are very useful as the last path in a list of alternates are nothing and default. Use nothing to blank the target if none of the paths is found, or default to leave the example text in place.

You can also test the existence of a path directly with the exists: expression type prefix. A path expression with exists: in front of it is true if the path exists, false otherwise. These examples both display an error message only if it is passed in the request:

```
<h4 tal:define="err request/form/errmsg | nothing"
  tal:condition="err" tal:content="err">Error!</h4>
```

```
<h4 tal:condition="exists:request/form/errmsg"
  tal:content="request/form/errmsg">Error!</h4>
```

2.2.8 Dummy Elements

You can include page elements that are visible in the template but not in generated text by using the builtin variable nothing, like this:

```
<tr tal:replace="nothing">
  <td>10213</td><td>Example Item</td><td>$15.34</td>
</tr>
```

This can be useful for filling out parts of the page that will take up more of the generated page than of the template. For instance, a table that usually has ten rows will only have one row in the template. By adding nine dummy rows, the template's layout will look more like the final result.

2.2.9 Inserting Structure

Normally, the `tal:replace` and `tal:content` statements quote the text that they insert, converting `<` to `<`, for instance. If you actually want to insert the unquoted text, you need to precede the expression with the `structure` keyword. Given a variable `copyright`, the following two lines:

```
<span tal:replace="copyright">Copyright 2000</span>
<span tal:replace="structure copyright">
  Copyright 2000</span>
```

...might generate "© 2001 By **Me**" and "© 2001 By Me" respectively.

This feature is especially useful when you are inserting a fragment of HTML that is stored in a property or generated by another Zope object. For instance, you may have news items that contain simple HTML markup such as bold and italic text when they are rendered, and you want to preserve this when inserting them into a "Top News" page. In this case, you might write:

```
<p tal:repeat="article topnewsitems"
  tal:content="structure article">A News Article</p>
```

2.2.10 Basic Python Expressions

The Python language is a simple and expressive one. If you have never encountered it before, you should read one of the excellent tutorials or introductions available at the website <http://www.python.org>.

A Page Template Python expression can contain anything that the Python language considers an expression. You can't use statements such as `if` and `while`, and Zope's security restrictions are applied.

Comparisons

One place where Python expressions are practically necessary is in `tal:condition` statements. You usually want to compare two strings or numbers, and there isn't any other way to do that. You can use the comparison operators `<` (less than), `>` (greater than), `==` (equal to), and `!=` (not equal to). You can also use the boolean operators `and`, `not`, and `or`. For example:

```
<p tal:repeat="widget widgets">
  <span tal:condition="python:widget.type == 'gear'">
    Gear #<span tal:replace="repeat/widget/number">1
  </span>:
  <span tal:replace="widget/name">Name</span>
</span>
</p>
```

Sometimes you want to choose different values inside a single statement based on one or more conditions. You can do this with the `test` function, like this:

```
You <span tal:define="name user/getUserName"
  tal:replace="python:test(name=='Anonymous User',
    'need to log in', default)">
  are logged in as
  <span tal:replace="name">Name</span>
  </span>

<tr tal:define="oddrow repeat/item/odd"
  tal:attributes="class python:test(oddrow, 'oddclass',
    'evenoddclass')">
```

Using other Expression Types

You can use other expression types inside of a Python expression. Each type has a corresponding function with the same name, including `path()`, `string()`, `exists()`, and `nocall()`. This allows you to write expressions such as:

```
"python:path('here/%s/thing' % foldername)"
"python:path(string('here/$foldername/thing'))"
"python:path('request/form/x') or default"
```

The final example has a slightly different meaning than the `path` expression `"request/form/x | default"`, since it will use the default text if `"request/form/x"` doesn't exist or if it is false.

2.2.11 Getting at Zope Objects

Much of the power of Zope involves tying together specialized objects. Your Page Templates can use Scripts, SQL Methods, Catalogs, and custom content objects. In order to use them, you have to know how to get access to them.

Object properties are usually attributes, so you can get a template's title with the expression `"template.title"`. Most Zope objects support acquisition, which allows you to get attributes from "parent" objects. This means that the Python expression `"here.Control_Panel"` will acquire the Control Panel object from the root Folder. Object methods are attributes, as in `"here.objectIds"` and `"request.set"`. Objects contained in a Folder can be accessed as attributes of the Folder, but since they often have Ids that are not valid Python identifiers, you can't use the normal notation. For example, instead of writing `"here.penguin.gif"`, you must write `"getattr(here, penguin.gif)"`. Some objects, such as `request`, `modules`, and Zope Folders support item access. Some examples of this are:

```
request['URL'], modules['math'], and here['thing']
```

When you use item access on a Folder, it doesn't try to acquire the name, so it will only succeed if there is actually an object with that Id contained in the Folder.

As shown in previous chapters, path expressions allow you to ignore details of how you get from one object to the next. Zope tries attribute access, then item access. You can write `"here/images/penguin.gif"` instead of `"python:getattr(here, images, penguin.gif)"`, and `"request/form/x"` instead of `"python:request.form!['x']"`.

The tradeoff is that path expressions don't allow you to specify those details. For instance, if you have a form variable named `"get"`, you must write `"python:request.form!['get']"`, since `"request/form/get"` will evaluate to the `"get"` method of the form dictionary.

2.2.12 Using Scripts

Script objects are often used to encapsulate business logic and complex data manipulation. Any time that you find yourself writing lots of TAL statements with complicated expressions in them, you should consider whether you could do the work better in a Script.

Each Script has a list of parameters that it expects to be given when it is called. If this list is empty, then you can use the Script by writing a path expression. Otherwise, you will need to use a Python expression, like this:

```
"python:here.myscript(1, 2)"
"python:here.myscript('arg', foo=request.form['x'])"
```

If you want to return more than a single bit of data from a Script to a Page Template, it is a good idea to return it in a dictionary. That way, you can define a variable to hold all the data, and use path expressions to refer to each bit. For example:

getPerson returns this: {'name': 'Fred', 'age': 25}

```
<span tal:define="person here/getPerson"
  tal:replace="string:${person/name} is
  ${person/age}">
  Name is 30</span> years old.
```

2.2.13 Calling DTML

Unlike Scripts, DTML Methods don't have an explicit parameter list. Instead, they expect to be passed a client, a mapping, and keyword arguments. They use these to construct a namespace.

When the ZPublisher publishes a DTML object, it passes the context of the object as the client, and the REQUEST as the mapping. When one DTML object calls another, it passes its own namespace as the mapping, and no client.

If you use a path expression to render a DTML object, it will pass a namespace with request, here, and the template's variables already on it. This means that the DTML object will be able to use the same names as if it were being published in the same context as the template, plus the variable names defined in the template.

2.2.14 Python Modules

The Python language comes with a large number of modules, which provide a wide variety of capabilities to Python programs. Each module is a collection of Python functions, data, and classes related to a single purpose, such as mathematical calculations or regular expressions.

Several modules, including "math" and "string", are available in Python Expressions by default. For example, you can get the value of π from the math module by writing "python:math.pi". To access it from a path expression, however, you need to use the modules variable. In this case, you would use "modules/math/pi". Please refer to the Zope Book or a DTML reference guide for more information about these modules.

The "string" module is hidden in Python expressions by the "string" expression type function, so you need to access it through the modules variable. You can do this directly in an expression in which you use it, or define a global variable for it, like this:

```
tal:define="global mstring modules/string"
tal:replace="python:mstring.join(slist, ':')"
```

Modules can be grouped into packages, which are simply a way of organizing and naming related modules. For instance, Zope's Python-based Scripts are provided by a collection of modules in the "PythonScripts" subpackage of the Zope "Products" package. In particular, the "standard" module in this package provides a number of useful formatting functions that are standard in the DTML "Var" tag. The full name of this module is "Products.PythonScripts.standard", so you could get access to it using either of the following statements:

```
tal:define="pps modules/Products.PythonScripts.standard"
tal:define="pps python:modules
[Products.PythonScripts.standard]"
```

Most Python modules cannot be accessed from Page Templates, DTML, or Scripts unless you add Zope security assertions to them. That's outside the scope of this document, and is covered by the Zope Security Guide.

2.3 TALES Specification Version 1.3

The *Template Attribute Language Expression Syntax* describes expressions that may be used to supply TAL and METAL with data. TALES is one possible expression syntax for these languages, but they are not bound to this definition. Similarly, TALES could be used in a context having nothing to do with TAL or METAL.

TALES expressions are described below with any delimiter or quote markup from higher language layers removed, since this is how expression strings must be passed to the TALES engine. Here is the basic definition of TALES syntax:

```
Expression ::= [type_prefix ':'] String
type_prefix ::= Name
```

See *EBNF for rules and terminals*. Here are some simple examples:

```
a/b/c
path:a/b/c
nothing
path:nothing
python: 1 + 2
string:Hello, ${username}
```

The optional *type prefix* determines the semantics and syntax of the *expression string* that follows it. A given implementation of TALES can define any number of expression types, with whatever syntax you like. It also determines which expression type is indicated by omitting the prefix.

Several expression types are required:

Required Type Prefixes

- ✎ *not* - evaluate the expression string (recursively) as a full expression, and returns the boolean negation of its value. If the expression supplied does not evaluate to a boolean value, *not* will issue a warning and *coerce* the expression's value into a boolean type based on the following rules: 1. integer 0 is *false* 2. integer > 0 is true 3. an empty string or other sequence is false 4. a non-empty string or other sequence is true 5. a *non-value* (e.g. void, None, Nil, NULL, etc) is *false* 6. all other values are implementation-dependent.

If no expression string is supplied, an error should be generated.

- ✎ *path* - interpret the expression string as the path to some object. The syntax and semantics of paths warrant their own section and thus are described below (see Path Expressions) If no expression string is supplied, the result is interpreted as the value nothing.

- ✎ *string* - interpret the expression string as text. If no expression string is supplied the resulting string is *empty*. The string can contain variable substitutions of the form `$name` or `${name}`, where `name` is an expression of type *path*. The escaped string value of the path expression is inserted into the string. To prevent a `$` from being interpreted this way, it must be escaped as `$$`. Note that leading and trailing spaces in the expression are included in the value.

Syntax:

```
string_expression ::= ( plain_string | [ varsub ] ) *
varsub             ::= ( '$' Path ) | ( '${' Path '}' )
plain_string       ::= ( '$$' | non_dollar ) *
non_dollar         ::= any character except '$'
```

Optional Type Prefixes

- ✎ *python* - interpret the expression string as restricted Python code. This code must be a legitimate python expression.

Path Expressions

A path expression consists of one or more **paths** separated by vertical bars (|). A **path** consists of one or more non-empty strings separated by slashes. The first string must be a variable name, and the remaining strings, the **path segments**, may contain letters, digits, spaces, and the punctuation characters underscore, dash, period, comma, and tilde.

Here is the syntax:

```
PathExpr ::= Path [ '|' Path ] *
Path      ::= variable [ '/' URL_Segment ] *
variable  ::= Name
```

For example:

```
request/cookies/oatmeal
nothing
here/some-file 2001_02.html.tar.gz/foo
root/to/branch | default
```

When a TALES path expression is evaluated, it attempts to traverse each path, from left to right, until it succeeds or runs out of paths. To traverse a path, it first fetches the object stored in the variable. For each path segment, it traverses from the current object to the subobject named by the path segment.

Once a path has been successfully traversed, the resulting object is the value of the expression. If it is a callable object, such as a method or class, it is called. The semantics of traversal (and what it means to be callable) are implementation-dependent..

If a traversal step fails, evaluation immediately proceeds to the next path. If there are no further paths, an error results.

Since every path must start with a variable name, you need a set of starting variables that you can use to find other objects and values. PageTemplates define the variable names listed below. Since variable names are looked up first in locals, then in globals, then in this list, these names act just like builtins in Python; They are always available, but they can be shadowed by a global or local variable declaration. You can always access the builtin names explicitly by prefixing them with *CONTEXTS*. (e.g. *CONTEXTS/root*, *CONTEXTS/nothing*, etc).

Builtin Names in Page Templates

- ✎ `nothing` - special singleton value used by TAL to represent a non-value (e.g. void, None, Nil, NULL).
- ✎ `default` - special singleton value used by TAL to specify that existing text should not be replaced.
- ✎ `options` - the keyword arguments passed to the template.
- ✎ `repeat` - the repeat variables (see RepeatVariable).
- ✎ `attrs` - a dictionary containing the initial values of the attributes of the current statement tag.
- ✎ `CONTEXTS` - the list of standard names (this list). This can be used to access a builtin variable that has been hidden by a local or global variable with the same name.

Optional Names in Page Templates (used in Zope Page Templates)

- ✎ `root` - the system's top-most object.
- ✎ `here` - the object to which the template is being applied.
- ✎ `container` - the template's container object.
- ✎ `template` - the template itself.
- ✎ `request` - the publishing request object.
- ✎ `user` - the authenticated user object.
- ✎ `modules` - a collection through which all Python modules and packages can be accessed. Some or many of these may not be usable in TALES, however, depending on the security policies of the template's implementation.

2.31 TAL Specification Version 1.4

This specification supercedes [TAL Specification 1.2](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL%20Specification%201.2) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL%20Specification%201.2>).

The *Template Attribute Language* is an [AttributeLanguage](http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage>) used to create dynamic templates. It allows elements of a document to be replaced, repeated, or omitted.

The statements of TAL are XML attributes from the TAL namespace. These attributes can be applied to an XML or HTML document in order to make it act as a template.

A **TAL statement** has a name (the attribute name) and a body (the attribute value). For example, an content statement might look like `tal:content="string:Hello"`. The element on which a statement is defined is its **statement element**. Most TAL statements require expressions, but the syntax and semantics of these expressions are not part of TAL. [TALES](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES>) is recommended for this purpose.

The TAL namespace URI and recommended alias are currently defined as:

```
xmlns:tal="http://xml.zope.org/namespaces/tal"
```

This is not a URL, but merely a unique identifier. Do not expect a browser to resolve it successfully.

TAL Statements

See [EBNF](http://www.zope.org/Wikis/DevSite/Projects/ZPT/EBNF) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/EBNF>) for rules and terminals. See [AttributeLanguage](http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage>) for a description of attribute language statements. The following are the TAL 1.4 statements: define, attributes, condition, content, replace, repeat, on-error, omit-tag (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL%20Specification%201.4>). Each statement is described below, along with its argument syntax. Also, the [order of operations](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL%20Specification%201.4#oop) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL%20Specification%201.4#oop>) is described.

Although [TAL](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL>) does not define the syntax of expression non-terminals, leaving that up to the implementation, a canonical expression syntax for use in [TAL](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL>) arguments is described in [TALES](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES>).

Expressions used in statements may return values of any type, although most statements will only accept strings, or will convert values into a string representation. The expression language must define a value named `nothing` (see [TALES](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES>)) that is not a string. In particular, this value is useful for deleting elements or attributes.

Evaluation of an expression may cause the action (the effect of the statement) to be cancelled. In this case, it is as though the statement (or part of the statement, in some cases) did not exist.

Define

Syntax:

```
argument ::= define_scope [';' define_scope]*
define_scope ::= ('local' | 'global') define_var
define_var ::= variable_name expression
variable_name ::= Name
```

Note: *If you want to include a semi-colon (;) in an expression, it must be escaped by doubling it (;,).*

You can define two different kinds of TAL variables: local and global. When you define a local variable in a statement element, you can only use that variable in that element and the elements it contains. If you redefine a local variable in a contained element, the new definition hides the outer element's definition within the inner element. When you define a global variable, you can use it in any element processed after the defining element. If you redefine a global variable, you replace its definition for the rest of the Template.

If the expression associated with a variable evaluates to `nothing`, then that variable has the value *nothing*, and may be used as such in further expressions. If the expression cancels the action, then that variable is not (re)defined. This means that if the variable exists in an enclosing scope, its value is unchanged, but if it does not, it is not created. Each variable definition is independent, so variables may be defined in the same statement in which some variable definitions are cancelled.

Examples:

```
tal:define="mytitle template/title; tlen python:len(mytitle)"
tal:define="global company_name string:Digital Creations, Inc."
```

Attributes

Syntax:

```
argument          ::= attribute_statement
                  [';' attribute_statement]*
attribute_statement ::= attribute_name expression
attribute_name     ::= [namespace ':' ] Name
namespace         ::= Name
```

Note: If you want to include a semi-colon (;) in an expression, it must be escaped by doubling it (;;).

If you want to replace the value of an attribute (or create an attribute) with a dynamic value, you need the attributes statement. You can qualify an attribute name with a namespace prefix, for example `html:table`, if you are generating an XML document with multiple namespaces. The value of each expression is converted to a string, if necessary.

If the expression associated with an attribute assignment evaluates to *nothing*, then that attribute is deleted from the statement element. If the expression cancels the action, then that attribute is left unchanged. Each attribute assignment is independent, so attributes may be assigned in the same statement in which some attributes are deleted and others are left alone due to cancellation. Examples:

```
<a href="/sample/link.html"
  tal:attributes="href here/sub/absolute_url">
<textarea rows="80" cols="20"
  tal:attributes="rows request/rows;cols request/cols">
```

When this statement is used on an element with an active replace command, the implementation may ignore the attributes statement. If it does not, the replacement must use the structure type, the structure returned by the expression must yield at least one element, and the attributes will be replaced on the first such element only. For example, for the first line below, either of the two outcomes that follow it is acceptable:

```
<span tal:replace="structure an_image"
  tal:attributes="border string:1">


```

When this is used on an element with a repeat statement, the replacement is made on each repetition of the element, and the replacement expression is evaluated fresh for each repetition.

Condition

Syntax:

```
argument ::= expression
```

To include a particular part of a Template only under certain conditions, and omit it otherwise, use the condition statement. If its expression evaluates to a true value, then normal processing of the element continues, otherwise the statement element is immediately removed from the document. It is up to the interface between TAL and the expression engine to determine the value of true and false. For these purposes, the value *nothing* is false, and cancellation of the action has the same effect as returning a true value.

Example:

```
<p tal:condition="here/copyright"
  tal:content="here/copyright">(c) 2000</p>
```

Replace

Syntax:

```
argument ::= ([ 'text' ] | 'structure') expression
```

To replace an element with dynamic content, use the replace statement. This replaces the statement element with either text or a structure (unescaped markup). The body of the statement is an expression with an optional type prefix. The value of the expression is converted into an escaped string if you prefix the expression with `text` or omit the prefix, and is inserted unchanged if you prefix it with `structure`. Escaping consists of converting "&" to "&", "<" to "<", and ">" to ">".

If the value is *nothing*, then the element is simply removed. If the action is cancelled, then the element is left unchanged (see the TALEs *default* value).

Note: The default replacement behavior is text.

Examples:

```
<span tal:replace="template/title">Title</span>
<span tal:replace="text template/title">Title</span>
<span tal:replace="structure table" />
<span tal:replace="nothing">This element is a
comment.</span>
```

Content

Syntax:

```
argument ::= ([ 'text' ] | 'structure') expression
```

Rather than replacing an entire element, you can insert text or structure in place of its children with the content statement. The statement argument is exactly like that of `replace`, and is interpreted in the same fashion. If the expression evaluates to *nothing*, the statement element is left childless. If the action is cancelled, then the element's contents are unchanged.

Note: The default replacement behavior is text.

Example:

```
<p tal:content="user/name">Fred Farkas</p>
```

Repeat

Syntax:

```
argument      ::= variable_name expression
variable_name ::= Name
```

When you want to replicate a subtree of your document once for each item in a sequence, you use `repeat`. The expression should evaluate to a sequence. If the sequence is empty, then the statement element is deleted, otherwise it is repeated for each value in the sequence. If the action is cancelled, then the element is left unchanged, and no new variables are defined. The `variable_name` is used to define a local variable and a repeat variable. For each repetition, the local variable is set to the current sequence element, and the repeat variable is set to an iteration object. You use iteration objects to access information about the current repetition (such as the repeat index). (Iteration objects are more properly the domain of [TALES](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES>.) The repeat variable has the same name as the local variable,

but is only accessible through the builtin variable named `repeat` (see [RepeatVariable](http://www.zope.org/Wikis/DevSite/Projects/ZPT/RepeatVariable) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/RepeatVariable>)). Examples:

```
<p tal:repeat="txt python:'one', 'two', 'three'">
  <span tal:replace="txt" />
</p>
<table>
  <tr tal:repeat="item here/cart">
    <td tal:content="repeat/item/index">1</td>
    <td tal:content="item/description">Widget</td>
    <td tal:content="item/price">$1.50</td>
  </tr>
</table>
```

On-Error

Syntax:

```
argument ::= ([ 'text' ] | 'structure') expression
```

You can provide error handling for your document using `on-error`. When a TAL statement produces an error, the TAL interpreter searches for an `on-error` statement on the same element, then on the enclosing element, and so forth. The first `on-error` found is invoked. It is treated as a content statement. The simplest sort of `on-error` statement has a literal error string or *nothing* for an expression. A more complex handler may call a script that examines the error and either emits error text or raises an exception to propagate the error outwards. See [RenderErrorHandlingStrategies](http://www.zope.org/Wikis/DevSite/Projects/ZPT/RenderErrorHandlingStrategies) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/RenderErrorHandlingStrategies>) for further information.

Example:

```
<p tal:on-error="string: Error! This paragraph is buggy!">
  My name is <span tal:replace="here/SlimShady" />. <br />
  (My login name is
  <b tal:on-error="string: Username is not defined!"
    tal:content="user">Unknown</b>)
</p>
```

In the above example, if `here/SlimShady` results in an error, the `on-error` statement catches it and replaces the paragraph with the string "Error! This paragraph is buggy!". If `here/SlimShady` evaluates correctly, but there is an error evaluating `user`, then "Username is not defined!" replaces `Unknown`, but the rest of the paragraph is processed normally.

Omit-Tag

Syntax:

```
argument ::= [expression]
```

To leave the contents of a tag in place while omitting the surrounding start and end tag, use the `omit-tag` statement. If its expression evaluates to a *false* value, then normal processing of the element continues. If the expression evaluates to a *true* value, or there is no expression, the statement tag is replaced with its contents. It is up to the interface between TAL and the expression engine to determine the value of *true* and *false*. For these purposes, the value *nothing* is false, and cancellation of the action has the same effect as returning a false value.

Examples:

```
<div tal:omit-tag="" comment="This tag will be
  removed">
  <i>...but this text will remain.</i>
</div>
```

```
<b tal:omit-tag="not:bold">I may not be bold.</b>
```

Order of Operations

When there is only one TAL statement per element, the order in which they are executed is simple. Starting with the root element, each element's statements are executed, then each of its child elements is visited, in order, to do the same.

Any combination of statements may appear on the same elements, except that the content and replace statements may not appear together.

When an element has multiple statements, they are executed in this order:

- define
- condition
- repeat
- content or replace
- attributes
- omit-tag

Since the `on-error` statement is only invoked when an error occurs, it does not appear in the list.

The reasoning behind this ordering goes like this: You often want to set up variables for use in other statements, so `define` comes first. The very next thing to do is decide whether this element will be included at all, so `condition` is next; since the condition may depend on variables you just set, it comes after `define`. It is valuable to be able to replace various parts of an element with different values on each iteration of a `repeat`, so `repeat` is next. It makes no sense to replace attributes and then throw them away, so `attributes` is last. The remaining statements clash, because they each replace or edit the statement element.

If you want to override this ordering, you must do so by enclosing the element in another element, possibly `div` or `span`, and placing some of the statements on this new element.

Examples:

```
<p tal:define="x /a/long/path/from/the/root"
  tal:condition="x"
  tal:content="x/txt"
  tal:attributes="class x/class">Ex Text</p>
```

2.4 Frequently Asked Questions about Zope Page Templates

Q1: What are Page Templates (ZPT)?

See [VisionStatement](http://www.zope.org/Wikis/DevSite/Projects/ZPT/VisionStatement) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/VisionStatement>) for an overview.

Q2: Will ZPT replace DTML?

ZPT is an alternative to DTML and may replace most if not all of its usage for a particular audience. If you are a [PresentationDesigner](http://www.zope.org/Wikis/DevSite/Projects/ZPT/PresentationDesigner) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/PresentationDesigner>), you may find ZPT more useful than DTML.

Q3: Is DTML being deprecated?

No, we encourage new projects (and developers) to take a look at ZPT and see if it fits. At DC, we will probably use ZPT as much as possible on new projects. However, if DTML is a better fit...

Q4: Can I use ZPT without knowing XML?

Yes, you can use it without knowing anything about XML. ZPT's [AttributeLanguage](http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage>) was designed for compatibility with XML markup syntax, but this mostly means that it is simple and very well-defined.

Q4a: So, Why XML?

As the Web moves to XHTML (assuming you believe this will happen), XML will be the common underlying markup format. There is enough flexibility through the usage of namespaces, that we can work within the structure of XML to do the [DTMLish](http://www.zope.org/Wikis/DevSite/Projects/ZPT/FAQ/editform?page=DTMLish) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/FAQ/editform?page=DTMLish>) things we wish to do. Also, we expect more HTML editors, in the future, to support XML than DTML ;-)

Q4b: Okay, but why be restricted to working within attributes?

Defining our own elements was certainly a possibility. Unfortunately, we expect that most HTML editors will continue to want to deal with HTML, not arbitrary XML. Defining your own element set (through a DTD or Schema) requires a lot of processing overhead on the editor side in order to do anything useful with that markup. Part of the core idea of ZPT is to allow [PresentationDesigners](#) to concentrate on the presentation (the HTML) rather than some newly defined elements that don't visually render themselves in the editor.

Q5: Does ZPT require XHTML?

ZPT will work within plain old HTML. The ZPT markup exists in attributes of your HTML tags, so if your HTML editor leaves unfamiliar attributes alone, it should work with ZPT templates.

Q6: What is XHTML (briefly, please)?

Loosely speaking, XHTML is HTML with stricter syntax and semantics. As a rule, all tags must have *end tags*. And, all attributes must follow a syntax of `attributename = "something always in quotes"`. Also, certain elements cannot contain other elements as children. That's pretty much it (with apologies to XML language lawyers).

Q7: My favorite editor produces plain old HTML. Can I use ZPT?

Yes, you have a couple of choices. The best is to use an editor that supports XML (meaning: it doesn't muck with your elements or use of *namespaced* attributes). If your editor doesn't mess with stuff like this:

```
<span tal:define="x str:1" />
```

Then you can probably use ZPT.

However, you can also use a tool like HTML Tidy to convert your HTML to XHTML (which ZPT naturally speaks). The side effect of this is that your output doesn't match your original source! This is not an optimal solution.

Q8: What HTML editors work well with ZPT?

We are still compiling a list, but on the commercial (industrial strength) front, [GoLive](http://www.zope.org/Wikis/DevSite/Projects/ZPT/GoLive) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/GoLive>) and Dreamweaver 4 work well with ZPT. Recent versions of Amaya work well in HTML mode.

Q9: Ideally, how does one use an HTML editor with ZPT?

ZPT wants to be fairly tightly integrated with your editing process. Right now, this means using the ZPT HTML edit form (yuck) or using a HTML editor that works with FTP or [WebDAV](http://www.zope.org/Wikis/DevSite/Projects/ZPT/WebDAV) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/WebDAV>). Since ZPT expects you to retrieve your editable sources from Zope, this is the most natural way to work: Zope has the canonical sources for the site.

Q10: Why does ZPT talk about TAL and METAL being non-Zope specific?

We thought that these ideas were so cool, that we shouldn't hog them all to ourselves. We, of course, expect to have the most butt kicking implementation of the specs, but we wanted to make sure we had a general enough specification so that people who *steal* the idea (and all good ideas should be stolen!), will not produce widely different implementations. We need convergence, not divergence!

Q11: Does this mean that they will be proposed as standards to a body such as the W3C?

Currently, we have no plans to propose TAL and METAL as a standard. Unlike many standards proposals, we want to have a working implementation first ;-)

Q12: How do I run the tests? Should they all pass?

The TAL README.txt file describes how to run the test suite for TAL. If you do not have expat installed, the XML tests will not pass. The ZPT tests can be run by executing run.py from the tests directory.

Q13: Does acquisition work with these path expressions in TALES?

Path expressions such as "here/master_page/macros/copyright" must always start with a builtin or defined variable ("here", in this case). Each step in the path will take advantage of acquisition if the current object supports it, so "master_page" may be acquired from "here". This is different from DTML, where every name is searched for in a large, diverse set of locations many of which support acquisition. locations

Q14: How do I write comments that won't be included in the rendered page?

Use tal:replace="nothing" or tal:condition="nothing", like this:

```
<span tal:replace="nothing">A Comment</span>  
  <span tal:replace="nothing"><-- A Commented  
  Comment --></span>
```

Q15: When I use tal:attributes to set the src or href attribute of a tag, it turns my ampersands into &s! How can I stop it?

You can't, and shouldn't. According to the standard, escaping is the proper thing to do here. It works properly with every browser we've tried.

3.1 METAL for Beginners

Created by [toner](http://www.zope.org/Members/toner) (<http://www.zope.org/Members/toner>). Last modified on 2001/08/01.

3.2 A beginners guide to METAL

This is part of an effort to 'give something back' into the pot of the ZPT (Zope Page Templates) system. I'm certainly not an expert in ZPT, so I'm likely to have got a few things wrong - please let me know and I'll try to get them fixed. This is not an introduction to ZPT - there's plenty of other resources to help you there.

3.2.1 What is METAL?

It's a macro language built into the ZPT system. I'll let Evan Simpson (leading light in the ZPT community) describe what a macro is used for

Macros are a way to share chunks of presentation, and have the shared stuff appear inline in the template. You don't use them for plugging in data values; That's what tal:replace and tal:content are for. - Evan Simpson

3.2.2 Getting Started with Macros

Firstly, create a page template document, call it 'mymacro'. The standard document looks like this;

```
<html>
<head>
  <title tal:content="template/title">The title</title>
</head>
<body>

  <h2><span tal:replace="here/title_or_id">content
    title or id</span>
    <span tal:condition="template/title"
      tal:replace="template/title">optional template
      id</span></h2>

  This is Page Template <em tal:content="template/id">
    template id</em>.
</body>
</html>
```

Ok, now change the html tag so it looks like this

```
<html xmlns:tal="http://xml.zope.org/namespaces/tal"
  xmlns:metal="http://xml.zope.org/namespaces/metal"
  metal:define-macro="master">
```

I dunno about you, but seeing all that XML namespace stuff made me a little worried initially (having been on a few XML lists where namespaces stoked up some dreadful religious wars didn't help...). The good news is you don't need them, so you can change the html tag to this;

```
<html
  metal:define-macro="master">
Much nicer.
```

Ok, we'll start using macros now, we'll add a navigation bar and a nice tabular outline - change the page template 'mymacro' so it looks like this

```
<html metal:define-macro="master">
  <head>
    <title tal:content="template/title">The title</title>
  </head>
  <body>
<table border="1">
<tr><td>I am a navigation bar - put links in here
  </td></tr>
<tr><td>
  <h2><span tal:replace="here/title_or_id">content
    title or id</span>
    <span tal:condition="template/title"
      tal:replace="template/title">optional template id
    </span></h2>
  <div metal:define-slot="main">
    If you supply a tag with a 'fill-slot="main"' attribute
    when using this macro, that tag will replace this text.
    This is the main block. It contains everything.
  </div>

</td>
</tr>
</table>
</body>
</html>
```

Use the 'test' tab to see what it looks like.

To use this macro effectively, you'll need to call it from another page template, so create one called 'use_mymacro' and enter the following;

```
<html metal:use-macro="here/mymacro/macros/master">
  <head>
    <title tal:content="template/title">The title</title>
  </head>
  <body>
    <p>I know nothing</p>
  </body>
</html>
```

BIG TIP: Don't select the 'Expand Macros when Editing' check box. It expands out all the macro content and you lose your clean document. I may have misunderstood how 'Expand Macros...' works though.

If you 'test' the 'use_mymacro' page template you've just created, you'll get a big surprise, the 'I know nothing' text isn't displayed.

However, the page itself gives you a huge clue as to what to do next. *If you supply a tag with a 'fill-slot="main"' attribute when using this macro, that tag will replace this text. This is the main block. It contains everything.* . Ok, change the 'use_mymacro' page template so it looks like this;

```
<html metal:use-macro="here/mymacro/macros/master">
  <head>
    <title tal:content="template/title">The title</title>
  </head>
  <body>
    <p metal:fill-slot="main">I know nothing</p>
  </body>
</html>
```

Ok, now 'test' this page template - success!. You could use standard tal:replace and tal:content structures in the <p> tag

and **that** content will be used instead of the static content I've put in.

A word about using slots. Evan Simpson says this about using slots on the ZPT list;

You only need to use slots if you want to be able to override parts of a macro in templates that use it. To do this, give a tag in the macro definition a slot name with `metal:define-slot`. When you use this macro, the slot contents will be used normally along with the rest of the macro, unless you add a `metal:fill-slot` statement with the slot's name. A `metal:fill-slot` statement tag completely replaces the slot tag from the macro definition, including the tag name and any attributes.

3.2.3 What I use Macros for

I use macros to 'manage away' the interface elements of my site. I should also be able to render completely different layouts by changing the `metal:use-macro="here/mymacro/macros/master"` attribute in the html tag, eg for visually impaired people. By changing the 'mymacro' part of the attribute, totally different output can be generated.

3.2.4 Links and Other Resources

You can get to a lot of resources from the ZPT site, but here's ones I use all the time

- 🔗 [The ZPT site](http://dev.zope.org/Wikis/DevSite/Projects/ZPT)
(<http://dev.zope.org/Wikis/DevSite/Projects/ZPT>)
- 🔗 [Recent changes to the ZPT site](http://dev.zope.org/Wikis/DevSite/Projects/ZPT/RecentChanges)
(<http://dev.zope.org/Wikis/DevSite/Projects/ZPT/RecentChanges>)
- 🔗 [Evan Simpsons' ZPT tutorial \(4 parts\)](http://dev.zope.org/Wikis/DevSite/Projects/ZPT/SimpleTutorial)
(<http://dev.zope.org/Wikis/DevSite/Projects/ZPT/SimpleTutorial>)
- 🔗 [Peter Bengtssons' DTML --> ZPT page](http://www.zope.org/Members/peterbe/DTML2ZPT)
(<http://www.zope.org/Members/peterbe/DTML2ZPT>)

I'll add to this HowTo when I get more METAL-Zen.

3.2.5 METAL Specification Version 1.0

The *Macro Expansion Template Attribute Language* is an [AttributeLanguage](http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage>) for structured macro preprocessing. It can be used in conjunction with or independently of [TAL](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL>), [TALES](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES>) and [ZPT](http://www.zope.org/Wikis/DevSite/Projects/ZPT) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT>).

Macros provide a way to define a chunk of presentation in one template, and share it in others, so that changes to the macro are immediately reflected in all of the places that share it. Additionally, macros are always fully expanded, even in a template's source text, so that the template appears very similar to its final rendering.

The METAL namespace URI and recommended alias are currently defined as:

```
xmlns:metal="http://xml.zope.org/namespaces/metal"
```

METAL Statements

See [EBNF](http://www.zope.org/Wikis/DevSite/Projects/ZPT/EBNF) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/EBNF>) for rules and terminals. *See [AttributeLanguage](http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage>) for a description of attribute language statements.*

The following are the names of the required TAL 1.0 statements: `define-macro`, `use-macro`, `define-slot`, `use-slot` (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/METAL%20Specification%201.0>). Each statement is described below, along with its argument syntax.

Although METAL does not define the syntax of expression non-terminals, leaving that up to the implementation, a canonical expression syntax for use in METAL arguments is described in [TALES Specification 1.0](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES%20Specification%201.0) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES%20Specification%201.0>).

Define-Macro

Syntax:
argument ::= Name

To define a macro, choose a name for the macro and add a `define-macro` attribute to a document element with the name as the argument. The element's subtree is the macro body. Example:

```
<p metal:define-macro="copyright">
  Copyright 2001, <em>Foobar</em> Inc.
</p>
```

Use-Macro

Syntax:
argument ::= expression

To use a macro, first choose the document element that you want to replace. Add a `use-macro` attribute to it, and set the value of the attribute to an expression that will return a macro definition. This will usually be some kind of path or template id and a macro name. It is important to remember that this expression will be evaluated separately from any TAL commands in the template, so it cannot depend on any such commands.

Note: [PageTemplates](http://www.zope.org/Wikis/DevSite/Projects/ZPT/PageTemplates) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/PageTemplates>) use [TALES](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES>) for the expression, and you can use any of the standard context names in path expressions. Since they expose their collection of macro definitions through a `macros` attribute, you can access individual macros easily.

For example:

```
<hr />
<p metal:use-macro="here/master_page/macros/
  copyright">
</hr />
```

Macro Expansion

The effect of expanding a macro is to graft a subtree from another document (or from elsewhere in the current document) in place of the statement element, replacing the exist-

ing subtree. Parts of the original subtree may remain, grafted onto the new subtree, if the macro has slots. If the macro body uses any macros, they are expanded first.

When a macro is expanded, its `define-macro` attribute is replaced with the `use-macro` attribute from the statement element. This makes the root of the expanded macro a valid `use-macro` statement element.

Define-Slot and Fill-Slot

Syntax:

argument ::= Name

Macros are much more useful if you can override parts of them when you use them. For example, you might want to reuse a complex table, but provide different contents for one of the table cells in every place that you use it. It would be possible to place the contents somewhere on each Template, define a variable for it, and insert that variable into the cell in the macro body. This, however, would violate the presentation structure, preventing you from seeing the table with the cell contents in place.

To make elements of the macro body overridable, add `define-slot` attributes with the value set to a slot name. Wherever the macro is used, choose corresponding sub-elements of the statement element and add `fill-slot` attributes with the value set to the slot name. When the macro is expanded, `fill-slot` elements will replace the `define-slot` elements in the macro body that use the same slot name.

Slot names must be unique within a single macro, and within a single macro use. It is legal, however, to define a slot in a macro and not fill it. This will simply cause the default contents of the slot definition to be copied into the expanded macro. If a `fill-slot` element names a slot that is not found in the macro body, it causes an error.

Examples:

In doc1:

```
<table metal:define-macro="sidebar">
  <tr><th>Links</th></tr>
  <tr><td metal:define-slot="links">
    <a href="/">A Link</a>
  </td></tr>
</table>
```

In doc2:

```
<table metal:use-macro="here/doc1/macros/sidebar">
  <tr><th>Links</th></tr>
  <tr><td metal:fill-slot="links">
    <a href="http://www.goodplace.com">Good
      Place</a><br>
    <a href="http://www.badplace.com">Bad
      Place</a><br>
    <a href="http://www.otherplace.com">Other
      Place</a>
  </td></tr>
</table>
```

Notice that doc2, which uses the macro defined in doc1, contains the entire text of the sidebar macro except for the links slot. This is because the macro is inserted every time the source of doc2 is edited.

Open Publication License

Draft v1.0, 8 June 1999

I. REQUIREMENTS ON BOTH UNMODIFIED AND MODIFIED VERSIONS

The Open Publication works may be reproduced and distributed in whole or in part, in any medium physical or electronic, provided that the terms of this license are adhered to, and that this license or an incorporation of it by reference (with any options elected by the author(s) and/or publisher) is displayed in the reproduction.

Proper form for an incorporation by reference is as follows:

Copyright (c) <year> by <author's name or designee>. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, vX.Y or later (the latest version is presently available at <http://www.opencontent.org/openpub/>).

The reference must be immediately followed with any options elected by the author(s) and/or publisher of the document (see section VI).

Commercial redistribution of Open Publication-licensed material is permitted.

Any publication in standard (paper) book form shall require the citation of the original publisher and author. The publisher and author's names shall appear on all outer surfaces of the book. On all outer surfaces of the book the original publisher's name shall be as large as the title of the work and cited as possessive with respect to the title.

II. COPYRIGHT

The copyright to each Open Publication is owned by its author(s) or designee.

III. SCOPE OF LICENSE

The following license terms apply to all Open Publication works, unless otherwise explicitly stated in the document.

Mere aggregation of Open Publication works or a portion of an Open Publication work with other works or programs on the same media shall not cause this license to apply to those other works. The aggregate work shall contain a notice specifying the inclusion of the Open Publication material and appropriate copyright notice.

SEVERABILITY. If any part of this license is found to be unenforceable in any jurisdiction, the remaining portions of the license remain in force.

NO WARRANTY. Open Publication works are licensed and provided "as is" without warranty of any kind, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose or a warranty of non-infringement.

IV. REQUIREMENTS ON MODIFIED WORKS

All modified versions of documents covered by this license, including translations, anthologies, compilations and partial documents, must meet the following requirements:

1. The modified version must be labeled as such.
2. The person making the modifications must be identified and the modifications dated.
3. Acknowledgement of the original author and publisher if applicable must be retained according to normal academic citation practices.
4. The location of the original unmodified document must be identified.
5. The original author's (or authors') name(s) may not be used to assert or imply endorsement of the resulting document without the original author's (or authors') permission.

V. GOOD-PRACTICE RECOMMENDATIONS

In addition to the requirements of this license, it is requested from and strongly recommended of redistributors that:

1. If you are distributing Open Publication works on hardcopy or CD-ROM, you provide email notification to the authors of your intent to redistribute at least thirty days before your manuscript or media freeze, to give the authors time to provide updated documents. This notification should describe modifications, if any, made to the document.
2. All substantive modifications (including deletions) be either clearly marked up in the document or else described in an attachment to the document.
3. Finally, while it is not mandatory under this license, it is considered good form to offer a free copy of any hardcopy and CD-ROM expression of an Open Publication-licensed work to its author(s).

VI. LICENSE OPTIONS

The author(s) and/or publisher of an Open Publication-licensed document may elect certain options by appending language to the reference to or copy of the license. These options are considered part of the license instance and must be included with the license (or its incorporation by reference) in derived works.

A. To prohibit distribution of substantively modified versions without the explicit permission of the author(s). "Substantive modification" is defined as a change to the semantic content of the document, and excludes mere changes in format or typographical corrections.

To accomplish this, add the phrase 'Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.' to the license reference or copy.

B. To prohibit any publication of this work or derivative works in whole or in part in standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.

To accomplish this, add the phrase 'Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.' to the license reference or copy.