Building an HTML macro toolbox

Andrew Peel (A.T.Peel@ukc.ac.uk)
Computing Laboratory
University of Kent at Canterbury
Canterbury, Kent CT2 7NF
United Kingdom

Telephone: +44 1227 82 3979 Fax: +44 1227 762 811

14th February 1996

Abstract

Nobody in their right mind writes in raw troff or TEX. Instead, a higher level of control is attained through the use of troff ms (or mm) macros, or LATEX, respectively. In the same way, HTML (Hypertext Markup Language) is crude and cumbersome.

Work on HTML macros (Peel, 1996) by the author has already demonstrated their value in creating complex web texts for publication on the World-Wide Web. The Tiler macro processor can be used to provide navigation and a consistent look and feel with very little effort. This is done by using standard HTML tags as building blocks for more more intricate and sophisticated components. The reduction in the amount of manual HTML markup required by the developer also results in a more robust and maintainable end product.

The paper discusses problems found with the macro processor software (Tiler v1.1) which expands macros in requested documents on-the-fly. The solution was to build a macro *pre*-processor (Tiler v2.0), complete with its own *Macro Definition Language*. The paper finishes with a description of Rummage, a tool for inserting structure-related macros into collections of web nodes based on information gleaned from a *framework file* which holds the structure of the web text.

Themes: User-centred methods and tools for designing Web structures and browsers
User-centred requirements for next generation Web authoring/navigation

1 Introduction

1.1 Review of the Tiler macro processor

The Tiler HTML macro processor is a complex CGI (Common Gateway Interface) (McCool, 1993) program. It receives a request for a document through the CGI GET method (ie. as the *query* part of a URL) which it retrieves from the local filestore. The document is searched for HTML macros which are then expanded in a pre-programmed way.

For example, a glossary button can be created using a Tiler `GLOSS` macro in the following way:

```
<GLOSS WORD="Arithmetic Logic Unit (ALU)" KEY="alu">
```

When included in an HTML file and filtered by Tiler, the following HTML is delivered to the browser:

```
<A HREF="/cgi-bin/tiler/glossary?key=alu>Arithmetic Logic
Unit (ALU)</A>
```

Thus the browser creates a hyperlink to the specified URL; that of another CGI script called `glossary`. The `WORD` parameter is used for the text of the link, and the `KEY` parameter is passed to the glossary script, which uses it to search for the matching key, and hence glossary entry, in a glossary database. The recovered glossary entry could also contain HTML macros, as CGI scripts can pass their output to Tiler's input using a Unix pipe.

1.2 Problems with the Tiler macro processor

There are two problems that prevent widespread use of the Tiler macro processor as it stands.

- Server-side on-the-fly expansion is expensive in terms of processor load, and use of the package by popular web sites would, probably, result in a decrease in response time and frequency.
- The Tiler software is written in Perl (Wall and Schwartz, 1992) and the macro definitions are hardwired into the source code. Hence adding extra macros requires the user to be able to program in Perl.
- If a user downloading a document which is being filtered for macros on-the-fly and they quit before the job is finished then the filter process remains running as a "zombie" process on the remote Unix machine.

2 The move to pre-processing

The Tiler macro processor puts an extra load onto web servers. Pre-processing can be used to reduce that load by expanding any HTML macros before-hand and removing the need for on-the-fly processing.

This has many implications, however:

- CGI tools, such as the glossary above, could previously produce HTML macros on their output which was then expanded on-the-fly. Making the macro processor static, rather than dynamic, takes away this functionality.

  A simple solution is for the CGI Scripts in question to save their output to a temporary file which they then expand with the pre-processor tool. The temporary file can then be returned to the client before it is deleted.

- With the previously dynamic macro processor, it would have been simple to add an ability to "tailor" documents as the each document requested is filtered on-the-fly already.

  Again, a solution would be simple to arrange. Each document would be to run the pre-processed document through a dynamic filter (built purely for tailoring). This does make the whole publishing process more complicated and error-prone none-the-less.

- There is no facility for persistence of user preferences (Peel, 1996).
- There is no "emergency exit button" functionality (Peel, 1996).

2.1 Some simple mechanics

Figure 1 illustrates the operation of a macro processor, as we have described above.

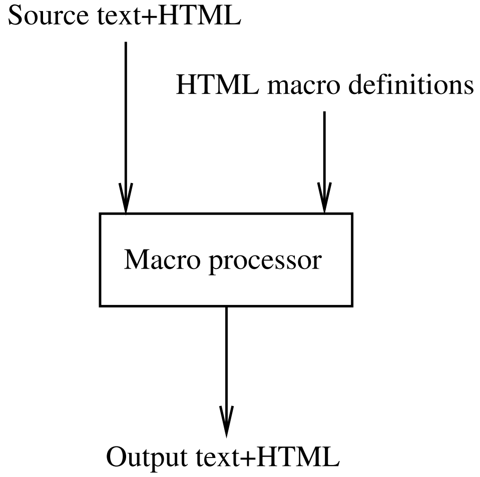

Figure 1: Operation of a macro processor

There are several methods available to achieve pre-processing.

2.2 The on-the-spot method

In this approach, the source file is replaced by the destination file. The destination file keeps details of the macros needed to create the resultant HTML so that the source file can be recreated for editing (see figure 2). This is done by wrapping the resultant HTML in an `EXPANSION` block, as can be seen in figure 3:

Currently, `EXPANSION` is not part of the HTML standard and is therefore not recognised by web browsers; it is ignored. So not only can the source file be recreated by removing the `EXPANSION` tags, they can also be used for debugging purposes.

The benefits of this scheme are that less disk space is required (as only one version of the file is ever physically stored), and that it is perhaps more intuitive for the novice user than the dual document tree arrangement below.

The main drawback found with constant use was that a lot of time was spent waiting for documents to be "contracted" before they could be edited – any changes made within an `EXPANSION` tag pair in the resultant document are lost when the document is contracted and then expanded.

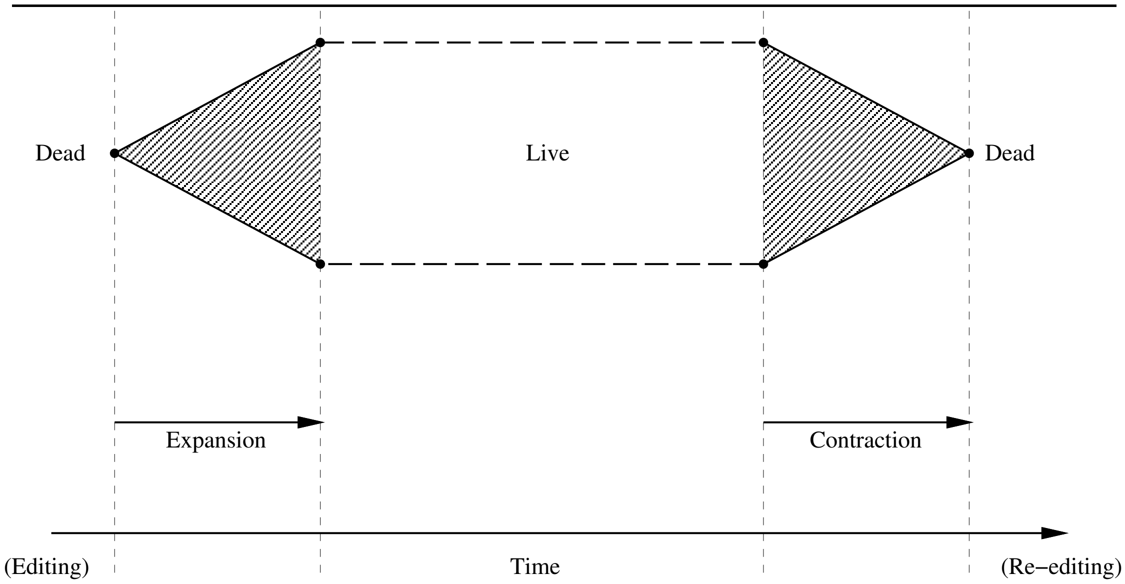

Figure 2: Expansion/contraction cycle time graph

```
<GLOSS WORD="Arithmetic Logic Unit (ALU)" KEY="alu">
<EXPANSION>
<A HREF="/cgi-bin/tiler/glossary?key=alu>Arithmetic Logic
Unit (ALU)</A>
</EXPANSION>
```

Figure 3: The `EXPANSION` tag

2.3 The dual document tree method

This approach really quite straightforward. There are two document trees: one contains the source documents (i.e. source text+HTML), the other – the "live" tree – contains the documents resulting from pre-processing (i.e. output text+HTML). The pre-processor takes one or more source documents (or document sub-trees) specified by the web text developer, expands it, and places it at the correct position in the document tree (see figure 4). To differentiate between the two types of file (source and resultant) we give them different filename extensions. A source file ends in *.mml* (Macro Markup Language) whereas its associated resultant file ends in *.html* as normal. It is this .html file that is made visible by the web server. Hence altering a link or deleting a node or any other changes to the source web text won't affect end users until they are updated in the destination tree.

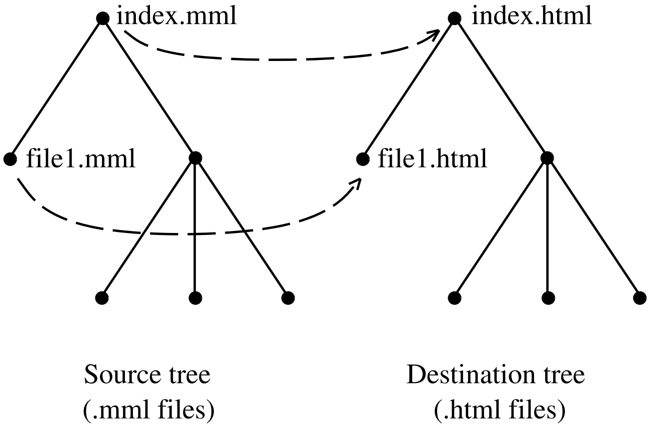

Figure 4: Pre-processing with a dual document tree

To make life easier for the web text developer, the two *logical* trees exist within the same *physical* tree. I.e. the pre-processor takes a .mml file, expands any macros it contains, and places the resultant .html in the same directory in the filing system.

This dual document tree was chosen as the preferred method as it adds other benefits to web publishing. Namely, a web text can maintain a network prescence whilst it is being edited by the developer. In addition, we can use the dual trees to create a dependency between .mml and .html file pairs; the pre-processor can check the timestamp of the source file to see if it has been modified since it was last expanded.

3 The HTML Macro Definition Language

3.1 Design for usability

As we have seen, decisions were taken during the design process to make the pre-processor tool usable, and hence popular, with web developers. It is necessary to make the writing, as well as the application, of macros possible for non-programmers. After all, this is one of the attractions of HTML: it is simple enough for the novice to master, yet provides them with great power.

In the same vein, the Macro Definition Language needed to be easy to use to write macros. The implementation involves writing macro definitions in text files (one per macro definition) using a very simple syntax; users no longer need to be able to program in Perl to be able to add new macros.

3.2 Design for simplicity

The macro definitions mirror the HTML style as well as its philosophy. As there are two basic forms of HTML tag (single and container) this is reflected in macro definitions. Each macro contains three basic parts: source, start replacement and end replacement.

- The source part contains the macro name and the names of any attributes it has. All this must be on line line only, whereas the remaining two parts may consist of multiple lines of text.
- The start part is used as the replacement text for the start tag. It can be more than one line of text.
- The end part is optional, and if used, must be separated from the start part by a `#terminator` control line. It can also be more than one line of text.

Figure 5 shows an example of the simplest form of macro definition, with just source and start replacement parts, whereas figure 6 shows a macro definiton containing all parts. Like HTML, macro definitions are case-insensitive.

```
// Comments begin with double slash (//)
// footer macro
// Create a footer consisting of a horizontal rule
// and my name as a hotlink to my email address
// atp
// 7th September 1995
<footer>
<BR><BR>
<HR>
<A HREF="mailto:A.T.Peel@ukc.ac.uk">Andrew Peel</A><P>
```

Figure 5: The simplest form of macro definition - The `FOOTER` macro

```
// screen
// Used for drawing screen shots of VT100 terminals
// atp
// 3rd October 1995
<screen>
<TABLE BORDER WIDTH=50% CELLPADDING=10><TR><TD><PRE>
#terminator
</PRE></TD></TR></TABLE>
```

Figure 6: A full macro definition - The `SCREEN` macro

This creates some complications with overlapping `SCREEN` and `EXPANSION` tags (amongst others), as this is not allowed in HTML. This is easily remedied by removing the insertion of expansion tags by the macro pre-processor but this has not been done since, as has already been pointed out, this “*meta-markup*” is beneficial to debugging incorrect resultant HTML.

3.3 Design for power

As Brown (1977) said, ”It is truly amazing how powerful a primitive operation an ordinary macro can be, provided that macro definitions may be dynamically created”.

Hence we have introduced several other constructions for use at *macro-time*. These are:

- `att::`*`attribute name`*

  Refers to the value of a macro attribute name.

- `const::`*`constant name`*

  Refers to the value of a pre-defined constant.

- `func::`*`Perl function name(text)`*

  Creates a call to a pre-defined Perl function.

- `#if` *`attribute name`*`..#else..#endif`

  Creates a traditional if..then..else control structure based on the *existance* of a value for a specified attribute name.

Note that the “reference” operators mentioned so far all use the `::` construct. This can be used literally in a definition by using an HTML entity-like construction: `&doublecolon;`. The same goes for control lines – a hash (#) indicates that the line contains a control command, unless it is escaped using `&hash;`.

Appendix A contains a BNF grammar for the Macro Definition Language.

Figure 7 shows a macro definition that can be used to create the glossary button demonstrated earlier.

```
// Glossary
// key takes a string to search for in the glossary data file
// word is used as the text of the hyperlink
// atp 28/7/95
<gloss key word>
<A HREF="const::glossary_url?key=att::key">att::word</A>
```

Figure 7: The `GLOSS` macro

Currently, nesting is not possible, although it would be straightforward to implement.

4 A more complex example - navigation bars

Figure 8 shows the macro definition for the `GO` macro. This is used to indicate that following a link will take the end user away from the local web server material.

Figure 9 shows an example *constant.cdl* file which describes constants available to macro definitions. See Appendix B for a BNF grammar for the Constant Definition Language.

Figure 10 contains a more complex macro definition – that of the `BAR` macro. It is used to create a tailored navigation bar for a web node. Figure 11 shows an example of the macro in action. It is the very first line that contains the `BAR` macro, whose attributes (`UP`, `LEFT` and `TITLE` in the example) describe the format of the navigation bar.

The macro definition, although quite complicated looking, is quite straightforward. The most complex action is takes is to use look up the titles of ajoining nodes using the GetLinkTitle(); function. This lets each node print the titles of its parents and siblings (if any) so showing the user what lies in each direction.

The rest of the figure 11 is used to create the content of the web node which is illustrated in figure 12.

```
// 7th Feb 1996
// atp
<goto>
<IMG SRC="/sicons/goto.gif" ALIGN=MIDDLE>
```

Figure 8: The `GO` macro definition

```
// Constants file
// lvalue and rvalue separated by =
// whitespace either side of values is ignored (as are
// comments and blank lines)

glossary_url = /cgi-bin/tiler/glossary
bibliography_url = /cgi-bin/tiler/bibliography
transweb_url = /cgi-bin/tiler/transweb_server
```

Figure 9: An example *constant.cdl* file

5 Framework files

Nielsen (1991) said that the disadvantage of hypertext compared with traditional computer systems is that there is “no central definition of the structure of the data, and therefore no easy way to specify general actions or computations on the data.”

But with HTML, we have the structure of the hypertext nodes; what is missing is a structure to contain the links between those nodes. If we assume a hierarchical hyper-structure the task becomes simpler, although some might say that this is restrictive, inflexible, and not a “proper” hypertext.

However, forcing the user into a hierarchy can be a good thing. As Brown (1991) says “A hierarchy is a powerful aid to orienting the user – indeed it is the only higher-level abstraction he has” and some go further in promoting the use of a hierarchy for web developers: “for the information provider such systems are easy to build by cross-linking existing filesystems” (Berners-Lee *et al.*, 1992).

So the choice of a hierarchy is perhaps a wise one, and they can be seen again and again at World-Wide Web sites, making life simpler for both end users and developers.

```
// atp
// 20th September 1995
//
//toolbar for HPCC TLTP courseware
<bar title left up right>
<TABLE>
<TR>
<TD ALIGN=CENTER>
#if left
<A HREF="att::left">
<IMG SRC="/sicons/Prev.gif" ALT="Previous page"></A>
#else
<IMG SRC="/sicons/Prev_dead.gif">
#endif
</TD>

<TD ALIGN=CENTER>
#if up
<A HREF="att::up">
<IMG SRC="/sicons/Up.gif" ALT="Up a page"></A>
#else
<IMG SRC="/sicons/Up_dead.gif">
#endif
</TD>

<TD ALIGN=CENTER>
#if right
<A HREF="att::right">
<IMG SRC="/sicons/Next.gif" ALT="Next page"></A>
#else
<IMG SRC="/sicons/Next_dead.gif">
#endif
</TD>

// do semi automatic linking
<TD ROWSPAN=2>
#if left
<A HREF="att::left">Previous: func::GetLinkTitle(left)<BR></A>
#endif
#if up
<A HREF="att::up">Up: func::GetLinkTitle(up)</A><BR>
#endif
#if right
<A HREF="att::right">Next: func::GetLinkTitle(right)</A><BR>
#endif
</TD>

</TR ALIGN=CENTER>
<TR>
#if left
<TD ALIGN=CENTER><A HREF="att::left">Previous</A></TD>
#else
<TD ALIGN=CENTER></TD>
#endif
#if up
<TD ALIGN=CENTER><A HREF="att::up">Up</A></TD>
#else
<TD ALIGN=CENTER></TD>
#endif
#if right
<TD ALIGN=CENTER><A HREF="att::right">Next</A></TD>
#else
<TD ALIGN=CENTER></TD>
#endif
</TR>
</TABLE>
#if title
<TITLE>att::title</TITLE>
<H1>att::title</H1>
#endif
<P><BR>
```

Figure 10: The BAR macro definition

```
<BAR UP="/courseware/modules/occam/1/index.html"
LEFT="/courseware/modules/occam/1/architecture.html"
TITLE="Mandelbrots">

<DL>
    <DD> <IMG SRC="images/mandelbrot1.gif">
<IMG SRC="images/mandelbrot2.gif"><P>

    <GO> <A HREF="http://www.cis.ohio-state.edu/hypertext
/faq/usenet/fractal-faq/faq.html">Fractal FAQ</A><P>

    <GO> <A HREF="http://www.comlab.ox.ac.uk/archive/other
/museums/computing/mandelbrot.html">Mandelbrot
Exhibition</A> <A HREF="http://www.comlab.ox.ac.uk
/archive/other/museums/computing.html">(Virtual
Museum of Computing)</A>
</DL>
```

Figure 11: The .mml file before macro expansion

If each node in a web text contains a BAR macro specifying links to its parent and immediate siblings this still involves large amounts of repetitive manual markup. Also, moving a node to a different position in the web text requires not only altering the BAR macro in the node to be moved, but also the links to that node from other nodes.

An improvement on Tiler would be for information on the structure of the web to be stored in a central database. This is similar to the concept suggested by (Berners-Lee, 1995) ("Some browsers have 'next' and 'previous' buttons to allow a document to be browsed serially") and implemented in Footsteps (Nicol *et al.*, 1995), but by using two, rather than just one dimension, we can store hierarchical hypertexts as opposed to sequential trails.

For example, the structure of a web text could be stored in a single text file in the format shown in figure 13, with children listed in brackets after their parents.

This would represent the following web text structure illustrated in figure 14.

The advantage of replacing "*hardlinking*" with "*softlinking*" is that any alterations can be made to this single file. The resulting changes to links in individual web nodes can then be made automatically. This would further ease the maintenance of the web text whilst making it still more robust.

Softlinking also allows simple tailoring of courseware: any unwanted nodes can be removed simply by deleting their entry in the framework file; any extra nodes can be inserted by adding their entry. Once the central database is complete, the necessary updates to the linking of the individual web nodes can be made automatically.

6 Conclusion

The pre-processor is a slow tool, perhaps better for batch jobs than processing in real time. Still, it is not in the least unusable, and a rewrite of the code from Perl, which is an interpreted

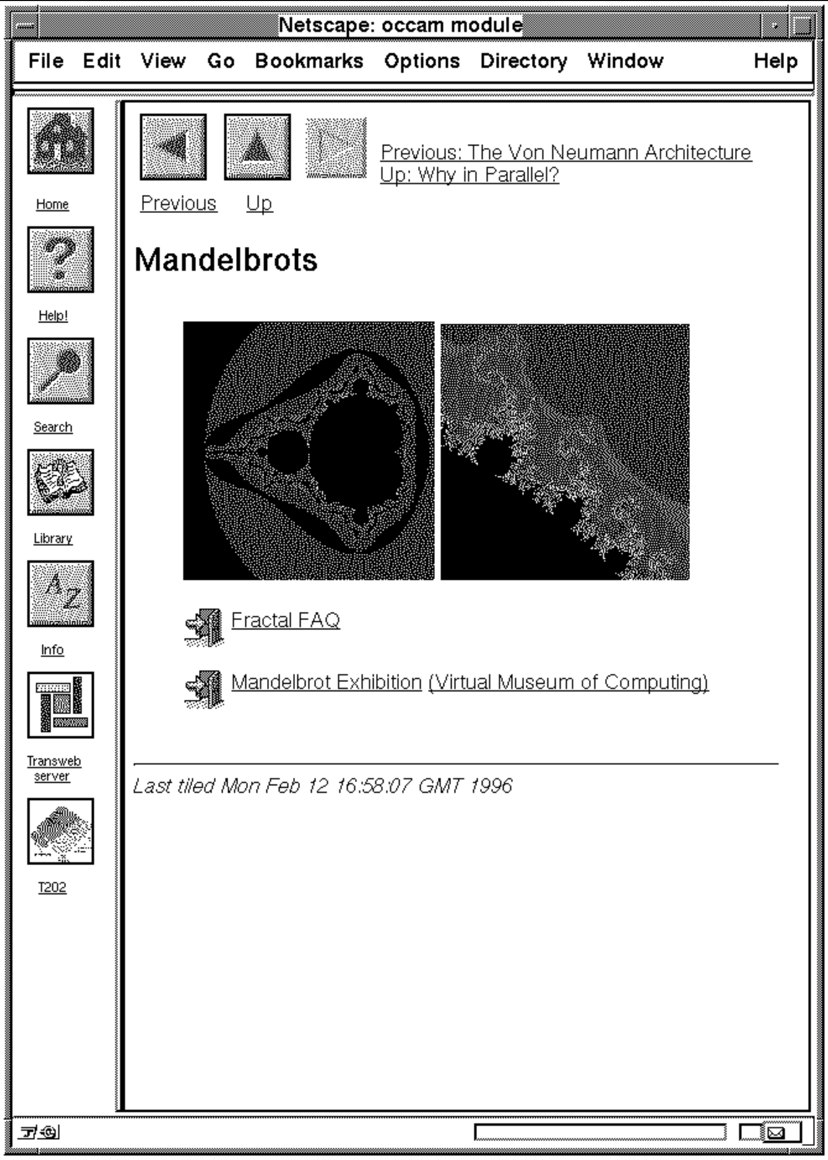

Figure 12: A web node containing an expanded BAR macro

```
dir/root.html {
    dir/sub-dir/child1.html
    dir/sub-dir/child2.html {
        dir/sub-dir/grandchild.html
    }
    dir/sub-dir/child3.html {
        dir/sub-dir/another-grandchild.html
        dir/sub-dir/yet-another-grandchild.html
    }
}
```

Figure 13: Example contents of a framework file

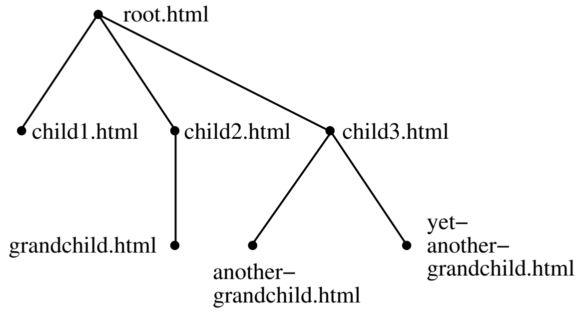

Figure 14: The Framework tree representing the data structure of figure 13

language, into C would no doubt resolve this negligable issue.

Although the dependency between .mml and .html files has been built into the pre-processor, the creation of .html files should also be determined by changes in the following:

- macro defintions
- user function definitions
- framework files (changes in sibling and parent node titles/links)

This would decrease the performance to a lower level still. It would simple take too long to check everything so the current implementation of Tiler (v2.1) relies on the user to force a "*retiling*" if necessary. Makefiles have previously been used to set up these dependencies, but they increase the complexity of operation for the novice user whilst making the pre-processor code less platform dependent.

References

Berners-Lee, T. (1995), Style guide for online hypertext, <URL: http://www.w3.org/hypertext/WWW/Provider/Style/Overview.html>.

Berners-Lee, T., Cailliau, R., Groff, J.-F., and Pollerman, B. (Spring 1992), *Electronic Networking: Research, Applications and Policy, Vol. 1 No. 2, Meckler Publishing, Westport, CT, USA* .

Brown, P. (1977), Macro Processors, in Brown, P., editor, *Software Portability*, Cambridge University Press.

Brown, P. (1991), Hypertext: Dreams and Reality, in Brown, H., editor, *Hypermedia/Hypertext and Object Oriented Databases*, Chapman & Hall.

McCool, R. (1993), The Common Gateway Interface, NCSA, <URL: http://hoohoo.ncsa.uiuc.edu/cgi/>.

Nicol, D., Smeaton, C., and Slater, A. (1995), Footsteps: Trail-blazing the Web, in *Third International World-Wide Web Conference*, Darmstadt, Germany, <URL: http://www.igd.fhg.de/www/www95/papers/60/footsteps.html>.

Nielsen, J. (1991), Usability considerations in Introducing Hypertext, in Brown, H., editor, *Hypermedia/Hypertext and Object Oriented Databases*, Chapman & Hall.

Peel, A. (1996), HTML macros - Easing the construction and maintenance of Web texts, Technical Report 4/96, University of Kent at Canterbury.

Wall, L. and Schwartz, R. (1992), *Programming Perl*, O'Reilly & Associates.

APPENDIX A

A BNF grammar for the Macro Definition Language

```
<macro_definition_file>      ::=  {<comment_line>}
                                  <macro_source_line>
                                  {<comment_line>}
                                  <macro_result_block>
                                  [<terminator_separator>
                                  <macro_result_block>];

<comment_line>               ::=  COMMENT TEXT NEWLINE;

<terminator_separator>       ::=  HASH TERMINATOR;

<macro_source_line>          ::=  LEFT_ANGLE_BRACKET
                                  MACRO_NAME
                                  <macro_attribute_line>
                                  RIGHT_ANGLE_BRACKET
                                  NEWLINE;

<macro_attribute_line>       ::=  {MACRO_ATTRIBUTE_NAME};

<macro_result_block>         ::=  {<macro_result_line | <macro_if_block>};

<macro_result_line>          ::=  {TEXT
                             |    <constant_reference>
                             |    <attribute_value_reference>
                             |    <function_call>
                             |    <mdl_entity>}
                             |    NEWLINE;

<constant_reference>         ::=  CONST DOUBLE_COLON CONST_NAME;

<attribute_value_reference>  ::=  ATT DOUBLE_COLON MACRO_ATTRIBUTE_NAME;

<function_call>              ::=  FUNC DOUBLE_COLON function_prototype;

<function_prototype>         ::=  FUNC_NAME
                                  LEFT_BRACKET
                                  {<attribute_value_reference>
                             |    <constant_value_reference>
                             |    TEXT}
                                  RIGHT_BRACKET;
```

<macro_if_block>	::=	HASH IF MACRO_ATTRIBUTE_NAME NEWLINE {<macro_result_line>} HASH ELSE NEWLINE {<macro_result_line>} HASH ENDIF NEWLINE;
<mdl_entity>	::=	AMPERSAND (DOUBLECOLON | HASHWORD | DOUBLESLASH) SEMICOLON;

A.1 Lexical tokens

COMMENT	::=	//;
TEXT	::=	{A-Za-z0-9_};
NEWLINE	::=	\n;
LEFT_ANGLE_BRACKET	::=	<;;
RIGHT_ANGLE_BRACKET	::=	>;;
MACRO_ATTRIBUTE_NAME	::=	{A-Za-z0-9_};
CONST	::=	const;
DOUBLE_COLON	::=	::;
CONST_NAME	::=	{A-Za-z0-9_};
ATT	::=	att;
FUNC	::=	func;
FUNC_NAME	::=	{A-Za-z0-9_};
LEFT_BRACKET	::=	(;
RIGHT_BRACKET	::=	);
HASH	::=	#;
TERMINATOR	::=	terminator;
IF	::=	if;
ELSE	::=	else;
ENDIF	::=	endif;
AMPERSAND	::=	&;
DOUBLECOLON	::=	doublecolon;
HASHWORD	::=	hash;
DOUBLESLASH	::=	doubleslash;
SEMICOLON	::=	;;

APPENDIX B

A BNF grammar for the Constant Definition Language

```
<constant_definition_file>    ::=  {<comment_line>
                              |    <constant_definition_line>
                              |    NEWLINE};

<comment_line>                ::=  COMMENT TEXT NEWLINE;

<constant_definition_line>    ::=  CONST_NAME EQUALS CONST_VALUE;
```

B.1 Lexical tokens

```
NEWLINE      ::=  \n;
COMMENT      ::=  //;
TEXT         ::=  {A-Za-z0-9_};
CONST_NAME   ::=  {A-Za-z0-9_};
EQUALS       ::=  =;
CONST_VALUE  ::=  {A-Za-z0-9_};
```

Vita

Andrew Peel graduated in 1993 with a B.Sc in Computer Systems Engineering from the University of Kent at Canterbury. He is currently working as a Research Associate in the Computing Laboratory at UKC.

His previous work has involved the production of "*An Introduction to Computer Aided Assessment in Higher Education*", a hypertext tutorial running under PC Guide which is now available in the UK. He has been involved in the production of High Performance Computing courseware for the World-Wide Web for the past two years, during which he developed the Tiler HTML macro pre-processor.

His research interests include Computer Based Learning and Assessment using the World-Wide Web, novel uses for Web browsers, such as remote compilation and execution of program source code and shell scripts over the Web. For example, using a Web browser as a GUI to compile Occam on the Transputer network at UKC from anywhere in the world. He is also interested in the future of the Web, HTML and browsers, and especially the problems involved with searching for relevant material on the Web without spending all day surfing.

He is currently research ways of putting hyper-functionality into the World-Wide Web using the Java programming language.