HTML PLAIN reference V1.1

Cyrille Artho

October 4, 1999

Abstract

While HTML (Hyper Text Markup Language) is widely used due to its simplicity, it is not powerful enough to cope with the representation of complex content. It does not separate information from its representation, and offers no means to access metadata (information about pages). As a result of this, maintaining and updating large web pages is often a nightmare, either involving much manual work or the use of complex tools that try to glue an inconsistent set of documents together.

By extending HTML, these problems can be solved in an elegant way. Customizable tags allow a better structuring, and variables decouple information from its appearance in documents. A preprocessor then generates a set of standard HTML pages based on this extra information and pages written in this new, higher level HTML. It also includes other powerful functions, such as file management and macros, which can automate many tedious tasks.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problems with HTML

HTML has many advantages. It is very simple, making it easy to learn. It has many built-in style tags which are quite expressive. Linking is also readily achieved, in a simple, standard way. However, most of these advantages also have a downside, which make the maintenance of large web pages a nightmare [Arb][1].

1.1.1 Limited structure

Due to its fixed set of tags, which are primarily concerned with formatting, the language is inherently limited. HTML is mainly concerned about appearance and content delivery, but not about structure or content. It allows only very limited reuse and automation.

1.1.2 Limited reuse

HTML has no notion of a "document template". A template allows the designer to create a fancy page and then use that layout for all web pages. However, with HTML this usually implies inserting the new set of tags into all documents. Because there are sometimes conflicts, these insertions often have to be done manually, which is very time consuming.

While HTML has style sheets, these do not solve all problems. Style sheets are a clean way to specify the appearance of the text based on its content. However, they cannot deal with the other problems that HTML causes (see below).

1.1.3 Limited linking

Because of the single standard linking mechanism, it is trivial to link to other pages or pages within the site. If these pages move, though, all links have to be updated with the new URL! This is a very tedious task, which is not always easy to automate (relative links to a document look always different

[1] The page referenced to uses a non-standard character encoding, so it is a bit awkward to read due to that.

depending in which directory you are). There is no way to specify a link based on a symbolic name, which allows changing the link by changing the definition of that symbol.

Because links are hard-coded, it is usually not possible to move a file to a different location, once the filename is given. Sometimes, this would help to restructure the page and keep the overview of all documents. Because it involves too much work to fix all links afterwards, this approach is usually abandoned.

Linking to external pages is also a nightmare. If a page moves, all links need to be changed.

1.1.4 Limited automation

HTML is not expressive enough to allow automation. It is hard to impossible to get information about the content, not only about the appearance, of a document. There is no way at all how to get metadata about a whole site encompassing a range of documents. Tasks like generating a site map or a list of links usually involve custom made scripts, which only work with a particular site, and/or a lot of manual work.

1.2 Common tasks for webmasters

1.2.1 Layout changes

It is not too uncommon to change the layout of all pages now and then. Usually this involves a lot of re-tagging of each document. Even tasks as simple as changing the color turn out to be a nightmare. This is all because HTML specifies the appearance completely, without any layer of abstraction. Therefore it is very hard to change the layout once a page is made.

1.2.2 Linking

One problem is to keep all links current (see Section 1.1.3). However, even then, there are many repetitive tasks left: Adding the ubiquitous “Back” link to a document on a higher level (or the main page), a set of links to related topics or making a site map.

1.2.3 Setting image attributes

For a faster rendering of the page, every webmaster should specify the height and width of each image. Modern HTML editors usually do this. This is one of the problems that are not dependent on other documents, and therefore it can be solved easily.

1.3 Approach to the problem

Usually the webmaster either uses a tool that helps him to make his life a little easier, or some scripts of his own, which are adapted to a particular web page and can automate some tasks well. However, most of these tools have major limitations and often do not go beyond aspects of single documents. Therefore, I decided to create a tool of my own to solve these problems.

Obviously the functionality of HTML itself had to be extended. Defining custom tags alone could not solve all the problems - accessing metadata about the pages is still not possible then. Therefore it was necessary to introduce an extra layer hiding the location of the documents and other properties such as document size or "content level" (see Section 2.2.6 on page 13). Because it is not possible to hard-code all features in the HTML preprocessor, the possibility to execute macro functions has been added.

1.3.1 Precompiling HTML documents

Because changing the documents on the web server itself (via CGI) would be too slow, it is necessary to preprocess documents before they are uploaded. The preprocessor takes a template (see Section 2.2), which specifies the general appearance of all documents, and applies that template to each document that has to be uploaded. Other settings are stored in a variable file (see Section 2.3), which works in the same way as templates do, although they are concerned about content, not appearance.

The preprocessor takes all documents within a source directory tree and parses them according to all given options. The output is then stored in the upload directory where the HTML pages should be uploaded from.

1.4 Notations used

1.4.1 HTML

A "tag" or "HTML tag" is considered to be composed of the "tag" itself and its "options", i. e.

```
<tag ([switch]|[option="value"])*>
```

An "entity" is a special notation for some non-printable characters or string constants, and always has the form `&entity-name;`.

A string within double quotes can denote a "variable" or "variable name", which is, if defined, replaced by another string, the "content" of the variable.

1.4.2 The software

The "template" is a special file which defines the appearance of all file that actually appear on the web page. The "variable file" is the file where the values of the variables are stored. The "file table" is an intermediate file which is generated in order to be able to dereference symbolic names later.

The "documents" are the HTML pages which actually contain the text that is later uploaded to the web server. These documents do not have to have fully complete HTML code, since the template can have much of what is required for this. The "source" or "pages" document tree contains the files that are actually edited (in the document "root" path). These are then processed and stored in the “output” or “upload” directory. The webmaster has then the upload these documents.

The “preprocessor” is the HTML preprocessor which takes the documents and produces the documents to be uploaded. The “file table” generator is the program which contains all the symbolic names and their meaning; it has to be run each time a new page is added.

A short text in square brackets usually means a reference to the bibliography, with the exception of Chapter 4, where it sometimes also denotes key strokes, with they key in square brackets (e. g. [Return]). The difference is always obvious.

1.4.3 Typesetting

HTML tags and entities appear in a `typewriter` font, as do file and program names. Longer parts of HTML or Perl code appear in an extra paragraph, while single HTML tags are usually written in the same line as the other text. Variable names and program settings appear between double quotes (like “title”).

1.5 Other HTML preprocessors

Because the need for such a tool is definitely considerable, there are quite a few HTML preprocessors around:

[Chpp] A generic preprocessor which can also be used for HTML.

[Fro] Frontier: inspired HTML PLAIN with its file management, and also some of its syntax.

[Meta] Meta HTML: A commercial tool, intended for SQL backed web pages.

[PHP] PHP3: A very widely used embedded scripting language, including a CGI module.

[QML] QML: A simple markup language consisting entirely of assignments.

[Wpp] Very similar to HTML PLAIN, but no file management.

1.6 Where to obtain HTML PLAIN

The latest version of HTML PLAIN can be obtained at the official HTML PLAIN web page at [HPL].

Chapter 2

Specification of the HTML extension

The tools build on a specialized set of HTML, allowing

- using names or symbols instead of filenames for linking
- using templates for customizing (and changing) the look of a page
- using custom-defined tags and entities (XML-like)

The most important part of these tool is the preprocessor, which takes the raw pages and translates them into fully functional HTML pages. It uses templates (see Section 2.2) for customizing the look and a dictionary (see Section 2.4) for referencing variables.

2.1 Customized HTML tags

The preprocessor will allow the user to define his own set of HTML tags. These can be replacements of existing tags or wholly new ones. The tags will all be looked up from a template file. The tag redefinitions are case insensitive, like normal HTML tags.

2.2 Templates

Templates are the core of the new package; together with a dictionary, they define the functionality that the user wants. The goal is to make these templates easy to edit, both via a GUI editor and as ASCII files. In later versions, a collection of templates will be provided to cover the basic functionality that the majority of all users want, so they do not have to create their own template in the first place.

2.2.1 Format

The template contains a definition in the following format:

tag or entity   redefinition

The tag may be already defined previously in the document; this allows recursive definitions. Of course this function should be used with care. The tag and the redefinition are separated with a tab `(\t)` character. Everything behind a second tab character is regarded as a comment and ignored. Using a simple whitespace character would have the disadvantage of not allowing redefinitions of tags that include options, like `<td nowrap>`. No reasonable HTML text needs tab characters, so it can be used as a separator. Of course tabs are often used for indenting HTML code, but formatting of HTML code can be done with spaces as well (even though this is a kludge compared to tabs). The HTML code for templates should not become too complicate, so indentation will not be needed.

New tags do not have to use the ubiquitous angular brackets; if these are missing as the first character (or last one, respectively), they are inserted automatically. Since only HTML tags are to be replaced, they have to be replaced by other HTML tags, so this rule can be applied. An exception is a HTML entity; if the redefinition starts with an ampersand sign `(&)`, it is of course not enclosed in angular brackets.

Comments

Comments will allow all people who are "ASCII hackers" to include their comments more easily. The GUI editor supports short comments at the end of the line, but cannot offer the wide variety of comments that is possible.

A comment can start with a hash symbol `(#)` like in Perl or two slashes `(//)` like in C++. Longer comments can be written in C style, like in `/* comment */`. Comments are only allowed in the third column, except for C comments, which usually go over multiple lines.

Variables

In the redefinition part of the template, variables can be used. They are enclosed in double quotes `("variable name")`, and their content is looked up in the dictionary (see Section 2.4).

Multi line redefinitions

If a definition does not fit into a single line, it can be continued on the next one. The first column is left empty, and the second (or first non-empty) column contains the continued text from the previous line. The line breaks will also appear in the final HTML text, unless a backslash `(\)` is used to combine two lines into one (like in the C preprocessor).

2.2.2 Overriding options

If there is a default appearance of a tag, e. g. `<body color="#000000">`, and the document itself contains the same tag with different or more options, these will override the options from the template. This allows the author of the page to override the settings given in a template, if this should be necessary.

Any options occurring in either the template or the document, but not in both files, are used in the compiled version of the document.

Have the template options always override settings from a document

Sometimes, the opposite behavior from the one described above is desirable: a template option should always have precedence over the document. This can be to enforce a uniform style over inconsistent pages, or to use a JavaScript function for all links. The latter can be tremendously useful for pages that have multiple frames, which need to be updated when a link is used.

In order to get this behavior, the *comment* for the tag defined in the template must start with a `"!"`. This will look like a Unix-shebang comment if the template is viewed in an ASCII editor such as vi. The GUI tools do not yet specifically support this feature (e. g. by a check box for each tag), but will do so in a later version. An example shows this:

```
href <a href=javascript:link("LINK")>"NAME"</a>     # de-
fault look for links
aa <a> # for <a name=...> tags, use <aa=...> be-
cause <a> tags are replaced with javascript
a <a href=javascript:link("LINK")> #! always overrides docu-
ment
```

The first template definition is for the default links, using symbolic names between double quotes. The last one overrides the `<a href>` tag *and* preserves the link content (this only works for links right now, because of the internal variable `LINK`). Like that, any link will be processed by a JavaScript function. You can easily use this to refresh multiple frames if required, or destroy frames if the URL is an external page.

If you still want to use some "normal" links, you now have to use a little trick, e. g. by defining an `<aa>` tag in the template. The middle definition does this and allows you to tell explicitly if you want a standard link, by using `<aa href>` or `<aa name>`. Note that this definition has to be in the template prior to the redefinition of the `<a>` tag, because HTML PLAIN would otherwise assume that it was a recursive definition (i. e. an alias).

2.2.3 Special tags

ˆ and $

ˆ stands for the beginning of a page, and `$` for the end of it. These definitions are quite intuitive for people who have used regular expressions under UNIX before; there, these characters mark the beginning of a line (or the end of it, respectively). Here, their meaning is the same, just on a higher level. The parser automatically breaks up the set of tags defining ˆ and $ into multiple lines, internally using an array.

2.2.4 Custom closing tags

Sometimes it is not possible to use the opening tags given, without options, as a closing tag. For example, in the case of nested tags, the closing tags have to appear in reverse order. Therefore the user can define a closing tag in a template which will be used in that situation and work like any other tag with a unique name.

If no such closing tag can be found, the preprocessor uses the opening tag after stripping the options. The order of the closing tags is reversed, so a custom closing tag should usually not be needed.

2.2.5 Example

```
#suggestion for style of templates
i       em      #enforce "nice" HTML style
#the (new or old) HTML element and the redefinition are separated by tab
characters (\t)
#angular brackets are optional (if not present, they will be inserted
automatically)
sans    font face="Arial,Helvetica"     #add new tags
# ^ refers to beginning of document (in Emacs/Perl expressions, it means
beginning of line)
^       <html>
        <head>
        <title>"company" - "title"</title>
        </head>
        <body>
/*
This one requires some explanation. First, if the left column is empty, it
means that the redefinition from the previous line is continued.
(This is also the case if there is only one column, not counting comments.)
"Company" refers to a variable. It can either be defined in the dictionary
file (see below) or in the document itself. In the latter case, the
variable is defined in the data document as <title>Title</title>, using
normal HTML tags. The parser tries to find an occurrence of that tag within
the data document and replaces the variable in the template with its value.
*/
&copy;  everything is copyrighted except GNU software which is copylefted
# longer comments in C style

$       <%MakeLinks("LEVEL"-1, "LEVEL"+1, <ol>, <li>, </ol>)>
        </body></html>  # footer of document
/*
similar again to normal regular expressions, $ means end of document here.
This is another example of a definition that takes several lines. % calls
a macro function (which is just some Perl code) and places its result
there. See Section 2.5 for more on macros.
*/
<img>       img align="left" border="0" alt=\""NAME"\" width="WIDTH" \
        height="HEIGHT"
/*
Again a very special case: most parameters are special variables that will
be inserted when the HTML page is built. This <img> tag can have several
options, which will be inserted only when not yet present. This allows the
user to use symbols instead of <img> tags, so he does not have to know it
in order to use images. However, the user can still provide the <img> tag
and some of the options in order to override the template settings. Two of
the double quotes in "alt" had to be escaped, in order to preserver them in
the final document. */
```

2.2.6 Different levels of content

The definitions shown above are suitable if all pages are in the same “level” (content wise). However, It does not yet have any notion for the level of a content, making it impossible to generate different versions of “navigation bars” depending on how deep in the “tree” of a page the user is.

Depending on their position on the “navigation tree”, pages have different "levels" of content, according to their depth in the tree. The depth can be calculated as $(number\,of\,/) + (1\,if\,file \neq index.html;\,0\,otherwise)$. Slashes in the

name	level
index.html	0
contact.html	1
about/index.html	1
about/arthur.html	2
about/zaphod.html	2
products/index.html	1
products/panGalacticGargleBlaster.html	2
products/encyc/index.html	2
products/encyc/encyclopaediaGalactica.html	3
products/encyc/hitchHikersGuideToTheGalaxy.html	3

Table 2.1: Different levels of content.

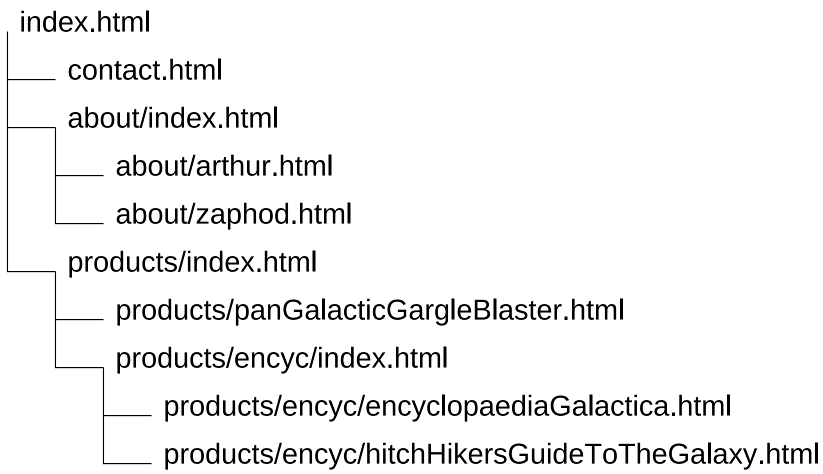

Figure 2.1: Full hierarchical tree of a site.

absolute path (before the document root) do not count. This formula looks (and is) cumbersome, but an example shows that this definition is quite intuitive:

In this example, it is presumed that the page is well organized. Leaf nodes are HTML files, non-leaf nodes a directory (with an index.html for an overview). The depth of a document is (the number of extra /s in their path), where "extra" means relative to the main page. If the file is not index.html, the depth is one higher.

This definition maps directly to a tree (or "site map").

The tree, should, in this example, look like Figure 2.1. Of course a HTML version cannot fully reproduce the graphical representation.

Generally, only a part of the tree is shown; all nodes with a depth between (current depth-1) and (current depth+1). If the depth is greater than the current depth, nodes are only shown if they lie within the subtree that has its root at the current node. Figures 2.2 and 2.3 show two examples.

Usually a link to the top level (0) is always provided, either as part of the usual navigation links or at a special position. Therefore, an option to

1. always include

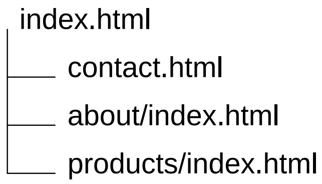

Figure 2.2: Current node = contact.html

Figure 2.3: Current node = about/index.html

2. always exclude

the root node should be there, as well as a third variant (treat it as normal).

Of course drawing a full site map based upon this is trivial, and the macro is not too hard to write. Filtering out the right part of the tree for obtaining a sub-tree is a bit harder, because the information is not represented as a tree, but simply as a list of strings; however, these strings contain enough information about their position in the tree.

2.2.7 Using macros in templates

The use of a macro is needed for more complex functions, such as the automatic generation of a footer. Redefinitions that contain a macro start with a `%` sign. Like this, a possible redefinition for a footer could be

```
$       <%MakeLinks("LEVEL"-1, "LEVEL"+1, <ol>, <li>, </ol>)>
```

The word after the `%` sign is the name of the macro that is called (just a Perl subroutine), the rest are the parameters passed to it (they have the same syntax like normal variables, see Section 2.3). `LEVEL` is the "depth" (see Section 2.2.6) of the current document in the tree of all documents. The last three arguments define the look of the image map. You can read more about macros in Section 2.5 on page 20. It is very important that the syntax of the macro is correct. There is not yet any complete syntax checking implemented. Specifically, all macros have to be called with an opening and closing bracket, even if no options are given. An example is `<%scalar localtime()>`. In Perl, the brackets are optional, but because the arguments may contain angular brackets, a closing round bracket is the only easy way of determining when the end of a macro has been reached. Another important aspect is that there should be no other tags at the same line; it is always possible to use line breaks between two tags to improve the structure. A macro should not be divided into several lines either. The fact that a macro should always stand alone in one line makes the syntax more restrictive than normal HTML, but this is not a problem.

2.3 Variables

Variables contain either some constants that are used throughout the document (but not as HTML replacements) or symbolic references to filenames. This eliminates the need of tracking file names when they change and also makes it possible to use abbreviations for long names, which are later expanded into their full name. The most common use will definitely be for replacing filenames (see Section 2.4.1). Variables are case sensitive (unlike tags redefined in a template).

2.3.1 Order of replacements

1. file names
2. special variables (with results from filenames), document defined variables and variables from the dictionary (see Section 2.4)
3. environment variables

First, the file dictionary is looked through for a string whose name matches. If a match occurs, the special variables (which are in the same table as the "normal" variables from the dictionary) are updated.

The second try is to find a matching name in the main variable table. That table contains all normal and special variables (see Section 2.3.2) and the content that has been found within the "magic tags" (see Section 2.7). In order to prevent name clashes, the user should refrain from having all uppercase names in his own documents and templates, even though people using AOL may find this very restrictive.

If the first two lookups (file names and variables) yield no result yet, the parser tries to find a matching variable name in the environment variables. This mechanism allows for setting some variables within shell scripts that call the parser. If no one of these three lookups are successful, the unchanged string, including the surrounding quotes, is returned.

2.3.2 Special variables

There are some special variables whose meaning is "hard coded". They are all written in CAPS and help to reference some properties of the current variable that is being replaced.

NAME File name that is replaced.

LINK The correct link to that file. With version 1.1. or higher, this also works with links including # signs (linking to a specific section of the page, such as `<a href="index.html#intro">`.

LEVEL Current “content level” (see Section 2.2.6).

HEIGHT Image height.

WIDTH Image width.

SIZE Size of document or image that the current variable refers to (in KB).

SELF Filename of the current document being processed by the parser.

THIS (Symbolic) name of the current document being processed by the parser.

THIS.SIZE Size of the current document being processed by the parser.

BACK Parent document; cleared before a document is compiled and then only set by some macros in order to speed up processing.

The values are usually returned including the quotes that surrounded them before the replacement. An exception to this rule are the variables “NAME”, “LEVEL” and “SIZE”, because they are not directly used in HTML tags and therefore may be used as plain text. This allows, in some cases, greater flexibility such as in

```
alt=\""NAME" ("SIZE")\"
```

where the quotes allow joining two variable names into one argument. Without this option (that makes templates harder to read), this flexibility could not be achieved.

Another very important aspect is the fact that the filename has to be the first evaluated argument, otherwise the variables SIZE etc. will refer to the previous file. There is no check being made for a valid return value.

Assignments to variables

As an alternative to definitions by “magic” HTML tags (see Section 2.7), a simple variable assignment such as

```
<"title" = Our products>
```

is often more intuitive, even though it breaks with any other syntax rule applied so far. If users want it, this feature will be included in the future.

2.4 The dictionary

The dictionary contains all symbols and names that should be used instead of filenames and URLs, for it is the key concept of HTML PLAIN to prevent the user from having to memorize filenames (and their paths, which can change). All instances of words within quotes (`"`) that can be found within the dictionary will replaced with their definition. Since the goal is to hide the location of the files from the user, this part of the dictionary is updated automatically.

The syntax of the variables is simple: any combination of letters, numbers and underscores (_) is allowed. Other characters such as punctuation characters are ignored. For references, the exact name of the page has to be used. By explicitly specifying the `<a href=...>` tag, the full flexibility of the linking tag can still be maintained. Variable redefinitions may contain other variables if they have been defined before. There is such a case in the example (see Section 2.4.3). For symbolic file names, spaces in the name are also significant.

2.4.1 The file dictionary

The format is very similar to templates (Section 2.2 on page 10).

Definition of files and documents

One part of the dictionary contains all references to the files and images used in the HTML documents. These are generated automatically as far as possible (more about that on Section 2.4.4). When the HTML files are built, it is assumed that all references are correct. Therefore, the file table has to be up to date before the preprocessor starts.

When a variable (i. e. any strings within quotes) is encountered during the build process, the file dictionary is first checked. There are three different cases in which a quoted string may appear:

1. Without a surrounding HTML tag. The program then just guesses the usage of the file.

   (a) If it is a HTML document, it is assumed that the string stands for a link. The proper (relative) link is inserted then.

   (b) If it is an image, the proper HTML tag is inserted. The settings (border, alignment etc.) can be specified in the template under `<img>` (see Section 2.2).

2. With a surrounding HTML tag. In that case, it can be assumed that the user just wants the filename to be inserted at the position of the variable. E. g.
   ```
   <a href="nice_image_large">Larger version ("SIZE" KB)</a>
   ```
   This overrides the standard behavior.

SIZE is a special variable here (see Section 2.3.2), which allows to include extra information like the file size of the referenced document.

The file dictionary contains the absolute location of all files, together with their “content depth” (see Section 2.2.6). It is used during the build process to look up the filenames, which are then converted to relative URLs.

2.4.2 The variable dictionary

The variable dictionary contains the definition of symbolic constants such as `"company"`. This is essentially the same like the redefinition of HTML tags, but since it is semantically not an HTML tag, it belongs into the dictionary. If a variable is found in the dictionary, it is replaced with its value.

Because longer comments should not be required, C style comments `/* comments */` are not allowed. However, Perl or C++ style comments at the beginning of a line are filtered out correctly.

2.4.3 Example

File dictionary

```
About Us          about/  2
About    "About Us"      2 # has been defined before
nice_image        nice_image.gif
```

Every time an instance of "About us" occurs in a document, it will be replaced by `<a href="about/">About Us</a>`. The example shows that it is sometimes convenient to define aliases; however, since the content should not be duplicated, a recursive definition is used. So far, there is no way to preserve manually inserted aliases in the file table. Such a mechanism might be introduced later, by extracting any recursive definitions before updating the file table, and then inserting them at the bottom.

Variable dictionary

```
company Artho Informatik          # :)
manager Cyrille Artho
```

In the example, it is shown how you can use symbols instead of names. This allows you to change the name of the manager in all documents by modifying only one file, should it ever change.

2.4.4 Maintaining the file dictionary

While the variable dictionary is maintained manually (either by editing the ASCII file or with the specialized editor), the file dictionary should be built up automatically as far as possible. The program `"plainfiles"` deals with this problem. It checks all documents that lie within a specified directory and scans their content. It uses the Perl module `File::Find` which is provided with any modern Perl distribution. Unfortunately, this module cannot follow symbolic links to directories, so all images have to be kept within the "root" or "pages" document tree. This makes it a bit harder to upload all pages easily, because the images have to be uploaded from the "pages" directory instead of the "upload" directory. However, it is possible to create a symbolic link in the "upload" tree of the documents. There is no option yet to specify that the file table generator should search the upload tree for images or binaries. This will be added later, as it is very important for non-Unix users, since they have no symbolic links in their file system (serves them right).

Overriding the alphabetical sorting in the macros

If you want some documents to appear on top of lists generated by macros such as `SiteMap`, you can now use the optional `sortingFile`. This configuration entry gives the location of a file in the following format:

```
search  replace
search2 replace2
intro   0
final   }
...     ...
```

The left hand side gives a search expression, the right hand side replaces it. The expressions are Perl regular expressions. Like this, you can assign a low ASCII character, such as `0`, to strings that should appear on the top of lists generated by macros. Giving a higher character such as `{`, | or `}`, makes items appear at the end. Note that the character ~ is not allowed in that file (unless preceded by a \).

2.5 Macros

Macros contain some Perl code that will be executed; the result will be used in the HTML code of the parsed page. They are run in a "safe" Perl interpreter, which means that they only have a very limited range of commands - for instance, they cannot write to files unless they have special permissions. The concept is much like Java's, with the exception that it is much easier to allow some external functions or variables to be used.

2.5.1 How macros work

Macros start with a `%` sign as the first character in the redefinition of a HTML tag in a template (see Section 2.2 on page 10). E. g. if the redefinition of `<time>` is `<%scalar localtime()>`, then the Perl function `scalar localtime` will be executed, which will insert the current time in the document.

A line containing a macro may not contain any more normal HTML tags in the same line. The entire line (except for the final closing angular bracket) is interpreted as a macro.

It is very important to notice that the macro functions have to return their result as a string, and *not* print it to `STDOUT`. This makes it possible to use all predefined Perl functions and new functions in the same way, and will prevent a lot of unneeded redirections of the standard output. It also eliminates the need of many `print` statements in macros. Even though it may look easier for novice programmers just to print out the results, the string concatenation function `"."` makes it very easy to emulate printing. For example, the code

```
$out .= "This is the same as printing a variable $var.\n";
```

appends another line to the string `$out`. Because all strings in double quotes are interpolated, nobody will miss the option to use

```
print "This is the same as printing a variable $var.\n";
```

except for the case when the variable that needs to be printed is an array. Then, the built in Perl function `sprintf` has to be used, or a `foreach` loop to iterate over the array.

IMP call	Service
GetFileTable()	Get array with all symbolic names of files
GetHTMLFiles()	Get array with all symbolic names of HTML documents
GetHTMLFilenames()	Get array with all full file names of HTML documents
GetFileInfo (symbolic_name)	Get array with file information: full file name, file size, document level (HTML files) or picture size (images)
GetVar(var_name)	Get variable content
PutVar(var_name, content)	Store variable content
EndingTags(tags)	Returns reversed closing tags, without any options

Table 2.2: IMP calls.

Argument	Example value
Minimum level	0 or "LEVEL" -1
Maximum level	"LEVEL" + 1
Starting tag for table	'<table border="0">'
Starting tag for parent	'<tr bgcolor="#ffffff"><td>'
Starting tag for sibling	'<tr><td>'
Starting tag for child	'<tr><td><b>'
Mode	4

Table 2.3: LinkTable arguments.

2.5.2 The IMP

Since macros cannot access any of the program variables without special permissions, access functions had to be provided for these. The interface over which variables can access metadata about the document is called the IMP = Interface for Macro Programming. The IMP provides macros with everything they need to know, such as the names of all files, their size or level.

2.5.3 Built-in macros

It is maybe the easiest way to learn how to deal with the IMP by looking at some provided macro functions. Because these functions are very powerful, they may not be always easy to understand. Here is a short description of them:

DumpArgs

This function just prints all arguments given. Use it for testing whether all variables have been correctly interpolated.

LinkTable

This macro creates a table with links to related topics. The arguments are given in Table 2.3.

Mode bit	Effect if set
0	include file size of documents after link
1	repeat starting tags from 0 each time (MakeLinks only)
2	always include root node in table
3	link to sibling nodes
4	link to siblings only if current page = index page

Table 2.4: LinkTable modes.

Argument	Example value
Minimum level	0 or `"LEVEL" -1`
Maximum level	`"LEVEL" + 1`
Starting tag for new level	`'<ul>'`
Starting tag for new entry	`'<li>'`
Ending tag for entry	`'</li>'`
Ending tag for level	`'</ul>'`
Mode	`20`

Table 2.5: MakeLinks arguments.

Except for the last argument, the meaning of the arguments is fairly s-traightforward. The minimum and maximum levels determine how many links are shown (see Section 2.2.6 on page 13 for more information). The nex-t four arguments control the appearance of the table. The last argument is a combination of several bits for some options:

In order to get the required value for the argument, sum up 2^n for all bits that need to be set, e. g. $2^0 + 2^2 + 2^3 + 2^4 = 29$ for using all options. 4 is a good value for the mode (always link to root node), 0 is default.

MakeLinks

This function is very similar to the one above, but it creates a hierarchical list of links (basically a subset of a site map).

See Table 2.4 for a description of the modes. Mode bit 1 is used here; in the case where non-breaking spaces `( )` are used for indentation rather than list tags (e. g. `<ul>, <dl>`), these have always to be repeated from 0 on. 20 is a good value for the mode, which means that the links to related topics will turn out right, and the main page will always be among the links.

SiteMap

"SiteMap" generates a complete list of all links, hierarchically structured, also known as a "site map".

This macro is a lot simpler, because there is no minimum or maximum level required, and each page appears on the list. The mode argument has two bits:

- bit 0: have the file size after the links included; usually the default value 0 is used (no file size).

Argument	Example value
Starting tag for new level	`'<ul>'`
Starting tag for new entry	`'<li>'`
Ending tag for entry	`'</li>'`
Mode	0

Table 2.6: SiteMap arguments.

- bit 1: don't include the separator before the first entry. Set this bit if you use something like a pipe character `("|")` as a separator, so it will not appear before the first entry.

Back

This macro generates a link to the parent page (the page one level higher in the site map tree whose child is the current page). Unfortunately this involves scanning all the metadata about the pages for the parent. This is the reason why the macros "LinkTable" and "MakeLinks" set the special variable "BACK" for the current page. This variable is cleared before each document is processed and holds the name of the parent page. If the macro "Back" finds the value to be set, it uses it and saves a few CPU cycles. Otherwise, it searches the whole file table for the parent page.

The only optional argument is a text to be included in the link, such as "Back to". You can use this text to link with both an image and text; a possible template definition could be

```
<%Back('<img src="/images/back.gif" bor-
der="0">  ')>
```

Prev

This macro creates a link to the previous page *within* the same subtree in a site map. This means that a page higher up in the hierarchy will not be linked to (the macro will return an empty string), nor will there be a link to a different directory. The best thing to find out how this thing works is to try it :-)

There is also an optional argument, like in the macro `Back`, for the link text that comes before the name of the page.

Next

This macro links to the next page *within* the same subtree in a site map and works otherwise like `Prev` (see above for more information).

LinkTo

This macro generates a correct link to a variable of which the content is a file within the "source" tree. This can be used to get a correct relative link to binaries, in case they have an exotic extension which has not been specified in the configuration as an extension for binary files. The only argument is the file to be linked to. This macro is now deprecated and only included as an example for IMP calls.

Pictures

Sometimes a page with all pictures within one directory is needed (e. g. for screen shots). This simple macro takes all files within a directory and creates a table for them. The first argument is the directory, the second (optional) one is the width of the table in cells (default: 2).

2.5.4 Adding new macros

If a macro is too long to be included in one line of code (e. g. the template), it is recommended to add it to the `macros.pl` file. Then its name has to be added to the configuration in order to allow the safe interpreter to execute the macro. Even then, malicious macros cannot do much more than screwing up the layout of the output page or force the preprocessor to quit (e. g. because of a syntax error). Unlike in a popular(?) word processor, macros do not have the full control over the computer they run on :-)

2.6 Other replacements

Other operations than the replacement of tags defined in the template and variables (see Section 2) is the replacement of absolute URLs (starting with `http://`, `ftp://` or `mailto:`) by a link, if this has not been done yet by the user. The same goes for e-mail addresses. There is not yet a way to override that behavior if a string that looks like a URL or e-mail address is not what it seems to be. However, the recognition of these strings is quite reliable, and I have not found a need to turn this feature off.

2.7 "Magic" tags

Sometimes, it is desirable to use information inside a document rather than within the template. This is implemented by giving the parser a set of "magic" tags that trigger a special behavior: before the content between that opening and closing tag is evaluated, the parser first searches through the file for the ending tag and then sets the variable with the name of the "magic" tag to the string between the tags. In the configuration file, a set of possible "magic" tags is specified. This may seem awkward but it is the only way to prevent having either two passes when compiling the files or maintaining a huge lookup table for any content between any opening and closing tags (not to mention the tags that will never be closed, such as `<hr>`). This would have been an overkill for such a feature that is not that important.

After setting the value of that variable (and potentially overwriting a user defined variable from the dictionary), parsing goes on, back after the opening tag.

E. g. if

```
<title>Hello, world!</title>
```

occurs in the document and "title" is one of these "magic" tags, then the variable "title" will be set to "Hello, world!" once the parser reaches that position in the document. After that, parsing will continue with `Hello, world</title>` and the rest of the document.

These magic tags can be used to include information from the document (such as the title) within the header and footer in the template[1].

2.8 Preprocessor directives

In the first line of a document, special options can now be specified in order to change the behavior of the preprocessor. This is sometimes more elegant than changing the options, and allows the inclusion of features that would otherwise not be possible to implement.

2.8.1 `skip`

If the first line of the document is

```
<!-- skip -->
```

then the document is not parsed at all. Instead, it is verbatim copied to the destination directory (after removing that first line). This allows for pages with a special layout, ignoring the template (e. g. the main page is usually different than the other ones).

2.8.2 `foreach`

Sometimes an HTML page containing variables should generate a set of output pages. An example is when there is a page about financial data for each year. The web master generates a document holding all information such as number of each product sold, costs associated etc. in variables. The parser should then generate a set of output files based on special supplementary data files (which are variable dictionaries).

This is a very powerful directive. The same keyword as in Perl has been chosen, which may make it easier to remember, but also easier to confuse. The first line looks like this:

```
<!-- foreach finance*.data year -->
```

The meaning is: use every file matching the "globbing pattern" (an asterix meaning any number if any character; other globbing options are not yet allowed) and generate an output page, based on this document. For instance, if there are two files: `finance1998.data` and `finance1999.data`, then the two pages `finance1998.html` and `finance1999.html` will be generated. (The extension will be taken from the document containing the `foreach` directive).

[1] In fact, this is the only occasion when I used this feature so far. The documents have their own `<title>` tags, which allows the title to be used in the template with `<title>"title"</title>`. See Section 2.2.5 for an example.

In order to be able to refer to the value of * (the year in this case), an optional second parameter specifies the name of the variable containing that value. This makes it very easy to design a page with "financial data of year xy" and use that page for generating pages for each year.

The format of the data files is exactly like the variable files. Indeed, some values are expected to be inside. The "startPat" option (see Section 5.2) has to contain a single HTML tag, and the tag name has to be defined as a variable as well. For instance, if the "startPat" option is set to `<title>`, the variable "title" has to be found in each `*.data` file. It will then be used in the document title, if that title contains the variable "title". A simple example makes this less obfuscated:

finance_template.html This is the "sub template" which generates a set of pages with financial data. Its first line is

```
<!-- foreach finance*.data year -->
```

as the first line. In all subsequent lines, variables from the data file can be referred to as usual, particularly in the line

```
<title>"title"</title>
```

which allows the inclusion of the title from the data document (otherwise the title would be the same for each output document).

.data files These files are simple variable files (see Section 2.3 on page 16), containing all values that are referred to in `finance_template.html`. The "title" variable is compulsory, since it has been defined as the pattern containing the name and thus the "keyword" for each document.

It has to be noted that the old values of variables which are defined in the data files will be overwritten and not restored later. This improves the performance of the preprocessor, and should not be a problem if the names of the data fields are chosen carefully. A more elaborate example can be found at Appendix A.1 on page 69.

2.8.3 `if/else/endif`

These preprocessor directives are quite simple to use. Their syntax is:

```
<!-- if (expression) -->
HTML text which appears if expression is true
<!-- else -->
HTML text which appears if expression is false
<!-- endif -->
```

The `else` part is optional. The expression can be any Perl statement; "true" is also defined according to the Perl standard, i. e. a value 0 or a string "" is interpreted as false, and any non-NULL or non-zero value as true. It is possible (though probably not meaningful) to call any IMP function that a macro is allowed to call. There is no % sign needed in that case. It has to be noted that despite the similarity to macros, the function of 'if/else' is on a different level: the body between the statement and 'endif' is used or not depending on the outcome of the expression.

The statements can be nested to any depth, like in any programming language. Because the 'endif' is compulsory, there can be no ambiguity in the case of nested statements. If more complex expressions have to be evaluated, it is better to include a short function in `macros.pl`, which, unlike a normal macro, does not return a string with HTML, but either true or false.

2.8.4 `include`

The include directive provides an alternative to tag redefinitions. It is not more powerful than tag redefinitions, but may be more convenient. The syntax is simple, but allows many options:

```
<!--[#]include [file|virtual=]"file" -->
```

The most simple way of using it is just `<!-- include "file" -->`. The other "flavors" of the syntax are compatible with server side includes (SSIs). Sometimes server side includes are used for static files, in order to organize a web page! With HTML PLAIN, these tricks are no longer needed. In order to prevent having to change all files, HTML PLAIN also accepts the SSI syntax here.

"File" may be a symbolic or a real file name (because the include file may not have the necessary tags to make it appear in the file table). If the include file has a tag like `<title>...</title>` which gives it a symbolic file name, this tag is later filtered out.

When using server side includes are used, the file may of course be dynamically generated (e. g. in form of daily updates), and a static inclusion is not the desired effect! Therefore, if an include file should be dynamic, it should *not* be present at "compile time", i. e. when `htmlplain` is run. The program will preserve the old include statement, and proceed as if nothing had happened. If the simpler HTML PLAIN syntax is used, an error message is generated if the file is missing.

Include pages are processed every time when they are included, which means that macros will always evaluate in the context of the page which has the `include` statement. Special variables (see Section 2.3.2) such as "SELF" are *not* changed while the include page is processed. — At this version, the include page is also processed, independently from being included somewhere, as a normal page, because the program does not check whether a page is included elsewhere. This may result in some unwanted output pages, and the "hide" configuration setting should include the file name of that page, in order to hide it in the site map.

Include files can be as complex as any other HTML file, i. e. they can use all macros, directives etc. that other HTML pages can use. However, the `foreach` statement does not make sense in this context (because that statement produces a set of pages, and `include` deals with a single page). If `foreach` is actually used in the include page, it has no effect when the page is included; however, it will still work as usual when the include page itself is processed. Nested includes are allowed.

Chapter 3

Parsing the documents

3.1 Why Perl?

The challenge is of course to parse documents efficiently, despite the large number of options allowed. Perl has been chosen as the implementation language. Unfortunately, not all Perl idioms and notations are easy to read (for a nice example see Section 3.3.1). Some idioms are very concise (and thus harder to read), and some expressions (especially built-in special variables and regular expression substitution) consist mainly of non-alphanumeric characters, which makes some lines of code look more like ASCII art than a programming language. This earned Perl the reputation as a "write only language". However, parts of the difficulties arise from slightly different notations, different operator precedence and data types which are not available in other languages (or only in a less flexible form). Another feature of Perl is that "it occupies half of the English name space" due to the huge number of built-in functions. Even without including Perl modules which introduce more names, it is possible to write poems in Perl (which even compile!), so there has to be something about it. You will find more about this in [Wal96, chapter 8].

Despite these minor disadvantages, Perl's features proved tremendously useful for implementing the parser. The parser heavily uses dynamic arrays, hash tables and more complex data types built upon these. The string substitution functions were of course invaluable. With "old fashioned" string libraries like C's `string.h,` the development would have been much slower. Regular expression matching and substitution allow very complex operations in just one line of code. There are other languages like Tcl which also offer regular expressions, but Perl's expressions have more powerful features than any other package I have found so far.

For prototyping, Perl's underlying philosophy was also very helpful.

> "Perl does not enforce private and public parts of its modules as you may have been used to in other languages like C++, Ada, or Modula-17. Perl doesn't have an infatuation with enforces privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun." [Wal96]

During the earlier development stages, when the interfaces still change now and again and it is not known which variables need to be accessed from outside, this is very useful. It saves the work of writing access functions for each variable that is used throughout the entire program. An example is the global line counter which hopefully gives more helpful error messages. It counts the line of the current template or document that is being processed. Because many modules have to access this variable, it is global. The alternative would have been to create an extra package (i. e. object) for this single variable, and an access method for incrementing it, and another one for obtaining its value. This is a nice design in theory, but not really needed in practice.

Perl still supports true privacy and encapsulation, and this feature has indeed been used frequently, e. g. in the buffer package (Section 3.4.1), where the number of operations is bigger, and a few auxiliary variables are involved. Here, the keyword `my` makes sure that no other packages can mess up the contents of the buffer. It provides *lexical scoping*, which means that the variable is only visible within the current block (similar to `auto` variables in C). The block may be just a `for` loop, even a single statement, or an entire package. Us usual, Perl is rather open minded about these things. Probably the most popular line of code which involves the keyword `my` is:

```
my $OS =~ s/Windows/Linux/;
```

As usual, Perl code is often close enough to natural language, so the reader should be able to understand the code without more comments :-) To sum it up, Perl's functionality, which specializes on text processing, perfectly suited the task.

3.2 Perl modules used

Table 3.1 shows an overview of all non-standard Perl modules that are used by the parser and the other programs. Stand alone programs that can be called from the command line have no extension, and modules used with `require` have the `.pl` extension. There are no `.pm` modules, because they are basically the same but require exporting the procedures that should be public.

3.3 Parsing the template and variables

First, the variable dictionary is read, and all variables are stored in a hash table. The same is then done with the template. All declared variables are replaced in the template while doing this. Finally, the file table is read.

3.3.1 The storage module

Since other languages do not have extensible hash tables as built-in data types, I encapsulated the storage of the definitions into a small Perl module. This very simple module handles all "load/store" requests, hence the names for the subroutines

```
sub load { # load (ref_to_storage, key)
  return ${$_[0]}{$_[1]};
```

Module name	File name	Purpose
Generator for argument parser	`args.pl`	Prints code for getargs.pl
Line buffer	`buffer.pl`	Buffers output, allows for changes later
Configuration file	`config.pl`	Sets global configuration variables
Converter for existing pages	`converter`	Converts a non-HTML PLAIN pages for being used with HTML PLAIN
File table generator	`plainfiles`	Updates file table; needs to be run each time a document is added or deleted
Argument parser	`getargs.pl`	Parses command line arguments
Include latex2html output	`latex2htmlinc`	Fixes a bug with images, and also adjusts filenames for being used in HTML PLAIN
File table loader	`ldfiletable.pl`	Loads file table from file into memory
Macros	`macros.pl`	Contains HTML PLAIN macros
Warnings/errors	`output.pl`	Prints warnings, errors and colored output
Main parser	`htmlplain`	Main program, parses each document
Parsing procedures	`parsing.pl`	Auxiliary parsing procedures
Storage module	`storage.pl`	Handles (and encapsulates) hash tables
Template parser	`template.pl`	Loads template from file into memory
Variable file parser	`variables.pl`	Loads variables from file into memory

Table 3.1: Perl modules and their file names.

```
}

sub store { # store (ref_to_storage, key, redefinition)
  ${$_[0]}{$_[1]} = $_[2];
}
```

The only difficulties arise from the Perl syntax. The operator precedence is crucial and can cause many hours of fruitless debugging if not known beforehand. The problem is the same for both subroutines.

First, `$_[0]` is evaluated, the first argument, giving the location of the storage. The curly brackets `{}` around it are needed to prevent the Perl parser from evaluating `$$_` first. This would use `$_` as a pointer, which makes of course no sense. The leftmost `$` dereferences the pointer to the hash table, and `{$_[1]}` takes the hash bucket with the second argument as the key, according to the common Perl syntax `$hash{key}`. A more precise explanation can be found in [Sri97, pp. 6 f.]:

> "The expression involving key lookups might cause some confusion. Do you read `$$rarray[1]` as `${$rarray[1]}` or `{$$rarray}[1]` or `${$rarray}[1]`?
>
> (Pause here to give your eyes time to refocus!)
>
> As it happens, the last one is the correct answer. Perl follows these two simple rules while parsing such expressions: (1) Key or index lookups are done at the end, and (2) the prefix closest to a variable name binds most closely. When Perl sees something like `$$rarray[1]` or `$$rhash{"browns"}`, it leaves index lookups

([1] and {"browns"} to the very end. That leaves $$rarray and $$rhash. It gives preference to the '$' closest to the variable name. So the precedence works out like this: ${$rarray} and ${$rhash}. Another way of visualizing the second rule is that the preference is given to the symbols from right to left (the variable is always to the right of a series of symbols).

Note that we are not really talking about operator precedence, since $, @, and % are not operators; the rules above indicate the way an expression is parsed."

3.3.2 The output module

This is a very simple module, allowing printing in colors (if the terminal supports this), which was helpful for early debugging stages. It also offers an interface for printing warning or error messages. In these cases, it checks for the global variables `fileName` and `line`. If they are defined, they are used in the error message, hopefully helping to find the error.

3.3.3 Auxiliary parsing procedures

This package contains methods that deal with a single HTML tag or variable or several of them. It allows operations like replacing the tag(s) with the corresponding redefinition(s) (and the same for variables), overriding options within HTML tags and getting the correct relative path from a document to a target file. These functions are used by the template parser, the variable file parser and of course the main parser. Due to the importance of this module, I will go more into depth of some of its functions.

replaceTags

This function replaces a number of tags with their redefinitions, if any. Tags that are not redefined are returned unchanged. The function `replaceTag` is used to actually change the tags; `replaceTags` just splits up a string containing several tags into its components. This is usually trivial unless macros are involved. Because macros can in turn contain tags as arguments, a simple regular expression cannot parse them any more.

breakUpTags

Breaks up a number of tags into an array, which contains the content of the tags. Every element holds the text between the angular brackets. This function is used for decomposing the header and footer of the document, allowing a faster processing of it later on.

replaceVars

Similarly to `replaceTags`, this function checks a string for all possible variables (text in double quotes) and calls `replaceVar` which tries to replace them.

Expression ($n)	Meaning
\s+(\w+)	Option (is always preceded by space)
\s*=\s*(.*?)\s	The value of the option (a string between spaces)
(\1)	Another option with the same name (and thus being obsolete)
\s*=\s*(.*)	Everything that comes after the equal sign (may contain more options)
$1=$2	replacement: discard second option and following text ("tail")

Table 3.2: Regular expression for removing duplicate arguments

replaceOptions

This procedure eliminates double occurrences of HTML options when an option should be replaced. For example, the redefinition for `<img>` could be `<img border="0" ...` (and more options for the height and width). Assume that the user wants to override this with `<img border="1" ....` By simply inserting the new options at the beginning (after the tag name), the result would be

```
<img border="1" border="0" ...
```

which is correctly interpreted by Netscape 4.x (it discards the second parameter), but looks *very* ugly. It is also not sure whether future versions of Netscape treat this differently. Therefore, duplicate options like these need to be removed. This is easier said than done (because this operation is often used and has to be very efficient). This is a slightly cut down version (some early checking for optimization is removed) of the main loop, which is used for each tag to remove duplicate options:

```
$old =~ s/\s+/$new /;       # add new arguments at front
while ($old =~ s/\s+(\w+)\s*=\s*(.*?)\s(\1)\s*=\s*(.*)/ $1=$2/i) {
   # check for duplicate options and remove them
   $tail = $4;
   # restore rest of string (everything after argument name)
   unless ($tail =~ s/^\s?\".*?\"\s?/ /) {
     # remove first argument in quotes
     $tail =~ s/^.*?\s/ /; # keep everything after space
   }
   $old .= $tail;
}
```

The first line simply inserts the new argument after the first space. The second line may cause some dizziness. The regular expression can be decomposed like in Table 3.2:

As the reader can now (after some regular expression acrobatics) see, this regular expression substitution solves almost the entire problem with one line of code. The only difficulty is the fact that the argument of the second option, which has to be removed, is not always enclosed in quotes. There are two cases:

1. The string is enclosed in double quotes ("), and the argument ends with the second quote.
2. The string is not enclosed in quotes and ends with the first whitespace.

The rest deals with this. If the "tail" contains a quoted string, it is removed, otherwise the text up to the first whitespace is removed. By using a non-greedy matching (`\".*?\"`), it is made sure that only the text between the first and the second quote is removed. After removing the duplicate argument, the whole procedure is applied again until no more duplicate arguments are found. It is recommended to keep a nice HTML style, with all arguments enclosed in quotes, in order to help the parser.

replaceTag

This procedure checks whether a tag has been redefined, and if so, replaces the tag with the new one. It calls `replaceOptions` (see above) for taking care of duplicate options. The user may define a different meaning for a closing tag, e. g.

```
/tt      </tt><sans>
tt       </sans><tt>
```

This deals with the fact that the text in the document body is enclosed in `<sans>` tags, which redefine the font. Then, the `<tt>` tag, which should force the browser to use a fixed space font, does not work properly anymore in Netscape. To work around this bug (in Netscape), the `<sans>` tag has to be closed before the `<tt>` tag is opened, and the `</tt>` tag has to include a new `<sans>` tag at the end. Note that the order in which these definitions occur in the template is significant. If the opening tag was defined first, then the template parser would replace the `</tt>` tag at the right hand side of the redefinition of `</tt>`. This "overzealous" recursion can be avoided by defining the closing tag first. Since the parser cannot match `<tt>` with the closing tag `</tt>`, everything works fine. In most cases, this behavior is what you want, but there are exceptions, like this one.

Another special property is the `<img>` tag. In order to avoid problems with images that are not found, the special variables "HEIGHT" and "WIDTH" (see Section 2.3.2) are set to the undefined value before the options within the `<img>` tag are processed.

replaceVar

That part was fairly straightforward in older versions of the parser, but creeping featurism demanded its tribute. Now, every time when a file (a document or image) is recognized as belonging to a variable, a few special variables have to be set in order to be able to "remember" attributes like file name, size etc. later. Other than that (and taking care of options/switches), the code is still rather simple.

adjustPath

This procedure creates a relative link when given two absolute locations. The only peculiar line of code is

```
"$path1\t$path2" =~ /^(.*\/)(.*?)\t\1/;
```

Here, it seems to be rather strange to do something with a "string constant" on the left hand side. As a matter of fact, that string that is generated there is *not* a constant, but an "anonymous" string, where only Perl knows how to retrieve it. This string is temporarily generated, and the regular expressions checks for a part in the path that is common to both locations. The `\1` is used here to represent a duplicate word. For more information, see [Chr98, pp. 194 ff.].

`URLify`

These lines of code are a slightly extended version of the example subroutine in [Chr98, Section 6.21].

Other subroutines

The other subroutines are less important, and easier to understand (hopefully). The code is well documented, so all information can be found there.

3.3.4 The template parser

The template parser takes the template file and stores its content in a hash table. It uses references to the hash tables, but does not really know that the storage is organized as a hash table. Using the module `storage.pl`, the structure of the storage is hidden.

After stripping out empty lines or lines with only comments (starting with `//` or `#),` the real parsing begins. C comments also require some care to be filtered out properly. The first tab `(\t)` character is searched, and it will separate the tag from its redefinition. The main challenge are definitions that go over multiple lines; they make the code more complicated. In order to allow multi line redefinitions, a redefinition is only stored when a new tag or entity is defined. In that case, the redefinition of the old tag is finished, and the old tag is stored. The redefinition of the new tag can then continue on the next lines.

All in all, the module is pretty simple, and more information can be found in the source code itself. `$` and `^` are not treated specially, because these names can just be used as a hash key; Perl allows not only letters and numbers, but pretty much anything to occur in the hash key.

The hash tables for entity and tag redefinition

Redefined `&entities;` are stored in `entity => redefinition` format, i. e. the ampersand `(&)` sign and the trailing semicolon are stripped out. Taking these two characters away makes the hash keys shorter. Similarly, HTML tag redefinitions are stored in `tag => redefinition` format, without the beginning and ending angular bracket. Any angular brackets in between are not removed. The two hash tables for entities and tags are separate, so there is no name space conflict after truncating the hash keys.

3.3.5 The variable file parser

This parser is much simpler than the template parser, because redefinitions going over multiple lines are not allowed. The only job it has to do is to split up the lines into the variable and the content and store them at a given position. It works in the same way as the template parser.

3.3.6 The command line argument parser

Using the module `Getopt::Long` from the standard Perl library, parsing command line options is quite easy. The syntax is like

```
GetOptions([option description]);
```

where the options are described as

```
'syntax' => reference to variable
```

See [Wal96, Chapter 7] for more information about this module.

However, including this for all options from the configuration file involves a lot of typing and is error prone. There is also the danger of inconsistency. Therefore, a small program has been written which extracts the options from the configuration files and generates Perl code which parses these options as the output. The main loop is quite simple and looks like this:

```
foreach  (@lines) {
  if (/\$(\w+) ?= ?'(.*?)';/) {
    # string "$value = 'value';..."
    if ($2 eq '0' or $2 eq '1') {
      print "\"$1\" => \\\$$1,\n";
    } else {
      print "\"$1=s\" => \\\$$1,\n";
    }
  }
}
```

The second line with the cryptic regular expression simply checks for strings containing a variable definition. If the value of the variable is either 0 or 1, it is assumed that the variable is a flag (I know that this may be wrong in some cases, but without any other information than the variable assignments, all you can do is guessing). The Perl code which is generated has the syntax described above. Two backslashes have to be used for quoting a backslash, and a backslash also precedes `"` and `$`, which makes the text within the `print` statements harder to read.

3.4 The document parser

The parser itself needs a few extra modules for its functionality. I will describe these first, and if the reader has not fallen asleep yet, he can then admire the description of the main module.

3.4.1 The buffer module

This module allows storing scalars (usually strings) in a buffer. This is needed for the second "pass" of the preprocessor, which deals with the header and footer of the document. It is also more efficient than a lot of write operations to a file, because all write operations happen in one run. The module offers one buffer and a lot of access functions, which allow appending a line to the buffer, getting, replacing, and inserting a line or several lines. Unlike the storage module, this one only supports a single buffer. However, it could easily be augmented to support multiple buffers, it that should be needed.

3.4.2 The macros

This package contains all the macro functions used. Note that even though they access all metadata through the IMP (see Section 2.5.2), they could also access variables or procedures directly, *even though* macros are run in a "safe" interpreter. This is because the safe interpreter allows access to the macro functions, which have in turn access to everything. The macro functions which can be accessed from the safe interpreter have to be defined beforehand. While they have full access to all Perl commands, it is not possible to smuggle in "rogue code" into the documents, because macros in these documents can only access predefined macros and IMP functions. Therefore, assuming that my macros are safe, macros cannot do anything worse than generating bad HTML output.

3.4.3 The file table loader

This package parses the output file from the file table generator (see Section 3.5). Due to the well-defined structure of that file, parsing it is trivial. After parsing it, some data structures are prepared in order to ease the work of the IMP (see Section 2.4). All IMP functions that deal with the file table are also contained in this module.

3.4.4 The parser (main module)

This module is the core of HTML PLAIN. It transforms the higher level documents and processes them as described in Section 2. First, a lot of preparation has to be done: the settings have to be read, storage has to be allocated, the template and variable files are read into memory via calls to these modules, a safe Perl interpreter is prepared, and the `mtime` (time of last modification) of some key files is checked. Then, each document is processed.

Before the text can be replaced, it has to be parsed first, though. This is the hard part. Unlike normal HTML, where all relevant entries are enclosed in angular brackets `(<>)`, there may now also be entities `(&entity;)` and strings enclosed in double quotes `(")`, representing a variable. This makes it impossible to just use `split` and dissect the string into all entities and non-entities. Instead, the parser needs several modes:

1. Standard mode. The parser is in this mode at the beginning or when reading plain text. It expects either a HTML tag or a string in double quotes. A missing ending double quote (before the end of the line) generates a *warning* here, since it may be used for quoting large parts of text.

2. HTML tag mode. No tags or entities inside tags (i. e. between the angular brackets) are allowed, but variables can still show up. Strings in double quotes are likely, either as simple arguments or variables.
   A missing ending double quote generates an *error* here, for it is needed in a correct document. If the closing bracket (>) is not found on the current line, the parser goes on with searching until the end of the file is reached (and an error generated) or the closing bracket is found. A warning reminds the user of adapting a better style of HTML next time. These warnings also appear very often if `latex2html` output is included (also see Section 5.5 on page 62).

3. Variable mode. Here the parser just has to look for the second double quote ending the variable name.

This yields the following (simplified) grammar for the new kind of HTML documents (as a regular expression):

Term		Definition
document	::=	(plain_text | HTML_tag | variable)*
plain_text	::=	(ASCII_character)*
HTML_tag	::=	< (plain_text | variable)* >
variable	::=	" plain_text "

This simple grammar is (fortunately) context free, which makes it easier to parse it. The parsing can be done with three important stages (as described above), without any extra information (context). “Magic tags” (see Section 2.7 on page 24) and macros make the implementation a bit more complex, but the general algorithm remains the same.

First prototype parser

The first prototype has been written in Perl within about two hours, and was not fully optimized. The main loop is still the same in the final version. Most of the small subroutines in that module are very simple, and not described here.

Mode 1 - standard mode (unrecognized text)

The main loop for each line in the parser goes like this:

```
while (&checkToken) { # either only " or " first
  if ($quotePos < $bracketPos) { # string in quotes
    &variable(1);
  }
  else {
    if ($bracketPos != -1) { # HTML tag, parse
      &HTMLTag;
    }
  }
}
```

Every line is checked for an occurrence of an HTML tag or a quoted string, and if one of these is found, it is further examined. A check for HTML entities has also been added; since they can only occur in plain text, the check is not hard to implement.

Mode 2 - parsing HTML tags

This part is now too long to include the whole source code here. Most of the first part deals with the problem that line breaks are allowed within HTML tags, even though they should not be, because this leads to an ugly style and makes parsing harder. The syntax is not fully checked, e. g. errors like `<h1 title</h1>`, where the closing > has been forgotten, are not recognized. Netscape gets into trouble as well, so there is no reason to be better than that highly sophisticated piece of software. It is up to the user to preview his documents in a browser to check whether the HTML is correct. He should not use line breaks within HTML tags, and take the warnings seriously and remove these line breaks.

The second part of the procedure dismantles the HTML tags into their parts: HTML within the angular brackets and options within double quotes, which may have to be replaced according to the variable dictionary (see Section 2.3). HTML tags have to be checked for occurring within the template, in which case they have to be replaced (see Section 2.2). If they are replaced, care needs to be taken for tags with options (like `<font face="Helvetica">`). The option has to be filtered out when the tag is closed. After this step, we finally get a full HTML tag with options, and have to take further actions:

1. The tag is a "magic tag". In that case, the text up to the closing tag has to be read, and stored in the variable which has the same name as the tag itself (see Section 2.7). The whole string after the opening tag has then to be processed again.

2. The tag is a "kill tag", which simply means that everything the opening and closing tag, including these tags themselves, is removed.

3. We have a macro that has to be executed.

4. We have a simple HTML tag with no more bells and whistles.

Mode 3 (string in quotes)

```
sub variable {
  # handle text after a double quote as variable *or* plain text
  if (/\"(.*?)\"/) {
    # syntax check for "..." and extraction of var. name
    plainText($`);     # text before quotes as plain text
    &printToBuffer(&replaceVar($1, 1));
    $_=$';             # postmatch = plain text or rest of tag
  }
  else {
    &warning ("Closing quote not found. Treating as plain text.");
    plainText($_);
    $_="";
    return;
  }
}
```

Value	Meaning
-1 (default)	outside an `if` statement
0	within an `else` statement
1	within an `if` statement (before `endif` or `else`)
2	"falsify" `if` statement if another `if` block lying outside the current one has been evaluated to false (and the current block should therefore be ignored)

Table 3.3: Meaning of 'if states'

Code	`@ifReturn`	`@ifState`
`...`	`(-1)`	`(-1)`
`if (true)`	`(1, -1)`	`(1, -1)`
`  if (false)`	`(0, 1, -1)`	`(1, 1, -1)`
`  else`	`(0, 1, -1)`	`(0, 1, -1)`
`  endif`	`(1, -1)`	`(1, -1)`
`else`	`(1, -1)`	`(0, -1)`
`  if (false)`	`(2, 1, -1)`	`(1, 0, -1)`
`  else`	`(2, 1, -1)`	`(0, 0, -1)`
`  endif`	`(1, -1)`	`(0, -1)`
`endif`	`(-1)`	`(-1)`

Table 3.4: 'If' return values and states within the program flow

This part is fairly straightforward. If a variable name (a string between double quotes) is found, it is replaced by its value. If the value (or the original string) happens to be a URL, it is "URLified", i. e. a `<a href="URL">URL</a>` is inserted. If no closing quote is found, a warning is printed. In most cases, these warning can be ignored (e. g. in quotations that go over several lines).

3.4.5 'If' evaluation

There are two lists involved with if evaluation: `@ifReturn` and `@ifState`. The first one corresponds to the list of return values of nested 'if's, the second one notes the position of the parser within a tree of 'if/then/else' statements. See Table 3.3 for the values that an element of `@ifState` can take.

Both lists are initialized with the single element -1, making an easier check (see below).

An example makes it clearer (see Table 3.4). The table shows the states *after* the statement has been evaluated.

As one can see from this table, the actions to be taken when a statement occurs are clearly defined. Table 3.5 shows an overview. The example does not show what happens if no 'else' statement occurs. Because an else statement only changes the first element of `@ifReturn` but does not change the number of elements, the states are still consistent without having an 'else' clause. Also, due to the way that both lists are defined, an arbitrary level of nesting can be allowed.

It is important that the lists are properly initialized, and that the list `@if-`

Code	Actions
if	- check current `@ifState`; if we should ignore this block, use 2 as "false" (not equal to 1) return value.
	- evaluate if, prepend return value to top of list `@ifReturn`
	- prepend '1' to `@ifState`
else	- no action for `@ifReturn`
	- change first element of `@ifState` to 0
endif	- remove first element of `@ifReturn`
	- remove first element of `@ifState`

Table 3.5: Actions to be taken for 'if', 'else', 'endif'

`State` always contains at least one element. This should be tested before an 'endif' statement is evaluated, in order to prevent the list from becoming inconsistent if an 'endif' statement occurs without a corresponding 'if'. Also, after parsing a document, the list can be checked for missing 'endif' statements in the same way. When an 'else' statement occurs, the first element of `@ifState` cannot be -1 — otherwise we have an error as well.

Optimizing the evaluation

By using '-1' as a special value for `@ifReturn`, denoting "outside any if statements", we can

- keep that list non-empty
- get an easy check whether a line should be included or not - `if $ifState[0] == $ifReturn[0]`, then include the line, otherwise don't.

The changes to the program are quite simple. `@ifReturn` just has to be initialized to (-1). The consistency test is now simply that the first element is not -1 (instead of non-empty).

Implementation issues

There is still a small problem left: if all lines are ignored until the 'else' or 'endif' is found, these statements themselves are ignored, too. Therefore the check for 'else' or 'endif' statements has to be made in the main loop (for each line), making the code harder to read. A check there is much more efficient than after parsing each line, though.

3.5 Maintaining the file table

The basic task for this program has been outlined in Section 2.4.1 on page 18. There are currently three different file types supported by this program:

1. HTML files. In that case, the title is extracted (which is any string between two regexp patterns), and the file table entry consists of the title, the full

file name, the file size and the content level (see Section 2.2.6). <!-- foreach --> files (see Section 2.8.2) result in several entries in the file table (one for each data file).

2. Image files. The file table entry is then the "base name" (file name without directory) without suffix, the full file name, the file size and the image size (height x width). In the case of the simple GIF format, the program can figure out the image size itself; otherwise, the utility `identify` is called. It returns a string like this:
   `penguin.gif 257x303+0+0 DirectClass 16335b GIF 1`
   If the user has a similar program, of which the output contains the information `"widthxheight"` after the first space, he may use that program instead. No guarantee is given that the file table will still be correct then. If someone has a GPL'ed Perl module that can find out the size of a JPG or PNG image, I will gladly include it :-)

3. Binaries. These are not further tested; the information in the file table is base name, full name, file size.

An important feature is that symbolic links are generated in the output directory for each image and each binary file that is not found there. This will make it a lot easier to upload the page later. Unfortunately, there exist still some operating systems that do not have symbolic links. If the linking does not work for some reason (e. g. broken file system/write protection), a warning is displayed.

3.6 Checking for dependencies

So far, only a rather simple (but effective) dependency check is implemented: if any of these configuration files (`config.pl`, file table, variable file and template) or the original document are newer than the output document, then the latter is updated.

3.7 Converting existing pages

The converter takes an existing page (which may also be the output of HTML PLAIN) and tries to "reverse engineer" it given a variable file and a file table. The goal is to reconstruct as much of the higher level information as possible. Because the syntactic structure is lost after parsing (and also in normal documents), the variable contents do no longer appear in double quotes and are harder to find. The only way to convert pages is to search in each line for *every* possible variable content and link from the file table, using a very big pattern for pattern matching. This makes the program rather slow.

Fortunately, Perl does a pretty good job at pattern matching, so the program is not too slow for smaller pages (converting larger pages is a very hard task anyway, because redundant HTML code among several pages is usually no longer fully consistent). Unfortunately, the relative links are different for each document, so the pattern matching string for relative links has to be rebuilt for each document.

If a pattern matches, the converter simply looks up the reversed variable and file tables and replaces the string. Reversing a hash table is quite simple in Perl, because

```
%newhash = reverse %oldhash
```

does the job, despite `reverse` being defined for lists. This is because the hash table gets converted to a list which looks like

```
key1, value1, key2, value2, ...
```

If that list is reversed and then converted back to a hash table, all values from the old hash table become keys for the new one! This is an excellent example that shows how powerful some Perl operators are.

For the file table, the value is a reference to an array with a set of values, so the reversion of the hash table has to be done manually by iterating over each element in the hash table, using `foreach (keys %hash)`. The full file name is extracted from the array and converted to a relative link, which is then used as a key for the hash table.

For all hash tables, a string has to be generated containing each key as an option for a pattern matching string. This string, which may be quite long, is then used to scan through the document and trying to reconstruct as much as possible from the higher level HTML code. In the case of file names and variables, this works quite well. For tags, the problem is more complex due to options occurring in tags, which make the search much harder. In the first version of the converter, redefined tags are not yet converted back to "higher level" tags.

Chapter 4

The graphical user interface

While many users may prefer to edit the files directly in ASCII mode, using an editor like Emacs or vi, there is always the danger that the user does not obey the syntax rules given by the files to edit. In this case, the user must always use tab to separate the entries; if he does not, the settings will not be read correctly.

The graphical front end prevents this by automatically generating the correct syntax. It also provides a more pleasant front end than vi, and is easier to use than Emacs. Because the front end only has the functions that are required, it is much more intuitive than a huge word processor or text editor.

4.1 Tcl/Tk

4.1.1 Advantages

Why did I choose Tcl/Tk, and not a different graphical toolkit? Due to my experience with Perl, something like Perl/Tk sounded very attractive, too, but Tcl/Tk is so widely used that I just had to try it. I had never programmed in Tcl before, but it was not too hard to learn. The syntax is rather unusual, without any round brackets and without having the equal sign for assignments. Tcl/Tk proved to be quite a good toolkit for developing the editors, mainly for two reasons:

1. It is interpreted. This has many advantages: no compile time, the ability to run small parts of the script interactively for testing, and introspection through the `tcl` shell. Especially the latter is sometimes quite useful for figuring out how some widgets work. You run `wish` (windowing shell) and load the program, and then you can experiment with all the widgets in the `wish.`

2. It uses composition, not inheritance. [Sri97, pp. 136 - 137] This basically means that objects (even though Tcl does not use object oriented notations such as classes) are composed of "completely encapsulated components with well-defined and documented interfaces". Therefore all GUI objects have a *has-a* relationship rather than an *is-a* relationship with the widgets it uses. This makes it much easier to create a graphical user interface, be-

cause it is not necessary to derive a new class for changing the behavior of the widgets - instead this is done through their configuration interface.

Another good reason for using Tk is its platform independence. Since Perl is platform independent, too, the whole suite of programs can run on any computer. I may later rewrite the editors in gtk to get some more advanced features for which a Tcl script may not be suitable, but for smaller GUI applications, Tcl/Tk is indeed a very good choice.

4.1.2 A Tk critique

Due to its advantages, Tcl (and later the windowing toolkit Tk) became fairly widespread, mainly in the Unix domain. Even though there are Mac OS and Windows versions available, these were hardly ever used. Tcl was invented as a "glue language" between applications, and there are still a few important Tcl applications left nowadays, such as "expect". ("expect" automates interactive programs such as ftp.)

Tk built on that philosophy, and allows to write small GUI applications rather easily and quickly. However, the number of predefined widgets is rather small compared to modern graphical toolkit such as the Gimp toolkit [gtk] and the Qt library [Qt]. For example, I missed a table-like widget, or a list box widget with multiple columns (including column titles). Both of the other libraries mentioned above have these as standard widgets.

Tk's canvas widget allows the programmer to build such more complex widgets; however, this still involves quite a bit of work. Probably this and the fact that Tcl is entirely different compared to other programming languages account for the relatively small number of Tcl/Tk applications which are released in these days. Tcl/Tk is still a very good tool for prototyping small applications, before the programmer moves on to more complex and more powerful toolkits.

4.1.3 Why not gtk+ (or Qt)?

Besides Tcl/Tk, gtk+ and Qt were two other alternatives. Both have a Windows version of their library, but Qt has no Mac OS version. While gtk+ is written in a sort of "object oriented C", Qt is a pure C++ library. This makes Qt easier to deal with, as C++'s object oriented syntax allows quite an intuitive way of developing graphical applications. There is also a C++ wrapper for gtk+ (called gtk--).

One of the reasons not to choose a compiled language for early development is the time that the programmer has to wait for each compilation. This influenced my choice in favor of Tcl/Tk. There are also other reasons, though: Towards the end of 1998, when the development of the GUI tools started, Qt still had license problems; it did not allow programmers to publish patched versions. Also, it is not very popular, and visually not more attractive than Tcl/Tk (of course this is also a matter of taste).

Gtk+ is highly customizable and by no doubt the best looking graphical toolkit. It also offers a wide range of widgets. On the downside, as it is often the case with libraries that develop quickly, the documentation was never quite up to date, and it was a fast moving target, with new releases every week. The

Module name	File name	Purpose
Color picker widget	`colorpicker.tcl`	User picks a color from the standard WWW (6^3 colors) palette.
Common functions	`common.tcl`	Functions used by both the variable and template editor.
Config tool 1	`config.tcl`	Standard version of config. tool.
Config tool 2	`config2.tcl`	Spreadsheet version of config. tool.
GUI tools demo	`demo.tcl`	Front end for running all GUI tools.
HTML input widget	`htmlinput.tcl`	HTML input with syntax coloring.
Installation program	`install.tcl`	Stand alone installation program.
Main front end	`main.tcl`	Front end for all (Perl and Tcl/ Tk) programs used in HTML PLAIN.
Template editor	`template.tcl`	GUI template editor.
Directory picker	`tkgetdir.tcl`	D. Roche's directory picker.
Two columns	`twocols.tcl`	"Spreadsheet" with two columns.
Two lists	`twolists.tcl`	Two linked list boxes.
	`twolists0.tcl`	Older version used for config. tools.
Variable editor	`variables.tcl`	GUI variable editor.

Table 4.1: Tcl/Tk modules and their file names.

C++ version, which I would have preferred over Tcl/Tk, was still in catching up with the numerous features that were added weekly to gtk+. Therefore, gtk-- was not a good option by the end of 1998 (there was not even an up to date tutorial for it). By now, version 1.2 of gtk+ and version 1.0 of gtk-- has been released. Gtk-- could almost catch up with the gtk+ development, so it is now a good time to write new applications using one of these two libraries.

If the graphical tools should become popular enough, I may rewrite them in gtk one day. I will also consider the gtk for future applications. However, it was not mature enough when I started the development of my applications.

4.2 Tcl/Tk modules used

Table 4.1 shows an overview of all Tcl/Tk modules that are used by the utilities for HTML PLAIN. All programs have a `.tcl` extension, even if they are called from the command line.

4.3 The configuration editor

This editor basically reads a file with Perl assignments in the form

```
$variable = 'value';    # optional comment
```

It is expected that every line to be edited has this form; otherwise, it is assumed that the line is either blank or a comment line. In the latter case, the line is not changeable and not shown in the configuration editor. However, it is inserted in the correct place when the user saves the document.

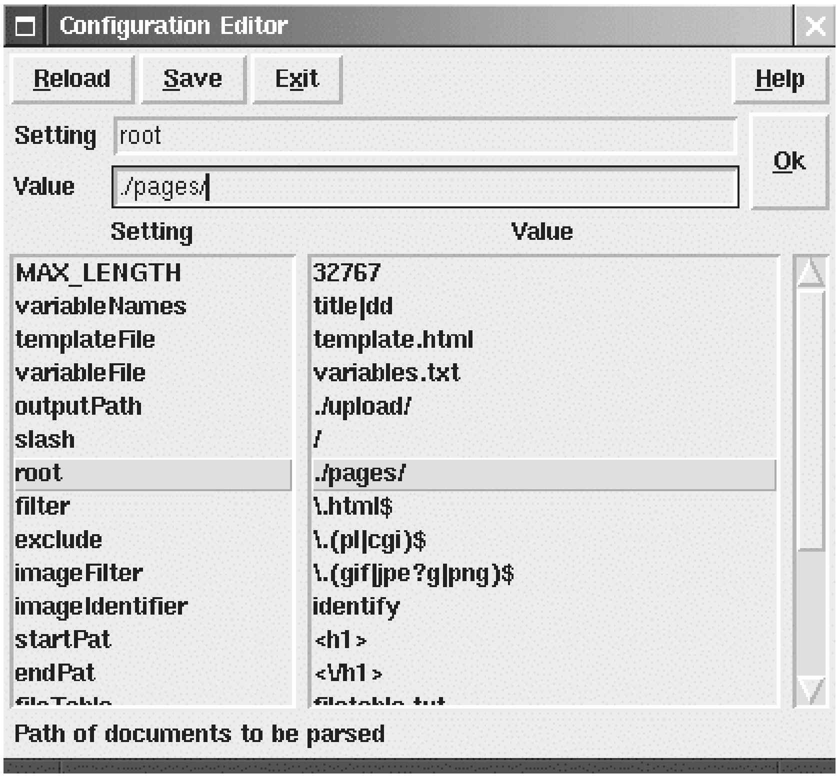

Figure 4.1: Configuration editor, standard version

All assignments with the structure mentioned above are shown as "Setting/Value" in a list with two columns. Because the name of the setting is read only and the number of the settings is fixed, the front end is quite simple. It allows the right hand columns to be changed, but not the left hand ones. Due to the fact that no entries can be deleted or inserted, it is also easy to add the comments again when saving the documents, and fit in any other lines that did not contribute to the settings.

4.3.1 The standard version

This was the first one of the series of configuration tools I wrote. The design is very basic; the only tricky component are the two linked list boxes. Even though it is extensively covered in [Wel97, pp. 513 ff.], the implementation given there is not completely bug free, and lacks any keyboard support. It tried my best to improve it, but there are still a few occasions where the two list boxes do not cooperate as expected - and it seems to be impossible to fix that.

This screen shot (as any other screen shot) has been taken under WindowMaker. It shows the configuration editor. The settings are shown to the left, their values to the right. As soon as a user selects on a setting (or its value) either with the mouse or with the keyboard, it appears in the upper part of the window. The first entry is read only, so the cursor will appear just at the second

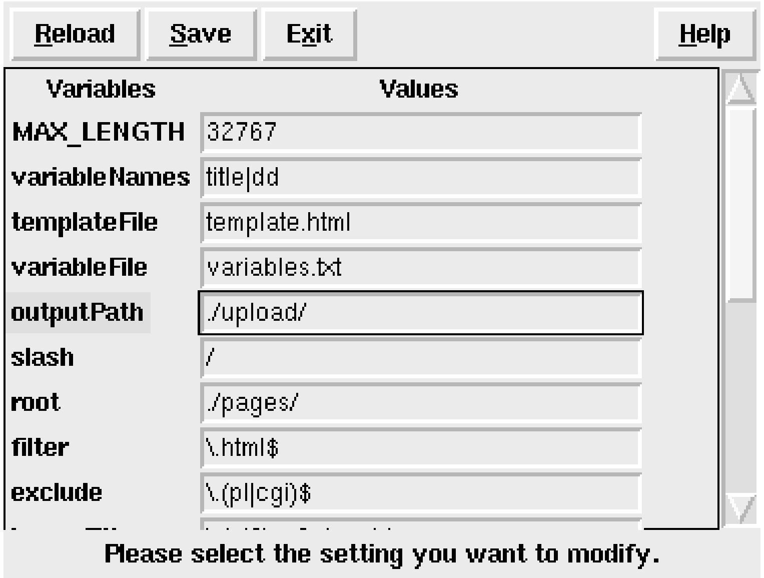

Figure 4.2: Configuration editor, spreadsheet version

part where the setting can be changed. The comments in the script file are used as status messages in the program, in order to provide the user some guidance for selecting the correct setting.

The user confirms the changes with [Return] or by pressing the "Ok" button. The change can be canceled by pressing [Esc] or selecting a different entry. The "Reload" button (later renamed to "Revert") reverts the changes back to the state of the last saved file.

4.3.2 The spreadsheet version

This program has exactly the same functionality as the standard version, but the fields are directly changeable, like in a table or spreadsheet. The user can choose between the two programs.

Implementing the spreadsheet-look-alike required using the Tk canvas, which allows the free placement of a wide range of widgets. Implementing the scrolling is surprisingly simple, and the example given in [Wel97, pp. 411 ff.] could almost directly be used in the actual program. However, there were again some subtleties and problems that were not covered by the book.

First, it seems to be impossible to make the text entries in the canvas object scale if the window size is changed. The canvas grows or shrinks with the surrounding parent window, but the size of the entry widgets is always the same. The book did not provide any hints on whether this can be implemented or not. The only solution that worked was to choose a generous size for the entry widgets and then disabling horizontal resizing, which is of course not a very nice solution.

Second, keyboard traversal and focus behavior were again not covered by the example, but because using a canvas yields a much cleaner implementation than by trying to make two list boxes cooperate, it was not hard to implement

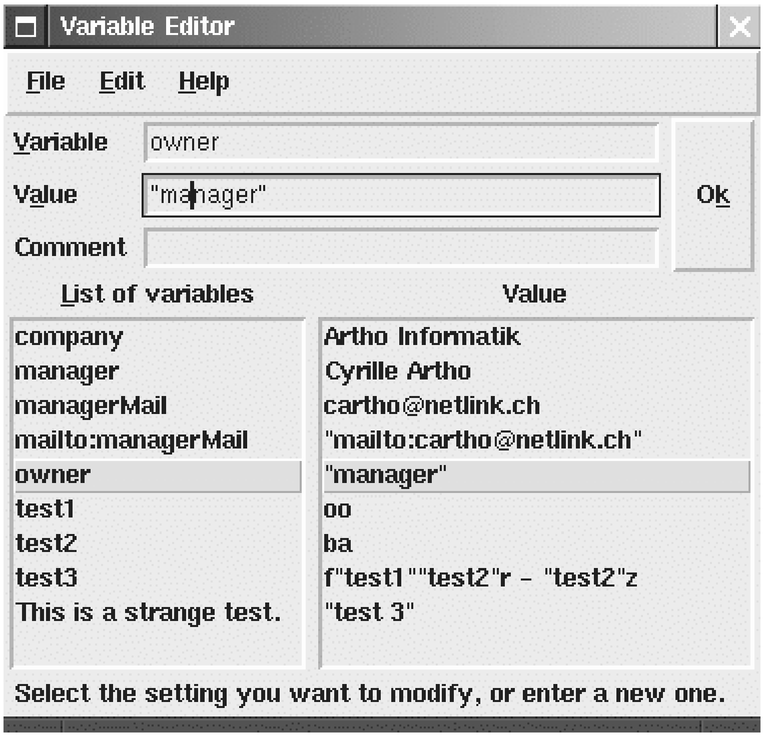

Figure 4.3: Variable editor

that.

4.4 The variable editor

This program lets the user change the values of the variables defined, and also add new settings or delete existing ones. The editor works very similarly to the configuration editor. Unlike in the configuration editor, the comments are changeable, and the number of entries can change, too. This required a whole range of new methods, some of which were not compatible with the object that handles the two list boxes for the configuration editor. Therefore, the code had to be forked, and the module that handles the two list boxes is slightly different.

The user can again select the entry that he wants to change with the mouse or the keyboard. Alternatively, he can press [Insert] or [Delete] in order to create an empty entry or delete an existing one, respectively. An “Undo” operation has been added in order to provide recovery from accidental deletions. The program also checks whether a changed or new entry collides with existing ones (since no two variable names may be the same).

Unfortunately, tracking the position of comments that go over an entire line is quite difficult when entries are added or deleted. Therefore, the method that loads the file stores these comments in a separate list and inserts them all at the beginning of the file when it is saved. This mangles the order in which the comments were inserted. It is recommended not to use comments that go over an entire line or multiple lines, and indeed this option is not allowed in the editor, which only allows the insertion of comments at the end of a definition. This should be all right in most cases, as comments that go over an entire line are commonly only used at the beginning of a file (e. g. for copyright notes).

4.4.1 The color picker

In order to save some typing when the user wants a WWW color (one out of 216 colors whose R, G, B values are multiples of 0x33) or a filename (e. g. if he wants to refer to a non-HTML document), two dialogs "Insert Color" and "Insert Filename" have been provided. They can both be invoked with the right button, in which case a pop-up menu appears. (On Mac computers with one button, a modifier key has to be pressed together with the mouse button for getting the same function.) The filename selection is not worth further description, because it just uses the given Tk dialog `tk_getOpenFile.` The color picker has been rewritten from scratch, though.

The `tk_chooseColor` dialog

There are two color pickers that ship with the standard Tk libraries. One is the `tk_chooseColor` dialog, which belongs to the Tk library and is available through a simple library call. That colorpicker is quite small, fast and simple, but it does not help the user in choosing a color that is compatible to the standard 216 color palette. Another disadvantage is that only the colors that are derived from the current RGB value by changing one number of the (R, G, B) triple are shown. It is therefore not always easy for a user to imagine which slider he has to move in order to end up with the color he likes, especially not if he is not familiar with the RGB model.

The other color picker

Therefore I tried to improve the other given color picker which is commonly found under `/usr/lib/tk8.0/demos/tcolor` or a similar name. That color picker supports the RGB, CMYK and HSV models and also shows a large number of predefined color names. This may sound very appealing (especially for professional artists). However, there are some serious drawbacks:

1. That color picker does again not support the restriction to a WWW color palette.
2. There are more than 400 color names to choose from, much of which are nearly identical.
3. It is not possible to enter the numerical value of the color directly in hex numbers.

The original selection of color names involved more than 600 colors (taken from the file `rgb.txt` which defines the X color names). With the programs `sort` and `uniq,` the selection was reduced by roughly a hundred, and eliminating obsolete gray scale entries by only using every fifth one reduces the

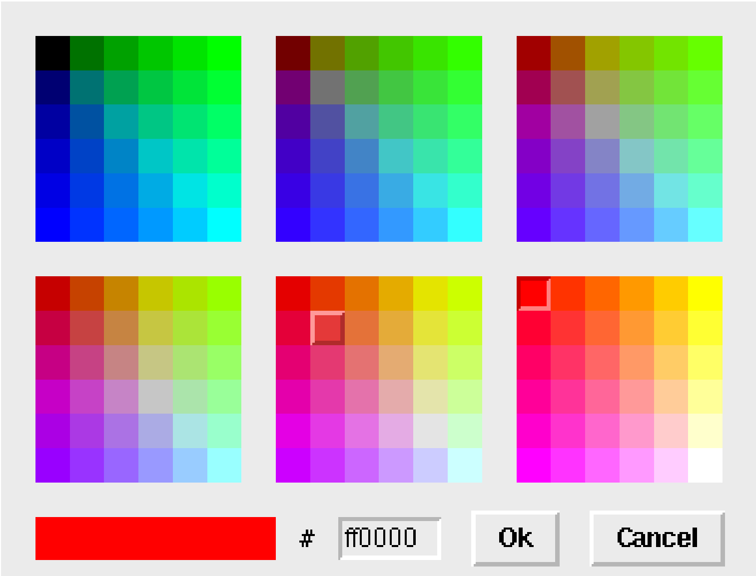

Figure 4.4: WWW Color picker

number further. However, there were still 422 color names left, clearly an overwhelming choice that does not make much sense.

My color picker

Therefore I decided to write a new color picker from scratch, which does not have any of these disadvantages. The user can conveniently select one of the 216 colors directly with the mouse, or enter a numerical value (in which case the color that matches the entered one best is chosen from the graphical representation). The rectangle to the lower left always shows the exact chosen color. The only disadvantage with this color picker is the fact that drawing all 216 colors in the first place takes a while under Tk, but the delay is acceptable (roughly one second, probably less with fast CPUs).

4.5 The template editor

The template editor has almost exactly the same functionality as the variable editor; therefore, the two programs share most of their code in a module called `common.tcl`. The major difference between both programs is the fact that tag redefinitions can span several lines; therefore the parsing of the file structure and the user interface become a bit more complicated.

4.5.1 Syntax coloring

One of the more outstanding features of the template editor is syntax coloring. This could be relatively easily achieved in tcl/tk, because the text entry widget supports "tags" which allow to change the style of a certain part of the text very easily, and also "marks" which enable the programmer to keep track of certain parts of the text, independently of absolute locations.

The basic idea

Of course the algorithm for the easy recognition of the portions of the text that need to be highlighted still needs to be written. I use the following approach: Every keystroke is ignored, except for the few special keystrokes which either delete a character or add one of the syntactically important ones, being "<", ">", "&", ";" and the double quote itself. For other texts than HTML, this could of course be extended. For HTML entities between "&" and ";", the action to be taken is quite simple: if there is a closing ";" within the current line, color everything that forms the HTML entity, otherwise use a warning color indicating a syntax error. The same goes for HTML tags, even though they can span multiple lines. However, for clarity and simplicity, this is not allowed in the template editor.

It soon turned out that just having some special code for each one of the delimiters of a symbol may be fast, but would soon lead to rather messy code. This is mainly because the deletion of such a character can yield to many possible states. E. g. the deletion of a "&" sign can be in one of the following situations (the underlined "&" gets deleted):

`&te&xt;` The second "&" was a typo, and the syntax coloring has to be extended to the previous "&".

`&text;` The text used to be a HTML entity but is now normal plain text; the syntax coloring has to be removed.

`&text` The text was always only plain text.

As one can see, this easily leads to many special cases of which the treatment becomes too complicated and is not worth the minimal gain of speed in the normal case (insertion of a character).

The implementation

Therefore, the whole line has to be parsed when a character changes. This can be speeded up by only doing that when one of the delimiters of a syntactic entity is added or deleted. For HTML, these symbols are "<", ">", "&", ";", "!" (for comments) and the double quote. If such a symbol is typed, the whole line is parsed and colored.

Deletion of a character is still a special case, because two lines may be joined. However, this was not much of a problem. For similar reasons, [Return] has also to be treated specially, because it separates two lines.

In the implementation, the default handler is carried out after the supplied custom handler. This means that if the user types a character, the custom handler is carried out before that character is actually inserted into the buffer! This is of course not acceptable for an editor with syntax coloring, since the true text is always known one keystroke too late. Fortunately, the order in which events are processed can be changed by the `bindtags` command. This allows to override the ordering in which the handlers are carried out.

However, if the default handler is carried out, we run into a problem when a character is deleted. Because the character is removed from the buffer before our handler is invoked, we can no longer find out which character was actually deleted. Therefore the only solution is to have our handler executed first and then execute the default handler. If our handler is called, we have to insert the typed key ourselves, in order to have an updated buffer. Of course we have to suppress the default handler in that case, since this would result in having every character inserted twice. The current implementation overrides the default handler for all special characters (the delimiter symbols, backspace, delete and return) and does not allow the default handler to run in these cases. For all normal characters, the default handler does the job of inserting the character.

The coloring itself

The procedure for coloring a line of text is not too complicated. A regular expression search shows us the first occurrence of a delimiting character, in which case we look for the character that ends the entity. In the case of an HTML tag, whenever we see a `"<"`, we look for the closing `">"` and color all the text between the two angular brackets (and the brackets itself) with the appropriate color. This can be easily achieved by using Tk "tags" for changing the appearance of text (see Section 4.5.1). We have to take care to delete all existing tags in a line first, before re-tagging that line.

Because different syntactical entities may occur within each other (e. g. a quoted string within an HTML tag), the procedure is recursive. Usually it is not allowed to have a different kind of entity within another one (e. g. a quoted string within an HTML entity), in which case we indicate this syntax error with a special color.

Due to the simple pattern matching used, the syntax coloring is not perfect and cannot detect all mistakes. E. g. if someone types

```
<body color="#000000" bgcolor="#FFFFFF"> >,
```

the second closing bracket is clearly a mistake, but since the program has already found one, it just treats that second bracket as plain text. Indeed most editors do this, and so does Netscape, so there is no need for a more complicated syntax coloring.

4.5.2 The actual editor

Other than the replacement of the simple entry widget by a more sophisticated text widget with syntax coloring, there are few differences between the template and the variable editor. The possibility to insert colors or filenames directly into the text has been omitted, because this should be done in the variable editor. The idea is that the user defines a variable for each color used, in order to be able to change that color more easily later on. This variable can then be used in several places in the template.

Of course the file format is slightly different, which leads to a different parsing method for loading the files. Namely the line spanning tags and C comments were tricky to handle. Because Tcl's regular expressions are not quite as powerful as Perl's, only C comments which go from the beginning to the end

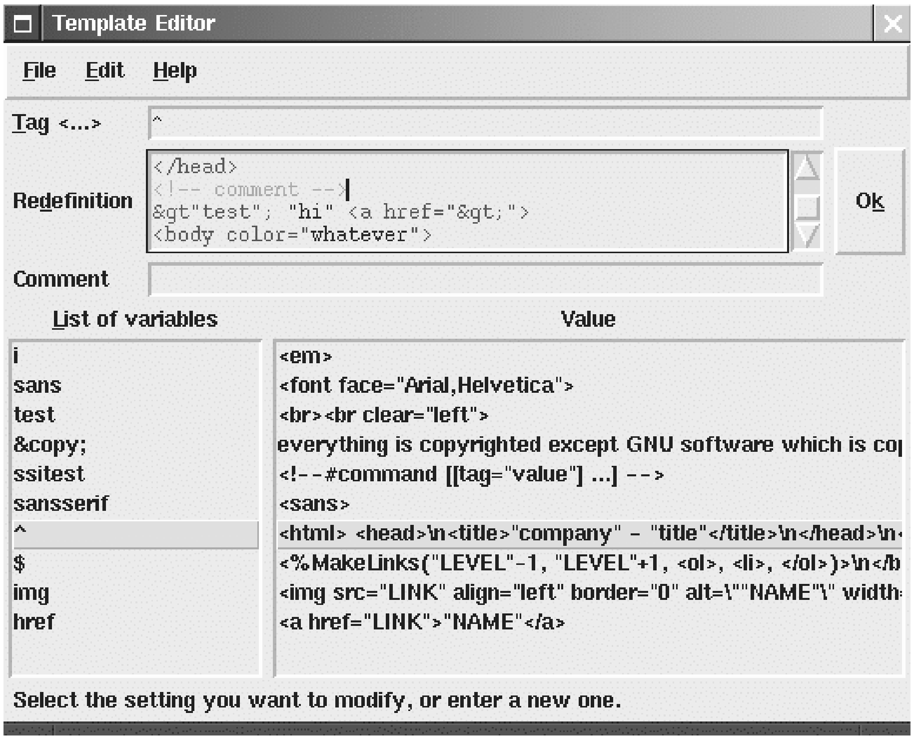

Figure 4.5: Template editor

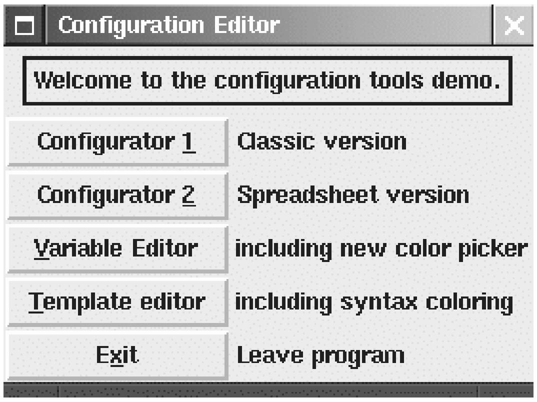

Figure 4.6: Demo program

of a line are allowed. Since C comments are only used by people who edit the file directly in ASCII mode, this should not be a problem. The template editor also inserts all long comments at the beginning of a file when saving it, thus having the same minor restriction as the variable editor.

4.6 The demo program

In order to allow easier testing by as many people as possible, a small demo program is provided that launches all the other programs so the user does not have to remember which `.tcl` files to use.

4.7 Testing

I first tried to find some testers via freshmeat `http://freshmeat.net/` (a page that announces many newly released software, mainly Open Source software for Linux), and there were indeed many people who downloaded it during the time when it was visible on the front page (about 40 people per hour during about five hours), which is quite surprising, because I did not release the back end yet, so the program did not actually do anything useful at that stage :-)

However, I did not get any feedback either. Probably no one cared that much about a simple Tcl/Tk user interface in order to help me to improve it. Therefore the next release would include the back end and all required documentation. And indeed, I got a little feedback, which proved quite useful - mainly for improving the documentation.

Chapter 5

Example and test pages

5.1 Using the programs

All programs still require the configuration file to be in either the directory where the program is started from or the `$HOME/.htmlplain` directory, and the configuration file should be writable (so the programs can change it). For more information about the installation, check the `README` file supplied with the archive.

5.1.1 Installation

There are two ways how to install the program: either manually or using the installation tool. The manual installation is very easy under Linux, if the default paths are used; otherwise, using the installation program is recommended.

Manual installation

First, the administrator has to untar the archive to `/usr/local/`. Then, each user has to copy the three files `config.pl`, `template.html` and `variables.txt` to the document root. While a Makefile can do the first part (with `make install`), which is the task of an administrator, the second part (copying the configuration file) has to be done by the user. A Makefile could check the user id and do that as well. The third step is to run the configuration tool and set up all the paths.

Using the installer

The installation program, `install.tcl`, has been written for users who want to install the program files under a different location that `/usr/local/`. Before that tool was written, the paths needed to be changed manually (since there was no compilation step), which became quite tedious with the frequent releases and additions to the program.

The installation program asks for the place where the program should be installed, and copies all files there. It also creates all directories, if needed, and searches through all programs in order to set the pathnames correctly. The first step of the installation can now be done quite conveniently.

In the second step, the configuration files are copied to the home directory of the person who installs the program. If this is the administrator, he then has to copy these files to the home directories of the users who want to use the program, or make them world readable. If a `HOME` environment variable exists, the path to the configuration file will not be hard coded, but dependent on the user's home directory, even if it appears not to be during the installation (because the `HOME` variable is inserted in the path).

The third step (running the configuration tool to set the directories for the HTML files) still has to be done explicitly. When writing the installation tool, it was the goal to make the first step easy; the rest is quite simple anyway.

5.1.2 The Tcl/Tk programs

The use of the Tcl/Tk programs is optional; they only provide a user friendly front end but are not concerned with the back end that actually parses the documents. The Tcl/Tk programs require Tcl/Tk 8.0 or newer. There is a main program called `"main.tcl"` which is a very simple "launch pad" for the other programs. It is not required to run the other programs; they can also be run as stand alone applications (if the permissions are set correctly).

Program	Function
`wish config.tcl`	Configurator. An alternative version is `config2.tcl.`
`wish variables.tcl`	Variable editor, needs configuration prior to usage.
`wish install.tcl`	Installation program.
`wish template.tcl`	Template editor, needs configuration prior to usage.

Under Windows, some obscure configuration has to be made in order to make the Tcl files launch the interpreter, since the "She-bang" notation `#!/usr/bin/wish` is not recognized. Under Unix, usually just using `"config.tcl"` works (if the path to the Tcl/Tk interpreter is correct).

5.1.3 The Perl programs

Only two Perl programs need to be run (preferably with a correct path in their first line): `plainfiles` and `htmlplain.` Again, they require `config.pl` (the file containing the configuration) to be set correctly. The program `plainfiles` has to be run first, and every time when new pages are created. `Htmlplain` has to be run in order to obtain a preview for all compiled pages, and of course before uploading the documents. It is recommended to set up the configuration first, then design the template, and then go on with the HTML documents.

A third program, `converter,` is optional, and allows existing pages to use some of HTML PLAIN's features.

5.2 An example configuration

See Section 2.2.5 on page 13 for an example template. The full (long) listing of all files is shown at Section A.1 on page 69. The example template that comes with the program almost identical. The example configuration looks like this (only the most important entries are shown):

```
$variableNames = 'title';             # 'Magic' tags (without angular brackets)
that can set variables
$slash = '/';                         # Directory/file separator (/ on Unices)
$root = '/home/foo/pages/';           # Path of documents to be parsed
$outputPath = '/home/foo/upload/';    # Path for files to be uploaded
$killTags = 'address';                # Remove everything here between <tag> and </tag>
$macros = 'SiteMap, MakeLinks, DumpArgs, LinkTable, Pictures, Back, LinkTo';
# Names of trusted macros
$searchPat = '(Last modified.*)|(<!-- hhmts.*-->)'; # These entries will be searched for ...
$replacePat = '';                     # And replaced by this (one line only)
$fastHeader = '0';                    # 1 = do not try to override header lines by document
$fastFooter = '0';                    # 1 = do not try to override footer lines by document
$glueHeader = 'body';                 # "glue" rest of header lines to this one
$glueFooter = '/body';                # "glue" rest of footer lines to this one
$startPat = '<title>';                # Pattern that occurs before document title
$endPat = '<\/title>';                # Pattern after document title, "\/" = "/"
$sizeDiv = '1024';                    # Divisor for file size - should be 1 or 1024
$sizeSuffix = 'KB';                   # Suffix for file size - should be none or KB
```

In case the short explanations are not clear enough, here are more details about some configuration entries:

Variable names

These are the "magic tags" of which the text between the opening and closing tag will be stored in a variable of the same name. Now this sounds even more complicated than the short version :-) It means that if you have a tag like `<title>Text</title>`, it will set the variable "title" to "Text". This will only occur if the tag has been specified beforehand, and the closing tag should come immediately after the tag (i. e. there should be no more tags in between such as in `<table>`).

Root, output path

These should be absolute directory names. The tilde is *not* expanded, and relative links do not always work. Don't worry, the web page that will come out will have relative links, not absolute ones.

Search/replace pattern

Sometimes it may be useful to have a search/replace function on the input text before it is processed. Regular expressions in Perl syntax are supported, but references to matching text are not (it never worked when I tried it). Usually this will be used to delete some text.

Fast header/footer

Usually a very complex algorithm is applied in order to combine the document with the template header. If the document contains no meta tags and is written from scratch (i. e. all markup language is in the template), then this option can speed up processing the documents.

Glue header/footer

For some tags (usually all tags after `<body>`), it is not possible to tell whether they are still part of the header, or whether they are part of the document itself. This goes for any "generic" tag such as `<br>, <p>, <img>` etc. In order to prevent the preprocessor from overwriting these, a fixed point has to be known, from which point on no line of the document will overlap with the header. Unless the template uses more unusual tags such as `<blockquote>`, which may also appear (redundantly) in the documents, this option should not be changed. The reason for the complex algorithm is to allow pages to have meta tags (for search engines) which cannot be set by templates, and for being able to include automatically generated documents such as `latex2html` output. And behold, this actually works pretty well!

Start/end pattern

These patterns determine where the symbolic name of the document will be found. If you do not want it to appear in the document, use your own comment tags for it, such as `<!-- name -->` and `<!-- endname -->`. I found it convenient to use the `<title>` tag because XEmacs always includes it in empty HTML pages, and you just have to enter the title in order to fill it in. If you want to use a different title later on, it is recommended to include the comment line

```
<!-- <title>old title</title> -->
```

with the old title tag inside before the actual title. This will "fool" the file table generator and make it use that fake title tag, which means that no links will be broken.

Size divisor/suffix

This allows it to include size information. Some lame editor includes the information up to the byte, which is not only hard to read but usually irrelevant. In this program, you can change the unit in which it appears. Set the divisor to '1' and the suffix to nothing or 'bytes' if you want to have the exact file sizes, otherwise use the example value.

5.2.1 Command line arguments

Command line arguments can be used to override settings from the configuration file. They always have exactly the same name like the entry in the configuration file and are preceded by `--`. For instance, in order to override the "root" setting, use

```
htmlplain --root=[new document root]
```

Some parameters are flags, which can be set by simply specifying `--flag_name` (e. g. `--verbose`). There is no way yet how to clear flags which are set in the configuration file - that file has to be edited in that case.

Other command line arguments

Besides all options from the configuration file, there is also the option `-o`, which is equivalent to `--outputPath`. Another option not occurring in the configuration file is `--all`. Setting this flag disables any dependency checking and forces the preprocessor to generate each document newly, even if it has not been updated since the last run.

Parameters without switches are interpreted as input files. Unless a file matches one of the pattern(s) given, it is not processed. This allows the use of Makefiles to check dependencies. Note that the `--all` option still needs to be specified if the built-in dependency checking should be disabled.

5.3 An example page

The example page uses the configuration from above (Section 5.2). The template is rather simple, in order to show the most outstanding features:

The template

```
hr      <br>        # Get rid of these ugly <hr> tags :)
stylish <font face="Geneva,Helvetica,Arial"> # redefine tags however you like
^       <html>
        <head>
        <title>HTML PLAIN demo - "title"</title>
        </head>
        <BODY text="foreground" bgcolor="background">  # header
$       <br>
        <%Back("Return to ")>
        <!-- The link back is made by a macro -->
        <br>
        </BODY>
        </HTML>  # footer
img     <img src="LINK" border="0" alt=\""NAME"\" width="WIDTH" height="HEIGHT">
# default look for images
href    <a href="LINK">"NAME"</a>    # default look for links
```

The first line redefines the ugly `<hr>` tag as a line break. The second line defines a fancy font. The third line is more interesting, as it is a special header tag (also see Section 2.2.3 on page 12). Up to and including the `<body>` tag, the parser will merge the header from the template with the header from the document. (It does this up to the tag specified under "glueHeader" in the configuration). Therefore, even `<!DOCTYPE...>` and meta tags will work, if inserted in the actual document. They will appear at the right position in the output.

The next tag, `$`, is the footer tag. It works like the header tag. The `<img>` tag also has a special importance, because it defines the appearance of all images which do not use tag options to override the template settings. Therefore, the default options should be used sparingly. Note how the variables in capital letters refer to special properties of the image (also see Section 2.3.2 on page 16). The last tag, finally, defines the appearance of links, if only a blank URL or variable name is used. The program automatically recognizes these as a link and inserts the link tag.

The variables

```
# variable settings for test documents
background "#cccccc" # use variables for colors
foreground "#000000"
mine "#000099" # A different color that overrides the template setting
bright "#ffffff" # another color variable
year 1999 # could be derived via localtime
Webtools main page http://artho.com/webtools/ # use filenames and URLs
```

The variables are mainly used for colors here, but they also show the use of referencing external URLs. Since you never know whether they change, you had better define a symbolic name. The tab characters are not properly reproduced here; they will appear differently in most editors (such as vi).

The example pages

There are a few example pages that come with the current release. Both pages are using the XEmacs standard template, which is partially overridden in order to get a fully customizable page. They show the most important features:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head>
    <meta name="example" content="Meta tags are correctly embedded in doc-
uments">
    <title>HTML PLAIN demo</title>
  </head>

  <body>
    <h1>HTML PLAIN demo</h1>
    Welcome to this demo page! You can go to
    <a href="Another example page">a page showing more features.</a>
    <br>
    Also check the
    "Webtools main page"
    for the latest version of
    <stylish>HTML PLAIN.</stylish>

    <hr>
    <address><a href="mailto:cartho@netlink.ch">Cyrille Artho</a></address>
<!-- Created: Tue Mar 16 22:05:12 GMT 1999 -->
<!-- hhmts start -->
Last modified: Tue May 25 20:12:27 BST 1999
<!-- hhmts end -->
  </body>
</html>
```

With the example configuration, most of the bottom part is filtered out. The output looks like this:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head>
    <meta name="example" content="Meta tags are correctly embedded in doc-
uments">
<title>HTML PLAIN demo - HTML PLAIN demo</title>
<mymeta>
  </head>
  <BODY text="#000000" bgcolor="#cccccc">
```

```
    <h1>HTML PLAIN demo</h1>
    Welcome to this demo page! You can go to
    <a href="example.html">a page showing more features.</a>
    <br>
    Also check the
    <a href="http://artho.com/webtools/">Webtools main page</a>
    for the latest version of
    <b><font face="Geneva,Helvetica,Arial">HTML PLAIN.</font></b>
    <br>
    <!-- Created: Tue Mar 16 22:05:12 GMT 1999 -->
<br>
<!-- The link back is made by a macro -->
<br>
  </BODY>
</HTML>
```

The second demo page exploits some other features:

1. The `<body>` tag overrides the setting of the template
2. "arrow" will be replaced by the image "arrow.gif"
3. The `<%Back()>` macro from the template is evaluated and replaced by a link back.

```
  <body bgcolor="bright" text="mine">
    <h1>Another example page</h1>

    These two pages show you
    <ul>
      <li>How to use templates, especially headers and footers
      <li>How to redefine tags, and override template settings
            (e. g. background)
      <li>How to link to other pages independently of their location
    </ul>
<br>
  <table border=0>
    <tr>
      <td>"arrow"</td>
      <td>Many other features can be exploited - check them out!
    </tr>
  </table>
    <hr>
    <address>
      <a href="mailto:cartho@netlink.ch">Cyrille Artho</a>
    </address>
<!-- Created: Tue Mar 16 22:09:37 GMT 1999 -->
<!-- hhmts start -->
Last modified: Tue Mar 16 23:27:36 GMT 1999
<!-- hhmts end -->
  </body>
</html>
```

The output is a page with different colors than the previous one. The link back to the previous page appears at the correct place, and the image also has been replaced. These two pages show all important features of HTML PLAIN.

5.4 Writing the pages

The most important step is writing the template. It should contain as much as possible of "recurring themes" throughout all pages. The header and footer might be as simple as a "complete"[1] set of tags up to <body> (or from </body> on, respectively), or they can contain a navigation bar, which is possibly generated by tables. In that case, the table tags that are opened in the header should be closed in the footer. Like this, a complex but uniform layout can be maintained throughout all pages.

The task of writing the web pages is now much simpler. You do not have to bother about recurring content (such as navigation bars) in every page once you have written a nice template. In the page itself, just use the actual text that goes into each document. There is no need to repeat all the header tags. If you want to include meta tags, though, you have to specify the <head> and </head> tags, too, in order to tell the preprocessor where to put the meta tags. You should also specify the <title> tags, as the are important. Other than that, you can use as little markup as line breaks, like in this example document:

```
<title>This is an example</title>
<h1>Example</h1>
This document is for use in HTML PLAIN.
<br>
As you can see, the amount of HTML code
that needs to be written is very small.
```

There is no need for <html>, <body> or other tags! Indeed these few lines can generate a much larger document with a given template. It is advised *not* to use tags other than <head>, <title> and <meta> anymore before the part where the actual document body (text) starts. The "glueHeader" option should be set to <body> in order to insure a correct insertion of the following tags. Following this scheme prevents having tags occurring twice (in the template and the document).

5.5 Including `latex2html` output

I found the program `latex2html` quite useful, because it allows to have a fairly good representation of a postscript file in HTML form[2]. In order to use its output in HTML PLAIN, a few intermediate steps are needed. First, it is required to rename node?.html to node 0?.html in order to get right order of all files in the site map etc.

```
ls -1 | grep 'node.\.html$' |
  sed -e 's/\(.*\)\(.\)\.html/mv \1\2.html \10\2.html/'
  | sh
```

is the wonderful shell command that does this job. Second, the hierarchy cannot get fully recognized since all output files have the same name (except for their number), regardless of their content level. Therefore that part of the site map will be flattened to a list.

The last step is to fix all the links inside the converted documents, because these still refer to `node1.html` to `node9.html`. This is done with the

[1] meaning all tags commonly found there: <html>, <head>, <title>, <body> except <meta> tags.

[2] except for this document, where it shows quite a few bugs I have never seen before.

short Perl script `latex2htmlinc.` In fact, the Perl script also does the file renaming. So you just have to run `latex2htmlinc` in the directory where the output files of `latex2html` are. That script accepts the command line switch `-imgbug`, which makes the program change all links to `img[n].gif` to `img[n+1].gif`. This is to work around a bug that I encountered with the `latex2html` output (before any other program is run); however, that bug may not always occur.

It is important that `latex2htmlinc` is only run once, as it will corrupt the files if run a second time. It also has to be run every time after `latex2html` was called.

Because the output of `latex2html` is not "clean" HTML, the preprocessor will complain *a lot* once the output from `latex2htmlinc` is included in a page processed with `htmlplain`. Fortunately, the HTML output of the first converter is still readable by browsers, and therefore all output messages are only warnings, no errors.

5.6 Uploading

It is intended that the pages from the "output path" are uploaded. In order to be able to upload the images as well, the program automatically creates symbolic links for each image found in the source tree. Because the program cannot follow symbolic links to directories, all images, binaries and HTML sources should be in the source tree. The file table program will create the necessary symbolic links in the output path for both images and binaries. On file systems with no symbolic links, a warning is issued; however, there is no alternative for symbolic links (of course the file could be copied, what a waste of space), so it is recommended that the file system is upgraded to a Unix-like file system that supports symbolic links.

5.7 Converting existing HTML pages

It is well worth using HTML PLAIN for existing pages, mainly for the sake of file management. While some aspects (such as templates) cannot be included anymore without some manual work, it is at least possible to replace all the relative links and variable contents with the symbolic names found in the HTMLPLAIN file table (see Section 2.4.1) and variable dictionary (see Section 2.3).

The converter takes the same configuration file as `htmlplain` does, but it does not need all the options. In particular, the "outputPath" option is crucial, since the resulting files will be stored there. A second configuration file should then be used later on with `htmlplain`, where the former "outputPath" is the new "root" path. It is not recommended to work with the old documents after a page has been converted for HTML PLAIN, but it is of course possible.

5.8 Managing multiple sites

The macro file is installed is the program directory (`/usr/local/htmlplain` by default). Therefore, if several users wish to create their own macros, they need write access to that file. In a later version, it may be possible to have a local copy of the macro file. Because the macro file is the most security sensitive part of the preprocessor, I thought it is better to install it such that only the superuser can modify it.

The configuration file is user specific, by default in the home directory. This is also where the variable and template files should commonly be stored. However, a problem occurs when several separate pages for different sites should be made for one user.

In that case, the user should take advantage of the fact that "local" configuration files (within the directory where the program was started from) are "preferred" to "global" (in the user's home directory) ones. Therefore, for multiple pages, the user simply has to copy all configuration files into the document root for each site, and modify them accordingly. The parser should then be started from the document root of each site.

Chapter 6

Summary and conclusions

6.1 Achievements

6.1.1 Custom tags, entities and variables

This was the easiest part and implemented first. As soon as the HTML parser was completed, every syntactic entity could simply be checked for occurring in the dictionary, and be replaced if it did.

This feature allows for a better structuring of the document, and especially the header and footer functionality can be tremendously useful, if used well.

6.1.2 Linking

This task involved setting up a table for all files, which was fairly easy to implement. Unfortunately, the filenames could not just be inserted by a simple lookup. Instead, many special variables (see Section 2.3.2) have to be set, and the path has to be adjusted (i. e. a relative path has to be derived from the absolute path).

Taking away all file management problems is a big relief for the web master, and greatly reduces the chance of broken links.

6.1.3 Macros

The part in the main parser that runs the macros is fairly simple. The macros themselves were not always easy to write, as there are many possible cases that have to be considered, and therefore the code for generating a set of links is sometimes a bit more complicated than anticipated.

It is not sure whether users will ever write their own macros; the supplied (though small) set of macros covers most of the functionality that a web page uses.

6.1.4 Performance

Because Perl is very good at handling strings, the preprocessor is quite fast. If only one or two documents have to be updated, the main time is taken by compiling the Perl scripts before executing them. Otherwise, macros using site maps or iterating through the whole file table account for the largest amount of time. This cannot be optimized by using C as the implementation language for the parser, because the macros will still be in Perl.

A fast CPU makes a big difference, because the I/O overhead is relatively low, especially after running `htmlplain` once (and all files are cached). The running time of the file table generator depends somewhat on I/O performance, but the speed of the main parser does not.

I found the performance satisfactory for myself, on a PC that is getting too slow for modern games. I have also tried the preprocessor on a new PC, where it is indeed quite fast (especially if no macros are used).

6.2 Remaining problems

6.2.1 Header and footer tags

The insertion of header and footer tags into the document was a very hard task. It is not always possible to know where to insert a tag, but the program usually makes correct guesses. In some rare cases, the insertion still does not work correctly, but this problem could usually be worked around by using a "glueHeader" setting.

6.2.2 No tutorial for using HTMLPLAIN

Section 5.3 covers most of the important functionality and should be sufficient for more experienced web page designers. A more detailed "step by step" tutorial would of course be preferable, but this could not (yet) be achieved due to time constraints.

6.2.3 Configuration tool weaknesses

The configuration tools are not perfect; in particular, the two list boxes do not work together properly, and the "spreadsheet-like" representation is not quite like a table either. This is mainly due to Tcl/Tk limitations; with a more advanced toolkit like gtk+ [gtk], the look and feel would be much better.

The main weakness is the fact that every configuration entry is presented in a list, as opposed to a hierarchical tree. This is because the configuration file does not include any metadata about itself, so a tree cannot be generated. Future versions may include this, yielding a much better (hierarchical) representation of the configuration options, like in Netscape Communicator. In Tcl/Tk, it is not possible to show such a tree (unless someone writes a new widget from scratch for it), so that newer version would definitely have to be written using the gtk.

Another flaw is that each configuration entry is treated as a simple string, regardless whether it is a flag, numerical value, search pattern or file path. This could be improved as well.

Since the emphasis is on the preprocessor and not on these auxiliary utilities, these features have not been implemented.

6.3 Future enhancements

Of course there are always ideas left about what could still be implemented. With the deadline for the 1.0 release coming closer, some features had to be left out. They will be implemented later, if either the author needs these features or users ask for them.

6.3.1 Manual entries in the file table

So far, the file table is generated fully automatically, with no chance of manual corrections (they should not be needed under normal circumstances). However, there are some cases where manual changes are desired. Currently, this can be achieved by editing a second file and appending it each time after `plain-files` has been called. Later versions of that program may support more elegant ways of achieving this.

Aliases

It may be helpful to have several names for one file. Currently, this can only be achieved by manually appending entries to the file table (see above).

ASCII text files

These files can only be referred to as binaries. It would be nice to have the program extract the first text line as title and allow using ASCII files like HTML files (including having them appear in a site map etc.)

External files or binaries

Linking to files on a different server (or just a different virtual server) has to be done through entries in the variable dictionary. In the case when the files are also present on the file system where the HTML documents are found, this task could be automated.

6.3.2 Multiple templates

Usually the main page looks different than all the other pages. It would be preferable to have different templates for different levels of content (see Section 2.2.6). Indeed this will not be hard to implement, as documents are now processed ordered by ascending content level. This will allow for a fairly easy extension; each time when a higher document level is reached, a new template is loaded, overriding options of the old one.

6.3.3 CGI extension

Sometimes the value of a variable is not known beforehand. In that case, a cut down version of the parser is needed, which works as a CGI program and fills in the variables that were not known at "compile time". This is not very hard to implement, as the more complicated part has been already done.

6.3.4 Supporting non-Unix operating systems

Because there is no standard path where applications are installed in non-Unix operating systems, writing Mac (or even Windows) versions involves quite a lot of extra work. With the installation program (see Section 5.1.1), running the program under a non-Unix platform should now be much easier. The lack of possibilities for testing Windows and Mac versions prevents a fast progress, though.

Appendix A

Test results

A.1 Example page

This example is discussed at Section 5.2 on page 56. This appendix lists all files, which are also included for reference in every distribution of HTML PLAIN.

A.1.1 Setup

Full configuration file

```
#!/usr/bin/perl
##############################################################
# Configurations for Perl files
# written in 1998/99 by Cyrille Artho, published under the GPL

# "all"
$MAX_LENGTH = '32767';                # Maximum line length
$verbose = '0';                       # verbosity level (0 or 1)
$variableNames = 'title';             # 'Magic' tags (without angular brackets) that can set variables

# parser.pl
$templateFile = 'template.html';      # Template file name
$variableFile = 'variables.txt';      # Variable file name
$slash = '/';                         # Directory/file separator (/ on Unices)
$root = '/home/cartho/strath/project/example'; # Path of documents to be parsed
$outputPath = '/home/cartho/strath/project/output'; # Path for files to be uploaded
$killTags = 'address';                # Remove everything here between <tag> and </tag>
$macros = 'SiteMap, MakeLinks, DumpArgs, LinkTable, Pictures, Back, LinkTo, Date, Test'; # Names of trusted macros
$searchPat = '(Last modified.*)|(<!-- hhmts.*-->)'; # These entries will be searched for ...
$replacePat = '';                     # And replaced by this (one line only)
$fastHeader = '0';                    # 1 = do not try to override header lines by document
$fastFooter = '0';                    # 1 = do not try to override footer lines by document
$glueHeader = 'body';                 # "glue" rest of header lines to this one
$glueFooter = '/body';                # "glue" rest of footer lines to this one
$quoteVars = '1';                     # put quotes around special variables (e. g. filename)

# filetable.pl
$filter = 'html$';                    # Filter for documents: html$ = html, phtml or shtml, but not *html~
$imageFilter = '\.(gif|jpe?g|png)$';  # Filter for image files
$binaries = '\.(tar|gz|ps|rpm|zip|css)$'; # filter for binaries (e. g. .tar.gz)
$exclude = '(\.(pl|cgi)$)|(template\.html)'; # Exclude these documents from parsing
$hide = '(include.html)';             # Pages that do not appear on site maps etc.
$imageIdentifier = 'identify';        # Image identifier equal to UNIX identify
$startPat = '<title>';                # Pattern that occurs before document title
$endPat = '<\/title>';                # Pattern after document title, "\/" = "/"
$fileTable = 'filetable.txt';         # File where to store file table
$sizeDiv = '1024';                    # Divisor for file size - should be 1 or 1024
$sizeSuffix = 'KB';                   # Suffix for file size - should be none or KB

# parsing.pl
# URLify
$urls = '(http|telnet|gopher|file|wais|ftp)'; # Set of valid URL protocols
$punc = '.,;:!?\-';                   # Punctuation characters
$any = '\w/#~:.?+=&%@!\-';            # Any character, \w = letters and numbers
1;
```

Input directory

See Table A.1.

File name	Purpose
`arrow.gif`	Picture with arrow
`config.pl`	Configuration file
`example.html`	Example page
`filetable.txt`	File table
`index.html`	First example
`skip.html`	A page that is not parsed
`template.html`	Template for all pages
`template_2.html`	Template for `*.data` pages
`test1.data`	First set of demo data
`test2.data`	Second set of demo data
`variables.txt`	Variable file

Table A.1: Input directory of example page

File table

The `.data` entries are quite long and do not fit on one line.

```
HTML PLAIN demo /home/cartho/strath/project/example/index.html 0 KB 0
Another example page /home/cartho/strath/project/example/example.html 0 KB 1
This page is not parsed. /home/cartho/strath/project/example/skip.html 0 KB 1
Test1 data /home/cartho/strath/project/example/test1.html 0 KB 1 /home/cartho/strath/project/example/test1.data
/home/cartho/strath/project/example/template_2.html 1
Test2 data /home/cartho/strath/project/example/test2.html 0 KB 1 /home/cartho/strath/project/example/test2.data
/home/cartho/strath/project/example/template_2.html 2
arrow /home/cartho/strath/project/example/arrow.gif 1 KB 32x32
```

Template

Because this file is quite small, it has also been included in full length in Section 5.3.

```
hr      <br>        # Get rid of these ugly <hr> tags :)
stylish <font face="Geneva,Helvetica,Arial"> # redefine tags however you like
^       <html>
        <head>
        <title>HTML PLAIN demo - "title"</title>
        </head>
        <BODY text="foreground" bgcolor="background">  # header
$       <br>
        <%Back("Return to ")>
        <!-- The link back is made by a macro -->
        <br>
        </BODY>
        </HTML>  # footer
img     <img src="LINK" border="0" alt=\""NAME"\" width="WIDTH" height="HEIGHT">
# default look for images
href    <a href="LINK">"NAME"</a>    # default look for links
```

Variable file

Because this file is quite small, it has also been included in full length in Section 5.3. The tab characters are not properly reproduced here; they will appear differently in most editors (such as vi).

```
# variable settings for test documents
background "#cccccc" # use variables for colors
```

```
foreground "#000000"
mine "#000099" # A different color that overrides the template setting
bright "#ffffff" # another color variable
year 1999 # could be derived via localtime
Webtools main page http://artho.com/webtools/ # use filenames and URLs
```

A.1.2 HTML documents

index.html

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head>
    <meta name="example" content="Meta tags are correctly embedded in doc-
uments">
    <title>HTML PLAIN demo</title>
  </head>

  <body>
    <h1>HTML PLAIN demo</h1>
    Welcome to this demo page! You can go to
    <a href="Another example page">a page showing more features.</a>
    <br>
    Also check the
    "Webtools main page"
    for the latest version of
    <stylish>HTML PLAIN.</stylish>

    <hr>
    <address><a href="mailto:cartho@netlink.ch">Cyrille Artho</a></address>
<!-- Created: Tue Mar 16 22:05:12 GMT 1999 -->
<!-- hhmts start -->
Last modified: Tue May 25 20:12:27 BST 1999
<!-- hhmts end -->
  </body>
</html>
```

example.html

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head>
    <title>Another example page</title>
  </head>

  <body bgcolor="bright" text="mine">
    <h1>Another example page</h1>

    These two pages show you
    <ul>
      <li>How to use templates, especially headers and footers
      <li>How to redefine tags, and override template settings
            (e. g. background)
      <li>How to link to other pages independently of their location
    </ul>
<br>
    <!-- include "include.html" -->
    <!--#include file="Include Page" -->
    <!--#include file="open_hours.html"-->

    <!-- if ("year" < 2000) -->
```

```
    Are you looking forward to the new Millennium (bug)?
    <!-- else -->
    We have the year <%(1900+"year" % 100)> - computers do not exist yet.
    Everything you see is just a product of your imagination...
    <!-- endif -->
    <table border=0>
      <tr>
<td>
  "arrow"
</td>
<td>Many other features can be exploited - check them out!
      </tr>
    </table>
    <b>Macros</b> are also very useful and let you easily generate
    nav bars and site maps:
    <br>
    <%SiteMap('<ul>','<li>','</ul>')>
    <br>
    The other pages show you some more advanced features.
    <br>
    <stylish>Some &quot;long quotation
      that goes over multiple lines&quot;</stylish>
    <br>
    A last example: environment variables, e. g. $USER = "USER".
    <hr>
    <address>
      <a href="mailto:cartho@netlink.ch">Cyrille Artho</a>
    </address>
<!-- Created: Tue Mar 16 22:09:37 GMT 1999 -->
<!-- hhmts start -->
Last modified: Mon Jul 12 00:22:28 CEST 1999
<!-- hhmts end -->
  </body>
</html>
```

skip.html

This page is simply copied to the destination directory and therefore of little interest.

template_2.html

This document generates `test1.html` and `test2.html`, using `<!-- foreach -->` (see Section 2.8.2 on page 25).

```
<!-- foreach test*.data count -->
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head>
    <title>"title"</title>
  </head>

  <body>
    <h1>"title"</h1>

    Another kind of template; this document has the line
    <tt>&lt;!-- foreach test*.data count--&gt;</tt>
    as its first line, which means that each file test*.data will be parsed,
    its values will be used (like in any variable file), and the output files
    will be test*.html.
```

File name	Remark
`arrow.gif`	Symbolic link to picture
`example.html`	Example page (overrides body tag)
`index.html`	Uses custom defined tag
`skip.html`	A page that is not parsed
`test1.html`	First output page of `template_2.html`
`test2.html`	Second output page of `template_2.html`

Table A.2: Output directory of example page

```
    The value of * can be used in &quot;count&quot; = "count".
    <br>
    This particular file only uses &quot;VarA&quot; = "VarA" and
    &quot;VarB&quot; = "VarB".
```

test1.data

```
VarA Variable A-1
VarB Variable B-1
title Test1 data
```

test2.data

```
VarA Variable A-2
VarB Variable B-2
title Test2 data
```

A.1.3 Output

Some of the changes are more subtle, and not visible on the first look. At the second look, it should become obvious that this preprocessor has many uses, some of which have not even been thought of by the author :-)

Output directory

See Table A.2.

index.html

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head>
    <meta name="example" content="Meta tags are correctly embedded in doc-
uments">
<title>HTML PLAIN demo - HTML PLAIN demo</title>
<mymeta>
  </head>
  <BODY text="#000000" bgcolor="#cccccc">
    <h1>HTML PLAIN demo</h1>
    Welcome to this demo page! You can go to
    <a href="example.html">a page showing more features.</a>
    <br>
    Also check the
    <a href="http://artho.com/webtools/">Webtools main page</a>
    for the latest version of
```

```
    <b><font face="Geneva,Helvetica,Arial">HTML PLAIN.</font></b>
    <br>
    <!-- Created: Tue Mar 16 22:05:12 GMT 1999 -->
<br>
<!-- The link back is made by a macro -->
<br>
  </BODY>
</HTML>
```

example.html

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head>
<title>HTML PLAIN demo - Another example page</title>
<mymeta>
  </head>
  <BODY bgcolor="#ffffff" text="#000099">
    <h1>Another example page</h1>
    These two pages show you
    <ul>
      <li>How to use templates, especially headers and footers
      <li>How to redefine tags, and override template settings
            (e. g. background)
      <li>How to link to other pages independently of their location
    </ul>
<br>

Testing includes...

Testing includes...
<!--#include file="open_hours.html"-->
    Are you looking forward to the new Millennium (bug)?
    <table border=0>
      <tr>
<td>
  <img src="arrow.gif" align="left" border="0" alt="arrow" width="32" height="32">
</td>
<td>Many other features can be exploited - check them out!
      </tr>
    </table>
    <b>Macros</b> are also very useful and let you easily generate
    nav bars and site maps:
    <br>
<ul><li><a href="index.html">HTML PLAIN demo</a>
<ul><li><a href="example.html">Another example page</a>
<li><a href="skip.html">This page is not parsed.</a>
<li><a href="test1.html">Test1 data</a>
<li><a href="test2.html">Test2 data</a>
</ul></ul>
    <br>
    The other pages show you some more advanced features.
    <br>
    <b><font face="Geneva,Helvetica,Arial">Some &quot;long quotation
      that goes over multiple lines&quot;</font></b>
    <br>
    A last example: environment variables, e. g. $USER = cartho.
    <br>
    <!-- Created: Tue Mar 16 22:09:37 GMT 1999 -->
<br>
<a href="index.html">Return to  HTML PLAIN demo</a><!-- The link back is made by a macro -
->
```

```
<br>
  </BODY>
</HTML>
```

skip.html

This page is simply copied to the destination directory and therefore of little interest.

test1.html

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head>
<title>HTML PLAIN demo - Test1 data</title>
<mymeta>
  </head>
  <BODY text="#000000" bgcolor="#cccccc">
    <h1>Test1 data</h1>
    Another kind of template; this document has the line
    <tt>&lt;!-- foreach test*.data count--&gt;</tt>
    as its first line, which means that each file test*.data will be parsed,
    its values will be used (like in any variable file), and the output files
    will be test*.html.
    The value of * can be used in &quot;count&quot; = "count".
    <br>
    This particular file only uses &quot;VarA&quot; = Variable A-1 and
    &quot;VarB&quot; = Variable B-1.
<br>
<a href="index.html">Return to  HTML PLAIN demo</a><!-- The link back is made by a macro -
->
<br>
</BODY>
</HTML>
```

test2.html

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head>
<title>HTML PLAIN demo - Test2 data</title>
<mymeta>
  </head>
  <BODY text="#000000" bgcolor="#cccccc">
    <h1>Test2 data</h1>
    Another kind of template; this document has the line
    <tt>&lt;!-- foreach test*.data count--&gt;</tt>
    as its first line, which means that each file test*.data will be parsed,
    its values will be used (like in any variable file), and the output files
    will be test*.html.
    The value of * can be used in &quot;count&quot; = "count".
    <br>
    This particular file only uses &quot;VarA&quot; = Variable A-2 and
    &quot;VarB&quot; = Variable B-2.
<br>
<a href="index.html">Return to  HTML PLAIN demo</a><!-- The link back is made by a macro -
->
<br>
</BODY>
</HTML>
```

A.2 Other test pages

There were a few other pages used for testing, but the features tested there were the same. Because it would take a large amount of paper to list these, too, I would rather spare a few trees. These pages can be obtained by request - go to artho.com `http://artho.com/` for a contact form.

Also, the entire `artho.com` page was made using HTML PLAIN, and many of the more advanced features could only be tested there, using a "real" page. The test pages from above were just too simple for showing some of the more subtle bugs. It was also a good way to test the inclusion of `latex2html` documents (see Section 5.5).

Appendix B

Changelogs

The two most important “Changelogs” are included in the HTML PLAIN distribution. They document the changes in the main programs. Some of the modules have their own changelog. If the version number is not yet at 1.0, this means that there were no or only minor changes since the last version. It means that the modules worked indeed, and required no more debugging for the jump the version 1.0 :-)

B.1 Perl programs

Converter

v0.1 Variable values are correctly replaced by variable names; and, more importantly, relative links are correctly converted back to the symbolic names.

File table generator

v0.1: Parses directories and searches the appropriate files for their title. No check for duplicate titles yet, and no automatic generation of "Title missing" (or similar) titles or something alike when no title could be found.

v0.2: Using now the module File::Find for platform independence. Module usage like in "The Perl Cookbook", Chapter 9.7.

v0.3: Can now parse image files for width and height. Only gifs are actually parsed, otherwise the system utility ’identify’ is used.

v0.4: Duplicate titles are now handled properly. For each duplicate, a number is appended after searching the namespace for a free number. Alas, a similar convention was once used by some software company trying to emulate longer filenames with much shorter ones, where this lead to much confusion. Therefore a warning message is printed, advising the user to change the title of one of both files.

v0.5: The file table is now sorted before it is saved, in order to allow documents being processed in ascending level order - this will allow the implementation of multiple levels of templates later.

v1.0: <!– foreach –> files are properly handled in the file table.

v1.0.2: Fixed a minor bug with the titles from <!– foreach –> files.

v1.1: The sorting can now be overridden with the sortingFile option, which contains regular expressions: the left part is replaced by the right part (separated by tabs) of each line.

Main program (parser)

Non-trivial changes to auxiliary modules which do not have their own changelog are also found here.

v0.1: First working version. Parses an HTML file from STDIN and prints the parsed output in colors (for debugging) to STDOUT. Handles HTML tags, entities and strings in quotes.

v0.2: Now imports template.pl and variables.pl, which parse the template and variable dictionary, given the right arguments (filename, pointer(s) to storage).

v0.3: Files are actually parsed now. Variables and tag/entity redefinitions are handled correctly, even tags with options.

v0.4: First major restructuring. The replacement of text is now handled by an external module, which offers this functionality also to the template and variable parser, allowing definitions that refer back to previous definitions. A few other details have also been improved.

v0.5: Parser now replaces HTML tags first, while ignoring all the options, and then the values of the options in double quotes. It is therefore not possible to redefine the name of an option (but this would not be needed anyway). E. g. <font face="myfont"> font and myfont can be changed but not 'face' (why should anyone change this?)

v0.5.1: The options for HTML tags are internally no longer separated by space, but by the 0x80 character for easier parsing. However, this change cannot cope with several requirements; eventually, the replaced tags will have to be stored in a more complex data structure:

The format will be <definition> => <reference to redefinition>

<redefinition> = (index, <list of references to tag redefinitions>) (length 1 unless several tags are specified); the index is the number of the list element with the same tag name as the definition. It does not always have to exist (and is 0 in that case). This helps to speed up option overriding.

<tag redefinition> = list: tagname, [option, value]

v0.6: Implementation with that data structure. Unfortunately, the access over this data structure is rather awkward and annihilates any performance gain by less frequent parsing.

v0.7: Back to the simple line based data storage, with improved parsing routines.

v0.8: Filenames etc. work now (if the file table is up to date), but there are no substitutions being made yet for "blank" names (without the surrounding tags). Also, the path has to be adjusted, otherwise the links will be broken.

v0.8.1: Links using symbolic names are now correct relative links. A bug in the option replacement has been fixed, so all redundant options should be erased.

v0.8.2: Strings between quotes that represent either links or images are now correctly replaced by the HTML code for a link or images.

v0.8.3: Options from config.pl are now used for getting the list of all files (via the filetable) and the output path. Redirection of input and output is no longer required, usage is now simply 'parser.pl'.

v0.8.4: Start of implementation of the header and footer. The problem of interweaving lines of the header/footer with the document (in case the options of the template header are overwritten in the document) are not yet solved.

v0.8.5: URLs and e-mail addresses in plain text are now "URLified", i. e. linked to with proper HTML.

v0.8.6: Overriding options is now properly done in the case of multiple HTML tags. Basic macro support is done, so far only for built-in Perl commands and bits of Perl code.

v0.9: Finally, the header and footer merging with the document is implemented. All that is left is now to write access functions for macros and also write the most important macros, too.

v0.9.1: Lots of small bug fixes, one major bug being the inability to cope with subdirectories (fixed). The header and footer insertion is still a bit buggy.

v0.9.2: A macro for header or footer links is implemented, along with a first incarnation of the IMP (Interface for Macro Programming).

v1.0beta1: The IMP has been enhanced, and the insertion of header and footer links is now much more reliable. Also lots of small changes and bug fixes. A notable change is that the program now runs from /usr/bin/local, using the configuration under $HOME/.htmlplain/.

v1.0beta2: A few minor bug fixes. Dependency checking has been added among a few minor tweaks. Another important new feature is that local configuration files are used, if they exist, instead of the ones in $HOME/.htmlplain/. The documentation has also been improved and is nearly complete.

v1.0: Command line arguments are parsed. The option to skip document parsing has been added, and also the 'foreach' operator that allows html documents to be used with a set of different variable files.

v1.0.1: It is now possible to include a pattern for input files in the command line. Also fixed a couple of bugs.

v1.0.2: A small new macro has been added: DataLinks. It is for use within <!– foreach –> documents to link to all other documents using the same data files.

v1.0.3: A couple of minor bugfixes, and hopefully better Windows compatibility.

v1.0.4: Some minor bugfixes. An 'if/else/endif' preprocessor directive has been added.

v1.0.5: An 'include' statement has been added - this may be preferable compared to redefining tags, if the redefinition is very large. Also, the header/footer merging part has been thoroughly revised, and less faults should occur now in that area - ideally, none :)

v1.0.6: A bugfix in the header/footer inclusion. Another bugfix for macros.

v1.0.7: Another minor bugfix in the header/footer inclusion. Empty variable names (two double quotes, "") are no longer replaced by one double quote; use a backslash instead, \", or "

v1.0.8: A few minor bugfixes, particularly in the template parsing part.

v1.1: Some internal data structures have been reorganized. Links with # signs should work correctly now. Two new features have been added: 1) Overriding the default sorting; 2) the possibility to have templates always override the document. Two new macros have been added (Prev and Next).

Template parser

v0.1: First working version. Parses templates from STDIN, prints parsed template to STDOUT (for debugging) and stores parsed content in a hash.

v0.2: Storing parsed lines was moved to an extra module.

v0.3: Recursive definitions are now possible. The implementation of that feature is still a bit clumsy and will need a rework for the final version.

v0.4: Recursive definitions are now better implemented.

v1.0: Many smaller changes.

v1.1: Support to have template take precedence over the document (if the comment starts with !).

Variable file parser

v0.1: Parses variable template, stores content in storage (filename and pointer given from args).

v0.2: Recursive definitions are now possible.

B.2 Tcl/Tk programs

There is only one changelog for all programs, because they have been developed in parallel, due to their highly similar functions.

Configuration tools

v0.9: Everything important is implemented. There is still a minor bug with scrolling with the first mouse button.

v0.9.1: User is now asked whether to save changes, and a simple session management is implemented.

v0.9.2: Some keyboard bindings are be defined.

v0.9.3: All keyboard bindings are defined, and input line automatically takes the focus once an element is selected.

v0.9.4: Some minor changes and tweaks in the user interface.

v0.9.5: Improved scrolling behavior when having the mouse button 1 pressed and scrolling. The behavior is still buggy, but partly due to tk itself.

v0.9.6: An OK button has been added, allowing from recovery if the user deletes the current setting in the entry buffer accidentally.

v0.9.7: Many small improvements and bug fixes (mainly with the two lists) or addition of missing implementations of event handling code. The variable editor has now all features implemented (adding/removing lines etc.)

v0.9.8: Only changes in the variable editor, which focused on focus. Also added a simple "about" dialog.

v0.9.9: Added a help menu entry which will launch Netscape, hopefully finding the help page then...

v0.9.10: Added "Insert color", "Insert filename" options.

v0.9.11: Restructured the above two options, allowing to use the selected text as input and replacing it with the result.

v0.9.12: Undo for line deletion/change implemented.

v0.9.13: Major restructuring of the code of the variable editor in order to allow to reuse as much as possible for the template editor.

v0.9.14: Many minor bug fixes, namely the undo operation for inserting a value.

v0.9.15: Lots of bug fixes after some minor changes introduced lots and lots of bugs...

v0.9.16: Unfortunately config.tcl (the oldest configuration tool) was no longer compatible with the modified version of the module that handled the two lists. A slightly modified version had to be forked off. Some time, config.tcl should be rewritten to use the newer module. - In the other modules, a few final bugs were fixed.

v0.9.17: Some final tweaks in the variable and template editors. Also changed "Close" to "New" and "Reload" to "Revert", which IMHO fits their function better.

v0.9.18: Some minor changes and bug fixes.

v0.9.19: More bug fixes.

v1.0beta1: A few tweaks, mainly with the right click menu. Also, external programs like the directory selector (newly added) or color picker are now sourced in instead of called via "wish", which saves forking another process.

v1.0: A few bug fixes and minor changes in the code (no new functionality added).

v1.0.1: Fixed a bug in the template editor with the closing angular brackets (">"). The help files are now also included and appear correctly (but only under Unix or Linux).

v1.0.2: D. Roche's directory picker changed the pwd, and a subsequent "Save File" fails. Fixed. An install program has also been added, making the installation to directories other than the default one very easy.

v1.0.3: Some improvements in the install.tcl program.

v1.0.4: Some bugfixes, although the install.tcl program still causes problems under Windows.

Appendix C

Acknowledgements

This project has been submitted for the Computer Science department of the University of Strathclyde, Glasgow. Duncan Smeed from the Computer Science department was the supervisor, and gave me a good idea of the initial concept, mainly of the file management aspect. Without these ideas, the project would have turned out quite differently, and the file management aspect would have been largely missing.

Many of these ideas stem from Frontier [Fro], a commercial tool for MacIntosh. HTML PLAIN implements about $\frac{2}{3}$ of Frontier's functionality, although not always in the same way. However, it lacks some of Frontier's bugs (e. g. a URL is surrounded with a link tag even if this has been done manually) and also has no support yet for dynamically generated pages (with CGI). Nevertheless, the first steps for this have been done. A future CGI expansion only has to replace the strings in double quotes - this can be done much faster than if a program had to parse everything.

I do not have Frontier and only had a glimpse at what it does. Therefore this program is in no way a replacement for it. Nor was the aim taken to make it compatible in any way.

Special thanks also go to Jens Jensen, for sending me a patch addressing one of the most difficult problems – a correct header/footer inclusion; Jens Benecke, for some good suggestions, especially about pointing out some important weaknesses in the README file; and to Tzafrir Cohen for testing the configuration tools under Windows 98, having the patience to change the path names (the installer was not written at that time).

Bibliography

[Chr98] *Perl Cookbook.* O'Reilly & Associates, Inc. Sebastopol, first edition, 1998.

[Sri97] Srinivasan, Sriram. *Advanced Perl Programming.* O'Reilly & Associates, Inc. Sebastopol, first edition, 1997.

[Wal96] Wall, Larry; Christiansen, Tom; Schwartz, Randal. *Programming Perl.* O'Reilly & Associates, Inc. Sebastopol, second edition, 1996.

[Wel97] Welch, Brent B. Practical Programming in Tcl and Tk. Prentice Hall. Upper Saddle River, second edition 1997.

[Arb] Arbortext: XML for managers `http://www.arbortext.com/think%5Ftank/xml%5F%5Fresources/xml%5Ffor%5Fmanagers/xml%5Ffor%5Fmanagers.html`

[Chpp] chpp `http://chakotay.ist.org/`

[Fro] Frontier `http://frontier.userland.com`

[gtk] The gimp toolkit `http://gtk.org/`

[HPL] HTML PLAIN `http://artho.com/webtools/plain/`

[Meta] Meta HTML `http://www.metahtml.com/`

[Perl] Perl `http://www.perl.com/`

[PHP] PHP3: P Hypertext Preprocessor `http://www.php.net/`

[QML] QML: Quick Markup Language `http://www.qml.org/`

[Qt] Troll tech `http://www.troll.no/`

[Tcl] Tcl/Tk `http://www.scriptics.com/`

[Wpp] Web Preprocessor `http://www.geocities.com/Tokyo/1474/wpp/`

Index