NAME

HTML::Tree − Perl extension for quickly parsing HTML files into trees

SYNOPSIS

```
use HTML::Tree;
$html = HTML::Tree->new( 'file.html' );

sub visitor {
       my( $node, $depth, $is_end_tag ) = @_;
       # ...
}
$html->visit( \&visitor );
```

or:

```
sub visitor {
       my( $hash_ref, $node, $depth, $is_end_tag ) = @_;
       # ...
}
%my_hash;
# ...
$html->visit( \%my_hash, \&visitor );
```

DESCRIPTION

HTML::Tree is a fast parser that parses an HTML file into a tree structure like the HTML DOM (Document Object Model). Once built, the nodes of the tree (elements and text from the HTML file) can be traversed by a user-defined *visitor* function.

HTML::Tree is very similar to the HTML::Parser and HTML::TreeBuilder modules by Gisle Aas, except that it:

1. Is several times faster. HTML::Tree owes its speed to two things: using *mmap*(2) to read the HTML file bypassing conventional I/O and buffering, and being written entirely in C++ as opposed to Perl.
2. Isn't a strict DTD (Document Type Definition) parser. The goal is to parse HTML files fast, not check for validity. (You should check the validity of your HTML files with other tools *before* you put them on your web site anyway.) For example, HTML::Tree couldn't care less what attributes a given HTML element has just so long as the syntax is correct. This is actually similar to browsers in that both are very permissive in what they accept.
3. Offers simple conditional and looping mechanisms assisting in the generation of dynamic HTML content.

Methods

For the methods below, the kind of node a method may be called on is indicated; $node means ''any kind of node.'' Calling a method for a node of the wrong kind is a fatal error.

$root_node = HTML::Tree->new(*file_name*)

Parse the given HTML file and return a reference to a new HTML::Tree object. If, for any reason, the file can not be parsed (file does not exist, insufficient permissions, etc.), undef is returned.

$element_node->att(*name*)

Returns the value of the element node's *name* attribute or undef if said node does not have one. Attribute names **must** be specified in lower case (regardless of how they are in the HTML file).

`$element_node->att( name, new_value )`
Sets the value of the element node's *name* attribute to *new_value*. If *new_value* is `undef`, then the attribute is deleted. Attribute names **must** be specified in lower case (regardless of how they are in the HTML file). If no *name* attribute existed, it is added.

`$element_node->atts()`
Returns a reference to a tied hash of all of an element node's attribute/value pairs or a reference to an empty hash if said node does not have any. Attribute names are returned in lower case (regardless of how they are in the HTML file). Because the hash is tied, assigning to a hash element changes that attribute's value; similarly, deleting an element deletes the attribute.

`$element_node->children()`
Returns all of an element node's child nodes or an empty list if said node does not have any.

`$node->is_text()`
Returns true (1) only if the current node is a text node; false (0), otherwise. (If a node isn't a text node, it must be an element node.) HTML comments are treated as text nodes.

`$element_node->name()`
Returns the HTML element name of an element node, e.g., `title`. All names are returned in lower case (regardless of how they are in the HTML file).

`$text_node->text()`
Returns the text of a text node.

`$text_node->text( new_value )`
Sets the text of a text node to *new_value*; returns the new text.

`$root_node->visit( \&visitor )`
Traverse the HTML tree by calling the *visitor* function for every node starting at the root node previously returned by the constructor.

`$root_node->visit( \%hash, \&visitor )`
Same as the previous method except that a hash reference is passed along (see **Arguments** below).

The Visitor Function

The user supplies a *visitor* function: a Perl function that is called when every node is visited (i.e., a ''call-back'') during an in-order tree traversal.

For HTML elements that have end tags, the *visitor* function may be called more than once for a given node based on the function's return value. (See **Return Value** below.)

Note that this occurs for such HTML elements even if said element's end tag is optional and was not present in the HTML file.

Arguments

`$hash_ref`	A reference to a hash that is passed only if the two-argument form of the `visit()` method is used. This provides a mechanism for additional data (or a blessed object) to be passed to and among the calls to the *visitor* function. The argument is not used at all by `HTML::Tree`.
`$node`	A reference to the current node.
`$depth`	An integer specifying how ''deep'' the node is in the tree. (Depths start at zero.)
`$is_end_tag`	True (1) only if the tag is an end tag of an HTML element; false (0), otherwise.

Return Value

The *visitor* function is expected to return a Boolean value (zero or non-zero for false or true, respectively). There are two meanings for the return value:

1. If the `$is_end_tag` argument is false, returning false means: do not visit any of the current node's child nodes, i.e., skip them and proceed directly to the current node's next sibling and also do not call the *visitor* again for the end tag; returning true means: do visit all child nodes and call the *visitor* again for the end tag.
2. If the `$is_end_tag` argument is true, returning false means: proceed normally to the next sibling; returning true means: loop back and repeat the visit cycle from the beginning by revisiting the start tag of the current element node (case 1 above).

EXAMPLE

Here is a sample visitor function that ''pretty prints'' an HTML file:

```
sub visitor {
        my( $node, $depth, $is_end_tag ) = @_;
        print "    " x $depth;
        if ( $node->is_text() ) {
                my $text = $node->text();
                $text =~ s/(?:^\n|\n$)//g;
                print "$text\n";
                return 1;
        }
        if ( $is_end_tag ) {
                print "</", $node->name(), ">\n";
                return 0;
        }
        print '<', $node->name();
        my $atts = $node->atts();
        while ( my( $att, $val ) = each %{ $atts } ) {
                print " $att=\"$val\"";
        }
        print ">\n";
        return 1;
}
```

NOTES

In order for an HTML file to be properly parsed, scripting languages **must** be ''comment hidden'' as in:

```
<SCRIPT LANGUAGE="JavaScript">
<!--
        ... script goes here ...
// -->
</SCRIPT>
```

SEE ALSO

perl(1), *mmap*(2), HTML:\fIs0:Parser(3), HTML:\fIs0:TreeBuilder(3).

World Wide Web Consortium Document Object Model Working Group. *Document Object Model*, December 1998. `http://www.w3.org/DOM/`

AUTHOR

Paul J. Lucas <*pjl@best.com*>

HISTORY

The HTML parser of the C++ part of the module is derived from code in SWISH++, a really fast file indexing and searching engine (also by the author).