

**NAME**

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

**SYNOPSIS**

```
use HTML::Tree;

$tree1 = HTML::Tree->from_file( 'file.html' );
$aref  = $tree1->as_array();

$tree2 = HTML::Tree->from_array( $aref );
$str   = $tree2->as_string();

$tree3 = HTML::Tree->from_string( $str );
$tree3->write( 'new_file.html' );
```

then:

```
sub visitor {
    my( $node, $depth, $is_end_tag ) = @_;
    # ...
}
$tree1->visit( \&visitor );
```

or:

```
sub visitor {
    my( $hash_ref, $node, $depth, $is_end_tag ) = @_;
    # ...
}
%my_hash;
# ...
$tree1->visit( \%my_hash, \&visitor );
```

also:

```
$aref = $node->children();
$node->delete();
$node = $node->find_if( \&predicate_function );
$node = $node->find_name( 'name' );
$bool = $node->is_element();
$bool = $node->is_comment();
$bool = $node->is_text();
$name = $node->name();
$text = $node->text();
```

**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 or compiled into an array-of-hashes data structure.

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.

```
$parent_node = HTML::Tree->from_file( $file_name [ { param_hash_ref } ] )
```

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.

Parameters that control how the data structure is built may be passed via a reference to a hash. If `Include_Comments` is given with a non-zero value, then comment nodes are included; otherwise, they are elided.

```
$array_ref = $node->as_array( [ { param_hash_ref } ] )
```

Returns a reference to an array-of-hashes data structure representing the nodes in the HTML tree starting at the specified node. Parameters that control how the data structure is built may be passed via a reference to a hash. The parameters are the same as for `from_file()` above.

For example, given this HTML:

```
<a href="file.html">
  Text A
  <b>Text B</b>
  <i>Text C</i>
</a>
```

the `Data::Dumper` representation of the resulting data structure would be:

```
$ref = [
  {
    'name' => 'a',
    'atts' => { 'href' => 'file.html' },
    'content' => [
      'Text A',
      {
        'name' => 'b',
        'content' => [ 'Text B' ]
      },
      {
        'name' => 'i',
        'content' => [ 'Text C' ]
      },
    ],
  }
]
```

Every HTML element at the same "depth" or "level" is contained in the same array, i.e., they are "siblings" in the tree. The order of the elements in the array matches the order of the HTML elements in the file.

A node is either a string (representing text or a comment) or a reference to a hash (representing an

HTML element).

Strings are tied scalars, so modifying them changes the underlying tree. Strings in the HTML file that are entirely whitespace are elided from the data structure.

A hash always has a `name` key whose value is the name of the HTML element and may also have an `atts` key and/or a `content` key.

The value of the `atts` key is a reference to a tied hash where the hash keys are attribute names and the hash values are the attribute values. 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 attribute changes that attribute's value; similarly, deleting an element deletes the attribute.

The value of the `content` key is a reference to an array containing all of the node's child nodes at the next level down.

**Note:** Modifying the arrays themselves (adding elements, deleting, etc.) does *not* modify the underlying tree. To do that, either use the `children()` method or “walk” the tree using a *visitor* function.

```
$parent_node = HTML::Tree->from_array( array_ref )
```

Create a new `HTML::Tree` object from a data structure in the form returned by `as_array()`. If, for any reason, the data structure isn't in the right form, the function will croak with an error message.

```
$string = $node->as_string( [ { param_hash_ref } ] )
```

Return the HTML text representation of the portion of the tree starting at the given node as a single string. Parameters that control how the HTML tree is converted to a string may be passed via a reference to a hash.

If the `Pretty_Print` parameter is given with a value greater than or equal to zero, then text nodes have leading and trailing whitespace removed, are indented according to their depth, and have a single newline appended. All other nodes appear on lines by themselves and are also indented according to their depth. Indentation is done by spaces where the number of spaces at a given depth is  $(\text{Pretty\_Print} + \text{depth}) * 2$ .

**Note:** pretty-printing is suspended inside `<PRE>` elements to preserve the original formatting.

```
$parent_node = HTML::Tree->from_string( string { param_hash_ref } ] )
```

This is the same as `from_file()` except that the HTML is parsed from the given string rather than a file.

```
$value = $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.

```
$attributes_ref = $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.

```
$child_nodes_ref = $parent_node->children()
```

Returns a reference to a tied array of all of an element node's child nodes. Because the array is tied, the Perl array manipulation functions `pop`, `push`, `shift`, and `unshift` work and affect the structure of the `HTML::Tree`. For example:

```
$orphan = unshift @{ $node1->children() };
```

“detaches” the first child node of `$node1` from the tree structure and returns a reference to it now as

its own distinct HTML::Tree. Conversely:

```
push @{$ $node2->children() }, $orphan;
```

“reattaches” the sub-tree but now at the end of the child nodes of `$node2` elsewhere in the tree.

Additionally, a child node can also be replaced by assignment as in:

```
$node->children()->[0] = expression
```

where *expression* is one of: a reference to a data structure in the form returned by `as_array()`, a reference to an HTML::Tree (in which case the whole tree is “inserted”), or a string (in which case the string is parsed as HTML).

`$node->delete()`

Delete the node and all of its child nodes, if any, from the tree. Once deleted, the reference to the node **must not** be used.

`$node = $node->find_if(func_ref)`

Find the first node for which the given predicate function is true starting the find from the given node. Returns `undef` if no such node is found. Closures work well to generate the predicate function since additional parameters can be used during the find. For example:

```
sub pred_att_re {
    my( $att, $re ) = @_;
    return sub {
        my $node = shift;
        return $node->is_element() &&
            $node->att( $att ) =~ /$re/;
    }
}

$node = $html->find_if( pred_att_re( 'href', '\.jpg$' ) );
```

This would find an element node having an attribute `href` that matches the regular expression `\.jpg$`.

`$element_node = $node->find_name(name)`

Find the first element node having the given name starting the find from the given node. The *name* **must** be specified in lower case. Returns `undef` if no such element node is found. (This function is a special case of `find_if()` and is much faster for finding by name alone.)

`$bool = $node->is_comment()`

Returns true (1) only if the current node is a comment node; false (0), otherwise.

`$bool = $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.)

`$name = $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 = $text_node->text( [new_text] )`

Returns the text of a text node as a string. If *new\_text* is given, the text is set to that first.

`$node->visit( \&visitor )`

Traverse the HTML tree by calling the *visitor* function for every node starting at the given node previously returned by a constructor.

```
$node->visit( \%hash, \&visitor )
```

Same as the previous method except that a hash reference is passed along (see **Arguments** below).

```
$success = $node->write( file_name [, { param_hash_ref } ] )
```

Write the HTML text representation of the portion of the tree starting at the given node as a single string to a file. Returns 1 upon success, 0 otherwise.

Parameters that control how the HTML is written may be passed via a reference to a hash. The `Pretty_Print` parameter has the same meaning as it does for `as_string()`.

## 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), *Data::Dumper* (3), *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 <[pauljlucas@mac.com](mailto:pauljlucas@mac.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).