NAME

HTML::Tree::Scanning −− article: "Scanning HTML"

SYNOPSIS

```
# This an article, not a module.
```

DESCRIPTION

The following article by Sean M. Burke first appeared in *The Perl Journal* #19 and is copyright 2000 The Perl Journal. It appears courtesy of Jon Orwant and The Perl Journal. This document may be distributed under the same terms as Perl itself.

(Note that this is discussed in chapters 6 through 10 of the book *Perl and LWP* <http://lwp.interglacial.com/> which was written after the following documentation, and which is available free online.)

Scanning HTML

−− Sean M. Burke

In *The Perl Journal* issue 17, Ken MacFarlane's article ''Parsing HTML with HTML::Parser'' describes how the HTML::Parser module scans HTML source as a stream of start-tags, end-tags, text, comments, etc. In TPJ #18, my ''Trees'' article kicked around the idea of tree-shaped data structures. Now I'll try to tie it together, in a discussion of HTML trees.

The CPAN module HTML::TreeBuilder takes the tags that HTML::Parser picks out, and builds a parse tree — a tree-shaped network of objects...

> Footnote: And if you need a quick explanation of objects, see my TPJ17 article ''A User's View of Object-Oriented Modules''; or go whole hog and get Damian Conway's excellent book *Object-Oriented Perl*, from Manning Publications.

...representing the structured content of the HTML document. And once the document is parsed as a tree, you'll find the common tasks of extracting data from that HTML document/tree to be quite straightforward.

HTML::Parser, HTML::TreeBuilder, and HTML::Element

You use HTML::TreeBuilder to make a parse tree out of an HTML source file, by simply saying:

```
use HTML::TreeBuilder;
my $tree = HTML::TreeBuilder->new();
$tree->parse_file('foo.html');
```

and then `$tree` contains a parse tree built from the HTML source from the file ''foo.html''. The way this parse tree is represented is with a network of objects — `$tree` is the root, an element with tag-name ''html'', and its children typically include a ''head'' and ''body'' element, and so on. Elements in the tree are objects of the class HTML::Element.

So, if you take this source:

```
<html><head><title>Doc 1</title></head>
<body>
Stuff <hr> 2000-08-17
</body></html>
```

and feed it to HTML::TreeBuilder, it'll return a tree of objects that looks like this:

```
              html
             /    \
         head      body
        /         /  |  \
    title   "Stuff" hr  "2000-08-17"
      |
   "Doc 1"
```

This is a pretty simple document, but if it were any more complex, it'd be a bit hard to draw in that style, since it's sprawl left and right. The same tree can be represented a bit more easily sideways, with

indenting:

```
. html
   . head
      . title
         . "Doc 1"
   . body
      . "Stuff"
      . hr
      . "2000-08-17"
```

Either way expresses the same structure. In that structure, the root node is an object of the class HTML::Element

> Footnote: Well actually, the root is of the class HTML::TreeBuilder, but that's just a subclass of HTML::Element, plus the few extra methods like `parse_file` that elaborate the tree

, with the tag name ''html'', and with two children: an HTML::Element object whose tag names are ''head'' and ''body''. And each of those elements have children, and so on down. Not all elements (as we'll call the objects of class HTML::Element) have children — the ''hr'' element doesn't. And note all nodes in the tree are elements — the text nodes (''Doc 1'', ''Stuff'', and ''2000−08−17'') are just strings.

Objects of the class HTML::Element each have three noteworthy attributes:

`_tag` — (best accessed as `$e->tag`) this element's tag-name, lowercased (e.g., ''em'' for an ''em'' element).

> Footnote: Yes, this is misnamed. In proper SGML terminology, this is instead called a ''GI'', short for ''generic identifier''; and the term ''tag'' is used for a token of SGML source that represents either the start of an element (a start-tag like '''') or the end of an element (an end-tag like ''''. However, since more people claim to have been abducted by aliens than to have ever seen the SGML standard, and since both encounters typically involve a feeling of ''missing time'', it's not surprising that the terminology of the SGML standard is not closely followed.

`_parent` — (best accessed as `$e->parent`) the element that is `$obj`'s parent, or undef if this element is the root of its tree.

`_content` — (best accessed as `$e->content_list`) the list of nodes (i.e., elements or text segments) that are `$e`'s children.

Moreover, if an element object has any attributes in the SGML sense of the word, then those are readable as `$e->attr('name')` — for example, with the object built from having parsed "<a **id='foo'**>bar", `$e->attr('id')` will return the string ''foo''. Moreover, `$e->tag` on that object returns the string ''a'', `$e->content_list` returns a list consisting of just the single scalar ''bar'', and `$e->parent` returns the object that's this node's parent — which may be, for example, a ''p'' element.

And that's all that there is to it — you throw HTML source at TreeBuilder, and it returns a tree built of HTML::Element objects and some text strings.

However, what do you *do* with a tree of objects? People code information into HTML trees not for the fun of arranging elements, but to represent the structure of specific text and images — some text is in this ''li'' element, some other text is in that heading, some images are in that other table cell that has those attributes, and so on.

Now, it may happen that you're rendering that whole HTML tree into some layout format. Or you could be trying to make some systematic change to the HTML tree before dumping it out as HTML source again. But, in my experience, by far the most common programming task that Perl programmers face with HTML is in trying to extract some piece of information from a larger document. Since that's so common (and also since it involves concepts that are basic to more complex tasks), that is what the rest of this article will be about.

Scanning HTML trees

Suppose you have a thousand HTML documents, each of them a press release. They all start out:

```
[...lots of leading images and junk...]
<h1>ConGlomCo to Open New Corporate Office in Ougadougou</h1>
BAKERSFIELD, CA, 2000-04-24 -- ConGlomCo's vice president in charge
of world conquest, Rock Feldspar, announced today the opening of a
new office in Ougadougou, the capital city of Burkino Faso, gateway
to the bustling "Silicon Sahara" of Africa...
[...etc...]
```

...and what you've got to do is, for each document, copy whatever text is in the ''h1'' element, so that you can, for example, make a table of contents of it. Now, there are three ways to do this:

- You can just use a regexp to scan the file for a text pattern.

  For many very simple tasks, this will do fine. Many HTML documents are, in practice, very consistently formatted as far as placement of linebreaks and whitespace, so you could just get away with scanning the file like so:

  ```
  sub get_heading {
    my $filename = $_[0];
    local *HTML;
    open(HTML, $filename)
      or die "Couldn't open $filename);
    my $heading;
   Line:
    while(<HTML>) {
      if( m{<h1>(.*?)</h1>}i ) {  # match it!
        $heading = $1;
        last Line;
      }
    }
    close(HTML);
    warn "No heading in $filename?"
     unless defined $heading;
    return $heading;
  }
  ```

  This is quick and fast, but awfully fragile — if there's a newline in the middle of a heading's text, it won't match the above regexp, and you'll get an error. The regexp will also fail if the ''h1'' element's start-tag has any attributes. If you have to adapt your code to fit more kinds of start-tags, you'll end up basically reinventing part of HTML::Parser, at which point you should probably just stop, and use HTML::Parser itself:

- You can use HTML::Parser to scan the file for an ''h1'' start-tag token, then capture all the text tokens until the ''h1'' close-tag. This approach is extensively covered in the Ken MacFarlane's TPJ17 article ''Parsing HTML with HTML::Parser''. (A variant of this approach is to use HTML::TokeParser, which presents a different and rather handier interface to the tokens that HTML::Parser picks out.)

  Using HTML::Parser is less fragile than our first approach, since it's not sensitive to the exact internal formatting of the start-tag (much less whether it's split across two lines). However, when you need more information about the context of the ''h1'' element, or if you're having to deal with any of the tricky bits of HTML, such as parsing of tables, you'll find out the flat list of tokens that HTML::Parser returns isn't immediately useful. To get something useful out of those tokens, you'll need to write code that knows some things about what elements take no content (as with ''hr'' elements), and that a '' '' end-tags are omissible, so a '' '' will end any currently open paragraph — and you're well on your way to pointlessly reinventing much of the code in HTML::TreeBuilder

Footnote: And, as the person who last rewrote that module, I can attest that it wasn't terribly easy to get right! Never underestimate the perversity of people coding HTML.

, at which point you should probably just stop, and use HTML::TreeBuilder itself:

- You can use HTML::Treebuilder, and scan the tree of element objects that you get back.

The last approach, using HTML::TreeBuilder, is the diametric opposite of first approach: The first approach involves just elementary Perl and one regexp, whereas the TreeBuilder approach involves being at home with the concept of tree-shaped data structures and modules with object-oriented interfaces, as well as with the particular interfaces that HTML::TreeBuilder and HTML::Element provide.

However, what the TreeBuilder approach has going for it is that it's the most robust, because it involves dealing with HTML in its ''native'' format — it deals with the tree structure that HTML code represents, without any consideration of how the source is coded and with what tags omitted.

So, to extract the text from the ''h1'' elements of an HTML document:

```
sub get_heading {
  my $tree = HTML::TreeBuilder->new;
  $tree->parse_file($_[0]);   # !
  my $heading;
  my $h1 = $tree->look_down('_tag', 'h1');  # !
  if($h1) {
    $heading = $h1->as_text;   # !
  } else {
    warn "No heading in $_[0]?";
  }
  $tree->delete; # clear memory!
  return $heading;
}
```

This uses some unfamiliar methods that need explaining. The`parse_file` method that we' ve seen before, builds a tree based on source from the file given. The`delete` method is for marking a tree' s contents as available for garbage collection, when you're done with the tree. The `as_text` method returns a string that contains all the text bits that are children (or otherwise descendants) of the given node — to get the text content of the `$h1` object, we could just say:

```
$heading = join '', $h1->content_list;
```

but that will work only if we're sure that the ''h1'' element's children will be only text bits — if the document contained:

```
<h1>Local Man Sees <cite>Blade</cite> Again</h1>
```

then the sub-tree would be:

```
. h1
  . "Local Man Sees "
  . cite
    . "Blade"
  . " Again'
```

so `join '', $h1->content_list` will be something like:

```
Local Man Sees HTML::Element=HASH(0x15424040) Again
```

whereas `$h1->as_text` would yield:

```
Local Man Sees Blade Again
```

and depending on what you're doing with the heading text, you might want the `as_HTML` method instead. It returns the (sub)tree represented as HTML source. `$h1->as_HTML` would yield:

```
<h1>Local Man Sees <cite>Blade</cite> Again</h1>
```

However, if you wanted the contents of `$h1` as HTML, but not the `$h1` itself, you could say:

```
join '',
  map(
    ref($_) ? $_->as_HTML : $_,
    $h1->content_list
  )
```

This `map` iterates over the nodes in `$h1`'s list of children; and for each node that's just a text bit (as "Local Man Sees " is), it just passes through that string value, and for each node that's an actual object (causing `ref` to be true), `as_HTML` will used instead of the string value of the object itself (which would be something quite useless, as most object values are). So that `as_HTML` for the "cite" element will be the string "<cite>Blade</cite>". And then, finally, `join` just puts into one string all the strings that the `map` returns.

Last but not least, the most important method in our `get_heading` sub is the `look_down` method. This method looks down at the subtree starting at the given object (`$h1`), looking for elements that meet criteria you provide.

The criteria are specified in the method's argument list. Each criterion can consist of two scalars, a key and a value, which express that you want elements that have that attribute (like "_tag", or "src") with the given value ("h1"); or the criterion can be a reference to a subroutine that, when called on the given element, returns true if that is a node you're looking for. If you specify several criteria, then that's taken to mean that you want all the elements that each satisfy *all* the criteria. (In other words, there's an "implicit AND".)

And finally, there's a bit of an optimization — if you call the `look_down` method in a scalar context, you get just the *first* node (or undef if none) — and, in fact, once `look_down` finds that first matching element, it doesn't bother looking any further.

So the example:

```
$h1 = $tree->look_down('_tag', 'h1');
```

returns the first element at-or-under `$tree` whose `"_tag"` attribute has the value `"h1"`.

Complex Criteria in Tree Scanning

Now, the above `look_down` code looks like a lot of bother, with barely more benefit than just grepping the file! But consider if your criteria were more complicated — suppose you found that some of the press releases that you were scanning had several "h1" elements, possibly before or after the one you actually want. For example:

```
<h1><center>Visit Our Corporate Partner
 <br><a href="/dyna/clickthru"
   ><img src="/dyna/vend_ad"></a>
</center></h1>
<h1><center>ConGlomCo President Schreck to Visit Regional HQ
 <br><a href="/photos/Schreck_visit_large.jpg"
   ><img src="/photos/Schreck_visit.jpg"></a>
</center></h1>
```

Here, you want to ignore the first "h1" element because it contains an ad, and you want the text from the second "h1". The problem is in formalizing the way you know that it's an ad. Since ad banners are always entreating you to "visit" the sponsoring site, you could exclude "h1" elements that contain the word "visit" under them:

```
my $real_h1 = $tree->look_down(
  '_tag', 'h1',
  sub {
    $_[0]->as_text !~ m/\bvisit/i
  }
);
```

The first criterion looks for ‘‘h1’’ elements, and the second criterion limits those to only the ones whose text content doesn’t match `m/\bvisit/`. But unfortunately, that won’t work for our example, since the second ‘‘h1’’ mentions "ConGlomCo President Schreck to *Visit* Regional HQ".

Instead you could try looking for the first ‘‘h1’’ element that doesn’t contain an image:

```
my $real_h1 = $tree->look_down(
  '_tag', 'h1',
  sub {
    not $_[0]->look_down('_tag', 'img')
  }
);
```

This criterion sub might seem a bit odd, since it calls `look_down` as part of a larger `look_down` operation, but that’s fine. Note that when considered as a boolean value, a `look_down` in a scalar context value returns false (specifically, undef) if there’s no matching element at or under the given element; and it returns the first matching element (which, being a reference and object, is always a true value), if any matches. So, here,

```
sub {
  not $_[0]->look_down('_tag', 'img')
}
```

means ‘‘return true only if this element has no ’img’ element as descendants (and isn’t an ’img’ element itself).’’

This correctly filters out the first ‘‘h1’’ that contains the ad, but it also incorrectly filters out the second ‘‘h1’’ that contains a non-advertisement photo besides the headline text you want.

There clearly are detectable differences between the first and second ‘‘h1’’ elements — the only second one contains the string ‘‘Schreck’’, and we could just test for that:

```
my $real_h1 = $tree->look_down(
  '_tag', 'h1',
  sub {
    $_[0]->as_text =~ m{Schreck}
  }
);
```

And that works fine for this one example, but unless all thousand of your press releases have ‘‘Schreck’’ in the headline, that’s just not a general solution. However, if all the ads-in-‘‘h1’’s that you want to exclude involve a link whose URL involves ‘‘/dyna/’’, then you can use that:

```
my $real_h1 = $tree->look_down(
  '_tag', 'h1',
  sub {
    my $link = $_[0]->look_down('_tag','a');
    return 1 unless $link;
      # no link means it's fine
    return 0 if $link->attr('href') =~ m{/dyna/};
      # a link to there is bad
    return 1; # otherwise okay
  }
);
```

Or you can look at it another way and say that you want the first "h1" element that either contains no images, or else whose image has a "src" attribute whose value contains "/photos/":

```
my $real_h1 = $tree->look_down(
  '_tag', 'h1',
  sub {
    my $img = $_[0]->look_down('_tag','img');
    return 1 unless $img;
      # no image means it's fine
    return 1 if $img->attr('src') =~ m{/photos/};
      # good if a photo
    return 0; # otherwise bad
  }
);
```

Recall that this use of `look_down` in a scalar context means to return the first element at or under `$tree` that matches all the criteria. But if you notice that you can formulate criteria that'll match several possible "h1" elements, some of which may be bogus but the *last* one of which is always the one you want, then you can use `look_down` in a list context, and just use the last element of that list:

```
my @h1s = $tree->look_down(
  '_tag', 'h1',
  ...maybe more criteria...
);
die "What, no h1s here?" unless @h1s;
my $real_h1 = $h1s[-1]; # last or only
```

A Case Study: Scanning Yahoo News's HTML

The above (somewhat contrived) case involves extracting data from a bunch of pre-existing HTML files. In that sort of situation, if your code works for all the files, then you know that the code *works* — since the data it's meant to handle won't go changing or growing; and, typically, once you've used the program, you'll never need to use it again.

The other kind of situation faced in many data extraction tasks is where the program is used recurringly to handle new data — such as from ever-changing Web pages. As a real-world example of this, consider a program that you could use (suppose it's crontabbed) to extract headline-links from subsections of Yahoo News (`http://dailynews.yahoo.com/`).

Yahoo News has several subsections:

http://dailynews.yahoo.com/h/tc/ for technology news
http://dailynews.yahoo.com/h/sc/ for science news
http://dailynews.yahoo.com/h/hl/ for health news
http://dailynews.yahoo.com/h/wl/ for world news
http://dailynews.yahoo.com/h/en/ for entertainment news

and others. All of them are built on the same basic HTML template — and a scarily complicated template it is, especially when you look at it with an eye toward making up rules that will select where the real

headline-links are, while screening out all the links to other parts of Yahoo, other news services, etc. You will need to puzzle over the HTML source, and scrutinize the output of `$tree->dump` on the parse tree of that HTML.

Sometimes the only way to pin down what you're after is by position in the tree. For example, headlines of interest may be in the third column of the second row of the second table element in a page:

```
my $table = ( $tree->look_down('_tag','table') )[1];
my $row2  = ( $table->look_down('_tag', 'tr' ) )[1];
my $col3  = ( $row2->look-down('_tag', 'td')   )[2];
...then do things with $col3...
```

Or they may be all the links in a "p" element that has at least three "br" elements as children:

```
my $p = $tree->look_down(
  '_tag', 'p',
  sub {
    2 < grep { ref($_) and $_->tag eq 'br' }
             $_[0]->content_list
  }
);
@links = $p->look_down('_tag', 'a');
```

But almost always, you can get away with looking for properties of the of the thing itself, rather than just looking for contexts. Now, if you're lucky, the document you're looking through has clear semantic tagging, such is as useful in CSS — note the class="headlinelink" bit here:

```
<a href="...long_news_url..." class="headlinelink">Elvis
seen in tortilla</a>
```

If you find anything like that, you could leap right in and select links with:

```
@links = $tree->look_down('class','headlinelink');
```

Regrettably, your chances of seeing any sort of semantic markup principles really being followed with actual HTML are pretty thin.

> Footnote: In fact, your chances of finding a page that is simply free of HTML errors are even thinner. And surprisingly, sites like Amazon or Yahoo are typically worse as far as quality of code than personal sites whose entire production cycle involves simply being saved and uploaded from Netscape Composer.

The code may be sort of "accidentally semantic", however — for example, in a set of pages I was scanning recently, I found that looking for "td" elements with a "width" attribute value of "375" got me exactly what I wanted. No-one designing that page ever conceived of "width=375" as *meaning* "this is a headline", but if you impute it to mean that, it works.

An approach like this happens to work for the Yahoo News code, because the headline-links are distinguished by the fact that they (and they alone) contain a "b" element:

```
<a href="...long_news_url..."><b>Elvis seen in tortilla</b></a>
```

or, diagrammed as a part of the parse tree:

```
. a  [href="...long_news_url..."]
  . b
    . "Elvis seen in tortilla"
```

A rule that matches these can be formalized as "look for any 'a' element that has only one daughter node, which must be a 'b' element". And this is what it looks like when cooked up as a `look_down` expression and prefaced with a bit of code that retrieves the text of the given Yahoo News page and feeds it to TreeBuilder:

```
use strict;
use HTML::TreeBuilder 2.97;
use LWP::UserAgent;
sub get_headlines {
  my $url = $_[0] || die "What URL?";

  my $response = LWP::UserAgent->new->request(
    HTTP::Request->new( GET => $url )
  );
  unless($response->is_success) {
    warn "Couldn't get $url: ", $response->status_line, "\n";
    return;
  }

  my $tree = HTML::TreeBuilder->new();
  $tree->parse($response->content);
  $tree->eof;

  my @out;
  foreach my $link (
    $tree->look_down(   # !
      '_tag', 'a',
      sub {
        return unless $_[0]->attr('href');
        my @c = $_[0]->content_list;
        @c == 1 and ref $c[0] and $c[0]->tag eq 'b';
      }
    )
  ) {
    push @out, [ $link->attr('href'), $link->as_text ];
  }

  warn "Odd, fewer than 6 stories in $url!" if @out < 6;
  $tree->delete;
  return @out;
}
```

...and add a bit of code to actually call that routine and display the results...

```
foreach my $section (qw[tc sc hl wl en]) {
  my @links = get_headlines(
    "http://dailynews.yahoo.com/h/$section/"
  );
  print
    $section, ": ", scalar(@links), " stories\n",
    map("  ", $_->[0], " : ", $_->[1], "\n"), @links),
    "\n";
}
```

And we've got our own headline-extractor service! This in and of itself isn't no amazingly useful (since if you want to see the headlines, you *can* just look at the Yahoo News pages), but it could easily be the basis for quite useful features like filtering the headlines for matching certain keywords of interest to you.

Now, one of these days, Yahoo News will decide to change its HTML template. When this happens, this will appear to the above program as there being no links that meet the given criteria; or, less likely, dozens of erroneous links will meet the criteria. In either case, the criteria will have to be changed for the new template; they may just need adjustment, or you may need to scrap them and start over.

Regardez, duvet!

It's often quite a challenge to write criteria to match the desired parts of an HTML parse tree. Very often you *can* pull it off with a simple `$tree->look_down('_tag', 'h1')`, but sometimes you do have to keep adding and refining criteria, until you might end up with complex filters like what I've shown in this article. The benefit to learning how to deal with HTML parse trees is that one main search tool, the `look_down` method, can do most of the work, making simple things easy, while still making hard things possible.

[end body of article]

[Author Credit]

Sean M. Burke (`sburke@cpan.org`) is the current maintainer of `HTML::TreeBuilder` and `HTML::Element`, both originally by Gisle Aas.

Sean adds: ''I'd like to thank the folks who listened to me ramble incessantly about HTML::TreeBuilder and HTML::Element at this year's Yet Another Perl Conference and O'Reilly Open Source Software Convention.''

BACK

Return to the HTML::Tree docs.