Fetching Web Pages, Parsing HTML, Writing Spiders & More

Perl & LWP

O'REILLY®

Sean M. Burke

Perl and LWP

Sean M. Burke

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 7

HTML Processing with Tokens

Regular expressions are powerful, but they're a painfully low-level way of dealing with HTML. You're forced to worry about spaces and newlines, single and double quotes, HTML comments, and a lot more. The next step up from a regular expression is an HTML tokenizer. In this chapter, we'll use HTML::TokeParser to extract information from HTML files. Using these techniques, you can extract information from any HTML file, and never again have to worry about character-level trivia of HTML markup.

HTML as Tokens

Your experience with HTML code probably involves seeing raw text such as this:

```
<p>Dear Diary,
<br>I'm gonna be a superstar, because I'm learning to play
the <a href="http://MyBalalaika.com">balalaika</a> &amp; the <a
href='http://MyBazouki.com'>bazouki</a>!!!
```

The HTML::TokeParser module divides the HTML into units called *tokens*, which means units of parsing. The above source code is parsed as this series of tokens:

start-tag token
: `p` with no attributes

text token
: `Dear Diary,\n`

start-tag token
: `br` with no attributes

text token
: `I'm gonna be a superstar, because I'm learning to play\nthe`

start-tag token
: `a`, with attribute `href` whose value is `http://MyBalalaika.com`

text token
: `balalaika`

end-tag token
: a

text token
: & the , which means & the

start-tag token
: a, with attribute href equals http://MyBazouki.com

text token
: bazouki

end-tag token
: a

text token
: !!!\n

This representation of things is more abstract, focusing on markup concepts and not individual characters. So whereas the two <a> tags have different types of quotes around their attribute values in the raw HTML, as tokens each has a start-tag of type a, with an href attribute of a particular value. A program that extracts information by working with a stream of tokens doesn't have to worry about the idiosyncrasies of entity encoding, whitespace, quotes, and trying to work out where a tag ends.

Basic HTML::TokeParser Use

The HTML::TokeParser module is a class for accessing HTML as tokens. An HTML::TokeParser object gives you one token at a time, much as a filehandle gives you one line at a time from a file. The HTML can be tokenized from a file or string. The tokenizer decodes entities in attributes, but not entities in text.

Create a token stream object using one of these two constructors:

```
my $stream = HTML::TokeParser->new($filename)
  || die "Couldn't read HTML file $filename: $!";
```

or:

```
my $stream = HTML::TokeParser->new( \$string_of_html );
```

Once you have that stream object, you get the next token by calling:

```
my $token = $stream->get_token( );
```

The $token variable then holds an array reference, or undef if there's nothing left in the stream's file or string. This code processes every token in a document:

```
my $stream = HTML::TokeParser->new($filename)
  || die "Couldn't read HTML file $filename: $!";

while(my $token = $stream->get_token) {
  # ... consider $token ...
}
```

The $token can have one of six kinds of values, distinguished first by the value of $token->[0], as shown in Table 7-1.

Table 7-1. Token types

Token	Values
Start-tag	["S", $tag, $attribute_hashref, $attribute_order_arrayref, $source]
End-tag	["E", $tag, $source]
Text	["T", $text, $should_not_decode]
Comment	["C", $source]
Declaration	["D", $source]
Processing instruction	["PI", $content, $source]

Start-Tag Tokens

If $token->[0] is "S", the token represents a start-tag:

```
["S",  $tag, $attribute_hash, $attribute_order_arrayref, $source]
```

The components of this token are:

`$tag`
: The tag name, in lowercase.

`$attribute_hashref`
: A reference to a hash encoding the attributes of this tag. The (lowercase) attribute names are the keys of the hash.

`$attribute_order_arrayref`
: A reference to an array of (lowercase) attribute names, in case you need to access elements in order.

`$source`
: The original HTML for this token.

The first three values are the most interesting ones, for most purposes.

For example, parsing this HTML:

```
<IMG SRC="kirk.jpg" alt="Shatner in r&ocirc;le of Kirk" WIDTH=352 height=522>
```

gives this token:

```
[
  'S',
  'img',
  { 'alt' => 'Shatner in rôle of Kirk',
     'height' => '522', 'src' => 'kirk.jpg', 'width' => '352'
  },
  [ 'src', 'alt', 'width', 'height' ],
  '<IMG SRC="kirk.jpg" alt="Shatner in r&ocirc;le of Kirk" WIDTH=352 height=522>'
]
```

Notice that the tag and attribute names have been lowercased, and the `&ocirc;` entity decoded within the `alt` attribute.

End-Tag Tokens

When `$token->[0]` is `"E"`, the token represents an end-tag:

```
[ "E", $tag, $source ]
```

The components of this tag are:

`$tag`
: The lowercase name of the tag being closed.

`$source`
: The original HTML for this token.

Parsing this HTML:

```
</A>
```

gives this token:

```
[ 'E', 'a', '</A>' ]
```

Text Tokens

When `$token->[0]` is `"T"`, the token represents text:

```
["T", $text, $should_not_decode]
```

The elements of this array are:

`$text`
: The text, which may have entities.

`$should_not_decode`
: A Boolean value true indicating that you should not decode the entities in `$text`.

Tokenizing this HTML:

```
&amp; the
```

gives this token:

```
[ 'T',
  ' &amp; the',
  ''
]
```

The empty string is a false value, indicating that there's nothing stopping us from decoding `$text` with `decode_entities( )` from HTML::Entities:

```
decode_entities($token->[1]) if $token->[2];
```

Text inside `<script>`, `<style>`, `<xmp>`, `<listing>`, and `<plaintext>` tags is not supposed to be entity-decoded. It is for such text that `$should_not_decode` is true.

Comment Tokens

When `$token->[0]` is `"C"`, you have a comment token:

```
["C", $source]
```

The `$source` component of the token holds the original HTML of the comment. Most programs that process HTML simply ignore comments.

Parsing this HTML

```
<!-- Shatner's best known r&ocirc;le -->
```

gives us this `$token` value:

```
[ 'C', #0: we're a comment
  '<!-- Shatner's best known r&ocirc;le -->'  #1: source
]
```

Markup Declaration Tokens

When `$token->[0]` is `"D"`, you have a declaration token:

```
["D", $source]
```

The `$source` element of the array is the HTML of the declaration. Declarations rarely occur in HTML, and when they do, they are rarely of any interest. Almost all programs that process HTML ignore declarations.

This HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

gives this token:

```
[ 'D',
  '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">'
]
```

Processing Instruction Tokens

When `$token->[0]` is `"PI"`, the token represents a processing instruction:

```
[ "PI", $instruction, $source ]
```

The components are:

`$instruction`
: The processing instruction stripped of initial `<?` and trailing `>`.

`$source`
: The original HTML for the processing instruction.

A processing instruction is an SGML construct rarely used in HTML. Most programs extracting information from HTML ignore processing instructions. If you do

handle processing instructions, be warned that in SGML (and thus HTML) a processing instruction ends with a greater-than (>), but in XML (and thus XHTML), a processing instruction ends with a question mark and a greater-than sign (?>).

Tokenizing:

```
<?subliminal message>
```

gives:

```
[ 'PI', 'subliminal message', '<?subliminal message>' ]
```

Individual Tokens

Now that you know the composition of the various types of tokens, let's see how to use HTML::TokeParser to write useful programs. Many problems are quite simple and require only one token at a time. Programs to solve these problems consist of a loop over all the tokens, with an `if` statement in the body of the loop identifying the interesting parts of the HTML:

```
use HTML::TokeParser;
my $stream = HTML::TokeParser->new($filename)
  || die "Couldn't read HTML file $filename: $!";
# For a string: HTML::TokeParser->new( \$string_of_html );

while (my $token = $stream->get_token) {
   if ($token->[0] eq 'T') { # text
     # process the text in $text->[1]

   } elsif ($token->[0] eq 'S') { # start-tag
     my($tagname, $attr) = @$token[1,2];
     # consider this start-tag...

   } elsif ($token->[0] eq 'E') {
     my $tagname = $token->[1];
     # consider this end-tag
   }

   # ignoring comments, declarations, and PIs
}
```

Checking Image Tags

Example 7-1 complains about any `img` tags in a document that are missing `alt`, `height`, or `width` attributes:

Example 7-1. Check <img> tags

```
while(my $token = $stream->get_token) {
  if($token->[0] eq 'S' and $token->[1] eq 'img') {
    my $i = $token->[2]; # attributes of this img tag
```

Example 7-1. Check <img> tags (continued)

```
    my @lack = grep !exists $i->{$_}, qw(alt height width);
    print "Missing for ", $i->{'src'} || "????", ": @lack\n" if @lack;
  }
}
```

When run on an HTML stream (whether from a file or a string), this outputs:

```
Missing for liza.jpg: height width
Missing for aimee.jpg: alt
Missing for laurie.jpg: alt height width
```

Identifying images has many applications: making HEAD requests to ensure the URLs are valid, or making a GET request to fetch the image and using Image::Size from CPAN to check or insert the `height` and `width` attributes.

HTML Filters

A similar `while` loop can use HTML::TokeParser as a simple code filter. You just pass through the `$source` from each token you don't mean to alter. Here's one that passes through every tag that it sees (by just printing its source as HTML::TokeParser passes it in), except for `img` start-tags, which get replaced with the content of their `alt` attributes:

```
while (my $token = $stream->get_token) {
  if ($token->[0] eq 'S') {
    if ($token->[1] eq 'img') {
      print $token->[2]{'alt'} || '';
    } else {
      print $token->[4];
    }
  }
  elsif($token->[0] eq 'E' ) { print $token->[2] }
  elsif($token->[0] eq 'T' ) { print $token->[1] }
  elsif($token->[0] eq 'C' ) { print $token->[1] }
  elsif($token->[0] eq 'D' ) { print $token->[1] }
  elsif($token->[0] eq 'PI') { print $token->[2] }
}
```

So, for example, a document consisting just of this:

```
<!-- new entry -->
<p>Dear Diary,
<br>This is me &amp; my balalaika, at BalalaikaCon 1998:
<img src="mybc1998.jpg" src="BC1998!  WHOOO!"> Rock on!</p>
```

is then spat out as this:

```
<!-- new entry -->
<p>Dear Diary,
<br>This is me &amp; my balalaika, at BalalaikaCon 1998:
BC1998!  WHOOO! Rock on!</p>
```

Token Sequences

Some problems cannot be solved with a single-token approach. Often you need to scan for a sequence of tokens. For example in Chapter 4, we extracted the Amazon sales rank from HTML like this:

```
<b>Amazon.com Sales Rank: </b> 4,070 </font><br>
```

Here we're looking for the text `Amazon.com Sales Rank:` , an end-tag for `b`, and the next token as a text token with the sales rank. To solve this, we need to check the next few tokens while being able to put them back if they're not what we expect.

To put tokens back into the stream, use the `unget_token( )` method:

```
$stream->unget_token(@next);
```

The tokens stored in `@next` will be returned to the stream. For example, to solve our Amazon problem:

```
while (my $token = $stream->get_token( )) {
  if ($token->[0] eq 'T' and
    $token->[1] eq 'Amazon.com Sales Rank: ') {
    my @next;
    push @next, $stream->get_token( );
    my $found = 0;
    if ($next[0][0] eq 'E' and $next[0][1] eq 'b') {
      push @next, $stream->get_token( );
      if ($next[1][0] eq 'T') {
        $sales_rank = $next[1][1];
        $found = 1;
      }
    }
    $stream->unget_token(@next) unless $found;
  }
}
```

If it's the text we're looking for, we cautiously explore the next tokens. If the next one is a `</b>` end-tag, check the next token to ensure that it's text. If it is, then that's the sales rank. If any of the tests fail, put the tokens back on the stream and go back to processing.

Example: BBC Headlines

Suppose, for example, that your morning ritual is to have the help come and wake you at about 11 a.m. as they bring two serving trays to your bed. On one tray there's a croissant, some *pain au chocolat*, and of course some *café au lait*, and on the other tray, your laptop with a browser window already open on each story from BBC News's front page (*http://news.bbc.co.uk*). However, the help have been getting mixed up lately and opening the stories on *The Guardian*'s web site, and that's a bit awkward, since clearly *The Guardian* is an after-lunch paper. You'd say something

about it, but one doesn't want to make a scene, so you just decide to write a program that the help can run on the laptop to find all the BBC story URLs.

So you look at the source of *http://news.bbc.co.uk* and discover that each headline link is wrapped in one of two kinds of code. There are lots of headlines in code such as these:

```
<B CLASS="h3"><A href="/hi/english/business/newsid_1576000/1576290.stm">Bank
of England mulls rate cut</A></B><BR>

<B CLASS="h3"><A href="/hi/english/uk_politics/newsid_1576000/1576541.stm">Euro
battle revived by Blair speech</A></B><BR>
```

and also some headlines in code like this:

```
<A href="/hi/english/business/newsid_1576000/1576636.stm">
  <B class="h2"> Swissair shares wiped out</B><BR>
</A>

<A href="/hi/english/world/middle_east/newsid_1576000/1576113.stm">
  <B class="h1">Mid-East blow to US anti-terror drive</B><BR>
</A>
```

(Note that the a start-tag's class value can be `h1` or `h2`.)

Studying this, you realize that this is how you find the story URLs:

- Every time there's a `B` start-tag with class value of `h3`, and then an `A` start-tag with an `href` value, save that `href`.
- Every time there's an `A` start-tag with an `href` value, a text token consisting of just whitespace, and then a `B` start-tag with a class value of `h1` or `h2`, save the first token's `href` value.

Translating the Problem into Code

We can take some shortcuts when translating this into `$stream->unget_token($token)` code. The following HTML is typical:

```
<B CLASS="h3">Top Stories</B><BR>
...
<B CLASS="h3"><A href="/hi/english/business/newsid_1576000/1576290.stm">Bank
of England mulls rate cut</A></B><BR>
```

When we see the first `B-h3` start-tag token, we think it might be the start of a `B-h3-A-href` pattern. So we get another token and see if it's an `A-href` token. It's not (it's the text token `Top Stories`), so we put it back into the stream (useful in case some other pattern we're looking for involves that being the first token), and we keep looping. Later, we see another `B-h3`, we get another token, and we inspect it to see if it's an `A-href` token. This time it is, so we process its `href` value and resume looping. There's no reason for us to put that `a-href` back, so the next iteration of the loop will resume with the next token being `Bank of England mulls rate cut`.

```
sub scan_bbc_stream {
  my($stream, $docbase) = @_;

 Token:
  while(my $token = $stream->get_token) {

    if ($token->[0] eq 'S'  and  $token->[1] eq 'b'  and
       ($token->[2]{'class'} || '') eq 'h3') {
      # The href we want is in the NEXT token... probably.
      # Like: <B CLASS="h3"><A href="magic_url_here">

      my(@next) = ($stream->get_token);

      if ($next[0] and $next[0][0] eq 'S'  and  $next[0][1] eq 'a'  and
          defined $next[0][2]{'href'} ) {
         # We found <a href="...">!  This rule matches!
         print URI->new_abs($next[0][2]{'href'}, $docbase), "\n";
         next Token;
      }
      # We get here only if we've given up on this rule:
      $stream->unget_token(@next);
    }

    # fall thru to subsequent rules here...

  }
  return;
}
```

The general form of the rule above is this: if the current token looks promising, pull off a token and see if that looks promising too. If, at any point, we see an unexpected token or hit the end of the stream, we restore what we've pulled off (held in the temporary array `@next`), and continue to try other rules. But if all the expectations in this rule are met, we make it to the part that processes this bunch of tokens (here it's just a single line, which prints the URL), and then call `next Token` to start another iteration of this loop *without* restoring the tokens that have matched this pattern. (If you are disturbed by this use of a named block and `last`ing and `next`ing around, consider that this could be written as a giant `if`/`else` statement at the risk of potentially greater damage to what's left of your sanity.)

Each such rule, then, can pull from the stream however many tokens it needs to either match or reject the pattern it's after. Either it matches and starts another iteration of this loop, or it restores the stream to exactly the way it was before this rule started pulling from it. This business of a temporary `@next` list may seem like overkill when we only have to look one token ahead, only ever looking at `$next[0]`. However, the `if` block for the next pattern (which requires looking two tokens ahead) shows how the same framework can be accommodating:

```
# Add this right after the first if-block ends.
if($token->[0] eq 'S'  and  $token->[1] eq 'a'  and
   defined $token->[2]{'href'} ) {
  # Like: <A href="magic_url_here"> <B class="h2">
```

```
  my(@next) = ($stream->get_token);
  if ($next[0] and $next[0][0] eq 'T' and $next[0][1] =~ m/^\s+/s ) {
    # We found whitespace.
    push @next, $stream->get_token;
    if ($next[1] and $next[1][0] eq 'S'  and  $next[1][1] eq 'b'  and
       ($next[1][2]{'class'} || '') =~ m/^h[12]$/s ) {
      # We found <b class="h2">!  This rule matches!
      print URI->new_abs( $token->[2]{'href'}, $docbase ), "\n";
      next Token;
    }
  }
  # We get here only if we've given up on this rule:
  $stream->unget_token(@next);
}
```

Bundling into a Program

With all that wrapped up in a pure function `scan_bbc_stream( )`, we can test it by first saving the contents of *http://news.bbc.co.uk* locally as *bbc.html* (which we probably already did to scrutinize its source code and figure out what HTML patterns surround headlines), and then calling this:

```
use strict;
use HTML::TokeParser;
use URI;

scan_bbc_stream(
  HTML::TokeParser->new('bbc.html') || die($!),
  'http://news.bbc.co.uk/' # base URL
);
```

When run, this merrily scans the local copy and say:

```
http://news.bbc.co.uk/hi/english/world/middle_east/newsid_1576000/1576113.stm
http://news.bbc.co.uk/hi/english/world/south_asia/newsid_1576000/1576186.stm
http://news.bbc.co.uk/hi/english/uk_politics/newsid_1576000/1576051.stm
http://news.bbc.co.uk/hi/english/uk/newsid_1576000/1576379.stm
http://news.bbc.co.uk/hi/english/business/newsid_1576000/1576636.stm
http://news.bbc.co.uk/sport/hi/english/in_depth/2001/england_in_zimbabwe/newsid_
1574000/1574824.stm
http://news.bbc.co.uk/hi/english/business/newsid_1576000/1576546.stm
http://news.bbc.co.uk/hi/english/uk/newsid_1576000/1576313.stm
http://news.bbc.co.uk/hi/english/uk_politics/newsid_1576000/1576541.stm
http://news.bbc.co.uk/hi/english/business/newsid_1576000/1576290.stm
http://news.bbc.co.uk/hi/english/entertainment/music/newsid_1576000/1576599.stm
http://news.bbc.co.uk/hi/english/sci/tech/newsid_1574000/1574048.stm
http://news.bbc.co.uk/hi/english/health/newsid_1576000/1576776.stm
http://news.bbc.co.uk/hi/english/in_depth/uk_politics/2001/conferences_2001/labour/
newsid_1576000/1576086.stm
```

At least that's what the program said once I got `scan_bbc_stream( )` in its final working state shown above. As I was writing it and testing bits of it, I could run and rerun the program, scanning the same local file. Then once it's working on the local

file (or files, depending on how many test cases you have), you can write the routine that gets what's at a URL, makes a stream pointing to its content, and runs a given scanner routine (such as `scan_bbc_stream( )`) on it:

```
my $browser;
BEGIN {
  use LWP::UserAgent;
  $browser = LWP::UserAgent->new;
  # and any other $browser initialization code here
}

sub url_scan {
  my($scanner, $url) = @_;
  die "What scanner function?" unless $scanner and ref($scanner) eq 'CODE';
  die "What URL?" unless $url;
  my $resp = $browser->get( $url );
  die "Error getting $url: ", $resp->status_line
    unless $resp->is_success;
  die "It's not HTML, it's ", $resp->content_type
    unless $resp->content_type eq 'text/html';

  my $stream = HTML::TokeParser->new( $resp->content_ref )
    || die "Couldn't make a stream from $url\'s content!?";
  # new( ) on a string wants a reference, and so that's what
  #  we give it!  HTTP::Response objects just happen to
  #  offer a method that returns a reference to the content.
  $scanner->($stream, $resp->base);
}
```

If you thought the contents of `$url` could be very large, you could save the contents to a temporary file, and start the stream off with `HTML::TokeParser->new($tempfile)`. With the above `url_scan( )`, to retrieve the BBC main page and scan it, you need only replace our test statement that scans the input stream, with this:

```
url_scan(\&scan_bbc_stream, 'http://news.bbc.co.uk/');
```

And then the program outputs the URLs from the live BBC main page (or will die with an error message if it can't get it). To actually complete the task of getting the printed URLs to each open a new browser instance, well, this depends on your browser and OS, but for my MS Windows laptop and Netscape, this Perl program will do it:

```
my $ns = "c:\\program files\\netscape\\communicator\\program\\netscape.exe";
die "$ns doesn't exist" unless -e $ns;
die "$ns isn't executable" unless -x $ns;
while (<>) { chomp; m/\S/ and system($ns, $_) and die $!; }
```

This is then called as:

```
C:\perlstuff> perl bbc_urls.pl | perl urls2ns.pl
```

Under Unix, the correct `system( )` command is:

```
system("netscape '$url' &")
```

More HTML::TokeParser Methods

Example 7-1 illustrates that often you aren't interested in every kind of token in a stream, but care only about tokens of a certain kind. The HTML::TokeParser interface supports this with three methods, `get_tag( )`, `get_text( )`, and `get_trimmed_text( )` that do something other than simply get the next token.

`$text_string = $stream->get_text( );`
: If the next token is text, return its value.

`$text_string = $stream->get_text('foo');`
: Return all text up to the next `foo` start-tag.

`$text_string = $stream->get_text('/bar');`
: Return all text up to the next `/bar` end-tag.

`$text = $stream->get_trimmed_text( );`
`$text = $stream->get_trimmed_text('foo');`
`$text = $stream->get_trimmed_text('/bar');`
: Like `get_text( )` calls, except with initial and final whitespace removed, and all other whitespace collapsed.

`$tag_ref = $stream->get_tag( );`
: Return the next start-tag or end-tag token.

`$tag_ref = $stream->get_tag('foo', '/bar', 'baz');`
: Return the next `foo` start-tag, `/bar` end-tag, or `baz` start-tag.

We will explain these methods in detail in the following sections.

The get_text( ) Method

The `get_text( )` syntax is:

```
$text_string = $stream->get_text( );
```

If `$stream`'s next token is text, this gets it, resolves any entities in it, and returns its string value. Otherwise, this returns an empty string.

For example, if you are parsing this snippet:

```
<h1 lang='en-GB'>Shatner Reprises Kirk R&ocirc;le</h1>
```

and have just parsed the token for `h1`, `$stream->get_text( )` returns "Shatner Reprises Kirk Rôle." If you call it again (and again and again), it will return the empty string, because the next token waiting is not a text token but an `h1` end-tag token.

The get_text( ) Method with Parameters

The syntax for `get_text( )` with parameters is:

```
$text_string = $stream->get_text('foo');
$text_string = $stream->get_text('/bar');
```

Specifying a `foo` or `/bar` parameter changes the meaning of `get_text()`. If you specify a tag, you get all the text up to the next time that tag occurs (or until the end of the file, if that tag never occurs).

For however many text tokens are found, their text values are taken, entity sequences are resolved, and they are combined and returned. (All the other sorts of tokens seen along the way are just ignored.)

Note that the tag name that you specify (whether `foo` or `/bar`) must be in lowercase.

This sounds complex, but it works out well in real use. For example, imagine you've got this snippet:

```
<h1 lang='en-GB'>Star of <cite>Star Trek</cite> in New R&ocirc;le</h1>
   <cite>American Psycho II</cite> in Production.
   <!-- I'm not making this up, folks. -->
   <br>Shatner to play FBI profiler.
```

and that you've just parsed the token for `h1`. Calling `$stream->get_text()`, simply gets `Star of` . If, however, the task you're performing is the extraction of the text content of `<h1>` elements, then what's called for is:

```
$stream->get_text('/h1')
```

This returns `Star of Star Trek in New Rôle`.

Calling:

```
$stream->get_text('br')
```

returns:

```
"Star of Star Trek in New Rôle\n  American Psycho II in Production.\n   \n  "
```

And if you instead called `$stream->get_text('schlock')` and there is no `<schlock...>` in the rest of the document, you will get `Star of Star Trek in New Rôle\n American Psycho II in Production.\n \n Shatner to play FBI profiler.\n`, plus whatever text there is in the rest of the document.

Note that this never introduces whitespace where it's not there in the original. So if you're parsing this:

```
<table>
<tr><th>Height<th>Weight<th>Shoe Size</tr>
<tr><th>6' 2"<th>180lbs<th>n/a</tr>
</table>
```

and you've just parsed the `table` token, if you call:

```
$stream->get_text('/table')
```

you'll get back:

```
"\nHeightWeightShoe Size\n6' 2"180lbsn/a\n"
```

Not all nontext tokens are ignored by `$stream->get_text()`. Some tags receive special treatment: if an `img` or `applet` tag is seen, it is treated as if it were a text token; if

it has an alt attribute, its value is used as the content of the virtual text token; otherwise, you get just the uppercase tag name in brackets: [IMG] or [APPLET]. For further information on altering and expanding this feature, see perldoc HTML::TokeParser in the documentation for the get_text method, and possibly even the surprisingly short HTML::TokeParser source code.

If you just want to turn off such special treatment for all tags:

```
$stream->{'textify'} = {}
```

This is the only case of the $object->{'thing'} syntax we'll discuss in this book. In no other case does an object require us to access its internals directly like this, because it has no method for more normal access. For more information on this particular syntax, see perldoc perlref's documentation on hash references.

The get_trimmed_text( ) Method

The syntax for the get_trimmed_text( ) method is:

```
$text = $stream->get_trimmed_text( );
$text = $stream->get_trimmed_text('foo');
$text = $stream->get_trimmed_text('/bar');
```

These work exactly like the corresponding $stream->get_text( ) calls, except any leading and trailing whitespace is removed and each sequence of whitespace is replaced with a single space.

Returning to our news example:

```
$html = <<<EOF ;
<h1 lang='en-GB'>Star of <cite>Star Trek</cite> in New R&ocirc;le</h1>
   <cite>American Psycho II</cite> in Production.
   <!-- I'm not making this up, folks. -->
   <br>Shatner to play FBI profiler.
EOF
$stream = HTML::TokeParser->new(\$html);
$stream->get_token( );                    # skip h1
```

The get_text( ) method would return Star of (with the trailing space), while the get_trimmed_text( ) method would return Star of (no trailing space).

Similarly, $stream->get_text('br') would return:

```
"Star of Star Trek in New Rôle\n  American Psycho II in Production.\n   \n  "
```

whereas $stream->get_trimmed_text ('br') would return:

```
"Star of Star Trek in New Rôle American Psycho II in Production."
```

Notice that the medial newline-space-space became a single space, and the final newline-space-space-newline-space-space was simply removed.

The caveat that get_text( ) does not introduce any new whitespace applies also to get_trimmed_text( ). So where, in the last example in get_text( ), you would have gotten \nHeightWeightShoe Size\n6' 2"180lbsn/a\n, get_trimmed_text( ) would return HeightWeightShoe Size 6' 2"180lbsn/a.

The get_tag( ) Method

The syntax for the get_tag( ) method is:

```
$tag_reference = $stream->get_tag( );
```

This returns the next start-tag or end-tag token (throwing out anything else it has to skip to get there), except while get_token( ) would return start and end-tags in these formats:

```
['S', 'hr', {'class','Ginormous'}, ['class'], '<hr class=Ginormous>']
['E', 'p' , '</P>']
```

get_tag( ) instead returns them in this format:

```
['hr', {'class','Ginormous'}, ['class'], '<hr class=Ginormous>']
['/p' , '</P>']
```

That is, the first item has been taken away, and end-tag names start with /.

Start-tags

Unless $tag->[0] begins with a /, the tag represents a start-tag:

```
[$tag, $attribute_hash, $attribute_order_arrayref, $source]
```

The components of this token are:

$tag
: The tag name, in lowercase.

$attribute_hashref
: A reference to a hash encoding the attributes of this tag. The (lowercase) attribute names are the keys of the hash.

$attribute_order_arrayref
: A reference to an array of (lowercase) attribute names, in case you need to access elements in order.

$source
: The original HTML for this token.

The first two values are the most interesting ones, for most purposes.

For example, parsing this HTML with $stream->get_tag( ) :

```
<IMG SRC="kirk.jpg" alt="Shatner in r&ocirc;le of Kirk" WIDTH=352 height=522>
```

gives this tag:

```
[
  'img',
  { 'alt' => 'Shatner in rôle of Kirk',
     'height' => '522', 'src' => 'kirk.jpg', 'width' => '352'
  },
  [ 'src', 'alt', 'width', 'height' ],
  '<IMG SRC="kirk.jpg" alt="Shatner in r&ocirc;le of Kirk" WIDTH=352 height=522>'
]
```

Notice that the tag and attribute names have been lowercased, and the `&ocirc;` entity decoded within the `alt` attribute.

End-tags

When `$tag->[0]` does begin with a `/`, the token represents an end-tag:

```
[ "/$tag", $source ]
```

The components of this tag are:

`$tag`
: The lowercase name of the tag being closed, with a leading `/`.

`$source`
: The original HTML for this token.

Parsing this HTML with `$stream->get_tag( )` :

```
</A>
```

gives this tag:

```
[ '/a', '</A>' ]
```

Note that if `get_tag( )` reads to the end of the stream and finds no tag tokens, it will return `undef`.

The get_tag( ) Method with Parameters

Pass a list of tags, to skip through the tokens until a matching tag is found:

```
$tag_reference = $stream->get_tag('foo', '/bar', 'baz');
```

This returns the next start-tag or end-tag that matches any of the strings you provide (throwing out anything it has to skip to get there). Note that the tag name(s) that you provide as parameters must be in lowercase.

If `get_tag( )` reads to the end of the stream and finds no matching tag tokens, it will return `undef`. For example, this code's `get_tag( )` looks for `img` start-tags:

```
while (my $img_tag = $stream->get_tag('img')) {
  my $i = $img_tag->[1];            # attributes of this img tag
  my @lack = grep !exists $i->{$_}, qw(alt height width);
  print "Missing for ", $i->{'src'} || "????", ": @lack\n" if @lack;
}
```

Using Extracted Text

Consider the BBC story-link extractor introduced earlier. Its task was to find links to stories, in either of these kinds of patterns:

```
<B CLASS="h3"><A href="/hi/english/business/newsid_1576000/1576290.stm">Bank
  of England mulls rate cut</A></B><BR>

<A href="/hi/english/world/middle_east/newsid_1576000/1576113.stm">
  <B class="h1">Mid-East blow to US anti-terror drive</B><BR>
</A>
```

and then to isolate the URL, absolutize it, and print it. But it ignores the actual link text, which starts with the next token in the stream. If we want that text, we could get the next token by calling `get_text( )`:

```
print $stream->get_text( ), "\n  ",
  URI->new_abs($next[0][2]{'href'}, $docbase), "\n";
```

That prints the text like this:

```
Bank
of England mulls rate cut
  http://news.bbc.co.uk/hi/english/business/newsid_1576000/1576290.stm
```

Note that the newline (and any indenting, if there was any) in the source hasn't been filtered out. For some applications, this makes no difference, but for neatness sake, let's keep headlines to one line each. Changing `get_text( )` to `get_trimmed_text( )` makes that happen:

```
print $stream->get_trimmed_text( ), "\n  ",
  URI->new_abs($next[0][2]{'href'}, $docbase), "\n";
Bank of England mulls rate cut
  http://news.bbc.co.uk/hi/english/business/newsid_1576000/1576290.stm
```

If the headlines are potentially quite long, we can pass them through Text::Wrap, to wrap them at 72 columns.

There's a trickier problem that occurs often with `get_text( )` or `get_trimmed_text( )`. What if the HTML we're parsing looks like this?

```
<B CLASS="h3"><A href="/unlikely/2468.stm">Shatner &amp; Kunis win Oscars
  for <cite>American Psycho II</cite> r&ocirc;les</A></B><BR>
```

If we've just parsed the `b` and the `a`, the next token in the stream is a text token, `Shatner & Kunis win Oscars for` , that's what `get_text( )` returns (`get_trimmed_text( )` returns the same thing, minus the final space). But we don't want only the first text token in the headline, we want the whole headline. So instead of defining the headline as "the next text token," we could define it as "all the text tokens until the next `</a>`." So the program changes to:

```
print $stream->get_trimmed_text('/a'), "\n  ",
  URI->new_abs($next[0][2]{'href'}, $docbase), "\n";
```

That happily prints:

```
Shatner & Kunis win Oscars for American Psycho II rôles
  http://news.bbc.co.uk/unlikely/2468.stm
```

Note that the `&` and `&ocirc;` entity references were resolved to `&` and `ô`. If you were using such a program to spit out something other than plain text (such as XML or RTF), a bare `&` and/or a bare high-bit character such as `ô` might be unacceptable, and might need escaping in some fashion. Even if you are emitting plain text, the `\xA0` (nonbreaking space) or `\xAD` (soft hyphen) characters may not be happily interpreted by whatever application you're reading the text with, in which case a `tr/\xA0/ /` and `tr/\xAD//d` are called for. If you're taking the output of `get_text()` or `get_trimmed_text()` and sending it to a system that understands only U.S. ASCII, then passing the text through a module such as Text::Unidecode might be called for to turn the `ô` into an `o`. This is not really an HTML or HTML::TokeParser matter at all, but is the sort of problem that commonly arises when extracting content from HTML and putting it into other formats.