NAME

HTML::Parser – HTML parser class

SYNOPSIS

```
use HTML::Parser ();

# Create parser object
$p = HTML::Parser->new( api_version => 3,
                        start_h => [\&start, "tagname, attr"],
                        end_h   => [\&end,   "tagname"],
                        marked_sections => 1,
                      );

# Parse document text chunk by chunk
$p->parse($chunk1);
$p->parse($chunk2);
#...
$p->eof;                 # signal end of document

# Parse directly from file
$p->parse_file("foo.html");
# or
open(my $fh, "<:utf8", "foo.html") || die;
$p->parse_file($fh);
```

DESCRIPTION

Objects of the `HTML::Parser` class will recognize markup and separate it from plain text (alias data content) in HTML documents. As different kinds of markup and text are recognized, the corresponding event handlers are invoked.

`HTML::Parser` is not a generic SGML parser. We have tried to make it able to deal with the HTML that is actually ''out there'', and it normally parses as closely as possible to the way the popular web browsers do it instead of strictly following one of the many HTML specifications from W3C. Where there is disagreement, there is often an option that you can enable to get the official behaviour.

The document to be parsed may be supplied in arbitrary chunks. This makes on-the-fly parsing as documents are received from the network possible.

If event driven parsing does not feel right for your application, you might want to use `HTML::PullParser`. This is an `HTML::Parser` subclass that allows a more conventional program structure.

METHODS

The following method is used to construct a new `HTML::Parser` object:

`$p` = HTML::Parser–>new(`%options_and_handlers`)

This class method creates a new `HTML::Parser` object and returns it. Key/value argument pairs may be provided to assign event handlers or initialize parser options. The handlers and parser options can also be set or modified later by the method calls described below.

If a top level key is in the form ''<event>_h'' (e.g., ''text_h'') then it assigns a handler to that event, otherwise it initializes a parser option. The event handler specification value must be an array reference. Multiple handlers may also be assigned with the 'handlers => [%handlers]' option. See examples below.

If **new()** is called without any arguments, it will create a parser that uses callback methods compatible with version 2 of `HTML::Parser`. See the section on ''version 2 compatibility'' below for details.

The special constructor option 'api_version => 2' can be used to initialize version 2 callbacks while still setting other options and handlers. The 'api_version => 3' option can be used if you don't want to

set any options and don't want to fall back to v2 compatible mode.

Examples:

```
$p = HTML::Parser->new(api_version => 3,
                       text_h => [ sub {...}, "dtext" ]);
```

This creates a new parser object with a text event handler subroutine that receives the original text with general entities decoded.

```
$p = HTML::Parser->new(api_version => 3,
                       start_h => [ 'my_start', "self,tokens" ]);
```

This creates a new parser object with a start event handler method that receives the `$p` and the tokens array.

```
$p = HTML::Parser->new(api_version => 3,
                       handlers => { text => [\@array, "event,text"],
                                     comment => [\@array, "event,text"],
                                   });
```

This creates a new parser object that stores the event type and the original text in `@array` for text and comment events.

The following methods feed the HTML document to the `HTML::Parser` object:

`$p->parse( $string )`
Parse `$string` as the next chunk of the HTML document. Handlers invoked should not attempt to modify the `$string` in-place until `$p->parse` returns.

If an invoked event handler aborts parsing by calling `$p->eof`, then `$p->`**parse()** will return a FALSE value. Otherwise the return value is a reference to the parser object ($p).

`$p->parse( $code_ref )`
If a code reference is passed as the argument to be parsed, then the chunks to be parsed are obtained by invoking this function repeatedly. Parsing continues until the function returns an empty (or undefined) result. When this happens `$p->eof` is automatically signaled.

Parsing will also abort if one of the event handlers calls `$p->eof`.

The effect of this is the same as:

```
while (1) {
   my $chunk = &$code_ref();
   if (!defined($chunk) || !length($chunk)) {
       $p->eof;
       return $p;
   }
   $p->parse($chunk) || return undef;
}
```

But it is more efficient as this loop runs internally in XS code.

`$p->parse_file( $file )`
Parse text directly from a file. The `$file` argument can be a filename, an open file handle, or a reference to an open file handle.

If `$file` contains a filename and the file can't be opened, then the method returns an undefined value and $! tells why it failed. Otherwise the return value is a reference to the parser object.

If a file handle is passed as the `$file` argument, then the file will normally be read until EOF, but not closed.

If an invoked event handler aborts parsing by calling `$p->eof`, then `$p->`**parse_file()** may not have

read the entire file.

On systems with multi-byte line terminators, the values passed for the offset and length argspecs may be too low if **parse_file()** is called on a file handle that is not in binary mode.

If a filename is passed in, then **parse_file()** will open the file in binary mode.

$p->eof
Signals the end of the HTML document. Calling the $p->eof method outside a handler callback will flush any remaining buffered text (which triggers the `text` event if there is any remaining text).

Calling $p->eof inside a handler will terminate parsing at that point and cause $p->parse to return a FALSE value. This also terminates parsing by $p->**parse_file()**.

After $p->eof has been called, the **parse()** and **parse_file()** methods can be invoked to feed new documents with the parser object.

The return value from **eof()** is a reference to the parser object.

Most parser options are controlled by boolean attributes. Each boolean attribute is enabled by calling the corresponding method with a TRUE argument and disabled with a FALSE argument. The attribute value is left unchanged if no argument is given. The return value from each method is the old attribute value.

Methods that can be used to get and/or set parser options are:

$p->attr_encoded
$p->attr_encoded($bool)
By default, the `attr` and `@attr` argspecs will have general entities for attribute values decoded. Enabling this attribute leaves entities alone.

$p->backquote
$p->backquote($bool)
By default, only ' and " are recognized as quote characters around attribute values. MSIE also recognizes backquotes for some reason. Enabling this attribute provides compatibility with this behaviour.

$p->boolean_attribute_value($val)
This method sets the value reported for boolean attributes inside HTML start tags. By default, the name of the attribute is also used as its value. This affects the values reported for `tokens` and `attr` argspecs.

$p->case_sensitive
$p->case_sensitive($bool)
By default, tagnames and attribute names are down-cased. Enabling this attribute leaves them as found in the HTML source document.

$p->closing_plaintext
$p->closing_plaintext($bool)
By default, "plaintext" element can never be closed. Everything up to the end of the document is parsed in CDATA mode. This historical behaviour is what at least MSIE does. Enabling this attribute makes closing "</plaintext>" tag effective and the parsing process will resume after seeing this tag. This emulates early gecko-based browsers.

$p->empty_element_tags
$p->empty_element_tags($bool)
By default, empty element tags are not recognized as such and the "/" before ">" is just treated like a normal name character (unless `strict_names` is enabled). Enabling this attribute make `HTML::Parser` recognize these tags.

Empty element tags look like start tags, but end with the character sequence "/>" instead of ">". When recognized by `HTML::Parser` they cause an artificial end event in addition to the start event. The `text` for the artificial end event will be empty and the `tokenpos` array will be undefined even though the token array will have one element containing the tag name.

$p->marked_sections
$p->marked_sections($bool)
By default, section markings like <![CDATA[...]]> are treated like ordinary text. When this attribute is enabled section markings are honoured.

There are currently no events associated with the marked section markup, but the text can be returned as skipped_text.

$p->strict_comment
$p->strict_comment($bool)
By default, comments are terminated by the first occurrence of "-->". This is the behaviour of most popular browsers (like Mozilla, Opera and MSIE), but it is not correct according to the official HTML standard. Officially, you need an even number of "--" tokens before the closing ">" is recognized and there may not be anything but whitespace between an even and an odd "--".

The official behaviour is enabled by enabling this attribute.

Enabling of 'strict_comment' also disables recognizing these forms as comments:

```
</ comment>
<! comment>
```

$p->strict_end
$p->strict_end($bool)
By default, attributes and other junk are allowed to be present on end tags in a manner that emulates MSIE's behaviour.

The official behaviour is enabled with this attribute. If enabled, only whitespace is allowed between the tagname and the final ">".

$p->strict_names
$p->strict_names($bool)
By default, almost anything is allowed in tag and attribute names. This is the behaviour of most popular browsers and allows us to parse some broken tags with invalid attribute values like:

```
<IMG SRC=newprevlstGr.gif ALT=[PREV LIST] BORDER=0>
```

By default, "LIST]" is parsed as a boolean attribute, not as part of the ALT value as was clearly intended. This is also what Mozilla sees.

The official behaviour is enabled by enabling this attribute. If enabled, it will cause the tag above to be reported as text since "LIST]" is not a legal attribute name.

$p->unbroken_text
$p->unbroken_text($bool)
By default, blocks of text are given to the text handler as soon as possible (but the parser takes care always to break text at a boundary between whitespace and non-whitespace so single words and entities can always be decoded safely). This might create breaks that make it hard to do transformations on the text. When this attribute is enabled, blocks of text are always reported in one piece. This will delay the text event until the following (non-text) event has been recognized by the parser.

Note that the offset argspec will give you the offset of the first segment of text and length is the combined length of the segments. Since there might be ignored tags in between, these numbers can't be used to directly index in the original document file.

$p->utf8_mode
$p->utf8_mode($bool)
Enable this option when parsing raw undecoded UTF-8. This tells the parser that the entities expanded for strings reported by attr, @attr and dtext should be expanded as decoded UTF-8 so they end up compatible with the surrounding text.

If `utf8_mode` is enabled then it is an error to pass strings containing characters with code above 255 to the **parse()** method, and the **parse()** method will croak if you try.

Example: The Unicode character "\x{2665}" is "\xE2\x99\xA5" when UTF-8 encoded. The character can also be represented by the entity "♥" or "&#x2665". If we feed the parser:

```
$p->parse("\xE2\x99\xA5&hearts;");
```

then `dtext` will be reported as "\xE2\x99\xA5\x{2665}" without `utf8_mode` enabled, but as "\xE2\x99\xA5\xE2\x99\xA5" when enabled. The later string is what you want.

This option is only available with perl-5.8 or better.

$p->xml_mode
$p->xml_mode($bool)
Enabling this attribute changes the parser to allow some XML constructs. This enables the behaviour controlled by individually by the `case_sensitive`, `empty_element_tags`, `strict_names` and `xml_pic` attributes and also suppresses special treatment of elements that are parsed as CDATA for HTML.

$p->xml_pic
$p->xml_pic($bool)
By default, *processing instructions* are terminated by ">". When this attribute is enabled, processing instructions are terminated by "?>" instead.

As markup and text is recognized, handlers are invoked. The following method is used to set up handlers for different events:

$p->handler(event => \&subroutine, $argspec)
$p->handler(event => $method_name, $argspec)
$p->handler(event => \@accum, $argspec)
$p->handler(event => "");
$p->handler(event => undef);
$p->handler(event);
This method assigns a subroutine, method, or array to handle an event.

Event is one of `text`, `start`, `end`, `declaration`, `comment`, `process`, `start_document`, `end_document` or `default`.

The `\&subroutine` is a reference to a subroutine which is called to handle the event.

The `$method_name` is the name of a method of `$p` which is called to handle the event.

The `@accum` is an array that will hold the event information as sub-arrays.

If the second argument is "", the event is ignored. If it is undef, the default handler is invoked for the event.

The `$argspec` is a string that describes the information to be reported for the event. Any requested information that does not apply to a specific event is passed as `undef`. If argspec is omitted, then it is left unchanged.

The return value from $p->handler is the old callback routine or a reference to the accumulator array.

Any return values from handler callback routines/methods are always ignored. A handler callback can request parsing to be aborted by invoking the $p->eof method. A handler callback is not allowed to invoke the $p->**parse()** or $p->**parse_file()** method. An exception will be raised if it tries.

Examples:

```
$p->handler(start =>  "start", 'self, attr, attrseq, text' );
```

This causes the "start" method of object `$p` to be called for 'start' events. The callback signature is $p->start(\%attr, \@attr_seq, `$text`).

```
$p->handler(start =>  \&start, 'attr, attrseq, text' );
```

This causes subroutine **start()** to be called for 'start' events. The callback signature is start(\%attr, \@attr_seq, `$text`).

```
$p->handler(start =>  \@accum, '"S", attr, attrseq, text' );
```

This causes 'start' event information to be saved in `@accum`. The array elements will be ['S', \%attr, \@attr_seq, `$text`].

```
$p->handler(start => "");
```

This causes 'start' events to be ignored. It also suppresses invocations of any default handler for start events. It is in most cases equivalent to `$p->handler(start =>` sub {}), but is more efficient. It is different from the empty-sub-handler in that `skipped_text` is not reset by it.

```
$p->handler(start => undef);
```

This causes no handler to be associated with start events. If there is a default handler it will be invoked.

Filters based on tags can be set up to limit the number of events reported. The main bottleneck during parsing is often the huge number of callbacks made from the parser. Applying filters can improve performance significantly.

The following methods control filters:

`$p->ignore_elements( @tags )`

Both the `start` event and the `end` event as well as any events that would be reported in between are suppressed. The ignored elements can contain nested occurrences of itself. Example:

```
$p->ignore_elements(qw(script style));
```

The `script` and `style` tags will always nest properly since their content is parsed in CDATA mode. For most other tags `ignore_elements` must be used with caution since HTML is often not *well formed*.

`$p->ignore_tags( @tags )`

Any `start` and `end` events involving any of the tags given are suppressed. To reset the filter (i.e. don't suppress any `start` and `end` events), call `ignore_tags` without an argument.

`$p->report_tags( @tags )`

Any `start` and `end` events involving any of the tags *not* given are suppressed. To reset the filter (i.e. report all `start` and `end` events), call `report_tags` without an argument.

Internally, the system has two filter lists, one for `report_tags` and one for `ignore_tags`, and both filters are applied. This effectively gives `ignore_tags` precedence over `report_tags`.

Examples:

```
$p->ignore_tags(qw(style));
$p->report_tags(qw(script style));
```

results in only `script` events being reported.

Argspec

Argspec is a string containing a comma-separated list that describes the information reported by the event. The following argspec identifier names can be used:

`attr`

Attr causes a reference to a hash of attribute name/value pairs to be passed.

Boolean attributes' values are either the value set by `$p->`boolean_attribute_value, or the attribute name if no value has been set by `$p->`boolean_attribute_value.

This passes undef except for `start` events.

Unless `xml_mode` or `case_sensitive` is enabled, the attribute names are forced to lower case.

General entities are decoded in the attribute values and one layer of matching quotes enclosing the attribute values is removed.

The Unicode character set is assumed for entity decoding.

`@attr`
Basically the same as `attr`, but keys and values are passed as individual arguments and the original sequence of the attributes is kept. The parameters passed will be the same as the `@attr` calculated here:

```
@attr = map { $_ => $attr->{$_} } @$attrseq;
```

assuming `$attr` and `$attrseq` here are the hash and array passed as the result of `attr` and `attrseq` argspecs.

This passes no values for events besides `start`.

`attrseq`
Attrseq causes a reference to an array of attribute names to be passed. This can be useful if you want to walk the `attr` hash in the original sequence.

This passes undef except for `start` events.

Unless `xml_mode` or `case_sensitive` is enabled, the attribute names are forced to lower case.

`column`
Column causes the column number of the start of the event to be passed. The first column on a line is 0.

`dtext`
Dtext causes the decoded text to be passed. General entities are automatically decoded unless the event was inside a CDATA section or was between literal start and end tags (`script`, `style`, `xmp`, `iframe`, `title`, `textarea` and `plaintext`).

The Unicode character set is assumed for entity decoding. With Perl version 5.6 or earlier only the Latin−1 range is supported, and entities for characters outside the range 0..255 are left unchanged.

This passes undef except for `text` events.

`event`
Event causes the event name to be passed.

The event name is one of `text`, `start`, `end`, `declaration`, `comment`, `process`, `start_document` or `end_document`.

`is_cdata`
Is_cdata causes a TRUE value to be passed if the event is inside a CDATA section or between literal start and end tags (`script`, `style`, `xmp`, `iframe`, `title`, `textarea` and `plaintext`).

if the flag is FALSE for a text event, then you should normally either use `dtext` or decode the entities yourself before the text is processed further.

`length`
Length causes the number of bytes of the source text of the event to be passed.

`line`
Line causes the line number of the start of the event to be passed. The first line in the document is 1. Line counting doesn't start until at least one handler requests this value to be reported.

`offset`
Offset causes the byte position in the HTML document of the start of the event to be passed. The first byte in the document has offset 0.

`offset_end`
: Offset_end causes the byte position in the HTML document of the end of the event to be passed. This is the same as `offset + length`.

`self`
: Self causes the current object to be passed to the handler. If the handler is a method, this must be the first element in the argspec.

  An alternative to passing self as an argspec is to register closures that capture `$self` by themselves as handlers. Unfortunately this creates circular references which prevent the HTML::Parser object from being garbage collected. Using the `self` argspec avoids this problem.

`skipped_text`
: Skipped_text returns the concatenated text of all the events that have been skipped since the last time an event was reported. Events might be skipped because no handler is registered for them or because some filter applies. Skipped text also includes marked section markup, since there are no events that can catch it.

  If an `""`–handler is registered for an event, then the text for this event is not included in `skipped_text`. Skipped text both before and after the `""`–event is included in the next reported `skipped_text`.

`tag`
: Same as `tagname`, but prefixed with "/" if it belongs to an `end` event and "!" for a declaration. The `tag` does not have any prefix for `start` events, and is in this case identical to `tagname`.

`tagname`
: This is the element name (or *generic identifier* in SGML jargon) for start and end tags. Since HTML is case insensitive, this name is forced to lower case to ease string matching.

  Since XML is case sensitive, the tagname case is not changed when `xml_mode` is enabled. The same happens if the `case_sensitive` attribute is set.

  The declaration type of declaration elements is also passed as a tagname, even if that is a bit strange. In fact, in the current implementation tagname is identical to `token0` except that the name may be forced to lower case.

`token0`
: Token0 causes the original text of the first token string to be passed. This should always be the same as `$tokens->[0]`.

  For `declaration` events, this is the declaration type.

  For `start` and `end` events, this is the tag name.

  For `process` and non-strict `comment` events, this is everything inside the tag.

  This passes undef if there are no tokens in the event.

`tokenpos`
: Tokenpos causes a reference to an array of token positions to be passed. For each string that appears in `tokens`, this array contains two numbers. The first number is the offset of the start of the token in the original `text` and the second number is the length of the token.

  Boolean attributes in a `start` event will have (0,0) for the attribute value offset and length.

  This passes undef if there are no tokens in the event (e.g., `text`) and for artificial `end` events triggered by empty element tags.

  If you are using these offsets and lengths to modify `text`, you should either work from right to left, or be very careful to calculate the changes to the offsets.

`tokens`
: Tokens causes a reference to an array of token strings to be passed. The strings are exactly as they were found in the original text, no decoding or case changes are applied.

  For `declaration` events, the array contains each word, comment, and delimited string starting with the declaration type.

  For `comment` events, this contains each sub-comment. If `$p->`strict_comments is disabled, there will be only one sub-comment.

  For `start` events, this contains the original tag name followed by the attribute name/value pairs. The values of boolean attributes will be either the value set by `$p->`boolean_attribute_value, or the attribute name if no value has been set by `$p->`boolean_attribute_value.

  For `end` events, this contains the original tag name (always one token).

  For `process` events, this contains the process instructions (always one token).

  This passes `undef` for `text` events.

`text`
: Text causes the source text (including markup element delimiters) to be passed.

`undef`
: Pass an undefined value. Useful as padding where the same handler routine is registered for multiple events.

`'...'`
: A literal string of 0 to 255 characters enclosed in single (') or double (") quotes is passed as entered.

The whole argspec string can be wrapped up in `'@{...}'` to signal that the resulting event array should be flattened. This only makes a difference if an array reference is used as the handler target. Consider this example:

```
$p->handler(text => [], 'text');
$p->handler(text => [], '@{text}']);
```

With two text events; `"foo"`, `"bar"`; then the first example will end up with [[“foo”], [“bar”]] and the second with [“foo”, “bar”] in the handler target array.

Events

Handlers for the following events can be registered:

`comment`
: This event is triggered when a markup comment is recognized.

  Example:

```
<!-- This is a comment -- -- So is this -->
```

`declaration`
: This event is triggered when a *markup declaration* is recognized.

  For typical HTML documents, the only declaration you are likely to find is <!DOCTYPE ...>.

  Example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
```

  DTDs inside <!DOCTYPE ...> will confuse HTML::Parser.

`default`
: This event is triggered for events that do not have a specific handler. You can set up a handler for this event to catch stuff you did not want to catch explicitly.

`end`
: This event is triggered when an end tag is recognized.

  Example:

```
</A>
```

`end_document`
: This event is triggered when `$p->eof` is called and after any remaining text is flushed. There is no document text associated with this event.

`process`
: This event is triggered when a processing instructions markup is recognized.

  The format and content of processing instructions are system and application dependent.

  Examples:

```
<? HTML processing instructions >
<? XML processing instructions ?>
```

`start`
: This event is triggered when a start tag is recognized.

  Example:

```
<A HREF="http://www.perl.com/">
```

`start_document`
: This event is triggered before any other events for a new document. A handler for it can be used to initialize stuff. There is no document text associated with this event.

`text`
: This event is triggered when plain text (characters) is recognized. The text may contain multiple lines. A sequence of text may be broken between several text events unless `$p->unbroken_text` is enabled.

  The parser will make sure that it does not break a word or a sequence of whitespace between two text events.

Unicode
: `HTML::Parser` can parse Unicode strings when running under perl-5.8 or better. If Unicode is passed to `$p->`**parse()** then chunks of Unicode will be reported to the handlers. The offset and length argspecs will also report their position in terms of characters.

  It is safe to parse raw undecoded UTF-8 if you either avoid decoding entities and make sure to not use *argspecs* that do, or enable the `utf8_mode` for the parser. Parsing of undecoded UTF-8 might be useful when parsing from a file where you need the reported offsets and lengths to match the byte offsets in the file.

  If a filename is passed to `$p->`**parse_file()** then the file will be read in binary mode. This will be fine if the file contains only ASCII or Latin-1 characters. If the file contains UTF-8 encoded text then care must be taken when decoding entities as described in the previous paragraph, but better is to open the file with the UTF-8 layer so that it is decoded properly:

```
open(my $fh, "<:utf8", "index.html") || die "...: $!";
$p->parse_file($fh);
```

  If the file contains text encoded in a charset besides ASCII, Latin-1 or UTF-8 then decoding will always be needed.

VERSION 2 COMPATIBILITY

When an `HTML::Parser` object is constructed with no arguments, a set of handlers is automatically provided that is compatible with the old HTML::Parser version 2 callback methods.

This is equivalent to the following method calls:

```
 $p->handler(start   => "start",   "self, tagname, attr, attrseq, text");
 $p->handler(end     => "end",     "self, tagname, text");
 $p->handler(text    => "text",    "self, text, is_cdata");
 $p->handler(process => "process", "self, token0, text");
 $p->handler(comment =>
             sub {
                 my($self, $tokens) = @_;
                 for (@$tokens) {$self->comment($_);}},
             "self, tokens");
 $p->handler(declaration =>
             sub {
                 my $self = shift;
                 $self->declaration(substr($_[0], 2, -1));},
             "self, text");
```

Setting up these handlers can also be requested with the ‘‘api_version => 2’’ constructor option.

SUBCLASSING

The `HTML::Parser` class is subclassable. Parser objects are plain hashes and `HTML::Parser` reserves only hash keys that start with ‘‘_hparser’’. The parser state can be set up by invoking the **init()** method, which takes the same arguments as **new()**.

EXAMPLES

The first simple example shows how you might strip out comments from an HTML document. We achieve this by setting up a comment handler that does nothing and a default handler that will print out anything else:

```
use HTML::Parser;
HTML::Parser->new(default_h => [sub { print shift }, 'text'],
                  comment_h => [""],
                 )->parse_file(shift || die) || die $!;
```

An alternative implementation is:

```
use HTML::Parser;
HTML::Parser->new(end_document_h => [sub { print shift },
                                     'skipped_text'],
                  comment_h      => [""],
                 )->parse_file(shift || die) || die $!;
```

This will in most cases be much more efficient since only a single callback will be made.

The next example prints out the text that is inside the <title> element of an HTML document. Here we start by setting up a start handler. When it sees the title start tag it enables a text handler that prints any text found and an end handler that will terminate parsing as soon as the title end tag is seen:

```
use HTML::Parser ();

sub start_handler
{
  return if shift ne "title";
  my $self = shift;
  $self->handler(text => sub { print shift }, "dtext");
  $self->handler(end  => sub { shift->eof if shift eq "title"; },
                         "tagname,self");
}

my $p = HTML::Parser->new(api_version => 3);
$p->handler( start => \&start_handler, "tagname,self");
$p->parse_file(shift || die) || die $!;
```

```
print "\n";
```

On a Debian box, more examples can be found in the /usr/share/doc/libhtml−parser−perl/examples directory. The program `hrefsub` shows how you can edit all links found in a document and `htextsub` how to edit the text only; the program `hstrip` shows how you can strip out certain tags/elements and/or attributes; and the program `htext` show how to obtain the plain text, but not any script/style content.

You can browse the *eg/* directory online from the *[Browse]* link on the http://search.cpan.org/~gaas/HTML−Parser/ page.

BUGS

The <style> and <script> sections do not end with the first ''</'', but need the complete corresponding end tag. The standard behaviour is not really practical.

When the *strict_comment* option is enabled, we still recognize comments where there is something other than whitespace between even and odd ''−−'' markers.

Once `$p−>`boolean_attribute_value has been set, there is no way to restore the default behaviour.

There is currently no way to get both quote characters into the same literal argspec.

Empty tags, e.g. ''<>'' and ''</>'', are not recognized. SGML allows them to repeat the previous start tag or close the previous start tag respectively.

NET tags, e.g. ''code/.../'' are not recognized. This is SGML shorthand for ''<code>...</code>''.

Unclosed start or end tags, e.g. ''<tt<b>...</b</tt>'' are not recognized.

DIAGNOSTICS

The following messages may be produced by HTML::Parser. The notation in this listing is the same as used in perldiag:

Not a reference to a hash
: (F) The object blessed into or subclassed from HTML::Parser is not a hash as required by the HTML::Parser methods.

Bad signature in parser state object at `%p`
: (F) The _hparser_xs_state element does not refer to a valid state structure. Something must have changed the internal value stored in this hash element, or the memory has been overwritten.

_hparser_xs_state element is not a reference
: (F) The _hparser_xs_state element has been destroyed.

Can't find '_hparser_xs_state' element in HTML::Parser hash
: (F) The _hparser_xs_state element is missing from the parser hash. It was either deleted, or not created when the object was created.

API version `%s` not supported by HTML::Parser `%s`
: (F) The constructor option 'api_version' with an argument greater than or equal to 4 is reserved for future extensions.

Bad constructor option '%s'
: (F) An unknown constructor option key was passed to the **new()** or **init()** methods.

Parse loop not allowed
: (F) A handler invoked the **parse()** or **parse_file()** method. This is not permitted.

marked sections not supported
: (F) The `$p−>`**marked_sections()** method was invoked in a HTML::Parser module that was compiled without support for marked sections.

Unknown boolean attribute (%d)
: (F) Something is wrong with the internal logic that set up aliases for boolean attributes.

Only code or array references allowed as handler
(F) The second argument for `$p->handler` must be either a subroutine reference, then name of a subroutine or method, or a reference to an array.

No handler for `%s` events
(F) The first argument to `$p->handler` must be a valid event name; i.e. one of ''start'', ''end'', ''text'', ''process'', ''declaration'' or ''comment''.

Unrecognized identifier `%s` in argspec
(F) The identifier is not a known argspec name. Use one of the names mentioned in the argspec section above.

Literal string is longer than 255 chars in argspec
(F) The current implementation limits the length of literals in an argspec to 255 characters. Make the literal shorter.

Backslash reserved for literal string in argspec
(F) The backslash character ''\'' is not allowed in argspec literals. It is reserved to permit quoting inside a literal in a later version.

Unterminated literal string in argspec
(F) The terminating quote character for a literal was not found.

Bad argspec (%s)
(F) Only identifier names, literals, spaces and commas are allowed in argspecs.

Missing comma separator in argspec
(F) Identifiers in an argspec must be separated with '','.

Parsing of undecoded UTF-8 will give garbage when decoding entities
(W) The first chunk parsed appears to contain undecoded UTF-8 and one or more argspecs that decode entities are used for the callback handlers.

The result of decoding will be a mix of encoded and decoded characters for any entities that expand to characters with code above 127. This is not a good thing.

The recommended solution is to apply **Encode::decode_utf8()** on the data before feeding it to the `$p->`**parse()**. For `$p->`**parse_file()** pass a file that has been opened in ''':utf8'' mode.

The alternative solution is to enable the `utf8_mode` and not decode before passing strings to `$p->`**parse()**. The parser can process raw undecoded UTF-8 sanely if the `utf8_mode` is enabled, or if the ''attr'', ''@attr'' or ''dtext'' argspecs are avoided.

Parsing string decoded with wrong endianness
(W) The first character in the document is U+FFFE. This is not a legal Unicode character but a byte swapped BOM. The result of parsing will likely be garbage.

Parsing of undecoded UTF-32
(W) The parser found the Unicode UTF-32 BOM signature at the start of the document. The result of parsing will likely be garbage.

Parsing of undecoded UTF-16
(W) The parser found the Unicode UTF-16 BOM signature at the start of the document. The result of parsing will likely be garbage.

SEE ALSO

HTML::Entities, HTML::PullParser, HTML::TokeParser, HTML::HeadParser, HTML::LinkExtor, HTML::Form

HTML::TreeBuilder (part of the *HTML-Tree* distribution)

<http://www.w3.org/TR/html4/>

More information about marked sections and processing instructions may be found at <http://www.is-thought.co.uk/book/sgml-8.htm>.

COPYRIGHT