NAME

Text::Balanced - Extract delimited text sequences from strings.

SYNOPSIS

```
use Text::Balanced qw (
  extract_delimited
  extract_bracketed
  extract_quotelike
  extract_codeblock
  extract_variable
  extract_tagged
  extract_multiple

  gen_delimited_pat
  gen_extract_tagged
        );

# Extract the initial substring of $text that is delimited by
# two (unescaped) instances of the first character in $delim.

($extracted, $remainder) = extract_delimited($text,$delim);

# Extract the initial substring of $text that is bracketed
# with a delimiter(s) specified by $delim (where the string
# in $delim contains one or more of '(){}[]<>').

($extracted, $remainder) = extract_bracketed($text,$delim);

# Extract the initial substring of $text that is bounded by
# an XML tag.

($extracted, $remainder) = extract_tagged($text);

# Extract the initial substring of $text that is bounded by
# a C<BEGIN>...C<END> pair. Don't allow nested C<BEGIN> tags

($extracted, $remainder) =
 extract_tagged($text,"BEGIN","END",undef,{bad=>["BEGIN"]});

# Extract the initial substring of $text that represents a
# Perl "quote or quote-like operation"

($extracted, $remainder) = extract_quotelike($text);

# Extract the initial substring of $text that represents a block
# of Perl code, bracketed by any of character(s) specified by $delim
# (where the string $delim contains one or more of '(){}[]<>').

($extracted, $remainder) = extract_codeblock($text,$delim);

# Extract the initial substrings of $text that would be extracted by
```

```
# one or more sequential applications of the specified functions
# or regular expressions

@extracted = extract_multiple($text,
         [ \&extract_bracketed,
    \&extract_quotelike,
    \&some_other_extractor_sub,
    qr/[xyz]*/,
    'literal',
         ]);
```

Create a string representing an optimized pattern (a la Friedl) # that matches a substring delimited by any of the specified characters # (in this case: any type of quote or a slash)

```
$patstring = gen_delimited_pat(q{'"`/});
```

Generate a reference to an anonymous sub that is just like extract_tagged # but pre-compiled and optimized for a specific pair of tags, and consequently # much faster (i.e. 3 times faster). It uses qr// for better performance on # repeated calls, so it only works under Perl 5.005 or later.

```
$extract_head = gen_extract_tagged('<HEAD>','</HEAD>');

($extracted, $remainder) = $extract_head->($text);
```

DESCRIPTION

The various `extract_...` subroutines may be used to extract a delimited substring, possibly after skipping a specified prefix string. By default, that prefix is optional whitespace (`/\s*/`), but you can change it to whatever you wish (see below).

The substring to be extracted must appear at the current `pos` location of the string's variable (or at index zero, if no `pos` position is defined). In other words, the `extract_...` subroutines *don't* extract the first occurrence of a substring anywhere in a string (like an unanchored regex would). Rather, they extract an occurrence of the substring appearing immediately at the current matching position in the string (like a `\G`-anchored regex would).

General behaviour in list contexts

In a list context, all the subroutines return a list, the first three elements of which are always:

[0]

The extracted string, including the specified delimiters. If the extraction fails `undef` is returned.

[1]

The remainder of the input string (i.e. the characters after the extracted string). On failure, the entire string is returned.

[2]

The skipped prefix (i.e. the characters before the extracted string). On failure, `undef` is returned.

Note that in a list context, the contents of the original input text (the first argument) are not modified in any way.

However, if the input text was passed in a variable, that variable's `pos` value is updated to point at the first character after the extracted text. That means that in a list context the various subroutines can be used much like regular expressions. For example:

```
while ( $next = (extract_quotelike($text))[0] )
{
 # process next quote-like (in $next)
}
```

General behaviour in scalar and void contexts

In a scalar context, the extracted string is returned, having first been removed from the input text. Thus, the following code also processes each quote-like operation, but actually removes them from $text:

```
while ( $next = extract_quotelike($text) )
{
 # process next quote-like (in $next)
}
```

Note that if the input text is a read-only string (i.e. a literal), no attempt is made to remove the extracted text.

In a void context the behaviour of the extraction subroutines is exactly the same as in a scalar context, except (of course) that the extracted substring is not returned.

A note about prefixes

Prefix patterns are matched without any trailing modifiers (`/gimsox` etc.) This can bite you if you're expecting a prefix specification like '.*?(?=<H1>)' to skip everything up to the first <H1> tag. Such a prefix pattern will only succeed if the <H1> tag is on the current line, since . normally doesn't match newlines.

To overcome this limitation, you need to turn on /s matching within the prefix pattern, using the `(?s)` directive: '(?s).*?(?=<H1>)'

extract_delimited

The `extract_delimited` function formalizes the common idiom of extracting a single-character-delimited substring from the start of a string. For example, to extract a single-quote delimited string, the following code is typically used:

```
($remainder = $text) =~ s/\A('(\\.|[^'])*')//s;
$extracted = $1;
```

but with `extract_delimited` it can be simplified to:

```
($extracted,$remainder) = extract_delimited($text, "'");
```

`extract_delimited` takes up to four scalars (the input text, the delimiters, a prefix pattern to be skipped, and any escape characters) and extracts the initial substring of the text that is appropriately delimited. If the delimiter string has multiple characters, the first one encountered in the text is taken to delimit the substring. The third argument specifies a prefix pattern that is to be skipped (but must be present!) before the substring is extracted. The final argument specifies the escape character to be used for each delimiter.

All arguments are optional. If the escape characters are not specified, every delimiter is escaped with a backslash (\). If the prefix is not specified, the pattern `'\s*'` - optional whitespace - is used. If the delimiter set is also not specified, the set `/["'`]/` is used. If the text to be processed is not specified either, $_ is used.

In list context, `extract_delimited` returns a array of three elements, the extracted substring (*including the surrounding delimiters*), the remainder of the text, and the skipped prefix (if any). If a suitable delimited substring is not found, the first element of the array is the empty string, the second

is the complete original text, and the prefix returned in the third element is an empty string.

In a scalar context, just the extracted substring is returned. In a void context, the extracted substring (and any prefix) are simply removed from the beginning of the first argument.

Examples:

```
 # Remove a single-quoted substring from the very beginning of $text:

  $substring = extract_delimited($text, "'", '');

 # Remove a single-quoted Pascalish substring (i.e. one in which
 # doubling the quote character escapes it) from the very
 # beginning of $text:

  $substring = extract_delimited($text, "'", '', "'");

 # Extract a single- or double- quoted substring from the
 # beginning of $text, optionally after some whitespace
 # (note the list context to protect $text from modification):

  ($substring) = extract_delimited $text, q{"'};

 # Delete the substring delimited by the first '/' in $text:

  $text = join '', (extract_delimited($text,'/','[^/]*')[2,1];
```

Note that this last example is *not* the same as deleting the first quote-like pattern. For instance, if `$text` contained the string:

```
 "if ('./cmd' =~ m/$UNIXCMD/s) { $cmd = $1; }"
```

then after the deletion it would contain:

```
 "if ('.$UNIXCMD/s) { $cmd = $1; }"
```

not:

```
 "if ('./cmd' =~ ms) { $cmd = $1; }"
```

See *extract_quotelike* for a (partial) solution to this problem.

extract_bracketed

Like `"extract_delimited"`, the `extract_bracketed` function takes up to three optional scalar arguments: a string to extract from, a delimiter specifier, and a prefix pattern. As before, a missing prefix defaults to optional whitespace and a missing text defaults to `$_`. However, a missing delimiter specifier defaults to `'{}()[]<>'` (see below).

`extract_bracketed` extracts a balanced-bracket-delimited substring (using any one (or more) of the user-specified delimiter brackets: '(..)', '{..}', '[..]', or '<..>'). Optionally it will also respect quoted unbalanced brackets (see below).

A "delimiter bracket" is a bracket in list of delimiters passed as `extract_bracketed`'s second argument. Delimiter brackets are specified by giving either the left or right (or both!) versions of the required bracket(s). Note that the order in which two or more delimiter brackets are specified is not

significant.A "balanced-bracket-delimited substring" is a substring bounded by matched brackets, such that any other (left or right) delimiter bracket *within* the substring is also matched by an opposite (right or left) delimiter bracket *at the same level of nesting*. Any type of bracket not in the delimiter list is treated as an ordinary character.

In other words, each type of bracket specified as a delimiter must be balanced and correctly nested within the substring, and any other kind of ("non-delimiter") bracket in the substring is ignored.

For example, given the string:

```
$text = "{ an '[irregularly :-(] {} parenthesized >:-)' string }";
```

then a call to `extract_bracketed` in a list context:

```
@result = extract_bracketed( $text, '{}' );
```

would return:

```
( "{ an '[irregularly :-(] {} parenthesized >:-)' string }" , "" , "" )
```

since both sets of `'{..}'` brackets are properly nested and evenly balanced. (In a scalar context just the first element of the array would be returned. In a void context, `$text` would be replaced by an empty string.)

Likewise the call in:

```
@result = extract_bracketed( $text, '{[' );
```

would return the same result, since all sets of both types of specified delimiter brackets are correctly nested and balanced.

However, the call in:

```
@result = extract_bracketed( $text, '{([<' );
```

would fail, returning:

```
( undef , "{ an '[irregularly :-(] {} parenthesized >:-)' string }"  );
```

because the embedded pairs of `'(..)'`s and `'[..]'`s are "cross-nested" and the embedded `'>'` is unbalanced. (In a scalar context, this call would return an empty string. In a void context, `$text` would be unchanged.)

Note that the embedded single-quotes in the string don't help in this case, since they have not been specified as acceptable delimiters and are therefore treated as non-delimiter characters (and ignored).

However, if a particular species of quote character is included in the delimiter specification, then that type of quote will be correctly handled. for example, if `$text` is:

```
$text = '<A HREF=">>>>">link</A>';
```

then

```
@result = extract_bracketed( $text, '<">' );
```

returns:

```
( '<A HREF=">>>>">', 'link</A>', "" )
```

as expected. Without the specification of " as an embedded quoter:

```
@result = extract_bracketed( $text, '<>' );
```

the result would be:

```
( '<A HREF=">', '>>>">link</A>', "" )
```

In addition to the quote delimiters ', ", and `, full Perl quote-like quoting (i.e. q{string}, qq{string}, etc) can be specified by including the letter 'q' as a delimiter. Hence:

```
@result = extract_bracketed( $text, '<q>' );
```

would correctly match something like this:

```
$text = '<leftop: conj /and/ conj>';
```

See also: "extract_quotelike" and "extract_codeblock".

extract_variable

extract_variable extracts any valid Perl variable or variable-involved expression, including scalars, arrays, hashes, array accesses, hash look-ups, method calls through objects, subroutine calls through subroutine references, etc.

The subroutine takes up to two optional arguments:

1. A string to be processed ($_ if the string is omitted or undef)
2. A string specifying a pattern to be matched as a prefix (which is to be skipped). If omitted, optional whitespace is skipped.

On success in a list context, an array of 3 elements is returned. The elements are:

[0]
: the extracted variable, or variablish expression

[1]
: the remainder of the input text,

[2]
: the prefix substring (if any),

On failure, all of these values (except the remaining text) are undef.

In a scalar context, extract_variable returns just the complete substring that matched a variablish expression. undef is returned on failure. In addition, the original input text has the returned substring (and any prefix) removed from it.

In a void context, the input text just has the matched substring (and any specified prefix) removed.

extract_tagged

extract_tagged extracts and segments text between (balanced) specified tags.

The subroutine takes up to five optional arguments:

1. A string to be processed ($_ if the string is omitted or undef)

2. A string specifying a pattern to be matched as the opening tag. If the pattern string is omitted (or `undef`) then a pattern that matches any standard XML tag is used.

3. A string specifying a pattern to be matched at the closing tag. If the pattern string is omitted (or `undef`) then the closing tag is constructed by inserting a `/` after any leading bracket characters in the actual opening tag that was matched (*not* the pattern that matched the tag). For example, if the opening tag pattern is specified as `'{{\w+}}'` and actually matched the opening tag `"{{DATA}}"`, then the constructed closing tag would be `"{{/DATA}}"`.

4. A string specifying a pattern to be matched as a prefix (which is to be skipped). If omitted, optional whitespace is skipped.

5. A hash reference containing various parsing options (see below)

The various options that can be specified are:

`reject => $listref`

The list reference contains one or more strings specifying patterns that must *not* appear within the tagged text.

For example, to extract an HTML link (which should not contain nested links) use:

```
        extract_tagged($text, '<A>', '</A>', undef, {reject =>
['<A>']} );
```

`ignore => $listref`

The list reference contains one or more strings specifying patterns that are *not* be be treated as nested tags within the tagged text (even if they would match the start tag pattern).

For example, to extract an arbitrary XML tag, but ignore "empty" elements:

```
        extract_tagged($text, undef, undef, undef, {ignore =>
['<[^>]*/>']} );
```

(also see *gen_delimited_pat* below).

`fail => $str`

The `fail` option indicates the action to be taken if a matching end tag is not encountered (i.e. before the end of the string or some `reject` pattern matches). By default, a failure to match a closing tag causes `extract_tagged` to immediately fail.

However, if the string value associated with <reject> is "MAX", then `extract_tagged` returns the complete text up to the point of failure. If the string is "PARA", `extract_tagged` returns only the first paragraph after the tag (up to the first line that is either empty or contains only whitespace characters). If the string is "", the the default behaviour (i.e. failure) is reinstated.

For example, suppose the start tag "/para" introduces a paragraph, which then continues until the next "/endpara" tag or until another "/para" tag is encountered:

```
        $text = "/para line 1\n\nline 3\n/para line 4";

        extract_tagged($text, '/para', '/endpara', undef,
                              {reject => '/para', fail => MAX );

        # EXTRACTED: "/para line 1\n\nline 3\n"
```

Suppose instead, that if no matching "/endpara" tag is found, the "/para" tag refers only to the immediately following paragraph:

```
        $text = "/para line 1\n\nline 3\n/para line 4";
```

```
        extract_tagged($text, '/para', '/endpara', undef,
                       {reject => '/para', fail => MAX );

        # EXTRACTED: "/para line 1\n"
```

Note that the specified `fail` behaviour applies to nested tags as well.

On success in a list context, an array of 6 elements is returned. The elements are:

[0]

the extracted tagged substring (including the outermost tags),

[1]

the remainder of the input text,

[2]

the prefix substring (if any),

[3]

the opening tag

[4]

the text between the opening and closing tags

[5]

the closing tag (or "" if no closing tag was found)

On failure, all of these values (except the remaining text) are `undef`.

In a scalar context, `extract_tagged` returns just the complete substring that matched a tagged text (including the start and end tags). `undef` is returned on failure. In addition, the original input text has the returned substring (and any prefix) removed from it.

In a void context, the input text just has the matched substring (and any specified prefix) removed.

gen_extract_tagged

(Note: This subroutine is only available under Perl5.005)

`gen_extract_tagged` generates a new anonymous subroutine which extracts text between (balanced) specified tags. In other words, it generates a function identical in function to `extract_tagged`.

The difference between `extract_tagged` and the anonymous subroutines generated by `gen_extract_tagged`, is that those generated subroutines:

- do not have to reparse tag specification or parsing options every time they are called (whereas `extract_tagged` has to effectively rebuild its tag parser on every call);
- make use of the new qr// construct to pre-compile the regexes they use (whereas `extract_tagged` uses standard string variable interpolation to create tag-matching patterns).

The subroutine takes up to four optional arguments (the same set as `extract_tagged` except for the string to be processed). It returns a reference to a subroutine which in turn takes a single argument (the text to be extracted from).

In other words, the implementation of `extract_tagged` is exactly equivalent to:

```
        sub extract_tagged
        {
```

```
        my $text = shift;
        $extractor = gen_extract_tagged(@_);
        return $extractor->($text);
}
```

(although `extract_tagged` is not currently implemented that way, in order to preserve pre-5.005 compatibility).

Using `gen_extract_tagged` to create extraction functions for specific tags is a good idea if those functions are going to be called more than once, since their performance is typically twice as good as the more general-purpose `extract_tagged`.

extract_quotelike

`extract_quotelike` attempts to recognize, extract, and segment any one of the various Perl quotes and quotelike operators (see *perlop(3)*) Nested backslashed delimiters, embedded balanced bracket delimiters (for the quotelike operators), and trailing modifiers are all caught. For example, in:

```
extract_quotelike 'q # an octothorpe: \# (not the end of the q!) #'

extract_quotelike '  "You said, \"Use sed\"."  '

extract_quotelike ' s{([A-Z]{1,8}\.[A-Z]{3})} /\L$1\E/; '

extract_quotelike ' tr/\\\/\\\\/\\\//ds; '
```

the full Perl quotelike operations are all extracted correctly.

Note too that, when using the /x modifier on a regex, any comment containing the current pattern delimiter will cause the regex to be immediately terminated. In other words:

```
'm /
        (?i)            # CASE INSENSITIVE
        [a-z_]          # LEADING ALPHABETIC/UNDERSCORE
        [a-z0-9]*       # FOLLOWED BY ANY NUMBER OF ALPHANUMERICS
   /x'
```

will be extracted as if it were:

```
'm /
        (?i)            # CASE INSENSITIVE
        [a-z_]          # LEADING ALPHABETIC/'
```

This behaviour is identical to that of the actual compiler.

`extract_quotelike` takes two arguments: the text to be processed and a prefix to be matched at the very beginning of the text. If no prefix is specified, optional whitespace is the default. If no text is given, `$_` is used.

In a list context, an array of 11 elements is returned. The elements are:

[0]

the extracted quotelike substring (including trailing modifiers),

[1]

the remainder of the input text,

[2]

the prefix substring (if any),

[3]

the name of the quotelike operator (if any),

[4]

the left delimiter of the first block of the operation,

[5]

the text of the first block of the operation (that is, the contents of a quote, the regex of a match or substitution or the target list of a translation),

[6]

the right delimiter of the first block of the operation,

[7]

the left delimiter of the second block of the operation (that is, if it is a `s`, `tr`, or `y`),

[8]

the text of the second block of the operation (that is, the replacement of a substitution or the translation list of a translation),

[9]

the right delimiter of the second block of the operation (if any),

[10]

the trailing modifiers on the operation (if any).

For each of the fields marked "(if any)" the default value on success is an empty string. On failure, all of these values (except the remaining text) are `undef`.

In a scalar context, `extract_quotelike` returns just the complete substring that matched a quotelike operation (or `undef` on failure). In a scalar or void context, the input text has the same substring (and any specified prefix) removed.

Examples:

```
        # Remove the first quotelike literal that appears in text

                $quotelike = extract_quotelike($text,'.*?');

        # Replace one or more leading whitespace-separated quotelike
        # literals in $_ with "<QLL>"

                do { $_ = join '<QLL>', (extract_quotelike)[2,1] } until
$@;

        # Isolate the search pattern in a quotelike operation from $text

                ($op,$pat) = (extract_quotelike $text)[3,5];
                if ($op =~ /[ms]/)
                {
                        print "search pattern: $pat\n";
                }
```

```
else
{
        print "$op is not a pattern matching operation\n";
}
```

extract_quotelike and "here documents"

`extract_quotelike` can successfully extract "here documents" from an input string, but with an important caveat in list contexts.

Unlike other types of quote-like literals, a here document is rarely a contiguous substring. For example, a typical piece of code using here document might look like this:

```
<<'EOMSG' || die;
This is the message.
EOMSG
exit;
```

Given this as an input string in a scalar context, `extract_quotelike` would correctly return the string "<<'EOMSG'\nThis is the message.\nEOMSG", leaving the string " || die;\nexit;" in the original variable. In other words, the two separate pieces of the here document are successfully extracted and concatenated.

In a list context, `extract_quotelike` would return the list

[0]

"<<'EOMSG'\nThis is the message.\nEOMSG\n" (i.e. the full extracted here document, including fore and aft delimiters),

[1]

" || die;\nexit;" (i.e. the remainder of the input text, concatenated),

[2]

"" (i.e. the prefix substring -- trivial in this case),

[3]

"<<" (i.e. the "name" of the quotelike operator)

[4]

"'EOMSG'" (i.e. the left delimiter of the here document, including any quotes),

[5]

"This is the message.\n" (i.e. the text of the here document),

[6]

"EOMSG" (i.e. the right delimiter of the here document),

[7..10]

"" (a here document has no second left delimiter, second text, second right delimiter, or trailing modifiers).

However, the matching position of the input variable would be set to "exit;" (i.e. *after* the closing delimiter of the here document), which would cause the earlier " || die;\nexit;" to be skipped in any sequence of code fragment extractions.

To avoid this problem, when it encounters a here document whilst extracting from a modifiable string, `extract_quotelike` silently rearranges the string to an equivalent piece of Perl:

```
        <<'EOMSG'
        This is the message.
        EOMSG
        || die;
        exit;
```

in which the here document *is* contiguous. It still leaves the matching position after the here document, but now the rest of the line on which the here document starts is not skipped.

To prevent <extract_quotelike> from mucking about with the input in this way (this is the only case where a list-context `extract_quotelike` does so), you can pass the input variable as an interpolated literal:

```
        $quotelike = extract_quotelike("$var");
```

extract_codeblock

`extract_codeblock` attempts to recognize and extract a balanced bracket delimited substring that may contain unbalanced brackets inside Perl quotes or quotelike operations. That is, `extract_codeblock` is like a combination of `"extract_bracketed"` and `"extract_quotelike"`.

`extract_codeblock` takes the same initial three parameters as `extract_bracketed`: a text to process, a set of delimiter brackets to look for, and a prefix to match first. It also takes an optional fourth parameter, which allows the outermost delimiter brackets to be specified separately (see below).

Omitting the first argument (input text) means process `$_` instead. Omitting the second argument (delimiter brackets) indicates that only `'{'` is to be used. Omitting the third argument (prefix argument) implies optional whitespace at the start. Omitting the fourth argument (outermost delimiter brackets) indicates that the value of the second argument is to be used for the outermost delimiters.

Once the prefix an dthe outermost opening delimiter bracket have been recognized, code blocks are extracted by stepping through the input text and trying the following alternatives in sequence:

1. Try and match a closing delimiter bracket. If the bracket was the same species as the last opening bracket, return the substring to that point. If the bracket was mismatched, return an error.
2. Try to match a quote or quotelike operator. If found, call `extract_quotelike` to eat it. If `extract_quotelike` fails, return the error it returned. Otherwise go back to step 1.
3. Try to match an opening delimiter bracket. If found, call `extract_codeblock` recursively to eat the embedded block. If the recursive call fails, return an error. Otherwise, go back to step 1.
4. Unconditionally match a bareword or any other single character, and then go back to step 1.

Examples:

```
        # Find a while loop in the text

                if ($text =~ s/.*?while\s*\{/{/)
                {
                        $loop = "while " . extract_codeblock($text);
                }

        # Remove the first round-bracketed list (which may include
        # round- or curly-bracketed code blocks or quotelike operators)
```

```
extract_codeblock $text, "(){}", '[^(]*';
```

The ability to specify a different outermost delimiter bracket is useful in some circumstances. For example, in the Parse::RecDescent module, parser actions which are to be performed only on a successful parse are specified using a `<defer:...>` directive. For example:

```
sentence: subject verb object
                <defer: {$::theVerb = $item{verb}} >
```

Parse::RecDescent uses `extract_codeblock($text, '{}<>')` to extract the code within the `<defer:...>` directive, but there's a problem.

A deferred action like this:

```
                <defer: {if ($count>10) {$count--}} >
```

will be incorrectly parsed as:

```
                <defer: {if ($count>
```

because the "less than" operator is interpreted as a closing delimiter.

But, by extracting the directive using `extract_codeblock($text, '{}', undef, '<>')` the '>' character is only treated as a delimited at the outermost level of the code block, so the directive is parsed correctly.

extract_multiple

The `extract_multiple` subroutine takes a string to be processed and a list of extractors (subroutines or regular expressions) to apply to that string.

In an array context `extract_multiple` returns an array of substrings of the original string, as extracted by the specified extractors. In a scalar context, `extract_multiple` returns the first substring successfully extracted from the original string. In both scalar and void contexts the original string has the first successfully extracted substring removed from it. In all contexts `extract_multiple` starts at the current `pos` of the string, and sets that `pos` appropriately after it matches.

Hence, the aim of of a call to `extract_multiple` in a list context is to split the processed string into as many non-overlapping fields as possible, by repeatedly applying each of the specified extractors to the remainder of the string. Thus `extract_multiple` is a generalized form of Perl's `split` subroutine.

The subroutine takes up to four optional arguments:

1. A string to be processed ($_ if the string is omitted or `undef`)

2. A reference to a list of subroutine references and/or qr// objects and/or literal strings and/or hash references, specifying the extractors to be used to split the string. If this argument is omitted (or `undef`) the list:

   ```
   [
           sub { extract_variable($_[0], '') },
           sub { extract_quotelike($_[0],'') },
           sub { extract_codeblock($_[0],'{}','') },
   ]
   ```

   is used.

3. An number specifying the maximum number of fields to return. If this argument is omitted (or

`undef`), split continues as long as possible.

If the third argument is *N*, then extraction continues until *N* fields have been successfully extracted, or until the string has been completely processed.

Note that in scalar and void contexts the value of this argument is automatically reset to 1 (under `-w`, a warning is issued if the argument has to be reset).

4. A value indicating whether unmatched substrings (see below) within the text should be skipped or returned as fields. If the value is true, such substrings are skipped. Otherwise, they are returned.

The extraction process works by applying each extractor in sequence to the text string.

If the extractor is a subroutine it is called in a list context and is expected to return a list of a single element, namely the extracted text. It may optionally also return two further arguments: a string representing the text left after extraction (like $' for a pattern match), and a string representing any prefix skipped before the extraction (like $` in a pattern match). Note that this is designed to facilitate the use of other Text::Balanced subroutines with `extract_multiple`. Note too that the value returned by an extractor subroutine need not bear any relationship to the corresponding substring of the original text (see examples below).

If the extractor is a precompiled regular expression or a string, it is matched against the text in a scalar context with a leading '\G' and the gc modifiers enabled. The extracted value is either $1 if that variable is defined after the match, or else the complete match (i.e. $&).

If the extractor is a hash reference, it must contain exactly one element. The value of that element is one of the above extractor types (subroutine reference, regular expression, or string). The key of that element is the name of a class into which the successful return value of the extractor will be blessed.

If an extractor returns a defined value, that value is immediately treated as the next extracted field and pushed onto the list of fields. If the extractor was specified in a hash reference, the field is also blessed into the appropriate class,

If the extractor fails to match (in the case of a regex extractor), or returns an empty list or an undefined value (in the case of a subroutine extractor), it is assumed to have failed to extract. If none of the extractor subroutines succeeds, then one character is extracted from the start of the text and the extraction subroutines reapplied. Characters which are thus removed are accumulated and eventually become the next field (unless the fourth argument is true, in which case they are discarded).

For example, the following extracts substrings that are valid Perl variables:

```
        @fields = extract_multiple($text,
                                   [ sub { extract_variable($_[0]) } ],
                                   undef, 1);
```

This example separates a text into fields which are quote delimited, curly bracketed, and anything else. The delimited and bracketed parts are also blessed to identify them (the "anything else" is unblessed):

```
        @fields = extract_multiple($text,
                    [
                         { Delim => sub { extract_delimited($_[0],q{'"}) }
},
                         { Brack => sub { extract_bracketed($_[0],'{}') } },
                    ]);
```

This call extracts the next single substring that is a valid Perl quotelike operator (and removes it from $text):

```
        $quotelike = extract_multiple($text,
                                      [
                                        sub { extract_quotelike($_[0]) },
                                      ], undef, 1);
```

Finally, here is yet another way to do comma-separated value parsing:

```
        @fields = extract_multiple($csv_text,
                                   [
                                         sub {
extract_delimited($_[0],q{'"}) },
                                         qr/([^,]+)(.*)/,
                                   ],
                                   undef,1);
```

The list in the second argument means: *"Try and extract a ' or " delimited string, otherwise extract anything up to a comma..."*. The undef third argument means: *"...as many times as possible..."*, and the true value in the fourth argument means *"...discarding anything else that appears (i.e. the commas)"*.

If you wanted the commas preserved as separate fields (i.e. like split does if your split pattern has capturing parentheses), you would just make the last parameter undefined (or remove it).

gen_delimited_pat

The `gen_delimited_pat` subroutine takes a single (string) argument and > builds a Friedl-style optimized regex that matches a string delimited by any one of the characters in the single argument. For example:

```
        gen_delimited_pat(q{'"})
```

returns the regex:

```
        (?:\"(?:\\\"|(?!\").)*\"|\'(?:\\\'|(?!\').)*\')
```

Note that the specified delimiters are automatically quotemeta'd.

A typical use of `gen_delimited_pat` would be to build special purpose tags for `extract_tagged`. For example, to properly ignore "empty" XML elements (which might contain quoted strings):

```
        my $empty_tag = '<(' . gen_delimited_pat(q{'"}) . '|.)+/>';

        extract_tagged($text, undef, undef, undef, {ignore => [$empty_tag]}
 );
```

`gen_delimited_pat` may also be called with an optional second argument, which specifies the "escape" character(s) to be used for each delimiter. For example to match a Pascal-style string (where ' is the delimiter and '' is a literal ' within the string):

```
        gen_delimited_pat(q{'},q{'});
```

Different escape characters can be specified for different delimiters. For example, to specify that '/' is the escape for single quotes and '%' is the escape for double quotes:

```
        gen_delimited_pat(q{'"},q{/%});
```

If more delimiters than escape chars are specified, the last escape char is used for the remaining

delimiters. If no escape char is specified for a given specified delimiter, '\' is used.

delimited_pat

Note that `gen_delimited_pat` was previously called `delimited_pat`. That name may still be used, but is now deprecated.

DIAGNOSTICS

In a list context, all the functions return `(undef,$original_text)` on failure. In a scalar context, failure is indicated by returning `undef` (in this case the input text is not modified in any way).

In addition, on failure in *any* context, the `$@` variable is set. Accessing `$@->{error}` returns one of the error diagnostics listed below. Accessing `$@->{pos}` returns the offset into the original string at which the error was detected (although not necessarily where it occurred!) Printing `$@` directly produces the error message, with the offset appended. On success, the `$@` variable is guaranteed to be `undef`.

The available diagnostics are:

`Did not find a suitable bracket: "%s"`

The delimiter provided to `extract_bracketed` was not one of `'()[]<>{}'`.

`Did not find prefix: /%s/`

A non-optional prefix was specified but wasn't found at the start of the text.

`Did not find opening bracket after prefix: "%s"`

`extract_bracketed` or `extract_codeblock` was expecting a particular kind of bracket at the start of the text, and didn't find it.

`No quotelike operator found after prefix: "%s"`

`extract_quotelike` didn't find one of the quotelike operators `q`, `qq`, `qw`, `qx`, `s`, `tr` or `y` at the start of the substring it was extracting.

`Unmatched closing bracket: "%c"`

`extract_bracketed`, `extract_quotelike` or `extract_codeblock` encountered a closing bracket where none was expected.

`Unmatched opening bracket(s): "%s"`

`extract_bracketed`, `extract_quotelike` or `extract_codeblock` ran out of characters in the text before closing one or more levels of nested brackets.

`Unmatched embedded quote (%s)`

`extract_bracketed` attempted to match an embedded quoted substring, but failed to find a closing quote to match it.

`Did not find closing delimiter to match '%s'`

`extract_quotelike` was unable to find a closing delimiter to match the one that opened the quote-like operation.

`Mismatched closing bracket: expected "%c" but found "%s"`

`extract_bracketed`, `extract_quotelike` or `extract_codeblock` found a valid bracket delimiter, but it was the wrong species. This usually indicates a nesting error, but may indicate incorrect quoting or escaping.

`No block delimiter found after quotelike "%s"`

`extract_quotelike` or `extract_codeblock` found one of the quotelike operators `q`, `qq`, `qw`, `qx`, `s`, `tr` or `y` without a suitable block after it.

`Did not find leading dereferencer`

`extract_variable` was expecting one of '$', '@', or '%' at the start of a variable, but didn't find any of them.

`Bad identifier after dereferencer`

`extract_variable` found a '$', '@', or '%' indicating a variable, but that character was not followed by a legal Perl identifier.

`Did not find expected opening bracket at %s`

`extract_codeblock` failed to find any of the outermost opening brackets that were specified.

`Improperly nested codeblock at %s`

A nested code block was found that started with a delimiter that was specified as being only to be used as an outermost bracket.

`Missing second block for quotelike "%s"`

`extract_codeblock` or `extract_quotelike` found one of the quotelike operators `s`, `tr` or `y` followed by only one block.

`No match found for opening bracket`

`extract_codeblock` failed to find a closing bracket to match the outermost opening bracket.

`Did not find opening tag: /%s/`

`extract_tagged` did not find a suitable opening tag (after any specified prefix was removed).

`Unable to construct closing tag to match: /%s/`

`extract_tagged` matched the specified opening tag and tried to modify the matched text to produce a matching closing tag (because none was specified). It failed to generate the closing tag, almost certainly because the opening tag did not start with a bracket of some kind.

`Found invalid nested tag: %s`

`extract_tagged` found a nested tag that appeared in the "reject" list (and the failure mode was not "MAX" or "PARA").

`Found unbalanced nested tag: %s`

`extract_tagged` found a nested opening tag that was not matched by a corresponding nested closing tag (and the failure mode was not "MAX" or "PARA").

`Did not find closing tag`

`extract_tagged` reached the end of the text without finding a closing tag to match the original opening tag (and the failure mode was not "MAX" or "PARA").

AUTHOR

Damian Conway (damian@conway.org)

BUGS AND IRRITATIONS

There are undoubtedly serious bugs lurking somewhere in this code, if only because parts of it give the impression of understanding a great deal more about Perl than they really do.

Bug reports and other feedback are most welcome.

COPYRIGHT