Removal of backslashes before delimiters

During the second pass, text between the starting and ending delimiters is copied to a safe location, and the \ is removed from combinations consisting of \ and delimiter–or delimiters, meaning both starting and ending delimiters will should these differ. This removal does not happen for multi-character delimiters. Note that the combination \\ is left intact, just as it was.

Starting from this step no information about the delimiters is used in parsing.

Interpolation

The next step is interpolation in the text obtained, which is now delimiter-independent. There are four different cases.

<<'EOF', m", s"', tr///, y///

No interpolation is performed.

", q//

The only interpolation is removal of \ from pairs \\.

"", ", qq//, qx//, <file*glob>

`\Q`, `\U`, `\u`, `\L`, `\l` (possibly paired with `\E`) are converted to corresponding Perl constructs. Thus, `"$foo\Qbaz$bar"` is converted to `$foo . (quotemeta("baz" . $bar))` internally. The other combinations are replaced with appropriate expansions.

Let it be stressed that *whatever falls between* `\Q` *and* `\E` is interpolated in the usual way. Something like `"\Q\\E"` has no `\E` inside. instead, it has `\Q`, `\\`, and `E`, so the result is the same as for `"\\\\E"`. As a general rule, backslashes between `\Q` and `\E` may lead to counterintuitive results. So, `"\Q\t\E"` is converted to `quotemeta("\t")`, which is the same as `"\\\t"` (since TAB is not alphanumeric). Note also that:

```
$str = '\t';
return "\Q$str";
```

may be closer to the conjectural *intention* of the writer of `"\Q\t\E"`.

Interpolated scalars and arrays are converted internally to the `join` and `.` catenation operations. Thus, `"$foo XXX '@arr'"` becomes:

```
$foo . " XXX '" . (join $", @arr) . "'";
```

All operations above are performed simultaneously, left to right.

Because the result of `"\Q STRING \E"` has all metacharacters quoted, there is no way to insert a literal `$` or `@` inside a `\Q\E` pair. If protected by \, `$` will be quoted to became `"\\\$"`; if not, it is interpreted as the start of an interpolated scalar.

Note also that the interpolation code needs to make a decision on where the interpolated scalar ends. For instance, whether `"a $b -> {c}"` really means:

```
"a " . $b . " -> {c}";
```

or:

```
"a " . $b -> {c};
```

Most of the time, the longest possible text that does not include spaces between components and which contains matching braces or brackets. because the outcome may be determined by voting based on heuristic estimators, the result is not strictly predictable. Fortunately, it's usually correct for ambiguous cases.

?RE?, /RE/, m/RE/, s/RE/foo/,

Processing of `\Q`, `\U`, `\u`, `\L`, `\l`, and interpolation happens (almost) as with `qq//` constructs, but the substitution of \ followed by RE-special chars (including \) is not performed. Moreover, inside `(?{BLOCK})`, `(?# comment )`, and a `#`-comment in a `//x`-regular expression, no processing is performed whatsoever. This is the first step at which the presence of the `//x` modifier is relevant.

Interpolation has several quirks: `$|`, `$(`, and `$)` are not interpolated, and constructs `$var[SOMETHING]` are voted (by several different estimators) to be either an array element or `$var` followed by an RE alternative.