



Exegesis 7

Feb 27, 2004 by Damian Conway

Editor's note: this document is out of date and remains here for historic interest. See Synopsis 7 (<http://dev.perl.org/perl6/doc/design/syn/S07.html>) for the current design information.

What a piece of work is Perl 6! How noble in reason! How infinite in faculty! In form how express and admirable!

– W. Shakespeare, “Hamlet” (Perl 6 revision)

Formats are Perl 5's mechanism for creating text templates with fixed-width fields. Those fields are then filled in using values from prespecified package variables. They're a useful tool for generating many types of plaintext reports – the *r* in *Perl*, if you will.

Unlike Perl 5, Perl 6 doesn't have a `format` keyword. Or the associated built-in formatting mechanism. Instead it has a `Form.pm` module. And a `form` function.

Like a Perl 5 `format` statement, the `form` function takes a series of format (or “picture”) strings, each of which is immediately followed by a suitable set of replacement values. It interpolates those values into the placeholders specified within each picture string, and returns the result.

The general idea is the same as for Perl's two other built-in string formatting functions: `sprintf` and `pack`. The first argument represents a template with *N* placeholders to be filled in, and the next *N* arguments are the data that is to be formatted and interpolated into those placeholders:

```
$text = sprintf $format_s, $datum1, $datum2, $datum3;
$text = pack $format_p, $datum1, $datum2, $datum3;
$text = form $format_f, $datum1, $datum2, $datum3;
```

Of course, these three functions use quite different mini-languages to specify the templates they fill in, and all three fill in those templates in quite distinct ways.

Apart from those differences in semantics, `form` has a syntactic difference too. With `form`, after the first *N* data arguments we're allowed to put a second format string and its corresponding data, then a third format and data, and so on:

```
$text = form $format_f1, $datum1, $datum2, $format_f2, $datum4, $format_f3, $datum5;
```

And if we prettify that function call a little, it becomes obvious that it has the same basic structure as a Perl 5 `format`:

```
form
    $format_f1,
    $datum1, $datum2, $datum3,
    $format_f2,
    $datum4,
    $format_f3,
    $datum5;
```

But the Perl 6 version is implemented as a vanilla Perl 6 subroutine, rather than hard-coded into the language with a special keyword and declaration syntax. In this respect it's rather like Perl 5's little-known `formline` function – only much, much better.

So, whereas in Perl 5 we might write:

TPRF
 (<https://www.perlfoundation.org>)
 Gold Sponsor



DuckDuckGo

(<https://duckduckgo.com>)

TPRF
 (<https://www.perlfoundation.org>)
 Silver Sponsor

webpros®

(<https://www.webpros.com/>)

TPRF
 (<https://www.perlfoundation.org>)
 Bronze Sponsor



(<https://www.proxmox.com/en>)

TPRF
 (<https://www.perlfoundation.org>)
 Bronze Sponsor



SUSE

(<https://www.suse.com/>)

TPRF
 (<https://www.perlfoundation.org>)
 Bronze Sponsor



(<https://geizhals.de/>)

TPRF
 (<https://www.perlfoundation.org>)
 Bronze Sponsor



Fastmail

(<https://www.fastmail.com/>)

in Perl 6 we could write:

And both of them would print something like:

At first glance the Perl 6 version may seem like something of a backwards step – all those extra quotation marks and commas that the Perl 5 format didn't require. But the new formatting interface does have several distinct advantages:

- ## LATEST COMMUNITY ARTICLES

Building a Lightweight Debugging Agent
(<https://dhirajpatra.medium.com/building-a-lightweight-debugging-agent-63da3215c7db?source=rss——per>
dhirajpatra.medium.com)

Is your account on blogs.perl.org registered with an [@cpan.org](https://cpan.org) email address?
(<https://blogs.perl.org/users/metalperch/26/05/at-cpan-dot-org-on-bpo.html>)

My Journey with Devel::ptkdb - Origins
(https://blogs.perl.org/users/matthew_persico/2026/05/my-journey-with-develptkdb--origins.html)
blogs.perl.org

PWC 373 Task 2: Let's Dance to Li
Division (<https://dev.to/boblied/pwc-373-task-2-lets-dance-to-list-division>)
1e0k)
 dev.to

**Enabling AddressSanitizer (ASan)
Makefile.PL for Linux Environments
(<https://dev.to/yukikimoto/enabling-addresssanitizer-asan-in-makefile-for-linux-environments-22fe>)**
dev.to

Introducing Time::Str
(<https://blogs.perl.org/users/chanrui/2026/05/introducing-timestr.html>)
blogs.perl.org

This week in PSC (224) | 2026-05-11.html)
blogs.perl.org

Weekly Challenge: Joining and splitting lists
(<https://dev.to/simongreenet/weekly-challenge-joining-and-splitting-lists-18gh>)
dev.to

Perl 🐪 Weekly #772 - PTS 2025
 (https://dev.to/szabgab/perl-week-772-pts-2025-inh)
 dev.to

 (https://www.reddit.com
perl/) links from
perl.reddit.com
(https://www.reddit.com
perl/)

(<https://www.reddit.com/r/perp/>)
Do you want AI posts in /r/perp/
(https://www.reddit.com/r/perp/comments/1rslw6z/do_you_want_ai_posts_in_rperp/)
14 points | 27 comments
(https://www.reddit.com/r/perp/comments/1rslw6z/do_you_want_posts_in_rperp/)

(<https://www.reddit.com/r/perl/conferences>) The Perl and Raku

Of course, this is Perl, not Puritanism. So those folks who happen to *like* package variables, global accumulators, and mysterious writes, can still have them. And, if they're particularly nostalgic, they can also get rid of all the quotation marks and commas, and even retain the dot as a format terminator. For example:

```
sub myster_rite {
    our ($name, $age, $ID, $comments);
    print form :interleave, <<'.'

=====
| NAME      |      AGE      | ID NUMBER |
|-----+-----+-----|
| {<<<<<<} | {|||||} | {>>>>>>} |
|-----+-----+-----|
| COMMENTS |
|-----+-----+-----|
| {|||||||||||||||||||||||||||||||||} |
|-----+-----+-----|

.
    $name,      $age,      $ID,
    $comments;

}

# and elsewhere in the same package...

($name, $age, $ID, $comments) = get_data();
myster_rite();

($name, $age, $ID, $comments) = get_more_data();
myster_rite();
```

Let's take a look...

What's in a name?

But before we do, here's a quick run-down of some of the highly arcane technical jargon we'll be using as we talk about formatting:

Format

A string that is used as a template for the creation of *text*. It will contain zero or more *fields*, usually with some literal characters and whitespace between them.

Text

A string that is created by replacing the fields of a format with specific *data* values. For example, the string that a call to `form` returns.

Field

A fixed-width slot within a format string, into which *data* will be formatted.

Data

A string or numeric value (or an array of such values) that is interpolated into a format, in order to fill in a particular field.

Single-line field

A field that interpolates only as much of its corresponding data value as will fit inside it within a single line of text.

Block field

A field that interpolates all of its corresponding data value, over a series of text lines – as many as necessary – producing a *text block*.

Text block

The column of newline-separated text lines. A text block is produced when data is formatted into a block field that is too small to contain the data in a single line

Column

The amount of space on an output device required to display one single-width character. One character will occupy one column in most cases, the most obvious exceptions being CJK double-width characters.

Editor's note: this document is out of date and remains here for historic interest. See Synopsis 7 (<http://dev.perl.org/perl6/doc/design/syn/S07.html>) for the current design information.

Why, how now, ho! From whence ariseth this?

Unlike `sprintf` and `pack`, the `form` subroutine isn't built into Perl 6. It's just a regular subroutine, defined in the `Form.pm` module:

Conference 2026, June 26-2
Greenville, SC
(<https://tprc.us/tprc-2026-g>
13 points | comment
(https://www.reddit.com/r/perl/comments/1r9xpgt/the_perl_and_u_conference_2026_june_2621
)

(<https://www.reddit.com/r/perl>
Is your account on blogs.per
registered with an @cpan.c
email address?
(<https://blogs.perl.org/users/ta/2026/05/at-cpan-dot-org-bpo.html>)
10 points | comment
(https://www.reddit.com/r/perl/comments/1tdtjz3/is_your_accoun_n_blogsp Perlorg_registered_wi

(<https://www.reddit.com/r/perl>
My Journey with Devel::ptk
Origins
(https://www.reddit.com/r/perl/comments/1tebgqg/my_jou_with_develptkdb_origins
1 point | comment
(https://www.reddit.com/r/perl/comments/1tebgqg/my_journey_develptkdb_origins/)

(<https://www.reddit.com/r/perl>
Why does a call to `ref` or
reference to `substr` produ
"LVALUE"?
(<https://stackoverflow.com/936046/2766176>)
12 points | 2 comments
(https://www.reddit.com/r/perl/comments/1tbwvf3/why_does_a_to_ref_on_a_reference_to_sub

(<https://www.reddit.com/r/perl>
Introducing Time::Str | Chri
Hansen [blogs.perl.org]
(https://blogs.perl.org/users/nsen/2026/05/introducing_timestr.html)
36 points | 1 comment
(https://www.reddit.com/r/perl/comments/1tbiov5/introducing_tir_christian_hansen_blogsp Perlc

(<https://www.reddit.com/r/perl>
The Perl Toolchain Summit 2
| Paul Johnson [blogs.perl.c
(https://blogs.perl.org/user:ul_johnson/2026/05/the-ptoolchain-summit-2026.htr
19 points | comment
(https://www.reddit.com/r/perl/mments/1tayt5f/the_perl_tool_summit_2026_paul_johnsoi

(<https://www.reddit.com/r/perl>
PTS 2026 | Leo Lapwort
[blogs.perl.org]
(https://blogs.perl.org/users/_lapworth/2026/05/pts-2026.html)

```

module Form
{
    type FormArgs ::= Str|Array|Pair;

    sub form (FormArgs *@args is context(Scalar)) returns Str
        is exported
    {
        ...
    }

    ...
}

```

That means that if we want to use `form` we need to be sure we:

```
use Form;
```

first.

Note that the above definition of `form` specifies that the subroutine takes a list of arguments (`*@args`), each of which must be a string, array or pair (`type FormArgs ::= Str|Num|Array|Pair`). And the `is context` trait specifies that each of those arguments will be evaluated in a scalar context.

That last bit is important, because normally a “slurpy” array parameter like `*@args` would impose a list context on the corresponding arguments. We don’t want that here, mainly because we’re going to want to be able to pass arrays to `form` (/pub/2004/02/27/exegesis7.html?page=3#therefore_put_you_in_your_best_array...) without having them flattened.

How called you...?

Like all Perl subroutines, `form` can be called in a variety of contexts.

When called in a scalar or list context, `form` returns a string containing the complete formatted text:

```

my $formatted_text = form $format, *@data;

@texts = ( form($format, *@data1), form($format, *@data2) ); # 2 elems

```

When called in a void context, `form` waxes lyrical about human frailty, betrayal of trust, and the pointlessness of calling out when nobody’s there to heed the reply, before dying in a highly theatrical manner.

He doth fill fields...

The format strings passed to `form` determine what the resulting formatted text looks like. Each format consists of a series of field specifiers, which are usually separated by literal characters.

`form` understands a far larger number of field specifiers than `format` did, but they’re easy to remember because they obey a small number of conventions:

- Each field is enclosed in a pair of braces.
- Within the braces, left or right angle brackets (`<` or `>`), bars (`|`), and single-quotes (`'`) indicate various types of single-line fields.
- Left or right square brackets (`[` or `]`), l’s (`l`), and double- quotes (`"`) indicate block fields of various types.
- The direction of the brackets within a field indicates the direction towards which text will be justified in that field. For example:

```

{<<<<<<<<<<<<} Justify the text to the left
{>>>>>>>>>>>>} Justify the text to the right
{>>>><<<<<<<<} Centre the text
{<<<<<>>>>>>} Fully justify the text to both margins

```

This is even true for numeric fields, which look like: `{>>>>>. <<<}` . The whole digits are right-justified before the dot and the decimals are left-justified after it.

- An `=` at either end of a field (or both ends) indicates the data interpolated into the field is to be vertically “middled” within the resulting block. That is, the text is to be centred vertically on the middle of all the lines produced by the complete format.
- An `_` at the start and/or end of a field indicates the interpolated data is to be vertically “bottomed” within the resulting block. That is, the text is to be pushed to the bottom of the lines produced by the format.

The fields are fragrant...

18 points | comment
(https://www.reddit.com/r/perl/comments/1ta0iw5/pts_2026_lec_worth_blogsperrorg/)

([https://www.reddit.com/r/perl/Perl Weekly Issue #772 - Feb 2025](https://www.reddit.com/r/perl/Perl%20Weekly%20Issue%20%23772%20-%20Feb%2025/))
(<https://perlweekly.com/archives/772.html>)

17 points | comment
(https://www.reddit.com/r/perl/comments/1t9y4eq/perl_weekly_e_772_pts_2025/)

([https://www.reddit.com/r/perl/Concerns about how developing a Perl version of the existing project might affect my car](https://www.reddit.com/r/perl/comments/1t90n6z/concerns_about_how_developing_a_perl_version_of_my_car_might_affect_my_car/))
(https://www.reddit.com/r/perl/comments/1t90n6z/concerns_about_how_developing_a_perl_version_of_my_car_might_affect_my_car/)
16 points | 11 comments
(https://www.reddit.com/r/perl/comments/1t90n6z/concerns_about_how_developing_a_perl_version_of_my_car_might_affect_my_car/)

Perl Resources

- Get Perl (<https://www.perl.org/get.html>)
- Learn About Perl (<https://www.perl.org/>)
- Get Perl Code (<https://www.perl.org/cpan.html>)
- Help Perl (<http://donate.perlfoundation.org/donate.html>)


```
print form
"...{<<<<<<<<<<<<<<<<<<<...{}}]]]]]...",
    $data1,          $data2;
```

```
...By the pricking of ... A horse!...
...                   ... A horse!...
...                   ...   My...
...                   ... kingdom...
...                   ...   for a...
...                   ...   horse!...
```

The usual reason for mixing line and block fields in this way is to allow numbered or bulleted points:

which might produce:

Editor's note: this document is out of date and remains here for historic interest. See Synopsis 7 (<http://dev.perl.org/perl6/doc/design/syn/S07.html>) for the current design information.

Obviously, as a call to `form` builds up each line of its output – extracting data from one or more data arguments and formatting it into the corresponding fields – it needs to keep track of where it's up to in each datum. It does this by progressively updating the `.pos` of each datum, in exactly the same way as a pattern match does.

And as with a pattern match, by default that updated `.pos` is only used internally and **not** preserved after the call to `form` is finished. So passing a string to `form` doesn't interfere with any other pattern matching or text formatting that we might subsequently do with that data.

But, if we want to apply a series of `form` calls to the same data we also need to be able to tell `form` to *respect* the `.pos` information of that data – to start extracting from the previously preserved `.pos` position, rather than from the start of the string.

Note that each ellipsis is a single, one-column wide Unicode HORIZONTAL ELLIPSIS character (`\c[2026]`), *not* three separate dots. The connotation of the ellipses is "*...then keep on formatting from where you previously left off, remembering there's probably still more to come...*". And the colons are the ASCII symbol most like a single character ellipsis (try tilting your head and squinting).

```
print "The best Shakespearean roles are:\n\n";  
  
for @roles -> $role {  
    print form " * {<<<<<<<<<<<<<<<<<<<<<<<<<<<} *{...<<<<<<<>>>>>>...}*"  
        , $role,  
        $disclaimer;  
}
```

```

The best Shakespearean roles are:

* Macbeth
* King Lear
* Juliet
* Othello
* Hippolyta
* Don John
* Katerina
* Richard
* Malvolio
* Bottom

*WARNING:
*This list of roles*
*constitutes a*
*personal opinion*
*only and is in no*
*way endorsed by*
*Shakespeare'R'Us. *
*It may contain*
*nuts.
*
```

Follow-on fields are similar to `^<<<<` fields in a Perl 5 format, except they don't destroy the contents of a data source; they merely change that data source's `.pos` marker.

Data, especially numeric data, is often stored in arrays. So `form` also accepts arrays as data arguments. It can do so because its parameter list is defined as:

which means that although its arguments may include one or more arrays, each such array argument is nevertheless evaluated in a scalar context. Which, in Perl 6, produces an array reference.

Once inside `form`, each array that was specified as the data source for a field is internally converted to a single string by joining it together with a newline between each element.

```
print "The best Shakespearean roles are:\n\n";

for @roles -> $role {
    print form " * {<<<<<<<<<<<<<<<<<<<<<<<<<} *{...<<<<<<<>>>>...}*",
              $role,                               $disclaimer;
}
```

we could just write:

Centred fields ({>>>><<<<} and {}}}}[[]]) likewise extract as much data as possible, and then pad both sides of it with (near) equal numbers of spaces. If the amount of padding required is not evenly divisible by 2, the one extra space is added *after* the data.

There is a second syntax for centred fields – a tip-o'-the-hat to Perl 5 formats: {|||||} and {IIIIIIII}. This variant also makes it easier to specify centering fields that are only three columns wide: {|} and {I}.

Note, however, that the behaviour of centering fields specified this way is exactly the same in every respect as the bracket-based versions, so we're free to use whichever we prefer.

Fully justified fields ({<<<<>>>>} and {[[[[]]]) extract a maximal substring and then distribute any padding as evenly as possible into the existing whitespace gaps in that data. For example:

```
print form '({<<<<<<<<>>>>>>>>})',
          "A fellow of infinite jest, of most excellent fancy";
```

would print:

```
(A fellow of infinite)
```

A fully-justified block field ({[[[[]]]) does the same across multiple lines, except that the very last line is always left-justified. Hence, this:

```
print form '({[[[[]]])',
          "All the world's a stage, And all the men and women merely players."
```

would print:

```
(All the world's a
(stage, And all
(the men and women
(merely players. )
```

By the way, with both centred fields ({>>>><<<<}) and fully justified fields ({<<<<>>>>}), the actual number of left vs right arrows is irrelevant, so long as there is at least one of each.

What, isn't too short?

One special case we need to consider is an empty set of field delimiters:

```
form 'ID number: {}'
```

This specification is treated as a two-column-wide, left-justified block field (since that seems to be the type of two-column-wide field most often required).

Other kinds of two-column (and single-column) fields can also be created using imperative field widths (/pub/2004/02/27/exegesis7.html?page=9#what_you_will_command_me_will_i_do...) and user-defined fields (/pub/2004/02/27/exegesis7.html?page=13#define_define_welleducated_infant.).

Editor's note: this document is out of date and remains here for historic interest. See Synopsis 7 (<http://dev.perl.org/perl6/doc/design/syn/S07.html>) for the current design information.

Command our present numbers be muster'd...

A field specifier of the form {>>>>.<<<<} or {}}}}.[[]] represents a decimal-aligned numeric field. The decimal marker always appears in exactly the position indicated and the rest of the number is aligned around it. The decimal places are rounded to the specific number of places indicated, but only "significant" digits are shown. For example:

```
@nums = (1, 1.2, 1.23, 11.234, 111.235, 1.0001);

print form "Thy score be: {}}}}.[[]]",
          @nums;
```

prints:

```
Thy score be: 1.0
Thy score be: 1.2
Thy score be: 1.23
Thy score be: 11.234
Thy score be: 111.235
Thy score be: 1.000
```

The points are all aligned, the minimal number of decimal places are shown, and the decimals are rounded (using the same rounding protocol that printf employs). Note in particular that, even though both 1 and 1.0001 would normally convert to the same 3-decimal-place value (1.000), a form call only shows all three

zeros in the second case since only in the second case are they “significant”.

In other words, unless we tell it otherwise (/pub/2004/02/27/exegesis7.html?page=10#but_say_thou_nought...), `form` tries to avoid displaying a number with more accuracy than it actually possesses (within the constraint that it must always show at least one decimal place).

Here are only numbers ratified.

You’re probably wondering what happens if we try to format a number that’s too large for the available places (as `123456.78` would be in the above format). Whereas `sprintf` would extend a numeric field to accommodate the number, `form` insists on preserving the specified layout; in particular, the position of the decimal point. But it obviously can’t just cut off the extra high-order digits; that would change the value:

```
Thy score be: 23456.78
```

So, instead, it indicates that the number doesn’t fit by filling the field with octothorpes (the way many spreadsheets do):

```
Thy score be: #####.###
```

Note, however, that it *is* possible to change this behaviour (/pub/2004/02/27/exegesis7.html?page=9#that_we_our_largest_bounty_may_extend...) should we need to.

It’s also possible that someone (not you, of course!) might attempt to pass a numeric field some data that isn’t numeric at all:

```
my @mixed_data = (1, 2, "three", {4=>5}, "6", "7-Up");

print form 'Thy score be: {}{}{}.[]{}',
           @mixed_data;
```

Unlike Perl itself, `form` doesn’t autoconvert non-numeric values. Instead it marks them with another special string, by filling the field with question-marks:

```
Thy score be:      1.0
Thy score be:      2.0
Thy score be:  ??????.???
Thy score be:  ??????.???
Thy score be:      6.0
Thy score be:  ??????.???
```

Note that strings per se aren’t a problem – `form` will happily convert strings that contain valid numbers, such as “6” in the above example. But it does reject strings that contain anything else besides a number (even when Perl itself would successfully convert the number – as it would for “7-Up” above).

Those who’d prefer Perl’s usual, more *laissez-faire* attitude to numerical conversion can just pre-numerify the values themselves using the unary numerification operator (shown here in its list form – `+<<` – since we have an array of values to be numerified):

```
print form 'Thy score be: {}{}{}.[]{}',
           +<< @mixed_data;
```

This version would print:

```
Thy score be:      1.0
Thy score be:      2.0
Thy score be:      0.0
Thy score be:      1.0
Thy score be:      6.0
Thy score be:      7.0
```

(The `1.0` on the fourth line appears because Perl 6 hashes numerify to the number of entries they contain).

See how the giddy multitude do point...

Of course, not everyone uses a dot for their decimal point. The other main contender is the comma, and naturally `form` supports that as well. If we specify a numeric field with a comma between the brackets:

```
@les_nums = (1, 1.2, 1.23, 11.234, 111.235, 1.0001);

print form 'Votre score est: {}{}[],[]{}',
           @les_nums;
```

the call prints:

```
Votre score est: 1,0
Votre score est: 1,2
Votre score est: 1,23
Votre score est: 11,234
Votre score est: 111,235
Votre score est: 1,000
```

In fact, `form` is extremely flexible about the characters we're allowed to use as a decimal marker: anything except an angle- or square bracket or a plus sign (/pub/2004/02/27/exegesis7.html?page=9#that_we_our_largest_bounty_may_extend...) is acceptable.

As a bonus, `form` allows us to use the specified decimal marker in the *data* as well as in the format. So this works too:

```
@les_nums = ("1", "1,2", "1,23", "11,234", "111,235", "1,0001");

print form 'Vos score est: {}]]], [{}]',
          @les_nums;
```

Or else be impudently negative...

Negative numbers work as expected, with the minus sign taking up one column of the field's allotted span:

```
@nums = ( 1, -1.2, 1.23, -11.234, 111.235, -12345.67);

print form 'Thy score be: {}]]], [{}]',
          @nums;
```

This would print:

```
Thy score be: 1.0
Thy score be: -1.2
Thy score be: 1.23
Thy score be: -11.234
Thy score be: 111.235
Thy score be: #####.###
```

However, `form` can also format numbers so that the minus sign *trails* the number. To do that we simply put an explicit minus sign inside the field specification, at the end:

```
print form 'Thy score be: {}]]], [{}-]',
          @nums;
```

which would then print:

```
Thy score be: 1.0
Thy score be: 1.2-
Thy score be: 1.23
Thy score be: 11.234-
Thy score be: 111.235
Thy score be: 12345.67-
```

`form` also understands the common financial usage where negative numbers are represented as positive numbers in parentheses. Once again, we draw an abstract picture of what we want (by putting parens at either end of the field specification):

```
print form 'Thy dividend be: {}]]], [{}]',
          @nums;
```

and `form` obliges:

```
Thy dividend be: 1.0
Thy dividend be: (1.2)
Thy dividend be: 1.23
Thy dividend be: (11.234)
Thy dividend be: 111.235
Thy dividend be: (12345.67)
```

Note that the parens have to go *inside* the field's braces. Otherwise, they're just literal parts of the format string:

```
print form 'Thy dividend be: ({}]]], [{}]',
          @nums;
```

and we'd get:

```

Thy dividend be: ( 1.0 )
Thy dividend be: ( -1.2 )
Thy dividend be: ( 1.23 )
Thy dividend be: ( -11.234 )
Thy dividend be: ( 111.235 )
Thy dividend be: ( #####.### )

```

And stand a comma 'tween their amities...

If we add so-called “thousands separators” inside a numeric field at the usual places, `form` includes them appropriately in its output. It can handle the five major formatting conventions:

```

my @nums = (0, 1, 1.1, 1.23, 4567.89, 34567.89, 234567.89, 1234567.89);

print form
  "Brittannic      Continental      Subcontinental      Tyrolean      Asiatic",
  "_____"
  "{},{}},{}},{}"
  @nums,      @nums,      @nums,      @nums,      @nums;

```

to produce:

Brittannic	Continental	Subcontinental	Tyrolean	Asiatic
0.0	0,0	0.0	0,0	0.0
1.0	1,0	1.0	1,0	1.0
1.1	1,1	1.1	1,1	1.1
1.23	1,23	1.23	1,23	1.23
4,567.89	4.567,89	4,567.89	4'567,89	4567.89
34,567.89	34.567,89	34,567.89	34'567,89	3,4567.89
234,567.89	234.567,89	2,34,567.89	234'567,89	23,4567.89
1,234,567.89	1.234.567,89	12,34,567.89	1'234'567,89	123,4567.89

It also accepts a space character as a “thousands separator” (with, of course, any decimal marker we might like):

```

print form
  "Hyperspatial",
  "_____"
  "{ } } } } : {} ",
  @nums;

```

to produce:

```

Hyperspatial
____
0:0
1:0
1:1
1:23
4 567:89
34 567:89
234 567:89
1 234 567:89

```

And gives to airy nothing a local habitation and a name

Of course, sometimes we don’t know ahead of time just where in the world our formatted numbers will be displayed. Locales were invented to address that very problem, and `form` supports them.

If we use the `:locale` option, `form` detects the current locale and converts any numerical formats it finds to the appropriate layout. For example, if we wrote:

```

@nums = ( 1, -1.2, 1.23, -11.234, 111.235, -12345.67 );

print form
  "{} , {} } , {} } . [ {} ] ",
  @nums;

```

then we’d get:

```

1.0
-1.2
1.23
-11.234
111.235
-12,345.67

```

wherever the program was run. But if we had written:

```
print form
    :locale,
    "{[,]}],)]].{[}",
    @nums;
```

then we'd get:

```
1.0
-1.2
1.23
-11.234
111.235
-12,345.67
```

or:

```
1,0
1,2-
1,23
11,23-
111,235
12.345,67-
```

or:

```
1,0
(1,2)
1,23
(11,23)
111,235
(12'345,67)
```

or whatever else the current locale indicated was the correct local layout for numbers.

That is, when the `:locale` option is specified, `form` ignores the actual decimal point, thousands separator, and negation sign we specified in the call, and instead uses the values for these markers that are returned by the POSIX `localeconv` function. That means that we can specify our numerical formatting in a style that seems natural to us, and at the same time allow the numbers to be formatted in a style that seems natural to the user.

Editor's note: this document is out of date and remains here for historic interest. See Synopsis 7 (<http://dev.perl.org/perl6/doc/design/syn/S07.html>) for the current design information.

Thou shalt have my best gown to make thee a pair...

Wait a minute...

Where exactly did we conjure that `:locale` syntax from? And what, exactly, did it create? What *is* an “option”?

Well, we're passing `:locale` as an argument to `form`, and `form`'s signature (/pub/2004/02/27/exegesis7.html?page=2#why_how_now_ho!_from_whence_ariseth_this) guarantees us that it can only accept a `Str`, or an `Array`, or a `Pair` as an argument. So an “option” must be one of those three types, and that funky `:identifier` syntax must be a constructor for the equivalent data structure.

And indeed, that's the case. An “option” is just a pair, and the funky `:identifier` syntax is just another way of writing a pair constructor.

The standard “option” syntax is:

```
:key( "value" )
```

which is identical in effect to:

```
key => "value"
```

Both specify an autoquoted key; both associate that key with a value; both evaluate to a pair object that contains the key and value. So why have a second syntax for pairs?

Because it allows us to optimize the pair constructor syntax in two different ways. The now-familiar “fat arrow” pair constructor takes a key and a value, each of which can be of any type. In contrast, the key of an “option” pair constructor can only be an identifier, which is always autoquoted...at compile-time. So, if we use the “option” syntax we're guaranteed that the key of the resulting pair is a string, that the string that contains a valid identifier, and that the compiler can check that validity before the program starts.

Moreover, whereas the “fat arrow” has only one syntax, “options” have several highly useful syntactic variations. For example, “fat arrow” pairs can be especially annoying when we want to use them to pass named boolean arguments to a subroutine. For example:

```
duel( $person1, $person2, to_death=>1, no_quarter=>1, left_handed=>1, bonetti=>1, capoferro=>1 );
```

In contrast, “options” have a special default behaviour. If we leave off their parenthesized value entirely, the implied value is 1. So we could rewrite the preceding function call as:

```
duel( $person1, $person2, :to_death, :no_quarter, :left_handed, :bonetti, :capoferro );
```

Better still, when we have a series of options, we don't have to put commas between them:

```
duel( $person1, $person2, :to_death :no_quarter :left_handed :bonetti :capoferro );
```

That makes them even more concise and uncluttered, especially in `use` statements:

```
use POSIX :errno_h :fcntl_h :time_h;
```

There are other handy “option” variants as well, all of which simply substitute the parentheses following their key for some other kind of bracket (and hence some other kind of value). The full list of “option”...err...options is:

Option syntax =====	Is equivalent to =====
:key("some value")	key => "some value"
:key	key => 1
:key{ a=>1, b=>2 }	key => { a=>1, b=>2 }
:key{ \$^arg * 2; }	key => { \$^arg * 2; }
:key[1, 2, 3, 4]	key => [1, 2, 3, 4]
:key«eat at Joe's»	key => ["eat", "at", "Joe's"]

Despite the deliberate differences in conciseness and flexibility, we can use “options” and “fat arrows” interchangeably in almost every situation where we need to construct a pair (except, of course, where the key needs to be something other than an identifier string, in which case the “fat arrow” is the only alternative). To illustrate that interchangeability, we'll use the “option” syntax throughout most of the rest of this discussion, except where using a “fat arrow” is clearly preferable for code readability.

Meanwhile, back in the fields...

Some tender money to me...

Formatting numbers gets even trickier when those numbers represent money. But `form` simply lets us specify how the local currency looks – including leading, trailing, or infix currency markers; leading, trailing, or circumfix negation markers; thousands separators; etc. – and then it formats it that way. For example:

```
my @amounts = (0, 1, 1.2345, 1234.56, -1234.56, 1234567.89);

my %format = (
  "Canadian (English)" => q/ {-}$,}}},}}}.{/},
  "Canadian (French)"  => q/ {-} }} } },[ $ }/,
  "Dutch"              => q/ { },}}},}}}.[-EUR}/,
  "German (pre-euro)"  => q/ {-}.}}}.}}},[DM}/,
  "Indian"             => q/ {-}}},}}},[ Rs}/,
  "Norwegian"          => q/ {kr -.}}}.}}},[ }/,
  "Portuguese (pre-euro)" => q/ {-}.}}}.}}}[ Esc}/,
  "Swiss"              => q/{Sfr -}'}}}'}}}.{/},
);

for %format.kv -> $nationality, $layout {
  print form "$nationality:",
    "    $layout",
    @amounts,
    "\n";
}
```

produces:

```

Swiss:
    Sfr 0.0
    Sfr 1.0
    Sfr 1.23
    Sfr 1'234.56
    Sfr -1'234.56
    Sfr 1'234'567.89

Canadian (French):
    0,0 $
    1,0 $
    1,23 $
    1 234,56 $
    -1 234,56 $
    1 234 567,89 $

Dutch:
    0.0EUR
    1.0EUR
    1.23EUR
    1,234.56EUR
    1,234.56-EUR
    1,234,567.89EUR

Norwegian:
    kr 0,0
    kr 1,0
    kr 1,23
    kr 1.234,56
    kr -1.234,56
    kr 1.234.567,89

German (pre-euro):
    0,0DM
    1,0DM
    1,23DM
    1.234,56DM
    -1.234,56DM
    1.234.567,89DM

Indian:
    0.0 Rs
    1.0 Rs
    1.23 Rs
    1,234.56 Rs
    -1,234.56 Rs
    12,34,567.89 Rs

Portuguese (pre-euro):
    0$0 Esc
    1$0 Esc
    1$23 Esc
    1.234$56 Esc
    -1.234$56 Esc
    1.234.567$89 Esc

Canadian (English):
    $0.0
    $1.0
    $1.23
    $1,234.56
    -$1,234.56
    $1,234,567.89

```

Nice, eh?

Able verbatim to rehearse...

But sometimes *too* nice. Sometimes all we want is an existing block of data laid out into columns – without any fancy reformatting or rejustification. For example, suppose we have an interesting string like this:

[illegible]

Men at some time are masters of their	G==C
fates: / the fault, dear Brutus, is not in	A==T
our genes, / but in ourselves, that we are	T=A
underlings. / Brutus and Caesar: what	A=T
should be in that 'Caesar'? / Why should	T==A
that DNA be sequenced more than yours? /	G===C
Extract them together, yours is as fair a	T==A
genome; / transcribe them, it doth become	C=G
mRNA as well; / recombine them, it is as	TA
long; clone with 'em, / Brutus will start a	AT
twin as soon as Caesar. / Now, in the names	A=T
of all the gods at once, / upon what	T==A
proteins doth our Caesar feed, / that he is	G===C
grown so great?	T==A

And that's the purpose of a *verbatim field*. A verbatim single-line field (`{ ' ' ' ' ' ' ' ' ' ' }`) grabs the next line of data it's offered and inserts as much of it as will fit in the field's width, preserving whitespace "as is". Likewise a verbatim block field (`{ " " " " " " " " " " }`) grabs every line of the data it's offered and interpolates it into the text without any reformatting or justification.

[illegible]

Men at some time are masters of their fates: / the fault, dear Brutus, is not in our genes, / but in ourselves, that we are underlings. / Brutus and Caesar: what should be in that 'Caesar'? / Why should that DNA be sequenced more than yours? / Extract them together, yours is as fair a genome; / transcribe them, it doth become mRNA as well; / recombine them, it is as long; clone with 'em, / Brutus will start a twin as soon as Caesar. / Now, in the names of all the gods at once, / upon what proteins doth our Caesar feed, / that he is grown so great?

G==C
A==T
T=A
A=T
T==A
G===C
T==A
C=G
TA
AT
A=T
T==A
G===C
T==A

[illegible]

results in gene slicing:

Men at some time are masters of their fates: / the fault, dear Brutus, is not in our genes, / but in ourselves, that we are underlings. / Brutus and Caesar: what should be in that 'Caesar'? / Why should that DNA be sequenced more than yours? / Extract them together, yours is as fair a genome; / transcribe them, it doth become mRNA as well; / recombine them, it is as long; clone with 'em, / Brutus will start a twin as soon as Caesar. / Now, in the names of all the gods at once, / upon what proteins doth our Caesar feed, / that he is grown so great?	G==C A== T A T== G===C T==A C=G TA AT A=T T==A G=== T=
---	---

rather than teratogenesis:

Men at some time are masters of their fates: / the fault, dear Brutus, is not in our genes, / but in ourselves, that we are underlings. / Brutus and Caesar: what should be in that 'Caesar'? / Why should that DNA be sequenced more than yours? / Extract them together, yours is as fair a genome; / transcribe them, it doth become mRNA as well; / recombine them, it is as long; clone with 'em, / Brutus will start a twin as soon as Caesar. / Now, in the names of all the gods at once, / upon what proteins doth our Caesar feed, / that he is grown so great?	G==C A== =T - T=A - A=T T== =A G===C T==A C=G TA AT A=T T==A G==- =C T- ==A
---	--

Editor's note: this document is out of date and remains here for historic interest. See Synopsis 7 (<http://dev.perl.org/perl6/doc/design/syn/S07.html>) for the current design information.

And now at length they overflow their banks.

It's not uncommon for a report to need a series of data fields in one column and then a second column with only single field, perhaps containing a summary or discussion of the other data. For example, we might want to produce recipes of the form:

===== [Hecate's Broth of Ambition] =====	
Preparation time: 66.6 minutes	Method: Remove the legs from the lizard, the wings from the owlet, and the tongue of the adder. Set them aside. Refrigerate the remains (they can be used to make a lovely white-meat stock). Drain the newts' eyes if using pickled. Wrap the toad toes in the bat's wool and immerse in half a pint of vegan stock in bottom of a preheated cauldron. (If you can't get a fresh vegan for the stock, a cup of boiling water poured over a vegetarian holding a sprouted onion will do). Toss in the fenny snake, then the legless lizard. Puree the tongues together and fold gradually into the mixture, stirring widdershins at all times. Allow to bubble for 45 minutes then decant into two tarnished copper chalices. Garnish each with an owlet wing, and serve immediately.
Serves: 2 doomed souls	
Ingredients: 2 snakes (1 fenny, 1 adder) 2 lizards (1 legless, 1 regular) 3 eyes of newt (fresh or pickled) 2 toad toes (canned are fine) 2 cups of bat's wool 1 dog tongue 1 common or spotted owlet	

There are several ways to achieve that effect. The most obvious is to format each column separately and then lay them out side-by-side with a pair of verbatim fields:

[illegible]

We could even chain the calls to `form` to eliminate the interim variables:

```
print form  
'===== [ {} ] ====='  
                                     $recipe,  
  
{ "}" { "  
form('Preparation time:  
    '{<<<<<<<<<<<<<<<}', $prep_time,  
  
    'Serves:  
        '{<<<<<<<<<<<<<<<}', $serves  
  
    'Ingredients:  
        '{[[]]}'', $ingredients,  
),  
form('Method:  
    '{[[]]}'',  
      $method,  
);
```

While it's impressive to be able to do that kind of nested formatting (and highly useful in extreme formatting scenarios), it's also far too ungainly for regular use. A cleaner, more maintainable solution is use a single format and just build the method column up piecemeal, like so:

[illegible]

That produces exactly the same result as the previous versions, because each follow-on `{...<<<<<<<<<...}` field in the “Method” column grabs one extra line from `$method`, and then the final follow-on `{...[[[[[[[[{` field grabs as many more as are required to lay out the rest of the contents of the variable. The only down-side is that the resulting code is still downright ugly. With all those tedious repetitions of the same variable, there’s far too much `$method` in our madness.

Having a series of follow-on fields like this – vertically continuing a single column across subsequent format lines – is so common that `form` provides a special shortcut: the `{VVVVVVVVV}` *overflow field*.

An overflow field automatically duplicates the field specification immediately above it. The important point being that, because that duplication includes copying the preceding field's data source, overflow fields don't require a separate data source of their own.

Using overflow fields, we could rewrite our quotation generator like this:

[illegible]

Which would once again produce the recipe shown earlier.

Note that the overflow fields interact equally well in formats with single-line and block fields. That's because block overflow fields have one other special feature: they're non-greedy. Unless we specify otherwise (/pub/2004/02/27/exegesis7.html?page=10#i_have_got_strength_of_limit), all types of block fields will consume their entire data source. For example, if we wrote:

[illegible]

we'd get:

Now is the winter of our discontent / Made glorious summer		
by this sun of York; / And all the clouds that lour'd upon		
our house / In		the deep bosom
of the ocean		buried. / Now
are our brows		bound with
victorious		wreaths; / Our
bruised arms		hung up for
monuments; /		Our stern
alarums	+-----+	changed to
merry		meetings, / Our
dreadful	Eat at Mrs Miggins!	marches to
delightful		measures. Grim-
visaged war	+-----+	hath smooth'd
his wrinkled		front; / And
now, instead		of mounting
barded steeds		/ To fright the
souls of		fearful
adversaries, /		He capers
nimbly in a		lady's chamber.

That's because the two `{...[[[[]]]]...}` block fields on either side of the verbatim advertisement field will eat all the data in `$speech`, leaving nothing for the final format. Then the advertisement will be centred on the two resulting columns of text.

But, block overflow fields are different. They only take as many lines as are required to fill the lines generated by the non-overflow fields in their format. So, if we changed our code to use overflows:

```
print form :layout«across»  
 '{<<<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>>>}', $speech,  
 '{vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv}',  
 '{vvvvvvvvvvvv} {=}"'=====}= {vvvvvvvvvvvv}', $advert,  
 '{vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv}';
```

we get both a cleaner specification and a more elegant result:

Now is the winter of our discontent / Made glorious summer
by this sun of York; / And all the clouds that lour'd upon
our house / In the deep bosom
of the ocean +-----+ buried. / Now
are our brows | | bound with
victorious | Eat at Mrs Miggins! | wreaths; / Our
bruised arms | | hung up for
monuments; / +-----+ Our stern
alarums changed to
merry meetings, / Our dreadful marches to delightful
measures. Grim-visaged war hath smooth'd his wrinkled
front; / And now, instead of mounting barded steeds / To
fright the souls of fearful adversaries, / He capers
nimble in a lady's chamber.

Notice that, in the third format line of the previous example, the two overflow fields on either side of the advertisement are each overflowing from the single field that's above both of them. This kind of multiple overflow is fine, but it does require that we specify *how* the various fields overflow (i.e. as two separate columns of text, or – as in this case – as a single, broken column across the page). That's the purpose of the : layout«across» option on the first line. This option is explained in detail below ([\).](http://pub/2004/02/27/exegesis7.html?page=8#lay out. lay out.)

The `{VVVVVVVV}` fields only consumed as much data from `$speech` as was required to sandwich the output lines created by the verbatim advertisement. This feature is important, because it means we can lay out a series of block fields in one column and a single overflowed field in another column without introducing ugly gaps. For example, because the `{VVVVVVVV}` fields in:

```
print form
"Name:"                                     ",
" {[[[[[[[[[[{                             ", $name,
" Biography:"                               ",
>Status:" {<<<<<<<<<<<<<<<<<<<<<<<<<"}", $bio,
" {[[[[[[[[[[{ {VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV}", $status,
" {VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV}",
"Comments:" {VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV}",
" {[[[[[[[[[[{ {VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV}", $comments;
```

only consume as much of the overflowing `$bio` field as necessary, the result is something like:

Name:	William Shakespeare
Biography:	
Status:	William Shakespeare was born on April 23, 1564 in Strathford-upon-Avon, England; he was third of eight children from Father John Shakespeare and Mother Mary Arden.
Deceased (1564-1616)	Shakespeare began his education at the age of seven when he probably attended the Strathford grammar school. The school provided Shakespeare with his formal education. The students chiefly studied Latin rhetoric, logic, and literature. His knowledge and imagination may have come from his reading of ancient authors and poetry. In November 1582, Shakespeare received a license to marry Anne Hathaway. At the time of their marriage, Shakespeare was 18 years old and Anne was 26. They had three children, the oldest Susanna, and twins- a boy, Hamneth, and a girl, Judith. Before his death on April 23 1616, William Shakespeare had written thirty-seven plays. He is generally considered the greatest playwright the world has ever known and has always been the world's most popular author.
Comments:	
Theories	
abound as to the true author of his plays. The prime alternative candidates being Sir Francis Bacon, Christopher Marlowe, or Edward de Vere	

If {VVVVVVVVVV} fields ate their entire data – the way {[[[[[[[[[[} or {IIIIIIIIII} fields do – then the output would be much less satisfactory. The first block overflow field for \$bio would have to consume the entire biography, before the comments field was even reached. So our output would be something like:

Name:
William
Shakespeare

Biography:
Status:
Deceased (1564
–1616)

William Shakespeare was born on April 23, 1564 in Strathford-upon-Avon, England; he was third of eight children from Father John Shakespeare and Mother Mary Arden. Shakespeare began his education at the age of seven when he probably attended the Strathford grammar school. The school provided Shakespeare with his formal education. The students chiefly studied Latin rhetoric, logic, and literature. His knowledge and imagination may have come from his reading of ancient authors and poetry. In November 1582, Shakespeare received a license to marry Anne Hathaway. At the time of their marriage, Shakespeare was 18 years old and Anne was 26. They had three children, the oldest Susanna, and twins– a boy, Hamneth, and a girl, Judith. Before his death on April 23 1616, William Shakespeare had written thirty-seven plays. He is generally considered the greatest playwright the world has ever known and has always been the world's most popular author.

Comments:
Theories
abound as to
the true
author of his
plays. The
prime
alternative
candidates
being Sir
Francis
Bacon,
Christopher
Marlowe, or
Edward de
Vere

Which is precisely why {VVVVVVVVVV} fields don't work that way.

Editor's note: this document is out of date and remains here for historic interest. See Synopsis 7 (<http://dev.perl.org/perl6/doc/design/syn/S07.html>) for the current design information.

Great floods have flown from simple sources...

When it comes to specifying the data source for each field in a format, `form` offers several alternatives as to where that data placed, several alternatives as to the order in which that data is extracted, and an option that lets us control how the data is fitted into each field.

A man may break a word with you, sir...

Whenever a field is passed more data than it can accommodate in a single line, `form` is forced to “break” that data somewhere.

If the field in question is W columns wide, `form` first squeezes any whitespace (as specified by the user's `:ws` option (/pub/2004/02/27/exegesis7.html?page=10#within_that_space_you_may_have_drawn_together...)) and then looks at the next W columns of the string. (Of course, that might actually correspond to less than W characters if the string contains wide characters. However, for the sake of exposition we'll pretend that all characters are one column wide here.)

`form`'s breaking algorithm then searches for a newline, a carriage return, any other whitespace character, or a hyphen. If it finds a newline or carriage return within the first W columns, it immediately breaks the data string at that point. Otherwise it locates the *last* whitespace or hyphen in the first W columns and breaks the string immediately after that space or hyphen. If it can't find anywhere suitable to break the string, it breaks it at the $(W-1)$ th column and appends a hyphen.

So, for example:

```
$data = "You can play no part but Pyramus;\nfor Pyramus is a sweet-faced man";

print form "|{[[[[]]|",
           $data;
```

prints:

```
|You can|
|play no|
|part   |
|but    |
|Pyramu-|
|s;     |
|for    |
|Pyramus|
|is a   |
|sweet- |
|faced  |
|man    |
```

Note the line-breaks after *can* (at a whitespace), *part* (after a whitespace), *sweet-* (after a hyphen), and *s;* (at a newline). Note too that *Pyramus;* doesn't fit in the field, so it has to be chopped in two and a hyphen inserted.

Of course, this particular style of line-breaking may not be suitable to all applications, and we might prefer that `form` use some other algorithm. For example, if `form` used the TeX breaking algorithm it would have broken *Pyramus;* less clumsily, yielding:

```
|You can|
|play no|
|part   |
|but    |
|Pyra-  |
|mus;   |
|for    |
|Pyramus|
|is a   |
|sweet- |
|faced  |
|man    |
```

To support different line-breaking strategies `form` provides the `:break` option. The `:break` option's value must be a closure/subroutine, which will then be called whenever a data string needs to be broken to fit a particular field width.

That subroutine is passed three arguments: the data string itself, an integer specifying how wide the field is, and a regex indicating which (if any) characters are to be squeezed ([/pub/2004/02/27/exegesis7.html?page=10#within_that_space_you_may_have_drawn_together...](http://pub/2004/02/27/exegesis7.html?page=10#within_that_space_you_may_have_drawn_together...)). It is expected to return a list of two values: a string which is taken as the "broken" text for the field, and a boolean value indicating whether or not any data remains after the break (so `form` knows when to stop breaking the data string). The subroutine is also expected to update the `.pos` of the data string to point immediately after the break it has imposed.

For example, if we always wanted to break at the exact width of the field (with no hyphens), we could do that with:

```
sub break_width ($data is rw, $width, $ws) {
    given $data {
        # Treat any squeezed or vertical whitespace as a single character
        # (since they'll subsequently be squeezed to a single space)
        my rule single_char { <$ws> | \v+ | . }

        # Give up if there are no more characters to grab...
        return ("", 0) unless m:cont/ (<single_char><1,$width>) /;

        # Squeeze the resultant substring...
        (my $result = $1) ~~ s:each/ <$ws> | \v+ /\c[SPACE]/;

        # Check for any more data still to come...
        my bool $more = m:cont/ <before: .* \S> /;

        # Return the squeezed substring and the "more" indicator...
        return ($result, $more);
    }
}

print form
      :break(&break_width),
      "|{[[[[]]|",
      $data;
```

producing:

```
|You can|
|play no|
|part bu|
|t Pyram|
|us; for|
|Pyramus|
|is a sw|
|eet-fac|
|ed man |
```

Or we might prefer to break on every single whitespace-separated word:

```

sub break_word ($data is rw, $width, $ws) {
    given $data {
        # Locate the next word (no longer than $width cols)
        my $found = m:cont/ \s* $?word:=(\S<1,$width>) /;

        # Fail if no more words...
        return ("", 0) unless $found{word};

        # Check for any more data still to come...
        my bool $more = m:cont/ <before: .* \S> /;

        # Otherwise, return broken text and "more" flag...
        return ($found{word}, $more);
    }
}

print form
:break(&break_word),
"|{[[[[[|",
$data;

```

producing:

```
| You |
| can |
| play |
| no |
| part |
| but |
| Pyramus |
| ; |
| for |
| Pyramus |
| is |
| a |
| sweet-f |
| aced |
| man |
```

We'll see yet another application of user-defined breaking when we discuss user-defined fields (/pub/2004/02/27/exegesis7.html?page=13#define,_define,_welleducated_infant.).

He, being in the vaward, placed behind...

There are (at least) three schools of thought when it comes to setting out a call to form that uses more than one format. The “traditional” way (i.e. the way Perl 5 formats do it) is to interleave each format string with a line containing the data it is to interpolate, with each datum aligned directly under the field into which it is to be fitted. Like so:

[illegible]

This approach has the advantage that it self-documents: to know what a particular field is supposed to contain, we merely need to look down one line.

It does, however, break up the “abstract picture” that the formats portray, which can make it more difficult to envisage what the final formatted text will look like. So some people prefer to put all the data to the right of the formats:

```
print form  
    "Name:"  
    " {[[[["  
        Biography:  
    "Status:"  
    " {[[[["  
    "  
    "Comments:"  
    " {[[[["
```

And that's perfectly acceptable too.

Sometimes, however, the data to be interpolated doesn't come neatly pre-packaged in separate variables that are easy to intersperse between the formats. For example, the data might be a list returned by a subroutine call (`get_info`) or might be stored in a hash (`%person{« name bio stat comm »}`). In such cases it's a nuisance to have to tease that data out into separate variables (or hash accesses) and then sprinkle them through the formats:

[illegible]

So `form` has an option that lets us put a single, multi-line format at the start of the argument list, place all the data together after it, and have that data automatically interleaved as necessary. Not surprisingly, that option is: `:interleave`. It's normally used in conjunction with a heredoc, since that's the easiest way to specify a multi-line string in Perl:

[illegible]

When `:interleave` is in effect, `form` grabs the first string argument it's passed and breaks that argument up into individual lines. It treats those individual lines as a series of distinct formats and grabs as many of the remaining arguments as are required to provide data for each format.

Of course, in this example we're also taking advantage of the new indenting behaviour of heredocs. The "Name:", "Status:", and "Comments:" titles are actually at the very beginning of their respective lines, because the start of a Perl 6 heredoc terminator marks the left margin of the entire heredoc string.

Would they were multitudes...

It's important to point out that, even when we're using `form`'s default **non**-interleaving behaviour, it's still okay to use a format that spans multiple lines. There *is* however a significant (and useful) difference in behaviour between the two alternatives.

The normal behaviour of `format` is to take each format string, fill in each field in the format with a substring from the corresponding data source, and then repeat that process until all the data sources have been exhausted. Which means that a multi-line format like this:

```
print form
    <<'EOFORMAT',
        Name:   {[{}]} Role: {[{}]}
        Address: {[{}]}
    EOFORMAT
@names, @roles, @addresses;
```

would normally produce this:

Name:	King Lear	Role: Protagonist
Address:	The Cliffs, Dover	
Name:	The Three Witches	Role: Plot devices
Address:	Dismal Forest, Scotland	
Name:	Iago	Role: Villain
Address:	Casa d'Otello, Venezia	

because the entire three-line format is repeatedly filled in as a single unit, line-by-line and datum-by-datum. On the other hand, if we tell `form` that it's supposed to automatically interleave the data coming after the format, like so:

```
print form :interleave,  
    <<'EOFORMAT',  
        Name: {[[[[]]]] Role: {[[]]}  
        Address: {[[]]  
  
EOFORMAT  
@names, @roles, @addresses;
```

then the call produces:

Name:	King Lear	Role:	Protagonist
Name:	The Three Witches	Role:	Plot devices
Name:	Iago	Role:	Villain
Address:	The Cliffs, Dover		
Address:	Dismal Forest, Scotland		
Address:	Casa d'Otello, Venezia		

because that second version is really equivalent to:

```
print form
    "Name: {[[[[]]]} Role: {[[[[]]]},"
        @names, @roles,
    "Address: {[[[[]]]}"
        @addresses,
    "
```

That's not much use in this particular example, but it was exactly what was needed for the biography example earlier. It's just a matter of choosing the right type of data placement to achieve the particular effect we want.

Editor's note: this document is out of date and remains here for historic interest. See Synopsis 7 (<http://dev.perl.org/perl6/doc/design/syn/S07.html>) for the current design information.

Lay out. Lay out.

As we saw earlier, with follow-on fields (/pub/2004/02/27/exegesis7.html?page=3#and_mark_what_way_i_make..) and overflow fields (/pub/2004/02/27/exegesis7.html?page=6#and_now_at_length_they_overflow_their_banks.), form is perfectly happy to have several fields in a single format that are all fed by the same data source. For example:

```
print form
"{{{[[[[[[]]]]]]]]}...}   {...[[[[[[]]]]]]]]}...}   {...[[[[[[]]]]]]]]}...",
    $soliloquy,              $soliloquy,              $soliloquy;
```

In fact, that kind of format is particularly useful for creating multi-column outputs (like newspaper columns, for example).

But a small quandry arises. In what order should `form` fill in these fields? Should the data be formatted down the page, filling each column completely before starting the next (and therefore potentially leaving the last column “short”):

Now is the winter of our discontent / Made glorious summer by this sun of York; / And all the clouds that lour'd upon our house / In the deep bosom of the ocean buried. / Now are our brows bound with vic-	torious wreaths; / Our bruised arms hung up for monuments; / Our stern alarums changed to merry meet- ings, / Our dreadful marches to delightful measures. / Grim- visaged war hath sm- oth'd his wrinkled	front; / And now, in- stead of mounting bar- red steeds / To frigh- t the souls of fear- ful adversaries, / He capers nimbly in a lady's chamber.
---	---	---

Or should the data be run line-by-line across all three columns (the way a Perl 5 `format` does it), filling one line completely before starting the next:

Now is the winter of this sun of York; / house / In the deep brows bound with vic- up for monuments; / ngs, / Our dreadful visaged war hath smo- stead of mounting ba- rful adversaries, /	our discontent / Made And all the clouds bosom of the ocean torious wreaths; / Our stern alarums ch- marches to delightful oth'd his wrinkled rded steeds / To fri- He capers nimbly in a	glorious summer by that lour'd upon our buried. / Now are our Our bruised arms hung anged to merry meeti- measures. / Grim- front; / And now, in- ght the souls of fea- lady's chamber.
--	---	---

Or should the text run down the columns, but in such a way as to leave those columns as evenly balanced in length as possible:

Now is the winter of
our discontent / Made
glorious summer by
this sun of York; /
And all the clouds
that lour'd upon our
house / In the deep
bosom of the ocean
buried. / Now are our
brows bound with vic-
torious wreaths; /
Our bruised arms hung
up for monuments; /
Our stern alarums ch-
anged to merry meeti-
ngs, / Our dreadful
marches to delightful
measures. / Grim-
visaged war hath smo-
oth'd his wrinkled
front; / And now, in-
stead of mounting bar-
ded steeds / To fri-
ght the souls of fea-
rful adversaries, /
He capers nimbly in a
lady's chamber.

Well, of course, there's no "right" answer to that; it depends entirely on what kind of effect we're trying to achieve.

The first approach (i.e. lay out the text down each column first) works well if we're formatting a news-column, or a report, or a description of some kind. The second (i.e. lay out the text across each line first), is excellent for putting diagrams or call-outs in the middle of a piece of text (as we did for Mrs Miggins (/pub/2004/02/27/exegesis7.html?page=6#and_now_at_length_they_overflow_their_banks.)). The third approach (i.e. lay out the data downwards but balance the columns) is best for presenting a single list of data in multiple columns – like `ls` does.

So we need an option with which to tell `form` which of these useful alternatives we want for a particular format. That option is named `:layout` and can take one of three string values: `"down"`, `"across"`, or `"balanced"`. So, for example, to produce three versions of Richard III's famous monologue in the order shown above, we'd use:

[illegible]

then:

```
print form :layout«across»,  
    "{[[[[[[[[]]]]]]]]...}" {..[[[[[[[[]]]]]]]]...} {..[[[[[[[[[[]]]]]]]]]}",  
        $soliloquy,           $soliloquy,           $soliloquy;
```

then:

[illegible]

By the way, the default value for the `:layout` option is "balanced" since formatting regular columns of data is more common than formatting news or advertising inserts.

For the table, sir, it shall be served...

The `:layout` option controls one other form of inter-column formatting: tabular layout.

So far, all the examples of tables we've created (for example, our normalized scores /pub/2004/02/27/exegesis7.html?page=3#therefore_put_you_in_your_best_array...) lined up nicely. But that was only because each item in each row happened to take the same number of lines (typically just one). So, a table generator like this:

```
my @play = map {"$_\r"} ( "Othello", "Richard III", "Hamlet" );
my @name = map {"$_\r"} ( "Iago", "Henry", "Claudius" );

print form
    "Character      Appears in ",
    "_____",         "_____",
    "{[[[[[[[[[[{   {[[[[[[[[[[{",
    @name,          @play;
```

correctly produces:

Character	Appears in
Iago	Othello
Henry	Richard III
Claudius	Hamlet

Note that we appended `"\r"` to each element to add an extra newline after each entry in the table. We can't use `"\n"` to specify a line-break within an array element, because `form` uses `"\n"` as an "end-of-element" marker (/pub/2004/02/27/exegesis7.html?page=3#therefore_put_you_in_your_best_array...). So, to allow line breaks within a single element of an array datum, `form` treats `"\r"` as "end-of-line-but-not-end-of-element" (somewhat like Perl 5's `format` does).

However, if we were to use the full titles for each character and each play:

```
my @play = map {"$_\r"} ( "Othello, The Moor of Venice",
                          "The Life and Death of King Richard III",
                          "Hamlet, Prince of Denmark",
                          );

my @name = map {"$_\r"} ( "Iago",
                          "Henry,\rEarl of Richmond",
                          "Claudius,\rKing of Denmark",
                          );
```

the same formatter would produce:

Character	Appears in
Iago	Othello, The Moor of Venice
Henry, Earl of Richmond	The Life and Death of King Richard III
Claudius, King of Denmark	Hamlet, Prince of Denmark

The problem is that the two block fields we're using just grab all the data from each array and format it independently into each column. Usually that's fine because the columns *are* independent (as we've previously seen (/pub/2004/02/27/exegesis7.html?page=5#able_verbatim_to_rehearse...)).

But in a table, the data in each column specifically relates to data in other columns, so corresponding elements from the column's data arrays ought to remain vertically aligned. To achieve this, we simply tell `form` that the data in the various columns should be laid out like a table:

```
print form :layout«tabular»,
  "Character      Appears in ",
  "              ",
  "{[[[[[[[[[[{  {[[[[[[[[[[{",
  @name,         @play;
```

which then produces the desired result:

Character	Appears in
Iago	Othello, The Moor of Venice
Henry, Earl of Richmond	The Life and Death of King Richard III
Claudius, King of Denmark	Hamlet, Prince of Denmark

Give him line and scope...

Sometimes we want to use a particular option or combination of options in every call we make to `form`. Or, more likely, in every call we make within a specific scope. For example, we might wish to default to a different line-breaking algorithm (/pub/2004/02/27/exegesis7.html?page=7#a_man_may_break_a_word_with_you,_sir...)

Normally in Perl 6, if we wanted to preset a particular optional argument we'd simply make an assumption:

But, of course, `form` collects all of its arguments in a single slurpy array ([/pub/2004/02/27/exegesis7.html?page=2#why_how_now_ho!_from_whence_ariseth_this](http://pub/2004/02/27/exegesis7.html?page=2#why_how_now_ho!_from_whence_ariseth_this)), so it doesn't actually *have* a `$layout` parameter that we can prebind.

```
my &down_form :=
  sub (FormArgs *@args is context(Scalar)) returns Str {
    return form( :layout«down», *@args );
  };
```

form provides one other mechanism by which options can be prebound. To use it, we (re-)load the Form module with an explicit argument list:

This causes the module to export a modified version of `form` in which the specified options are prebound. That modified version of `form` is exported lexically, and so `form` only has the specified defaults preset for the scope in which the `use Form` statement appears.

```
use Form :layout«across»,
      :interleave,
      :page{ :header("Draft $(localtime)\n\n") };

print form $introduction_format, *@introduction_data;

for @sections -> $format, @data {
  print form $format, *@data;
}

print form $conclusion_format, *@conclusion_data;
```

[illegible]

When specific field widths are required (perhaps by some design document or data formatting protocol) laying out wide fields can be error-prone. For example, most people can't visually distinguish between a 52-column field and a 53-column field and are therefore forced to manually verify the width of the corresponding field

When such fields are part of a larger format, errors like that can easily result in a call to `form` producing, say, 81-column lines. That would merely be messy if the extra characters wrapped, but could be disastrous if they happened to be chopped instead. Suppose, for example, that the last 4 columns of output contain nuclear reactor core temperatures and then consider the difference between an apparently normal reading of 567 Celsius and what might actually be happening if the reading were in fact a truncated 5678 Celsius.

```
print form '{[[[( 15 )[[[[] {<<<<(17)<<<<< {}]](14)]]].[]',
            *@data;
```

Inconsistent width for field 3.
Specified as '{}]](14)]]].[[]' but actual width is 15
in call to &form at demo.pl line 1

```
print form
'{'[[[( 15 )[[[]] {<<<<<(17)<<<<< }]](14.2)]]}.'[',
  *@data;
```

Of course, in some instances it would be much more convenient if we could simply *tell* `form` that we want a particular field to be a particular width, instead of having to explicitly *show* it.

```
print form
'[{15}] {<17><< }]]]{14.2}]]) .[[]',
*@data;
```

[illegible]

```
my $width = length(+@reasons)+1;

for @reasons.kv -> $n, $reason {
    my $n = @reasons - $index ~ '.';
    print form "      >>{$width}>>" {[[[["]]]]],
        $n,          $reason,
        ""};
}
```

By evaluating `@reasons` in a numeric context (`+@reasons`) we determine the number of reasons we have, and hence the largest number that need ever fit into the first field. Taking the length of that number (`length`) gives us the number of digits in that largest number and hence the width of a field that can format that number. We add one extra column (for the dot we're appending to each number) and that's our required width. Then we just tell `form` to make the first field that wide (`{>>{$width}>>}`).

And every one shall share...

A special form of imperative width field is the *starred field*. A starred field is one that contains an imperative width specification in which the number is replaced by a single asterisk.

The width of a starred field is not fixed, but rather is *computed* during formatting. That width is whatever is required to cause the entire format to fill the current page width of the format (by default, 78 columns).

Consider, for example:

```
print form
'{}{}{}{}{}{}{}{}{}{} {}{}.{[]} {{{{*}[] } ',
  @names,          @scores, @comments;
```

The width of the starred comment field in this case is 49 columns – the default page width of 78 columns minus the 29 columns consumed by the fixed-width portions of the format (including the other two fields).

If a format contains two or more starred fields, the available space is shared equally between them. So, for example, to create two equal columns (say, to compare the contents of two files), we might use:

```
print form
"{{{{{*}[]}} } {{{{*}[]}} },
  slurp($file1), slurp($file2);
```

And, yes, Perl 6 does have a built-in `slurp` function that takes a filename, opens the file, reads in the entire contents, and returns them as a single string. For more details see the `Perl6::Slurp` module (now on the CPAN).

There is one special case for starred fields: a starred verbatim field:

```
{""""{*}""""}
```

It acts like any other starred field, growing according to the available space, except that it will never grow any wider than the widest line of the data it is formatting. For example, whereas a regular starred field:

```
print form
'| {{{{*}[] } |',
  $monologue;
```

expands to the full page width:

```
| Now is the winter of our discontent           |
| Made glorious summer by this sun of York;    |
| And all the clouds that lour'd upon our house |
| In the deep bosom of the ocean buried.        |
| Now are our brows bound with victorious wreaths |
| Our bruised arms hung up for monuments;      |
| Our stern alarums changed to merry meetings, |
| Our dreadful marches to delightful measures.  |
| Grim-visaged war hath smooth'd his wrinkled front; |
| And now, instead of mounting barded steeds  |
| To fright the souls of fearful adversaries,  |
| He capers nimbly in a lady's chamber.        |
```

a starred verbatim field:

```
print form
'| {""""{*}"""" } |',
  $monologue;
```

only expands as much as is strictly necessary to accommodate the data:

```
| Now is the winter of our discontent           |
| Made glorious summer by this sun of York;    |
| And all the clouds that lour'd upon our house |
| In the deep bosom of the ocean buried.        |
| Now are our brows bound with victorious wreaths; |
| Our bruised arms hung up for monuments;      |
| Our stern alarums changed to merry meetings, |
| Our dreadful marches to delightful measures.  |
| Grim-visaged war hath smooth'd his wrinkled front; |
| And now, instead of mounting barded steeds  |
| To fright the souls of fearful adversaries,  |
| He capers nimbly in a lady's chamber.        |
```

That we our largest bounty may extend...

By now you've probably noticed that there is quite a large overlap between the functionality of `form` and that of `(s)printf`. For example, the call:

```
for @procs {
    print form
    "{>>>} {<<<<<<<(20)<<<<<<< }>>>>>>} {>>.%}",
    .{pid}, .{cmd},          .{time}, .{cpu};
}
```

has approximately the same effect as the call:

```
for @procs {
    printf "%5d %-20s %8s %5.1f%%\n",
    .{pid}, .{cmd}, .{time}, .{cpu};
}
```

One is more WYSIWYG, the other more concise, but (placed in a suitable loop), they would both print out lines like these:

2461	vi -ii henry	0:55.83	11.6%
2395	ex cathedra	0:06.59	3.5%
2439	head anne.boleyn	0:00.18	0.1%
2581	dig -short grave	0:01.04	0.0%

There is, however, a crucial difference between these two formatting facilities; one that only shows up when one of our processes runs over 99 hours. For example, suppose our browser has been running continuously for a few months (or, more precisely, for 1214:23.75 hours). Then the calls to `printf` would print:

2461	vi -ii henry	0:55.83	11.6%
2395	ex cathedra	0:06.59	3.5%
27384	lynx www.divorce.com	1214:23.75	0.8%
2439	head anne.boleyn	0:00.18	0.1%
2581	dig -short grave	0:01.04	0.0%

whilst the calls to `form` would print:

2461	vi -ii henry	0:55.83	11.6%
2395	ex cathedra	0:06.59	3.5%
27384	lynx www.divorce.com	1214:23-	0.8%
2439	head anne.boleyn	0:00.18	0.1%
2581	dig -short grave	0:01.04	0.0%

In other words, field widths in a `printf` represent *minimal* spacing (even if that throws off the overall layout), whereas field widths in a `form` represent *guaranteed* spacing (even if that truncates some of the data).

Of course, in a situation like this – where we knew that the data might not fit and we didn't want it truncated – we could use a block field instead:

```
for @procs {
    print form
    "{>>>} {<<<<<<<(19)<<<<<<< {}]]]]]} {>>.%}",
    .{pid}, .{cmd},          .{time}, .{cpu};
}
```

in which case we'd get:

2461	vi -ii henry	0:55.83	11.6%
2395	ex cathedra	0:06.59	3.5%
27384	lynx www.divorce.com	1214:23-	0.8%
		.75	
2439	head anne.boleyn	0:00.18	0.1%
2581	dig -short grave	0:01.04	0.0%

That preserves the data, but the results are still ugly, and it also requires some fancy footwork – making the percentage sign part of the field specification, as if it were a currency marker (/pub/2004/02/27/exegesis7.html?page=5#some_tender_money_to_me...) – to make the last field work correctly. In other words: it's a kludge. The sad truth is that sometimes variable-width fields are a better solution.

So `form` provides them too. Any field specification may include a plus sign (+) anywhere between its braces, in which case it specifies an *extensible field*: a field whose width is minimal, rather than absolute. So, in the above example, our call to `form` should actually look like this:

```
for @procs {
    print form
    "{>>>} {<<<<<<<(20)<<<<<<< }>>>>>>+} {>>.%}",
    .{pid}, .{cmd},          .{time}, .{cpu};
}
```

and would produce this:

2461	vi -ii henry	0:55.83	11.6%
2395	ex cathedra	0:06.59	3.5%
27384	lynx www.divorce.com	1214:23.75	0.8%
2439	head anne.boleyn	0:00.18	0.1%
2581	dig -short grave	0:01.04	0.0%

just like printf does.

Likewise, if we thought the command names might exceed 20 columns we could let that field stretch too:

```
for @procs {
    print form
        "{>>>} {<<<<<<<(20+)<<<<<< {>>>>>+} {>.>}%",
        .{pid}, .{cmd}, .{time}, .{cpu};
}
```

Note that the field width specifier would still warn us if the field's "picture" was not exactly 20 columns wide, but the resulting field would nevertheless stretch as necessary to accommodate longer data.

Editor's note: this document is out of date and remains here for historic interest. See Synopsis 7 (<http://dev.perl.org/perl6/doc/design/syn/S07.html>) for the current design information.

Within that space you may have drawn together...

When a field is being filled in, whitespace is normally left as-is (except for justification, and wrapping of lines in block fields). However, this behaviour can be altered by specifying a *whitespace squeezing* strategy. Squeezing replaces those substrings of the data that match a specified pattern (for example: `/\s+/`), substituting a single space character.

If we don't want the default (non-)squeezing strategy we can use the `:ws` option specify the particular pattern that is to be used for squeezing:

```
print form
      :ws(/h+/,),          # squeeze any horizontal whitespace
      $format1, *@data1,
      :ws(/<comment>|\s+/,), # now squeeze comments or whitespace
      $format2, *@data2;
```

For example, suppose we have a eulogy generator:

```
sub eulogize ($who, $to, $blaming) {...}
```

that (rather poorly) drops the appropriate names into a pre-formatted template, to produce strings like:

Friends, Romans, countrymen, lend me your ears;
I come to bury Caesar, not to praise him.
The evil that men do lives after them;
The good is oft interred with their bones;
So let it be with Caesar. The noble Brutus
Hath told you Caesar was ambitious:
If it were so, it was a grievous fault,
And grievously hath Caesar answer'd it.

If we interpolate that string, with its extra spaces and its embedded newlines, into a `form` field:

```
print form  
"| {[[[[]]]] } |",  
    eulogize('Caesar', to=>'Romans', blaming=>'Brutus');
```

we'd get:

	Friends, Romans , countrymen, lend me	
	your ears;	
	I come to bury Caesar , not to praise	
	him.	
	The evil that men do lives after them;	
	The good is oft interred with their bones;	
	So let it be with Caesar . The noble	
	Brutus	
	Hath told you Caesar was	
	ambitious:	
	If it were so, it was a grievous fault,	
	And grievously hath Caesar answer'd	
	it.	

Note that the extra spaces and the embedded newlines are preserved in the resulting text.

But, if we told `form` to squeeze all whitespaces:

we'd get:

Friends, Romans , countrymen, lend me your
ears; I come to bury Caesar , not to
praise him. The evil that men do lives
after them; The good is oft interred with
their bones; So let it be with Caesar .
The noble Brutus Hath told you Caesar was
ambitious: If it were so, it was a
grievous fault, And grievously hath Caesar
answer'd it.

with each sequence of characters that match `/\s+/` being reduced to a single space.

On the other hand, if we wanted to preserve the newlines and squeeze only horizontal whitespace, that would be:

which produces:

Friends, Romans , countrymen, lend me your	
ears;	
I come to bury Caesar , not to praise him.	
The evil that men do lives after them;	
The good is oft interred with their bones;	
So let it be with Caesar . The noble	
Brutus	
Hath told you Caesar was ambitious:	
If it were so, it was a grievous fault,	
And grievously hath Caesar answer'd it.	

Of course, for this particular text, none of these solutions is entirely satisfactory since squeezing the whitespaces to a single space still leaves a single space in places like "Caesar ." and "Romans ,".

To remove those blemishes we need to take advantage of a more sophisticated aspect of `form`'s whitespace squeezing behaviour. Namely that, when squeezing whitespace using a particular pattern, `form` detects if that pattern captures anything and *doesn't* squeeze the captured items.

More precisely, if the squeeze pattern matches but doesn't capture, `form` simply replaces the entire match with a single space character. But if the squeeze pattern *does* capture, `form` doesn't insert a space character, but instead replaces the entire match with the concatenation of the captured substrings.

That means we can completely eliminate any whitespace before a punctuation character with:

which produces the desired:

Friends, Romans, countrymen, lend me your	
ears;	
I come to bury Caesar, not to praise him.	
The evil that men do lives after them;	
The good is oft interred with their bones;	
So let it be with Caesar. The noble Brutus	
Hath told you Caesar was ambitious:	
If it were so, it was a grievous fault,	
And grievously hath Caesar answer'd it.	

This works because, in those instances where the pattern `/\h+ (<punct>)?/` matches some whitespace followed by a punctuation character, the punctuation character is captured, and the captured character is then used to replace the entire whitespace-plus-punctuator. On the other hand, if the pattern matches whitespace but no punctuator (and it's allowed to do that because the punctuator is optional), then nothing is captured, so `form` falls back to replacing the whitespace with a single space.

He doth fill fields with harness...

Fields are (almost) always of a fixed width. So, if there isn't enough data to fill a particular field, the unused portions of that field are filled in with spaces to preserve the vertical alignment of other columns of formatted data. However, spaces are only the default. The `:hfill` (horizontal fill) option can be used to change fillers. For example:

```
print form
:hfill("=-"),          # Fill next fields with "=-"
"{[*]}\n",            # Full width field for title
"[ Table of Contents ]", # Title
:hfill(" ."),          # Fill next fields with spaced dots
'  {[[[[[*]][[[[{}]]]}  ', # Two indented block fields
  @contents,    @page;    # Data for those blocks
```

This fills the empty space either side of the centred title with a repeated `=-` sequence. It then fills the gaps to the right of the left-justified contents field, and to left of the right-justified pages field, with spaced dots. Which, rather prettily, produces something like:

```
===== [ Table of Contents ]=====

Foreword. . . . . i
Preface . . . . . iii
Glossary. . . . . vi
Introduction. . . . . 1
The Tempest . . . . . 7
Two Gentlemen of Verona . . . . . 17
The Merry Wives of Windsor . . . . . 27
Twelfth Night. . . . . 39
Measure for Measure . . . . . 50
Much Ado About Nothing. . . . . 62
A Midsummer Night's Dream . . . . . 73
Love's Labour's Lost. . . . . 82
The Merchant of Venice. . . . . 94
As You Like It. . . . . 105
```

Note that the fill sequence doesn't have to be a single character and that the fill pattern is consistent across multiple fields and between adjacent lines. That is, it's as if every field is first filled with the same fill pattern, then the actual data written over the top. That's particularly handy in the above example, because it ensures that the fill pattern seamlessly bridges the boundary between the adjacent contents and pages fields.

It's also possible to specify separate fill sequences for the left- and right-hand gaps in a particular field, using the `:lfill` and `:rfill` options. This is particularly common for numerical fields. For example, this call to `form`:

```
print form
'Name          Bribe (per dastardry)',
'=====      =====',
'{[[[[[[[[[[{  }]],]].[[{  ',
@names,        @bribes;
```

would print something like:

```
Name          Bribe (per dastardry)
=====      =====
Crookback      12.676
Iago           1.62
Borachio       45,615.0
Shylock        19.0003
```

with the numeric field padded with whitespace and only showing as many decimal places as there are in the data.

However, in order to prevent subsequent...err...creative calligraphy (they *are*, after all, villains and would presumably not hesitate to add a few digits to the front of each number), we might prefer to put stars before the numbers and show all decimal places. We could do that like so:

```
print form
'Name          Bribe (per dastardry)',
'=====      =====',
'{[[[[[[[[[[{  }]],]].[[{  ',
@names,        :lfill('*'), :rfill('0'),
                @bribes;
```

which would then print:

```
Name          Bribe (per dastardry)
=====      =====
Crookback      *****12.6760
Iago           *****1.6200
Borachio       *45,615.0000
Shylock        *****19.0003
```

Note that the `:lfill` and `:rfill` options are specified *after* the format string and, more particularly, before the data for the second field. This means that those options only take effect for that particular field and the previous fill behaviour is then reasserted for subsequent fields. Many other `form` options – for example `:ws`, `:height`, or `:break` – can be specified in this way, so as to apply them only to a particular field.

Forbear, and eat no more.

Name:	Biography:
William Shakespeare	William Shakespeare was born on April 23, 1564 in Strathford-upon-Avon, England; he was third of etc.
Status:	

So it has to be a block field, to “keep up” with however much output the multi-line Name field produces. Unfortunately, starting the Biography column with a normal block field doesn’t solve the problem either. In fact we get:

Name:	Biography:
William Shakespeare	William Shakespeare was born on April 23, 1564 in Strathford-upon-Avon, England; he was third of eight children from Father John Shakespeare and Mother Mary Arden. Shakespeare began his education at the age of seven when he probably attended the Strathford grammar school. The school provided Shakespeare with his formal education. The students chiefly studied Latin rhetoric, logic, and literature. His knowledge and imagination may have come from his reading of ancient authors and poetry. In November 1582, Shakespeare received a license to marry Anne Hathaway. At the time of their marriage, Shakespeare was 18 years old and Anne was 26. They had three children, the oldest Susanna, and twins- a boy, Hamneth, and a girl, Judith. Before his death on April 23 1616, William Shakespeare had written thirty-seven plays. He is generally considered the greatest playwright the world has ever known and has always been the world's most popular author.
Status:	
Deceased (1564–1616)	
Comments:	
Theories abound as to the true author of his plays. The prime alternative candidates being Sir Francis Bacon, Christopher Marlowe, or Edward de Vere	

Normal block fields are remorseless in consuming all of their data. So the first Biography field absolutely will not stop formatting, ever, until your entire `$biography` string is gone.

What we really need here, is a kinder, gentler block field; a block field that formats minimally, like an overflow field. And we get that with yet another `:height` option: `:height«minimal»`. Like so:


```
print form
  'To Do:',
  '  {{{{{{}}}}}}',
  @todo;
```

might produce something like:

```
To Do:
Dissemble
Deceive
Dispute
Defy
Duel
Defeat
Dispatch
```

That looks fine but, because each line is produced by the large left-justified field that is automatically filled with whitespace, the output contains several hundred more space characters than are strictly necessary (you probably didn't notice them, but they're all there – hanging off the right sides of the individual To-Do items).

Fortunately, however, `form` is smarter than that. Extraneous trailing whitespace on the right-hand side of any output line is automatically trimmed. So the above example actually produces:

```
To Do:
Dissemble
Deceive
Dispute
Defy
Duel
Defeat
Dispatch
```

Of course, if you really do need those “invisible” trailing whitespaces for some reason, `form` provides a way to keep them – the `:untrimmed` option:

```
print form :untrimmed,
  'To Do:',
  '  {{{{{{}}}}}}',
  @todo;
```

Editor's note: this document is out of date and remains here for historic interest. See Synopsis 7 (<http://dev.perl.org/perl6/doc/design/syn/S07.html>) for the current design information.

Master Page, I am glad to see you...

Normally, `form` assumes that whatever data it is formatting is supposed to produce a single, arbitrarily long, unbroken piece of text. But `form` can also format data into multiple pages of fixed length and width, inserting customized, page-specific headers, footers, and pagefeeds for each page.

All these features are controlled by the `page` option (or more precisely, by its various suboptions):

```
print form
  :page{ :length( $page_len ),      # Default: 60 lines
         :width( $page_width ),     # Default: 78 columns
         :number( $first_page_num ), # Default: 1
         :header( &make_header ),   # Default: no header
         :footer( &make_footer ),   # Default: no footer
         :feed( &make_pagefeed ),   # Default: no pagefeed
         :body( &adjust_body ),     # Default: no chiropracty
      },
  $format,
  *@args;
```

Measure his woe the length and breadth of mine...

The `:page{ :length(...) }` suboption determines the number of output lines per page (including headers and footers). Normally, this suboption is set to infinity, which produces that single, arbitrarily long, unbroken page of text. But the suboption can be set to any positive integer value, to cause `form` to generate distinct pages of that many lines each.

The value of the `:page{ :width(...) }` suboption is used to determine the width of distributive fields (/pub/2004/02/27/exegesis7.html?page=9#and_every_one_shall_share...) and in some page body postprocessors (/pub/2004/02/27/exegesis7.html?page=12#do_to_this_body_what_extremes_you_can...). By default, this suboption is set to 78 (columns), but it may be set to any positive integer value.

The `:page{ :number(...) }` suboption specifies the current page number. By default it starts at 1, but may be set to any numeric value. This suboption is generally only of use in headers and footers (see below).

From the crown of his head to the sole of his foot...

The `:page{ :header(...) }` suboption specifies a hash containing a set of strings or subroutines that are to be used to create page headers. Each key of the hash indicates a particular kind of page that the corresponding value will provide the header for. For example:

```
:header{ first => "          'The Tempest' by W. Shakespeare      ",
         last  => "          -- The End --                      ",
         odd   => "Act $act, Scene $scene                        ",
         even  => "                                              ",
         other => "          [Thys hedder intenshunally blanke]  ",
}
```

Given the above specification, `form` will:

- use the full title and author as the header of the first page,
- write `-- The End --` across the top of the last page,
- prepend the act and scene information to the start of any odd page (except, of course, the first or the last), and
- provide an empty line as the header of any even page (except the last, if it happens to be even).

Note that, in this case, since we've provided specific headers for every odd and even page, the `"other"` header will never be used. On the other hand, if we'd specified:

```
:header{ first => "          'The Tempest' by W. Shakespeare      ",
         other => "                                              'The Tempest'",
}
```

then every page except the first would have just a right-justified title at the top.

Of course, if we want every page to have the same header, we can just write:

```
:header{ other => "                                              'The Tempest' }
```

But that's a little klunky, so `form` also accepts a single string instead of a hash, to specify a header to be used for every page:

```
:header("                                              'The Tempest'")
```

Headers don't all have to be the same size either. For example, we might prefer a more imposing first header:

```
:header{ first => "          'The Tempest'                      \n"
               ~ "          by                                  \n"
               ~ "          W. Shakespeare                     \n"
               ~ "          _____",
         other => "                                              'The Tempest'",
}
```

`form` simply notes the number of lines each header requires and then reduces the available number of lines within the page accordingly, so as to preserve the exact overall page length.

Often we'll need headers that aren't fixed strings. For example, we might want each page to include the appropriate page number. So instead of a string, we're allowed to specify a particular header as a subroutine. That subroutine is then called each time that particular header is required, and its return value is used as the required header.

When the subroutine is called, the current set of active formatting options are passed to it as a list of pairs. Typically, then, the subroutine will specify one or more named-only parameters corresponding to the options it cares about, followed by a starred hash parameter to collect the rest. For example if every page should have its (left-justified) page number for a header:

```
:header( sub (+$page, *%_) { return $page{number}; } )
```

Of course, this is also an excellent candidate for the cleaner (but equivalent) syntax of placeholder variables in raw blocks:

```
:header{ $^page{number}; }
```

And, naturally, we can mix-and-match static and dynamic headers:

```
:header{ odd  => "                                              'The Tempest'",
         even => { $^page{number}; },
}
```

Footers work in exactly the same way in almost all respects; the obvious exception being that they're placed at the end of a page, rather than the start.

Pagefeeds work the same way too. A pagefeed is a string that is placed between the footer of one page and the header of the next. They're like formfeeds, except they can be any string we choose (not just `\c[FF]`). They're called "pagefeeds" instead of "formfeeds" because they're placed between pages, not between calls to `form`.

Here's a complete example to illustrate the full set of features:

```
my @tobe = slurp 'Soliloquy.txt' or die;

my %page = (
  :length(15),
  :header{ first => "Hamlet's soliloquy begins...\n\n",
            odd  => "Hamlet's soliloquy continues...\n\n",
            even => { form '>>{*}>>}', "Hamlet's soliloquy continues...\n\n"; },
            last => "Hamlet's soliloquy concludes...\n\n",
          },
  :footer{
    last => { form "\n{||{*}||}", "END OF TEXT"; }
    other => { form "\n{>>{*}>>}", "../"~($^page{number}+1); },
  },
  :feed("\f"),
);

print form
  :page(%page),
  '{|||||}' {"{*}"} {'[[[[|]',
  [1..@tobe], @tobe, [1..@tobe];
```

which prints:

Hamlet's soliloquy begins...

1	To be, or not to be -- that is the question:	1
2	Whether 'tis nobler in the mind to suffer	2
3	The slings and arrows of outrageous fortune	3
4	Or to take arms against a sea of troubles	4
5	And by opposing end them. To die, to sleep --	5
6	No more -- and by a sleep to say we end	6
7	The heartache, and the thousand natural shocks	7
8	That flesh is heir to. 'Tis a consummation	8
9	Devoutly to be wished. To die, to sleep --	9
10	To sleep -- perchance to dream: ay, there's the rub,	10
11	For in that sleep of death what dreams may come	11

../2

^L

Hamlet's soliloquy continues...

12	When we have shuffled off this mortal coil,	12
13	Must give us pause. There's the respect	13
14	That makes calamity of so long life.	14
15	For who would bear the whips and scorns of time,	15
16	Th' oppressor's wrong, the proud man's contumely	16
17	The pangs of despised love, the law's delay,	17
18	The insolence of office, and the spurns	18
19	That patient merit of th' unworthy takes,	19
20	When he himself might his quietus make	20
21	With a bare bodkin? Who would fardels bear,	21
22	To grunt and sweat under a weary life,	22

../3

^L

Hamlet's soliloquy continues...

23	But that the dread of something after death,	23
24	The undiscovered country, from whose bourn	24
25	No traveller returns, puzzles the will,	25
26	And makes us rather bear those ills we have	26
27	Than fly to others that we know not of?	27
28	Thus conscience does make cowards of us all,	28
29	And thus the native hue of resolution	29
30	Is sicklied o'er with the pale cast of thought,	30
31	And enterprise of great pitch and moment	31
32	With this regard their currents turn awry	32
33	And lose the name of action. -- Soft you now,	33

../4

^L

Hamlet's soliloquy concludes...

34	The fair Ophelia! -- Nymph, in thy orisons	34
35	Be all my sins remembered.	35

END OF TEXT

^L

Note, in particular, the nested calls to `form` within some of the subroutines – to center or right-justify a particular header or footer. Permitting just this kind of “recursive” formatting is one of the main reasons Perl 5’s built-in `format` has become the (reentrant) `form` subroutine in Perl 6.

Do to this body what extremes you can...

Sometimes it’s useful to be able to grab the entire body of a page (i.e. the contents of the page between the header and footer) *after* it’s been formatted together. For example, we might wish to centre those contents, or to crop them at a particular column.

To this end, the `:page{ :body(...) }` suboption allows us to specify a page body post-processor. That is, a subroutine or format that lays out the page’s formatted text between the page’s header and footer. Like the `:header`, `:footer`, and `:feed` suboptions, the `:body` suboption can take either a closure, a hash, or a string.

If the value of the `:body` suboption is a string or a hash of pairs, the text of the body is (recursively) `form`’ed using that string (or those string values) as its format. A very common usage is to arrange for the formatted text to be horizontally and vertically centred on each page:

```
:body(' {=I{*}I=}')

```

A more sophisticated variation on this is to use a hash to insert a left or right “gutter” for each page:

```
$gutter = " " x $gutter_width;

:body{ odd    => $gutter ~ '{""{*}""}',
      even    => '{""{*}""}' ~ $gutter,
      }
}
```

On the other hand, if the value of the `:body` suboption is closure, the body text is passed to that closure as an array of lines. A second array is also passed in, containing as many newlines as would be needed to pad out the body text to the correct number of lines for the page. Finally, the current formatting options are passed as a list of pairs. As with the `:header` etc. suboption, the closure is expected to return a single string (representing the final formatting of the page body).

For example, to add line numbers to the text each page (but *not* to the headers or footers or filler lines):

[illegible]

which produces:

1 Now is the winter of our discontent /
2 Made glorious summer by this sun of
3 York; / And all the clouds that lour'd
4 upon our house / In the deep bosom of
5 the ocean buried. / Now are our brows
6 bound with victorious wreaths; / Our
7 bruised arms hung up for monuments; /
8 Our stern alarums changed to merry
9 meetings, / Our dreadful marches to

10 delightful measures. Grim-visaged war
11 hath smooth'd his wrinkled front; / And
12 now, instead of mounting barded steeds
13 / To fright the souls of fearful
14 adversaries, / He capers nimbly in a
15 lady's chamber.

16 To be, or not to be -- that is the question: /
17 Whether 'tis nobler in the mind to suffer /
18 The slings and arrows of outrageous fortune /
19 Or to take arms against a sea of troubles /
20 And by opposing end them. To die, to sleep --
21 / No more -- and by a sleep to say we end /
22 The heartache, and the thousand natural shocks
23 / That flesh is heir to. 'Tis a consummation /
24 Devoutly to be wished. To die, to sleep -- /

25 To sleep — perchance to dream: ay, there's
26 the rub, / For in that sleep of death what
27 dreams may come / When we have shuffled off
28 this mortal coil, / Must give us pause.
29 There's the respect / That makes calamity of
30 so long life.

```
my &ensor := expurgate «villain plot libel treacherous murderer false deadly 'G'»;

print form
"[Ye following tranfcript hath been cenfored by Order of ye King]\n\n",
"    {[[[{}]]]}",
    censor($speech);
```

to produce:

```
[Ye following tranfcript hath been cenfored by Order of ye King]
```

```
And therefore, since I cannot prove a lover,  
To entertain these fair well-spoken days,  
I am determined to prove a XXXXXXX  
And hate the idle pleasures of these days.  
XXXXs have I laid, inductions dangerous,  
By drunken prophecies, XXXXXs and dreams,  
To set my brother Clarence and the king  
In XXXXXX hate the one against the other:  
And if King Edward be as true and just  
As I am subtle, XXXXX and XXXXXXXXXXXX,  
This day should Clarence closely be mew'd up,  
About a prophecy, which says that XXX  
Of Edward's heirs the XXXXXXXX shall be.
```

Of course, if this were Puritanism and not Perl, we might have a long list of proscribed words that we needed to excise from *every* formatted text. In that case, rather than explicitly running every data source through the same censorious subroutine, it would be handy if `form` had a built-in field that did that for us automatically.

Naturally, `form` doesn't have such a field built-in...but we can certainly give it one.

User-defined field specifiers can be declared using the `:field` option, which takes as its value an array of pairs. The key of each pair is a string or a rule (i.e. regex) that specifies the syntax of the user-defined field. The value of each pair is a closure/subroutine that constructs a standard field specifier to replace the user-defined specifier. Alternatively, the value of a pair may be a string, which is taken as the (static) field specifier to be used instead of the user-defined field.

In other words, each pair is a macro that maps a user-defined field (specified by the pair's key) onto a standard `form` field (specified by the pair's value). For example:

```
:field[ /\{ X+ \}/ => &cursor_field ]
```

This tells `form` that whenever it finds a brace-delimited field consisting of one or more X's, it should call a subroutine named `cursor_field` and use the return value of that call instead of the all-X field.

When the key of a `:field` pair matches some part of a format, its corresponding subroutine is called. That subroutine is passed the result (i.e. `$0`) of the rule match, as well as the hash of active options for that field. Changes to the options hash will affect the subsequent formatting behaviour of that field.

So `cursor_field` could be implemented like so:

```
# Constructor subroutine for user-defined cursor fields...  
sub cursor_field ($field_spec, %opts) {  
  
    # Set up the field's 'break' option with a censorious break...  
    %opts{break} = break_and_cursor(%opts{break});  
  
    # Construct a left-justified field with the appropriate width  
    # specified imperatively...  
    return '[[{' ~ length($field_spec) ~ '}][]';  
}
```

The `cursor_field` subroutine has to change the field's `:break` option, creating a new line breaker that also expurgates unsuitable words. To do this it calls `break_and_cursor`, which returns a new line breaker subroutine:

```
# Create a new 'break' sub...  
sub break_and_cursor (&original_breaker) {  
    return sub (*@args) {  
  
        # Call the field's original 'break' sub...  
        my ($nextline, $more) = original_breaker(*@args);  
  
        # X out any doubleplus ungood words  
        $nextline =~ s:ei/(@proscribed)/$( 'X' x length $1 )/;  
  
        # Return the "corrected" version...  
        return ($nextline, $more);  
    }  
}
```

Having created a subroutine to translate cursor fields and another to break-and-expurgate the data placed in them, we are now in a position to create a module that encapsulates the new formatting functionality:

```
module Ministry::Of::Truth {

  # Internal mechanism (as above)...
  my @proscribed = «villain plot libel treacherous murderer false deadly 'G'»;
  sub break_and_censor (&original_breaker) {...}
  sub censor_field ($field_spec, %opts) {...}

  # Make the new field type standard by default in this scope...
  use Form :field[ /\{ X+ \}/ => &censor_field ];

  # Re-export the specialized &form that was imported above...
  sub form is exported {...}

}
```

Okay, admittedly that's quite a lot of work. But the pay-off is huge: we can now trample on free speech *much* more easily:

[illegible]

And we'd get the same carefully XXXX'ed output as before.

Put thyself into the trick of singularity...

User-defined fields are also a handy way to create single-character markers for single-column fields (in order to preserve the one-to-one spacing of a format). For example:

```
print form
      :field{ '^' => '{<III{1}III}',      # 1-char-wide, top-justified block
            '^' => '{<=II{1}II=}',        # 1-char-wide, middle-justified block
            '-' => '{<_II{1}II}',         # 1-char-wide, bottom-justified block
      },
      '~~~~~',
      '^ _ = _ ^',      «*like round and orient perls»,
      '~~~~~';
```

```
prints:
```

~~~~~  
l o p  
i r a r e  
k o n i r  
e u d e l  
n n s  
d t  
~~~~~

Note that we needed to use a unary `*` to flatten the `«like round and orient perls»` data list. That's because every argument of `form` is evaluated in scalar context, and an unflattened `«...»` list in scalar context becomes an array reference, rather than the five separate strings we needed to fill our five single-character fields.

Single fields are particularly useful for labelling the vertical axes of a graph:

```
use Form :field[ '=' => '{<=II{1}II=}' ];

@vert_label = «Villain's fortunes»
$hor_label = "Time";

print form
'      ^                                     ',
'  =  | { "*****" } ', *@vert_label, @data,
'    +----->',
'    { ||| ||| ||| ||| ||| ||| ||| ||| } ', $hor_label;
```

which produces:



But the:

```
:field[ '=' => '{<=II{1}II=}' ]
```

define a single-character field:

```
:single( '=' )
```

or to define several at once:

```
:single['#', '*', '+']
```

resulting block is top-justified. So our previous example could also have been written:

```
print form
      :single("="),
      ' = = | { "....." } ', *@vert_label, @data,
      ' +-----> ',
      ' { | | | | | | | | | | | | | | | | } ', $hor_label;
```

These paper bullets of the brain...

Bulleted lists of items are a very common feature of reports, but as we saw earlier (/pub/2004/02/27/exegesis7.html?page=3#therefore_put_you_in_your_best_array...) they're surprisingly hard to get right.

Suppose, for example, we want a list of items bulleted by “diamonds”:

- A rubber sword (laminated with mylar to look suitably shiny).
- Cotton tights (summer performances).
- Woolen tights (winter performances or those actors who are willing to admit to being over 65 years of age).
- Talcum powder.
- Codpieces (assorted sizes).
- Singlet.
- Double.
- Triplet (Kings and Emperors only).
- Supercilious attitude (optional).

Something like this works well enough:

```
for @items -> $item {  
    print form  
        '<> {' <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<', $item;  
        '      {' vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv';  
}
```

The first format produces the bullet plus the first line of text for the item, then the second format handles any overflow of the item data.

Alternatively, we could achieve the same result with a single format string by interpolating the bullet as well:

```
my $bullet = "<>";

for @items -> $item {
    print form
        "'{*}'" {[[[["",
            $bullet, $item;
        }
    }
}
```

Here we use a single-line starred verbatim field (`{' '*}`), so that the bullet is interpolated “as-is” and the field is only as wide as the bullet itself. Then for the item itself we use a block field, which will format the item data over as many lines as necessary. Meanwhile, because the bullet’s field is single-line, after the first line the bullet field will be filled with spaces (instead of a “diamond”), leaving a bullet only on the first line.

This second approach also has the advantage that we could change the bullet string at run-time and the format would adapt automatically.

However, it’s still a little irritating that we have to set up a loop and call `form` separately for each element of `@items`. After all, if we didn’t need to bullet our list we could just write:

```
print form
    "{[[[["",
    @items;
```

and `form` would take care of iterating over the `@items` for us. It seems that things ought to be that easy for bulleted lists as well.

And, of course, things *are* that easy.

All we need to do is tell `form` that whenever the string `"<>"` appears in a format, it should be treated as a bullet. That is, it should appear only beside the *first* line of text produced when formatting each element of the adjacent field’s data.

To tell `form` all that we use the `:bullet` option:

```
print form
    :bullet("<>"),
    "<> {[[[["",
    @items;
```

or, more permanently:

```
use Form :bullet("<>");

# and later...

print form
    "<> {[[[["",
    @items;
```

The presence of this `:bullet` option causes `form` to treat the sequence `"<>"` as a special field. That special field interpolates the string `"<>"` when the field immediately to its right begins to format a new data element, but thereafter interpolates only spaces until the adjacent field finishes formatting that data element.

Or, more simply, if we tell `form` that `"<>"` is a bullet, `form` treats it like a bullet that’s attached to the very next field.

So we could finally fix our Shakespearean roles example (/pub/2004/02/27/exegesis7.html?page=3#therefore_put_you_in_your_best_array...), like so:

```
print "The best Shakespearean roles are:\n\n";

print form
    :bullet("* "),
    " * {[[[[" *{[[[["]]]]]]]*",
    @roles, $disclaimer;
```

This would then produce:

The best Shakespearean roles are:

* Either of the 'two foolish officers': Dogberry and Verges	*WARNING:	*
* That dour Scot, the Laird Macbeth	*This list of roles*	
* The tragic Moor of Venice, Othello	*constitutes a*	
* Rosencrantz's good buddy Guildenstern	*personal opinion*	
* The hideous and malevolent Richard III	*only and is in no*	
	way endorsed by	
	*Shakespeare'R'Us. *	
	It may contain	
	*nuts. *	
	* *	

- control over line-breaking, whitespace squeezing, and filling of empty fields; and
- support for creating plaintext lists, tables, and graphs.

And because it's now part of a module, rather than a core component, `form` will be able to evolve more easily to meet the needs of its community. For example, we are currently investigating how we might add facilities for specifying numerical bullets, for formatting text using variable-width fonts, and for outputting HTML instead of plaintext.

If you're a regular user of Perl 5's `format` you might like to try the `form` function instead. It's available right now in the Perl6::Form module, which waits upon thy pleasure at the CPAN.

Tags

[perl-6 \(/categories/perl-6\)](#) [exegesis \(/tags/exegesis\)](#) [formats \(/tags/formats\)](#) [perl-6 \(/tags/perl-6\)](#)

Damian Conway

[\(/authors/damian-conway/\)](#)

Browse their articles [\(/authors/damian-conway/\)](#)

[\(/authors/damian-conway/\)](#)

Feedback

Something wrong with this article? Help us out by opening an issue or pull request on GitHub [\(https://github.com/perladvnt/perldotcom/blob/master/content/legacy/_pub_2004_02_27_exegesis7.md\)](https://github.com/perladvnt/perldotcom/blob/master/content/legacy/_pub_2004_02_27_exegesis7.md)

Site Map

- [Home \(/\)](#)
- [About \(/about\)](#)
- [Authors \(/authors\)](#)
- [Categories \(/categories\)](#)
- [Tags \(/tags\)](#)

Contact Us


To get in touch, submit an issue to [perladvnt/perldotcom](https://github.com/perladvnt/perldotcom/issues) (<https://github.com/perladvnt/perldotcom/issues>) on GitHub.

 (<https://perl.org>)  (</article/index.xml>)

 (<https://github.com/perladvnt/perldotcom>)

License

This work is licensed under a Creative Commons Attribution-NonCommercial 3.0 Unported License (<https://creativecommons.org/licenses/by-nc/3.0/>).

 (<https://creativecommons.org/licenses/by-nc/3.0/>)

Legal

Perl.com and the authors make no representations with respect to the accuracy or completeness of the contents of all work on this website and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. The information published on this website may not be suitable for every situation. All work on this website is provided with the understanding that Perl.com and the authors are not engaged in rendering professional services. Neither Perl.com nor the authors shall be liable for damages arising herefrom.