May you live in interesting times

Carl Mäsak

Granada, YAPC::EU 2015-09-03

Maybe you thought I was going to talk about macros.

Nope. Not today.

If you're starving for macros, check out 007.

Turning points. Bootstrapping a user base.

What is **production**?

It means different things to different people.

For some people it means something like “usable for work stuff”. For others, it means “we are actually using this for work stuff”.

For yet others it means “very bad things happen if this stops working”.

In this talk...

Production means...

...code I wrote...

...not for the Perl 6 tool chain itself...

...to serve me (or my employer) in some way...

...that actually gets run.

Thesis of this talk

Perl 6 is pretty cool, but all the features that make it cool are individually *pretty mundane*.

They're mundane because they're well-tested, and dependable. Things like types, lexical scoping, and good APIs.

As a corollary, the cool features that have traditionally been advertised as being the awesomest, are pretty marginal in Perl 6.

Challenge of this talk

I just commit to focusing only on boring features, didn't I?

Oh, shucks. ☺

…

As compensation, I promise to be very grumpy and cynical, so I sound like I have some “wisdom” and hard-earned “experience” to share.

Get off my lawn!

The `today` script

Let's start simple. This script runs every time I open a new terminal window or tab.

(But only once a day.)

```
  wed  26
  thu  27
  fri  28  X's birthday
  sat  29  Y's birthday
  sun  30
  mon  31
  tue   1  hours to $boss
  wed   2

masak@siddharta ~ $
```

It reminds me of important dates, like birthdays and anniversaries.

Think of it like a modern-day digital almanac.

slurp and spurt

First off, I note that I use `slurp` and `spurt` a lot in smaller scripts.

```
my $LAST_RUN_FILE = "%*ENV<HOME>/.today-last-run";
exit
    if try slurp($LAST_RUN_FILE) eq $today;
spurt($LAST_RUN_FILE, ~$today);
```

For Perl 5 users, CPAN provides, as usual: `File::Slurp` has your back (with `read_file` and `write_file`). But I'm glad these are built in.

qqw

I found I really liked `qqw` list quoting for hashes. For when you want rich strings but don't want to type any fat arrows and commas.

```
my %year-events = «
    "month 8, day 4"      "Obama's birthday"
    "month 7, day 14"     "Trump's birthday"
»;
```

Perl has always been good at making this kind of list convenient to write. Perl 6 even more so.

I think post-GLR there should be a `flat` there.

Naming your lexicals

Just like most of you, I've agonized about what to name my lexical variables. Perl 6 makes that slightly easier in some cases.

```
my $today = Date.today;
for ^7 -> $days {
    my $date = $today.later(:$days);
    # ...
}
```

I consider it a win every time I get to do this. It's like the callee decides the name of my variable, saving me the responsibility of choice.

Cautious `perl6`

If you have something running at every terminal startup, it turns out you need to cover the case where your Rakudo is rebuilding.

```
#!/bin/sh
if test $RAKUDO/src/gen/m-main.nqp -nt $RAKUDO/install/bin/perl6
    then exit
fi

# otherwise, invoke perl6
```

I learn shell/bash on a strictly need-to-know basis. In this case, the brief foray into the manpages meant that I could de-clutter my life a bit.

The `month` script

At the turn of every month, I send a work summary to my boss.

I used to manually list all the days from the previous month in a text file, but I got tired of that. So I wrote this script which does it for me.

Each month, it saves me a minute or two of tedious, repetitive, error-prone typing.

Output is always fiddly

I wanted the output to come out like you see on the right. The rule is: empty lines before and after a weekend.

In code, this comes out as:

```
say "" if $weekday eq "Monday" | "Saturday"
    && $_ !== $first;
say "$weekday $monthday $monthname";
```

Note how a “before” viewpoint is slightly forced upon us. Ergh.

```
Wednesday 1 July
Thursday 2 July
Friday 3 July

Saturday 4 July
Sunday 5 July

Monday 6 July
Tuesday 7 July
Wednesday 8 July
Thursday 9 July
Friday 10 July

Saturday 11 July
Sunday 12 July

...
```

Junctions

Hey, you probably noticed that *junction* flash by.

```
say "" if $weekday eq "Monday" | "Saturday"
    && $_ !== $first;
```

So what about not caring much for cool features?

Well, I don't. I use junctions as syntax, not as values. I consider the other ways an anti-pattern.

YMMV.

The pinnacle of Perl 6: MAIN

```
multi MAIN() {
    MAIN(LAST_MONTH.year, LAST_MONTH.month);
}

multi MAIN('current') {
    MAIN(TODAY.year, TODAY.month);
}

multi MAIN($year, $month) {
    # most of the code
}
```

Many excellent Perl 6 features (types, multi subs) meet and blend into a completed whole in MAIN.

constant

Constants look quite un-Perlish when you first see them. No sigil? What with the long keyword!?

```
constant TODAY = Date.today;
constant LAST_MONTH = TODAY.earlier(:month(1));
```

But they are very Perlish. They allow you to clearly specify intent (“this is not going to change”).

And `constant` isn't so long once you realize that a `BEGIN`-time declaration is included in the deal.

The sigil is optional.

Unexpected success: dates

The month script relies heavily on Perl 6's built-in date library. (As does the today script.) So do many Perl 6 scripts and programs.

We struggled to get dates and times right in Perl 6. Part of the problem is that lots of people have opinions about it, but it's actually really, really gnarly to get right. Like, horror-stories gnarly.

Let's look at an example.

Later

CPAN has a `DateTime::Duration`. It lets you specify a span of time in components: years, months, …, down to seconds.

The problem comes when you start to mix components. Do you add seconds first, or years? Does it matter? It kinda does, doesn't it? Augh!

```
my $date2 = $date.later(months => 1);
```

Perl 6's solution? Forcing you to be explicit.

We can loop over a range of dates

Just a small thing, but... ranges of dates work!

```
sub mday($day) { Date.new: $year, $month, $day }
my $first = mday 1;
my $last  = mday $first.days-in-month;

for $first .. $last {
    # ...
}
```

In fact, let's go back and improve the `today` script in the same way:

```
for Date.today .. Date.today.later(:7days) -> $date {
    # ...
}
```

Not bad.

mishu

Many years ago, for an old `$dayjob`, I wrote an IRC bot (in Perl 5) called `zarah`. She was a roaring success, and is still operational.

Ever since then, I've thought “I can do better”. This is the first sign of Second System Syndrome.

Now, finally, mishu is taking shape.

Restarting is how you stay alive

Steve Yegge wrote a great blog post, *The Pinocchio Problem*, about how great systems never reboot.

That's a tall order. Sometimes a system goes down, and you need to reload all the data and get back to where you were.

So a design principle of mishu is that it runs for very short pieces of time, catches up, and shuts down. Most of the time it's off.

A DSL instead of language hackery

For the longest time, I wanted to implement the restarting with *continuations*, and cool stuff!

But no. Bad masak! Not even Perl 6 has continuations, that's how insane they are.

Instead, look at the problem from a different angle, and implement something sane. In this case, basically a small interpreter that can be paused whenever we want.

Building small, building big

Perl 6 has this ladder of features that you start to lean on as your program scales up and becomes more “enterprisey”.

Packages. Classes. Roles. Grammars. Test.pm.

The experience of migrating from small to big is something Perl 6 does surprisingly well.

Hexagonal architecture

Speaking of testing.

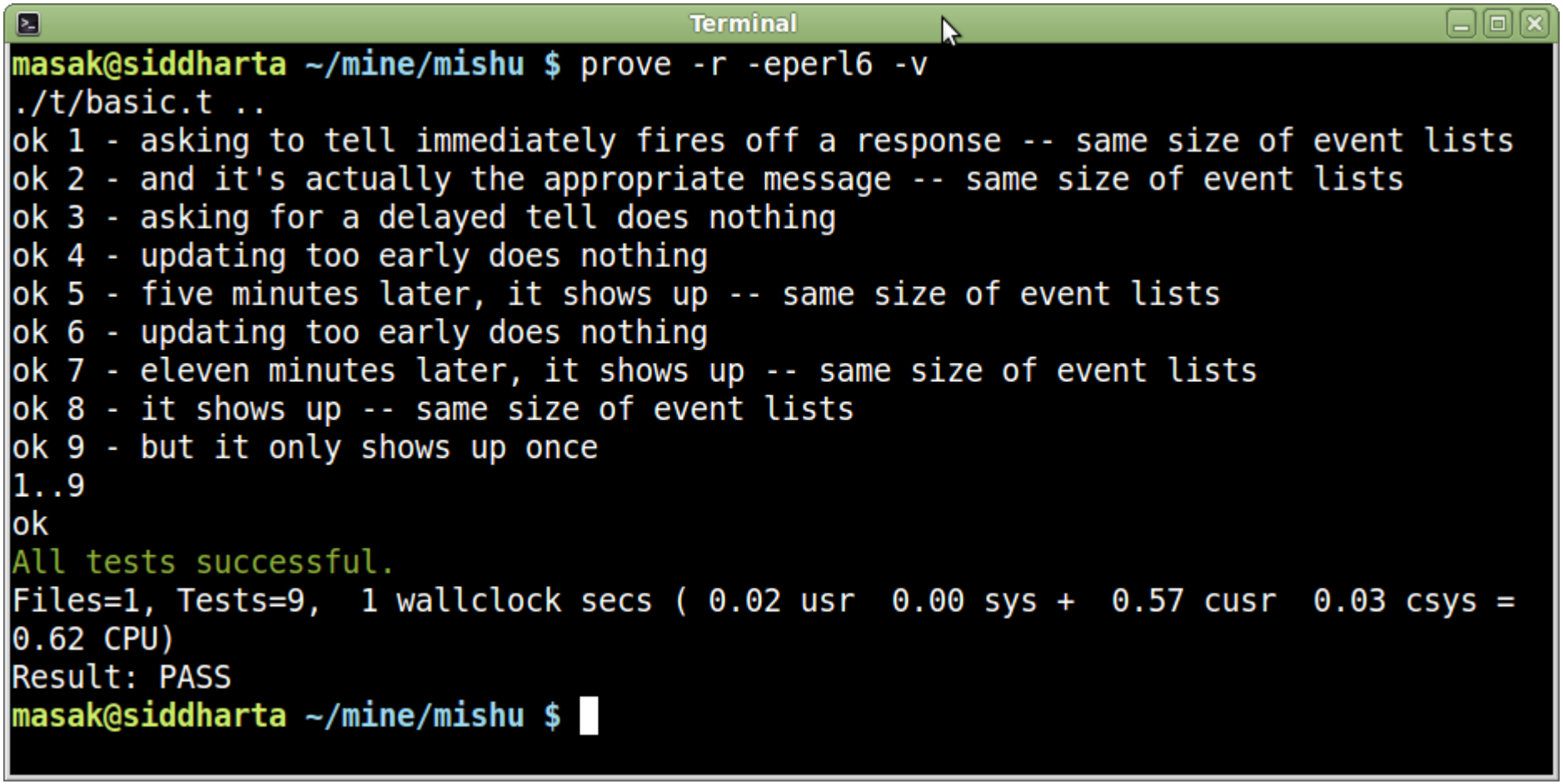
There's this idea that you can make an easy-to-test "core" model, and then plug in all these GUI/DB/IO things.

I feel like I'm doing that more and more. Most things have such a core, especially after you discover events. Not always, but really often.

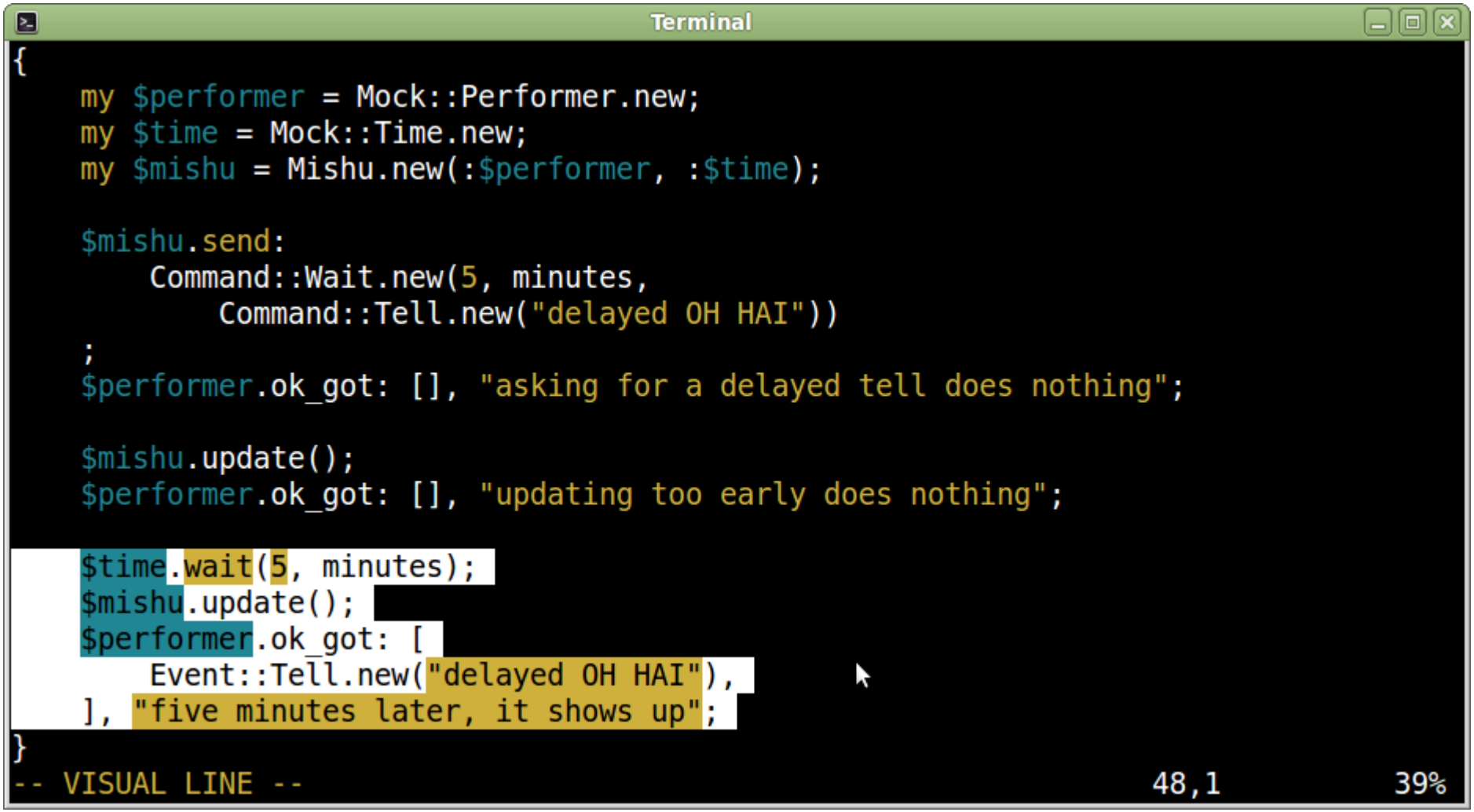
Testing gets to be *really* nice.

Look! Tests! And they pass!

This must mean something exists!



Looking at one of the tests



The course-abstracts repository

The “course-abstracts” repository is exactly what it says on the tin: a collection of course abstracts. They're in a Markdown-compliant data format.

But there's also a script that makes that makes a number of consistency checks against the data files. Kind of a custom linter for course abstracts.

Here's how the data format looks

Just so you know what we're dealing with.

```
# title

Rakudo Perl 6 and NQP Internals

# tweet

Take a deep dive into the Rakudo Perl 6 and NQP internals.

# blurb

Take a deep dive into the Rakudo Perl 6 and NQP internals. Build a small
compiler, complete with a simple class-based object system, to understand how
the toolchain works.

# abstract

This intensive 2-day workshop takes a deep dive into many areas of the Rakudo
Perl 6 and NQP internals...
```

What we're linting for

- No Byte-Order Marks in the file, injected by people who are either evil or use Windows.
- No missing required keys.
- No mistyped or made-up keys.
- No duplicate keys.
- No field values left empty.
- The “tweet” field shouldn't exceed 140 chars.
- ...and so on.

Many of these were installed on a “fool me once” basis.

What the linting code looks like

```
given Courses.new {
    expect_no .boms,                    "no files set up us the BOM";
    expect_no .missing_keys,            "all courses have the required keys";
    expect_no .unrecognized_keys,       "all entries have recognized keys";
    expect_no .empty_values,            "all values are non-empty";
    expect_no .overlong_tweets,         "all tweets are within 140 chars";
    expect_no .duplicate_keys,          "there are no duplicate keys";
    expect_no .tweets_without_dot,      "no tweet keys are missing final dot";

    expect_no .duplicate_codes,         "there are no duplicate codes";
    expect_no .en_courses_without_sv,   "all en courses have sv courses";
    expect_no .sv_courses_without_en,   "all sv courses have en courses";

    expect_no .todo_entries,            "there are no TODO entries";
}
```

It's actually an API on top of Test.pm. The `&expect_no` function uses `&is` under the hood.

Roles for great good

It took a while until I realized that I could factor the different concerns into individual roles.

```
class Courses
    does CheckKeys[
        required => <title tweet blurb abstract>,
        optional => <audience prerequisites agenda material length>
    ]
    does CheckValues
    does CheckTweets
    does CheckBom
    does CheckDuplicates {

    # much less stuff in here
}
```

This is an example of roles-for-reuse, because there's another class Talks which does another combination of these roles.

Subs *and* methods?

To a Perl 5 person, it might seem weird that Perl 6 allows both subs and methods in a class.

```
sub en_courses { ... }
sub sv_courses { ... }
sub courses { ... }

sub extract_code { ... }

method en_courses_without_sv { ... }
method sv_courses_without_en { ... }
```

But the distinction is very useful. Subs do what they always did: provide convenience functionality. Methods specify the object API. Inner language vs outer language.

These things but not those things

Most of us are familiar with this pattern. Hashes are useful in the sense that they provide a solution for this use case.

```
method en_courses_without_sv {
    my %courses = map { $_ => 1 }, en_course_codes();
    %courses{ sv_course_codes() } :delete;
    return %courses.keys.map($_
        ~ " has an English course but not a Swedish one");
}
```

But if you go “hey wait”, you might realize that this is not a hash problem in the first place.

Sets and hypers

What we really want is set difference.

```
method en_courses_without_sv {
    @( en_course_codes() (-) sv_course_codes() )
        >>~>> " has an English course but not a Swedish one";
}
```

Here hypers and sets work together to allow us to describe entire collections without `for` loops.

Pipes

Pipes are nice for when you want your data flow to run in the same direction as people read.

```
method missing_keys(@required) {
    @required ==>
        grep { %.entries{$_} :!exists } ==>
        map { $.file_and_key($_) }
}
```

Watch out, though! You still can't end a line with a closing curly (}) without also ending the statement!

A grammar

Almost as an afterthought, there's a grammar in there that just parses the necessary bits of Markdown-ish data syntax.

```
my grammar MarkData {
    token TOP {
        ^ <bom>? <entry>* $
    }

    token bom { \xFEFF }
    token entry { <heading> <data> \n? }

    # ...
}
```

psyde

Ever been to strangelyconsistent.org? The blog posts are generated by a ~240 line Perl 6 script.

The name “psyde” is a bad pun that stuck. The generator is based on Hakyll, a Haskell-based static site generator. Jekyll and Hyde, Hakyll and psyde.

Those 240 lines could be greatly generalized/de-duplicated, but over the years this script has done its job and hasn't needed improvement.

Overall structure

Roughly, psyde does four things:

- Generate the list of all posts
- Generate all individual articles (as needed)
- Generate the RSS feed (with last 10)
- Generate the index.html page (with last 3)

(Hakyll has a framework for doing dataflow; psyde just hardcodes the flow.)

The one thing to call out

I think this is the nicest function I ever wrote.

```
sub nonexistent-or-older($target, :than(@sources)!) {
    return $target.IO !~~ :e
           || $target.IO.changed before any(@sources).IO.changed;
}
```

Text::Markdown

I keep coming back to Markdown as something I need to parse. I don't know if you noticed, but...

...psyde uses it. A presentation framework I want to write for $work could really use it. Even course-abstracts got in on the action.

What I really want is a Markdown DOM in Perl 6. That's the holy grail.

Data-driven TDD

Here I just want to share a very different way to do TDD, that involves writing zero tests.

I have these 369 posts. They are test input. And I have a p5 Markdown module that can process them; so I have test output.

I just go through them in a sensible order and find the first thing that blows up. When it gets through all 369, I'll know I'm done.

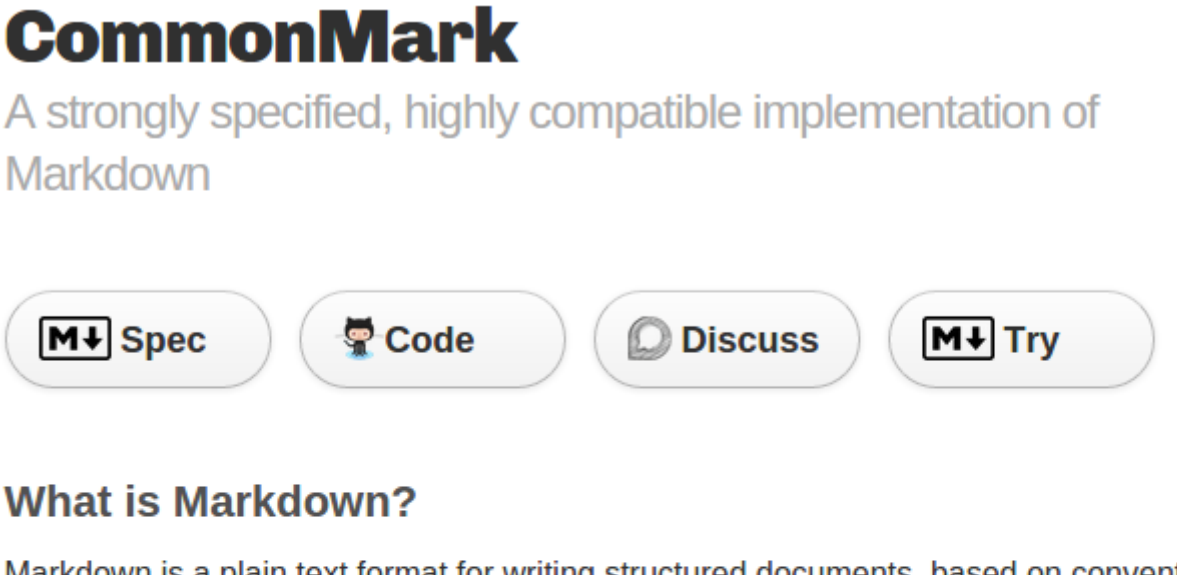
```
november-21-2008-a-torrent-of-things.markdown ..................................... [same]
november-22-2008-theres-more-than-one-way-to-write-it.markdown ..................... [same]
november-23-2008-the-rite-to-write.markdown ........................................ [same]
november-24-2008-evolving-things.markdown .......................................... [same]
november-25-2008-the-lexbug-of-death-is-dead.markdown .............................. [same]
november-26-2008-we-come-in-peace-bzzz.markdown .................................... [same]
november-27-2008-its-just-a-fleshwound.markdown .................................... [same]
november-28-2008-take-thee-annam-whatley.markdown .................................. [same]
november-29-2008-i-will-call-it-the-graphophone.markdown ........................... [same]
november-30-2008-improvements.markdown ............................................ [same]
that-was-the-month-that-was.markdown ............................................... [DIFF]
        --- p5-output/that-was-the-month-that-was.html  2015-09-03 10:31:27.335585677 +0200
        +++ p6-output/that-was-the-month-that-was.html  2015-09-03 10:31:29.819585632 +0200
        @@ -71,9 +71,11 @@
         <p>Finally, here are the things I promised I'd do, annotated with comments on what I
         <ul>
        -<li><dl>
        -<dt>Create three nice-looking layouts, which can all be used in p6w.</dt>
        +<li><dl></li>
        +</ul>
        +<p><dt>Create three nice-looking layouts, which can all be used in p6w.</dt>
         <dd>I created <strong>one</strong> really nice-looking layout for p5w, but I still h
s++ contributed a second layout.</dd>
         <dd> </dd>
        -</dl></li>
        +</dl></p>
        +<ul>
         <li><dl>
         <dt>Read a synopsis.</dt>

Successfully translated 38 Markdown files.
masak@siddharta ~/mine/simple-markdown-parser $
```

CommonMark

Once the parser can handle all my posts, I want to become compatible with CommonMark, a kind of gold standard for Markdown.

And yes, I expect fallout from my blog posts when doing this.



Conclusion

Perl 6 works well in production

I use it a lot in just that way.

Yay for boring features

Perl 6 has cool features. But the reason I stick with the language, and prefer to code in it, is that it provides a lot of sensible, safe, reliable, and well-designed features.

That also happen to mesh really well together.

But the interesting times should come from the problem domain, not from the language.

The future is stable

We've been living in the future for a while now.

Not everything in the future is flying cars and cool neon and talking animals. Sometimes it's just another day at work, snuggling up with your favorite cozy language, Perl 6.

And I like that.