Perl 6: what can you do today?

The state of the butterfly

Jonathan Worthington

Perl 6 in a slide

A new Perl

Multi-paradigm

Gradually typed

Still about getting stuff done

Scales better from script to application

Specified by its test suite and standard grammar

Designed and implemented by a great community

Show, don't tell

We'll take a concrete problem, and use Perl 6 to solve it

See how it copes, and if we get a good solution

I try to use Perl 6 every so often, to get a feel for where we're at and where the pain points are. For example:

2012: Implemented Rakudo::Debugger in Perl 6

2013: Used Perl 6 at a client to do static analysis of other languages, aiming to find hopefully-dead code and instrument it with logging. Confidently threw away over half a million lines in a 3 million line codebase.

Today's problem

There's a lot of interesting free data sets out there

A few weeks ago, I found a huge archive of historical global temperature data

No, it's not in JSON, or XML. Just structured-ish text. ☹

Felt like exploring it a little

Typical data-munging task

No agenda, just curiosity

A sample data file

Some meta-data at the top, followed by the average for each month of the year below a header "Obs:"

```
Number= 010010
Name= Jan Mayen
Country= NORWAY
Lat=   70.9
Long=    8.7
Height= 10
Start year= 1921
End year= 2009
...
Obs:
1921 -4.4 -7.1 -6.8 -4.3 -0.8  2.2  4.7  5.8  2.7 -2.0 -2.1 -4.0
1922 -0.9 -1.7 -6.2 -3.7 -1.6  2.9  4.8  6.3  2.7 -0.2 -3.8 -2.6
...
2008 -2.8 -2.7 -4.6 -1.8  1.1  3.3  6.1  6.9  5.8  1.2 -3.5 -0.8
2009 -2.3 -5.3 -3.2 -1.6  2.0  2.9  6.7  7.2  3.8  0.6 -0.3 -1.3
```

Let's parse it!

Perl has always been great for text processing - but too often the code becomes a tangle of ad-hoc regexes

Perl 6: break free of the limits of regular languages, by adding support for grammars

Enable building of well-structured, composable, extensible, maintainable parsers

Start at the TOP

We create a grammar, and in the special TOP rule describe the overall structure of the document

```
grammar StationDataParser {
    token TOP { ^ <keyval>+ <observations> $ }
    # ...
}
```

```
Number= 010010
Name= Jan Mayen
...
Obs:
1921 -4.4 -7.1 -6.8 -4.3 -0.8  2.2  4.7  5.8  2.7 -2.0 -2.1 -4.0
...
2009 -2.3 -5.3 -3.2 -1.6  2.0  2.9  6.7  7.2  3.8  0.6 -0.3 -1.3
```

Meta-data

Next, parse the key/value pairs; the $<key> syntax makes a named capture, and [...] is a non-capturing group

```
grammar StationDataParser {
    token TOP    { ^ <keyval>+ <observations> $          }
    token keyval { $<key>=[\w+] '=' \h* $<val>=[\N+] \n  }
    # ...
}
```

```
Number= 010010
Name= Jan Mayen
...
Obs:
1921 -4.4 -7.1 -6.8 -4.3 -0.8  2.2  4.7  5.8  2.7 -2.0 -2.1 -4.0
...
2009 -2.3 -5.3 -3.2 -1.6  2.0  2.9  6.7  7.2  3.8  0.6 -0.3 -1.3
```

Observations

The observations section starts with a literal string, followed by one or more (years of) observations

```
grammar StationDataParser {
    token TOP          { ^ <keyval>+ <observations> $          }
    token keyval       { $<key>=[\w+] '=' \h* $<val>=[\N+] \n  }
    token observations { 'Obs:' \h* \n <observation>+          }
    # ...
}
```

```
Number= 010010
Name= Jan Mayen
...
Obs:
1921 -4.4 -7.1 -6.8 -4.3 -0.8  2.2  4.7  5.8  2.7 -2.0 -2.1 -4.0
...
2009 -2.3 -5.3 -3.2 -1.6  2.0  2.9  6.7  7.2  3.8  0.6 -0.3 -1.3
```

Each year's observation

Each line has a year followed by temperatures separated by whitespace; %% specifies a quantifier separator

```
grammar StationDataParser {
    token TOP          { ^ <keyval>+ <observations> $          }
    token keyval       { $<key>=[\w+] '=' \h* $<val>=[\N+] \n  }
    token observations { 'Obs:' \h* \n <observation>+          }
    token observation  { $<year>=[\d+] \h* <temp>+ %% [\h*] \n }
    # ...
}
```

```
Number= 010010
Name= Jan Mayen
...
Obs:
1921 -4.4 -7.1 -6.8 -4.3 -0.8  2.2  4.7  5.8  2.7 -2.0 -2.1 -4.0
...
2009 -2.3 -5.3 -3.2 -1.6  2.0  2.9  6.7  7.2  3.8  0.6 -0.3 -1.3
```

Start at the TOP

Finally, we need a simple token that will match a temperature; little new except using quoting syntax

```
grammar StationDataParser {
    token TOP          { ^ <keyval>+ <observations> $          }
    token keyval       { $<key>=[\w+] '=' \h* $<val>=[\N+] \n  }
    token observations { 'Obs:' \h* \n <observation>+          }
    token observation  { $<year>=[\d+] \h* <temp>+ %% [\h*] \n }
    token temp         { '-'? \d+ \. \d+                       }
}
```

```
Number= 010010
Name= Jan Mayen
...
Obs:
1921 -4.4 -7.1 -6.8 -4.3 -0.8  2.2  4.7  5.8  2.7 -2.0 -2.1 -4.0
...
2009 -2.3 -5.3 -3.2 -1.6  2.0  2.9  6.7  7.2  3.8  0.6 -0.3 -1.3
```

Will it work?

Using our grammar to parse a file is just a case of calling its parsefile method, passing a file name:

```
say StationDataParser.parsefile('data/01/010010');
```

And...

Will it work?

Using our grammar to parse a file is just a case of calling its parsefile method, passing a file name:

```
say StationDataParser.parsefile('data/01/010010');
```

And...

```
#<failed match>
```

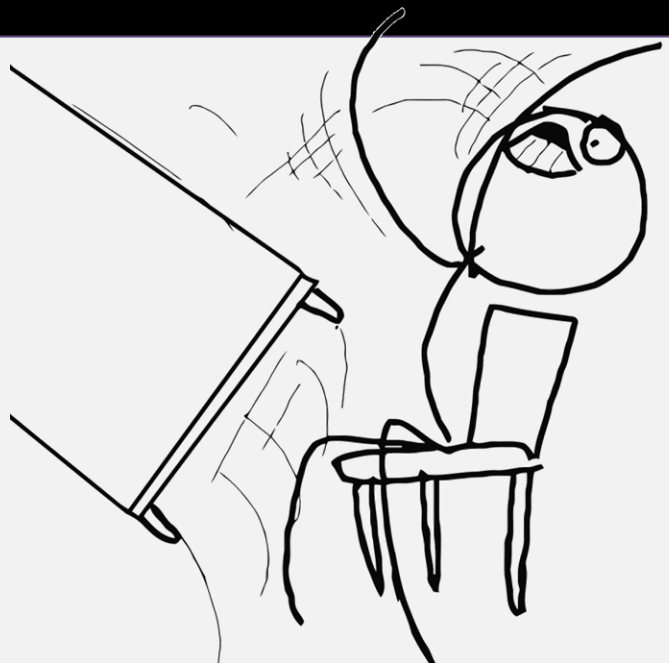
Will it work?

Using our grammar to parse a file is just a case of calling its parsefile method, passing a file name:

```
say StationDataParser.parsefile('data/01/010010');
```

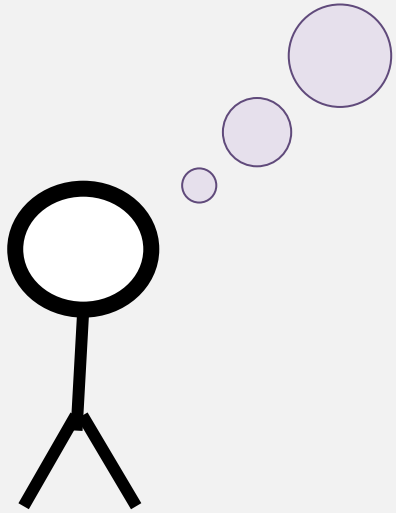
And...

```
#<failed match>
```



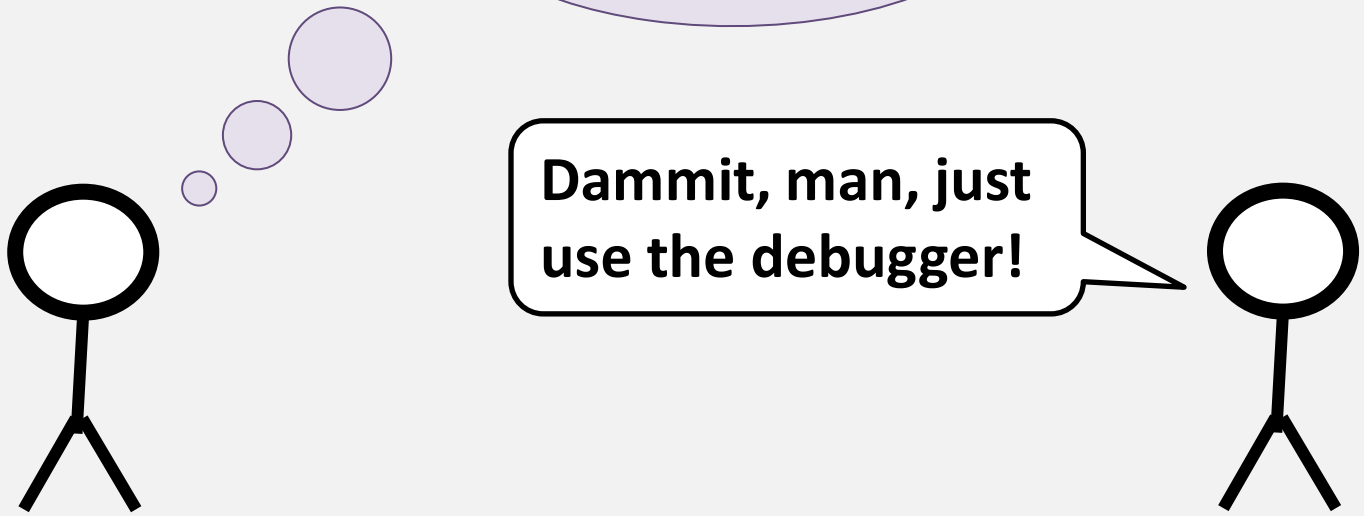
How on earth to debug this?

```
grammar StationDataParser {
    token TOP { ^ <keyval>+ <observations> $ }
    token keyval {
        $<key>=[\w+] '=' \h* $<val>=[\N+] \n
        { say "Got $<key> and $<val>" }
    }
}
```



How on earth to debug this?

```
grammar StationDataParser {
    token TOP { ^ <keyval>+ <observations> $ }
    token keyval {
        $<key>=[\w+] '=' \h* $<val>=[\N+] \n
        { say "Got $<key> and $<val>" }
    }
}
```



The debugger

Built specially for Perl 6, and written in Perl 6

Happily copes with stepping through...

BEGIN-time
EVAL'd code
macros
regexes
grammars

Let's try it!

<STOP! Demo time!>

A tree for free

The structure of the grammar is used to make us a tree of match objects (which work like arrays and hashes), which we might even use directly to mine the data!

```
my $parsed  = StationDataParser.parsefile('data/01/010010');

my $lowest  = Inf;
my $highest = -Inf;
for $parsed<observations><observation> -> $ob {
    for $ob<temp> -> $t {
        $lowest  min= +$t;
        $highest max= +$t;
    }
}

say "Lowest: $lowest, Highest: $highest";
```

```
Lowest: -14.5, Highest: 8
```

Great for a quick hack, but...

The tree we've got isn't too bad to pull data out of

However, if our grammar changes, so may the tree

Also, the parse tree keeps lots of information that we may not need, or the structure it provides may not be an ideal one for the way we want to use the data

So, we'll turn to...

Perl 6's object system (to create a nicer model)

Parse actions (to map from grammar to that model)

A StationData class

We'll declare a class to hold the data we find most interesting from each weather station

```
class StationData {
    has $.name;
    has $.country;
    has @.data;

    # Interesting methods come here...
}
```

All attributes in Perl 6 are private, and named $!foo

However, if we declare then with as $.foo instead then a public read-only accessor is generated for us

Actions class

Has methods named after the tokens in the grammar

```
class StationDataActions {
    method TOP($/) {
        ...
    }
    method keyval($/) {
        ...
    }
    method observations($/) {
        ...
    }
    method observation($/) {
        ...
    }
}
```

Will be called on a successful match, passing the appropriate piece of the tree as an argument

Keys and values

Produce a Pair for each key/value pair

```
class StationDataActions {
    method TOP($/) {
        ...
    }
    method keyval($/) {
        make ~$<key> => ~$<val>;
    }
    method observations($/) {
        ...
    }
    method observation($/) {
        ...
    }
}
```

Note that the ~ prefix operator grabs the text string that was matched, so we don't store the entire Match object

A year's observation

Produce a Pair mapping a year to a list of temperatures

```
class StationDataActions {
    method TOP($/) {
        ...
    }
    method keyval($/) {
        make ~$<key> => ~$<val>;
    }
    method observations($/) {
        ...
    }
    method observation($/) {
        make +$<year> => $<temp>.map(*.Num);
    }
}
```

The + prefix numifies; we when take each temperature Match object and coerce it to a Num

All the observations

Become a list of Pairs of year and temperatures

```
class StationDataActions {
    method TOP($/) {
        ...
    }
    method keyval($/) {
        make ~$<key> => ~$<val>;
    }
    method observations($/) {
        make $<observation>.map(*.ast).grep(*.value.none <= -99);
    }
    method observation($/) {
        make +$<year> => $<temp>.map(*.Num);
    }
}
```

Note that .ast gets the thing observation stored with make; we filter out years with any invalid readings

Constructing StationData

Finally, we bring it all together to make a StationData

```
class StationDataActions {
    method TOP($/) {
        make StationData.new(
            info => $<keyval>.map(*.ast).hash,
            data => $<observations>.ast
        );
    }
    method keyval($/) {
        make ~$<key> => ~$<val>;
    }
    method observations($/) {
        make $<observation>.map(*.ast).grep(*.value.none <= -99);
    }
    method observation($/) {
        make +$<year> => $<temp>.map(*.Num);
    }
}
```

Object construction

We've one remaining task: to map the named arguments we provide in object construction to the attributes

Here's a fairly simple way to go about this

```
class StationData {
    has $.name;
    has $.country;
    has @.data;

    submethod BUILD(:%info, :@data) {
        $!name    = %info<Name>;
        $!country = %info<Country>;
        @!data    = @data;
    }
}
```

Attributive parameters

If we're feeling lazy, though, there's a few shortcuts

First, we can specify that we want to bind the data parameter directly to the attribute

```
class StationData {
    has $.name;
    has $.country;
    has @.data;

    submethod BUILD(:%info, :@!data) {
        $!name    = %info<Name>;
        $!country = %info<Country>;
        @!data    = @data;
    }
}
```

Unpacking

Second, we can exploit data structure unpacking

We pull out the Name and Country keys, and have them bound directly to the appropriate attributes

```
class StationData {
    has $.name;
    has $.country;
    has @.data;

    submethod BUILD(
            :%info (:Name($!name), :Country($!country), *%),
            :@!data) {
        $!name    = %info<Name>;
        $!country = %info<Country>;
    }
}
```

Using the actions

The actions can be passed as an extra argument to the parsefile method

```
say StationDataParser.parsefile(
    'data/01/010010',
    :actions(StationDataActions)).ast;
```

The ast method on the final result is used to get the StationData instance made by TOP

And...it works!

```
StationData.new(name => "Jan Mayen", country => "NORWAY", data =>
Array.new(1921 => (-4.4e0, -7.1e0, -6.8e0, -4.3e0, -0.8e0, 2.2e0,
4.7e0, 5.8e0, 2.7e0, -2e0, -2.1e0, -4e0).list.item, 1922 => (-
0.9e0, -1.7e0, -6.2e0, -3.7e0, -1.6e0, 2.9e0, 4.8e0, 6.3e0, 2.7e0,
-0.2e0, -3.8e0, -2.6e0).list.item, ...))
```

So far...

```
grammar StationDataParser {
    token TOP          { ^ <keyval>+ <observations> $             }
    token keyval       { $<key>=[<-[=]>+] '=' \h* $<val>=[\N+] \n }
    token observations { 'Obs:' \h* \n <observation>+             }
    token observation  { $<year>=[\d+] \h* <temp>+ %% [\h*] \n    }
    token temp         { '-'? \d+ \. \d+                          }
}

class StationData {
    has $.name;
    has $.country;
    has @.data;

    submethod BUILD(:%info (:Name($!name), :Country($!country), *%), :@!data) {
    }
}

class StationDataActions {
    method TOP($/) {
        make StationData.new(
            info => $<keyval>.map(*.ast).hash,
            data => $<observations>.ast
        );
    }
    method keyval($/) {
        make ~$<key> => ~$<val>;
    }
    method observations($/) {
        make $<observation>.map(*.ast).grep(*.value.none <= -99);
    }
    method observation($/) {
        make +$<year> => $<temp>.map(*.Num);
    }
}

say StationDataParser.parsefile('data/01/010010', :actions(StationDataActions)).ast;
```

36 lines of code (including whitespace)

A nice parser

A basic model class

Action methods to perform a mapping

Yearly means

We'd like to produce a graph of the mean temperature each year for a station

We create a method that will calculate them...

```
class StationData {
    ...

    method year_means() {
        ...
    }
}
```

Yearly means

We'd like to produce a graph of the mean temperature each year for a station

...define inside it a (lexical) mean sub...

```
class StationData {
    ...

    method year_means() {
        sub mean(@values) {
            [+](@values) / @values
        }
        ...
    }
}
```

Yearly means

We'd like to produce a graph of the mean temperature each year for a station

...and map each data pair to a pair of year and mean.

```
class StationData {
    ...

    method year_means() {
        sub mean(@values) {
            [+](@values) / @values
        }
        @!data.map(-> $year_pair {
            $year_pair.key => mean $year_pair.value
        });
    }
}
```

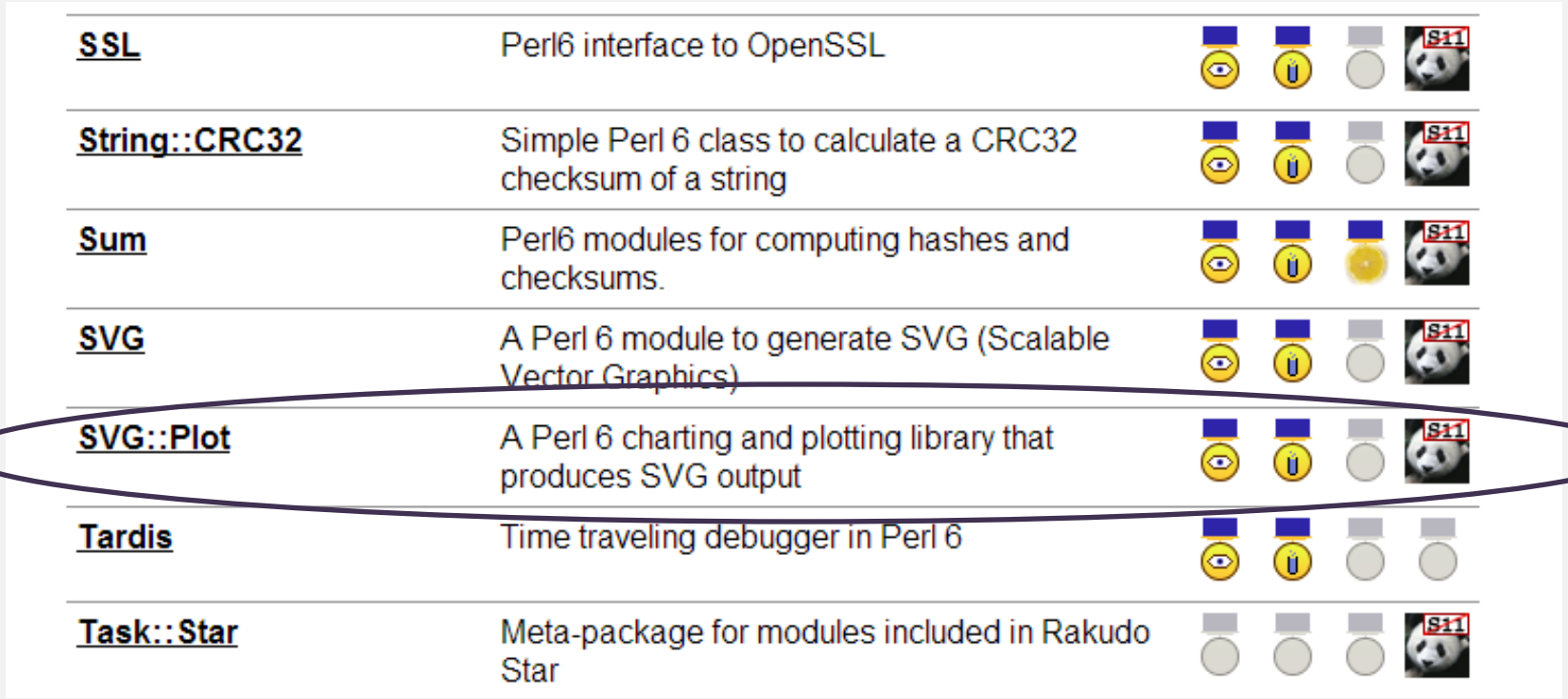
What's the plot for graphs?

I really don't want to write my own graph plotter

What's the plot for graphs?

I really don't want to write my own graph plotter

modules.perl6.org to the rescue!



SSL	Perl6 interface to OpenSSL
String::CRC32	Simple Perl 6 class to calculate a CRC32 checksum of a string
Sum	Perl6 modules for computing hashes and checksums.
SVG	A Perl 6 module to generate SVG (Scalable Vector Graphics)
SVG::Plot	A Perl 6 charting and plotting library that produces SVG output
Tardis	Time traveling debugger in Perl 6
Task::Star	Meta-package for modules included in Rakudo Star

Producing a graph

Use the SVG and SVG::Plot modules:

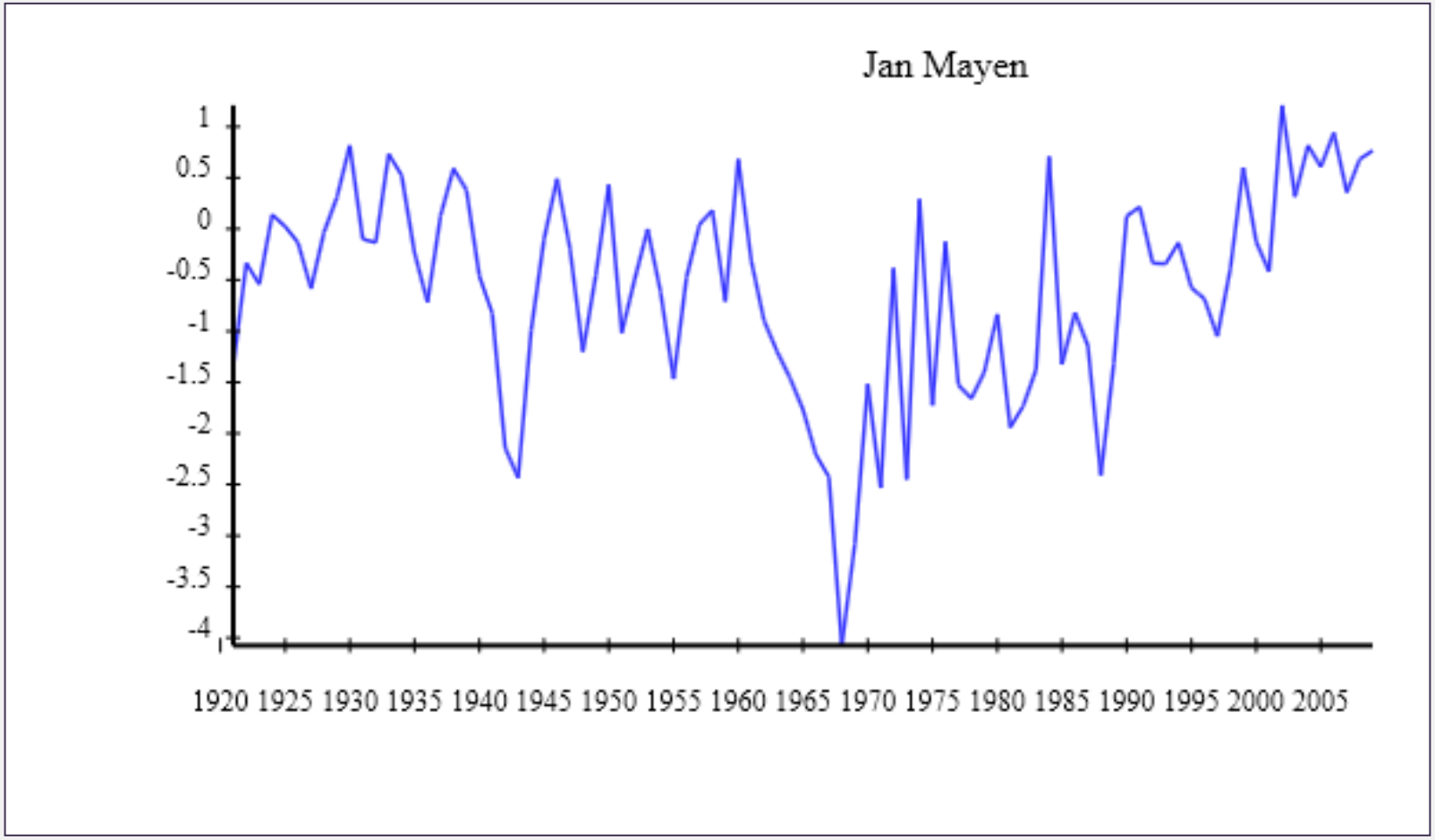
```
use SVG;
use SVG::Plot;
```

Then use it to produce a graph of the yearly means:

```
my $station = StationDataParser.parsefile('data/01/010010',
    :actions(StationDataActions)).ast;
my @means = $station.year_means;

spurt "means.svg", SVG.serialize(SVG::Plot.new(
    width  => 600,
    height => 350,
    x      => @means.map(*.key),
    values => [[@means.map(*.value)]],
    title  => $station.name
).plot(:xy-lines));
```

The first temperature graph

Here is the result:



Statistics by country

It would be interesting to look at yearly means of an entire country

This can be computed by finding the mean of the mean temperatures for all stations in the country

We can start by defining a CountryData class:

```
class CountryData {
    has $.name;
    has @.stations;

    submethod BUILD(:$!name, :@!stations) { }
}
```

Many stations, categorized by country

We now start to read in all of the station files in a directory, getting a StationData for each one

```
my @stations = do for dir('data/01') -> $file {
    StationDataParser.parsefile($file,
        :actions(StationDataActions)).ast;
};
```

These can then be categorized by country, and a CountryData constructed for each one

```
my @countries = do for @stations.categorize(*.country) {
    CountryData.new(name => .key, stations => .value)
}
```

Duplicate the yearly means code?

Calculating the yearly means over a bunch of stations will clearly be a bit different than doing it for one

We could implement it separately in both StationData and CountryData...but once we start calculating other things, it will be a lot of duplication ☹

Insight:

Handling data for just one station is a special case of handling data for many stations

This will allow us to factor out the yearly means code ☺

StationData provides one data set

The StationData class is updated with a datasets method, which packages its single set of data up in an array

```
class StationData {
    has $.name;
    has $.country;
    has @.data;

    submethod BUILD(
        :%info (:Name($!name), :Country($!country), *%),
        :@!data) { }

    method datasets() { [@.data] }
}
```

CountryData provides many data sets

Meanwhile, CountryData maps the data sets out of each of the stations, caching this work in an attribute

```
class CountryData {
    has $.name;
    has @.stations;
    has @.datasets;

    submethod BUILD(:$!name, :@!stations) {
        @!datasets = @!stations.map(*.data.item);
    }
}
```

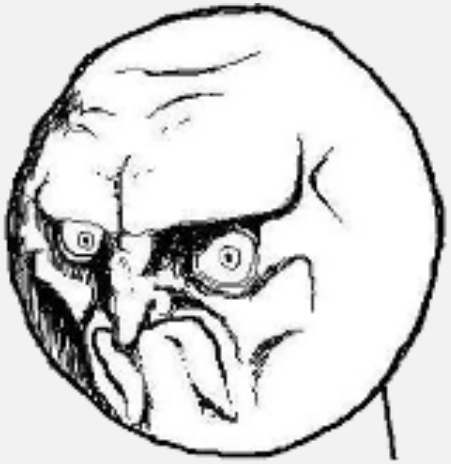
The attribute gets a public accessor, meaning that there is also a datasets method available

Where to put year_means?

In a common base class, maybe?

Where to put year_means?

In a common base class, maybe?



Inheritance is a mechanism for specialization and extension, and classes are about responsibility

Perl 6 provides us with roles for code re-use

Yearly statistics calculations go in a role

The code for year_means is moved into a role, and updated to work over many datasets

```
role YearlyStatistics {
    method year_means() {
        sub mean(@values) {
            [+](@values) / @values
        }
        my %station_means;
        for self.datasets -> @year_pairs {
            for @year_pairs {
                my ($year, $temps) = .key, .value;
                push %station_means{$year}, mean $temps;
            }
        }
        %station_means.sort(*.key).map({ $_.key => mean $_.value })
    }
}
```

Doing the role

Roles are incorporated into classes using does

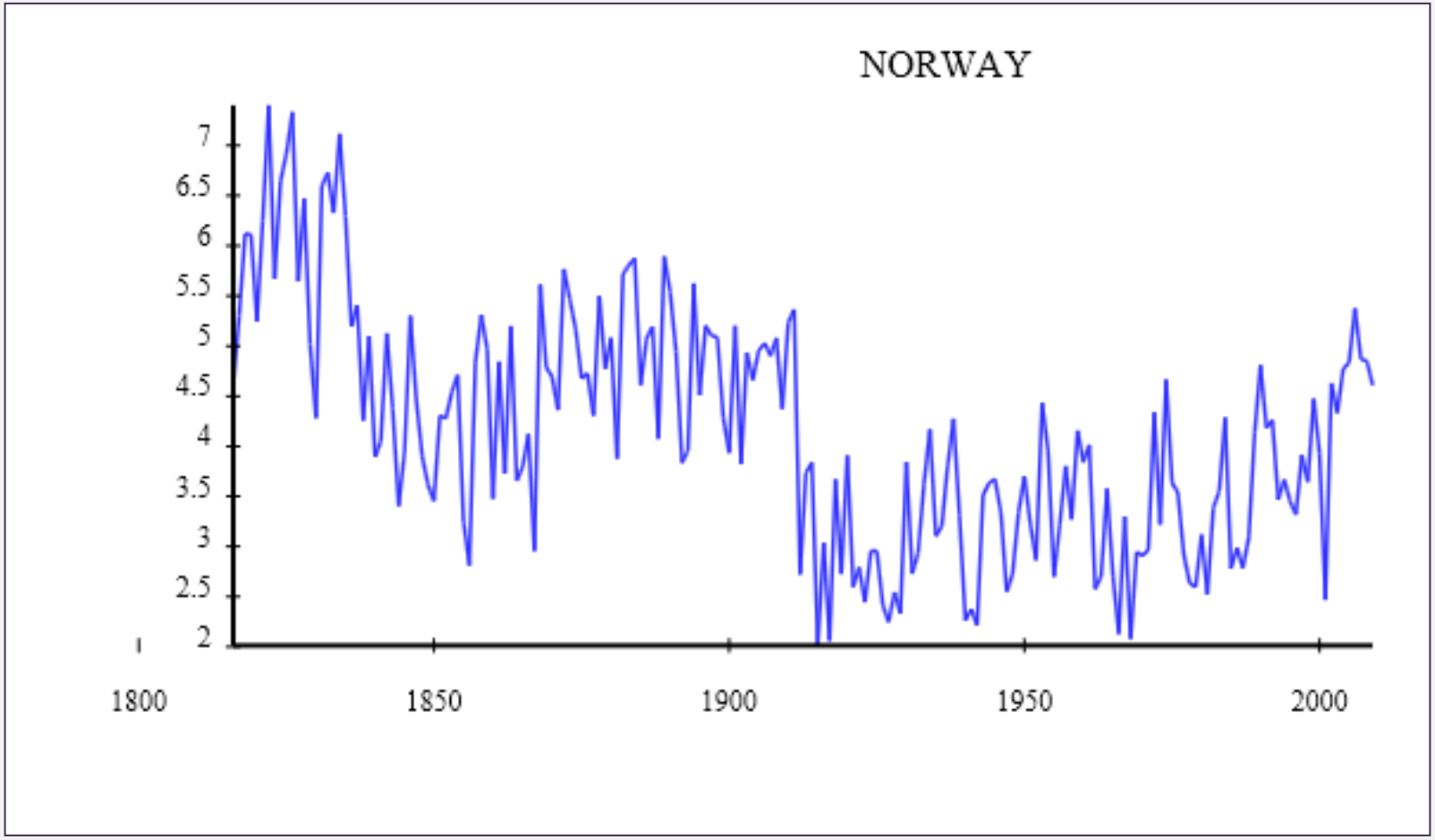
```
class StationData does YearlyStatistics {
    ...
}

class CountryData does YearlyStatistics {
    ...
}
```

This performs flattening composition, meaning any methods in the role are "copied" into the class

If you compose two roles that try to bring in a method of the same name, it is a compile-time error

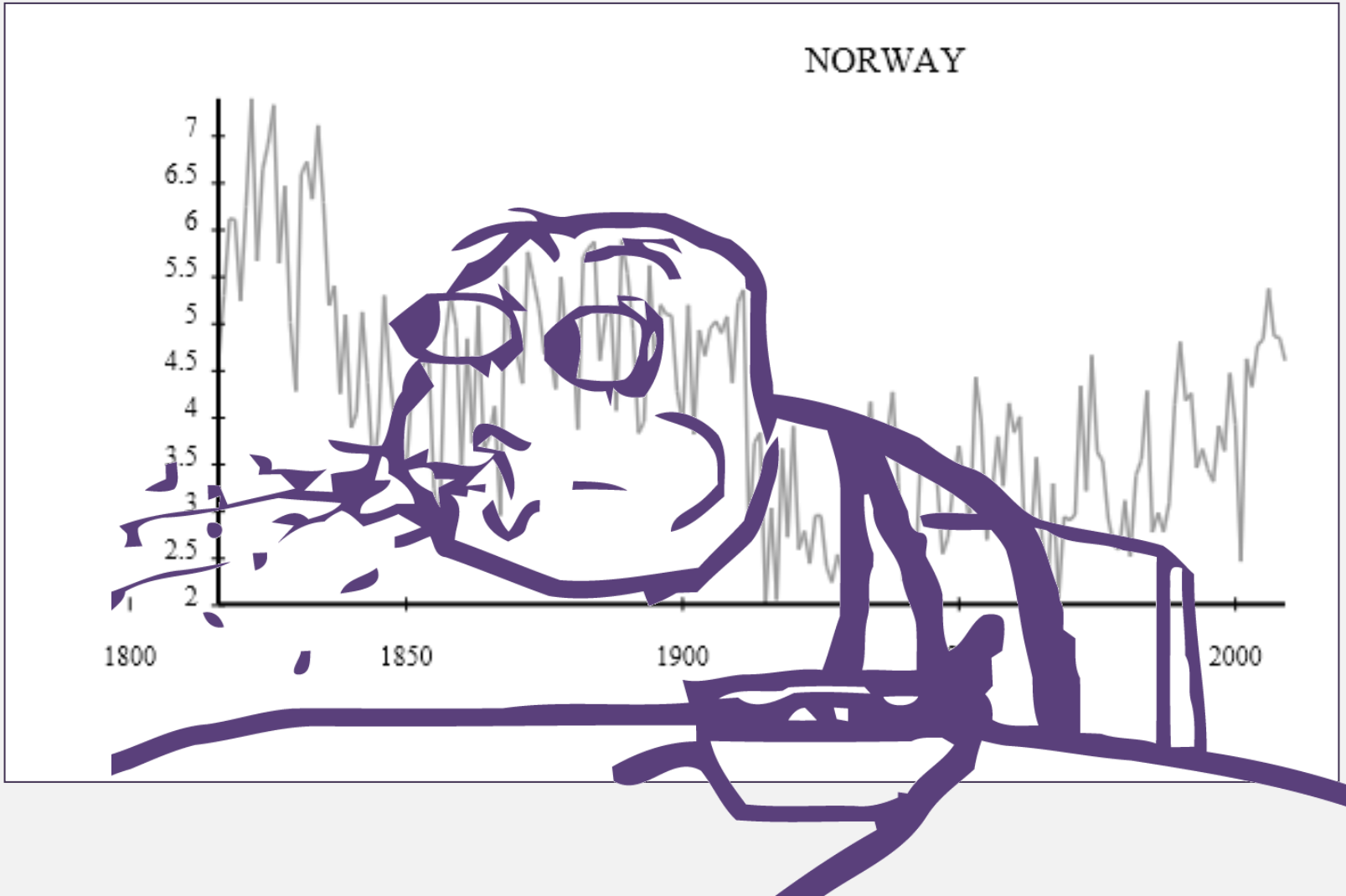
The graph

Here it is!



The graph

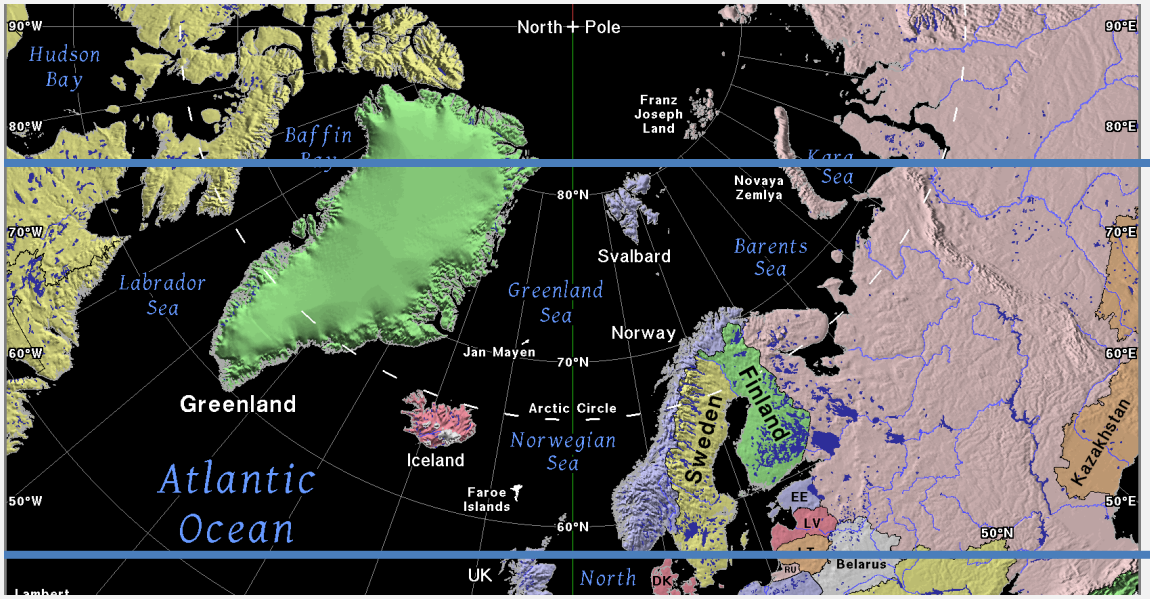
Apparently, Norway has been cooling massively!



Lies, dammed lies, and data

We don't have data from every year for every station. The two big transition points in the graph are in years where we gain data from new stations.

Norway is a huge country north to south, especially once you factor in the islands in the Arctic!



> 80° North

< 60° North

Valid years

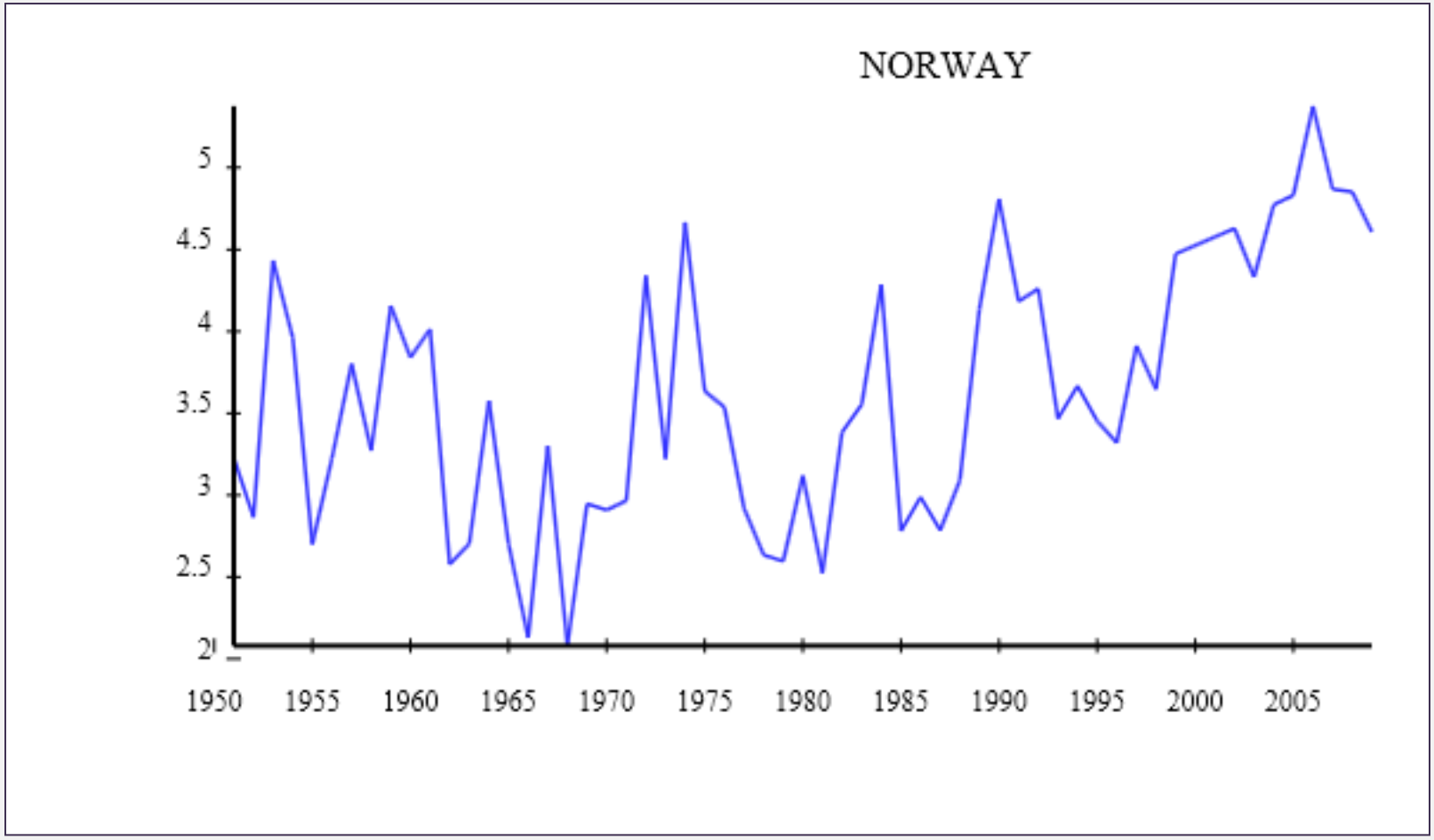
Build up a hash whose keys are years where all datasets can provide us with data

```
role YearlyStatistics {
    has %!valid_years = do {
        my %valid_count;
        my $num_datasets = self.datasets.elems;
        for self.datasets -> @year_pairs {
            for @year_pairs {
                %valid_count{.key}++;
            }
        }
        %valid_count.grep(*.value == $num_datasets)
    }
    ...
}
```

Skip invalid years in year_means

The graph

Looks rather more reasonable



Highs and lows

Could also plot yearly high and low temperatures

First, factor out the code that loops over datasets into a private method

```
role YearlyStatistics {
    ...

    method !for_valid_years(YearlyStatistics: &thing_to_do) {
        for self.datasets -> @year_pairs {
            for @year_pairs {
                my ($year, $temps) = .key, .value;
                next unless %!valid_years{$year}:exists;
                thing_to_do($year, $temps);
            }
        }
    }
}
```

Highs and lows

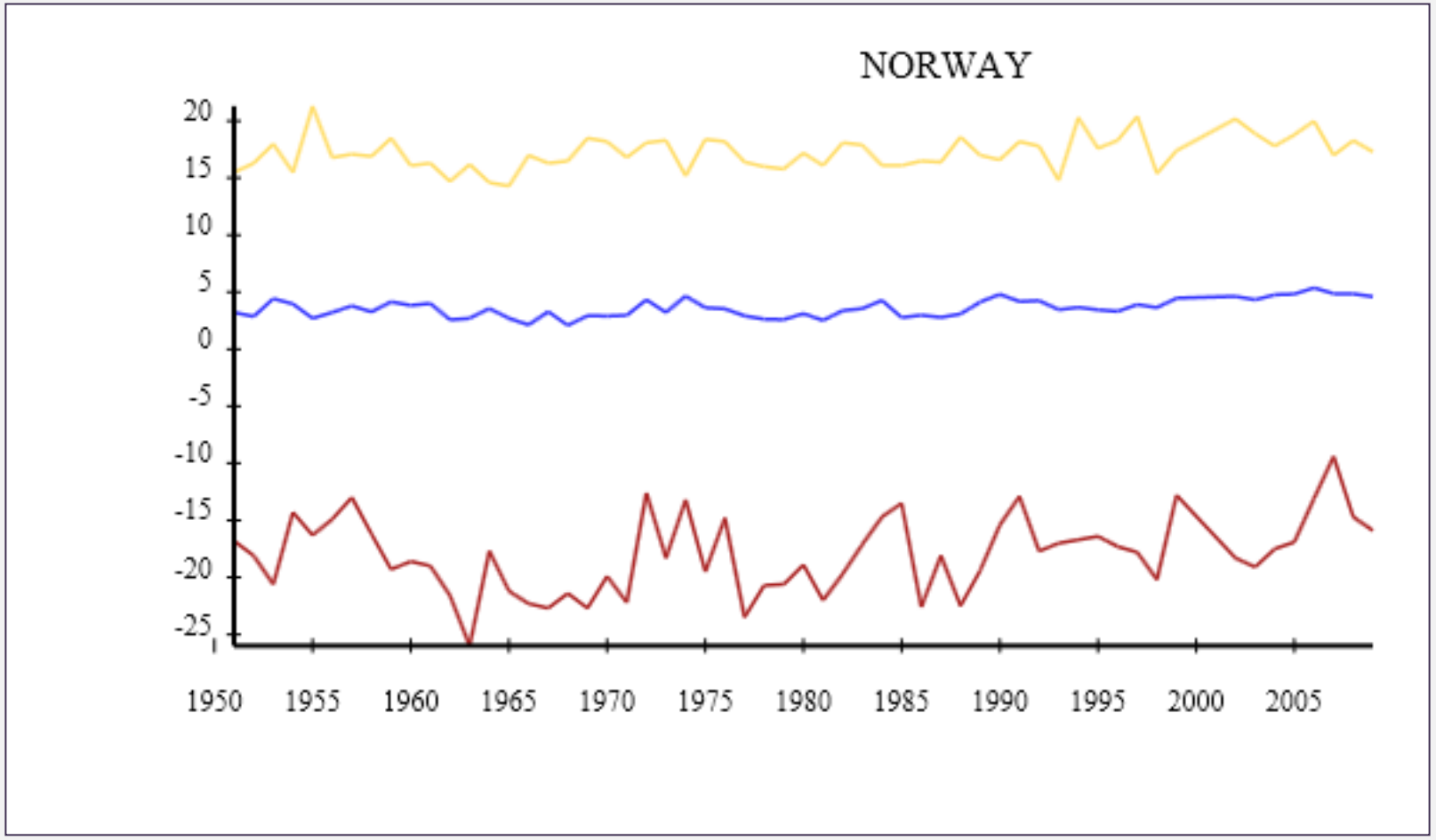
Then just use a hash to keep track of the lowest temperature we saw for each year across all stations

```
role YearlyStatistics {
    ...
    method year_lows() {
        my %year_lows;
        self!for_valid_years: -> $year, @temps {
            %year_lows{$year} min= @temps.min;
        }
        %year_lows.sort(*.key).map(*.value)
    }

    ...
}
```

Code for maximum temperatures is similar

Highs and lows in Norway

Probably the most interesting so far...



MoarVM

So far, we've been running the Rakudo Perl 6 implementation atop of the Moar Virtual Machine

Rather new, though already quite stable

Running Rakudo on MoarVM, rather than Parrot or the JVM, offers lower startup times and lower memory use

Moar is designed to enable a lot of interesting optimizations, especially type specialization, however these are still being implemented

Still typically faster than Rakudo on Parrot

Rakudo on the JVM

Awful startup time

But often, once you get past the slow start and it's had a chance to JIT comple things, this is (sometimes by a long way) the fastest Rakudo Perl 6 backend

Good for longer-running things

So what does it take to switch?

Nothing much, for us!

Just use perl6-j instead of perl6-m

Use all the cores!

Modern CPUs are typically multi-core

Rakudo on the JVM is the current leader in terms of implementing Perl 6's concurrency related features

We've quite a few files of climate data to crunch through, and it could be interesting to see how well we can exploit our multi-core CPU

Parallel processing of the stations

The data is spread over many directories...

```
my @stations = do for dir('data') -> $subdir {
    do for dir($subdir) -> $file {
        StationDataParser.parsefile($file,
            :actions(StationDataActions)).ast;
    }
}
```

Parallel processing of the stations

The data is spread over many directories...

```
my @stations = do for dir('data') -> $subdir {
    do for dir($subdir) -> $file {
        StationDataParser.parsefile($file,
            :actions(StationDataActions)).ast;
    }
}
```

So let's process them concurrently!

```
my @stations = await do for dir('data') -> $subdir {
    start {
        do for dir($subdir) -> $file {
            StationDataParser.parsefile($file,
                :actions(StationDataActions)).ast;
        }
    }
}
```

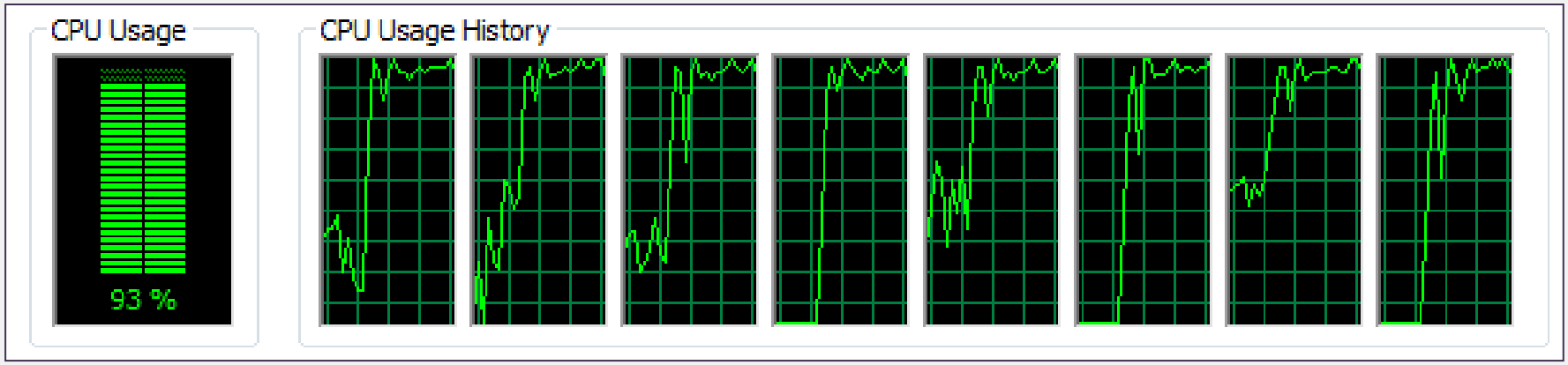
Parallel generation of the graphs

We can also produce the graphs for each of the countries in parallel too - by introducing exactly the same kind of small changes into our program

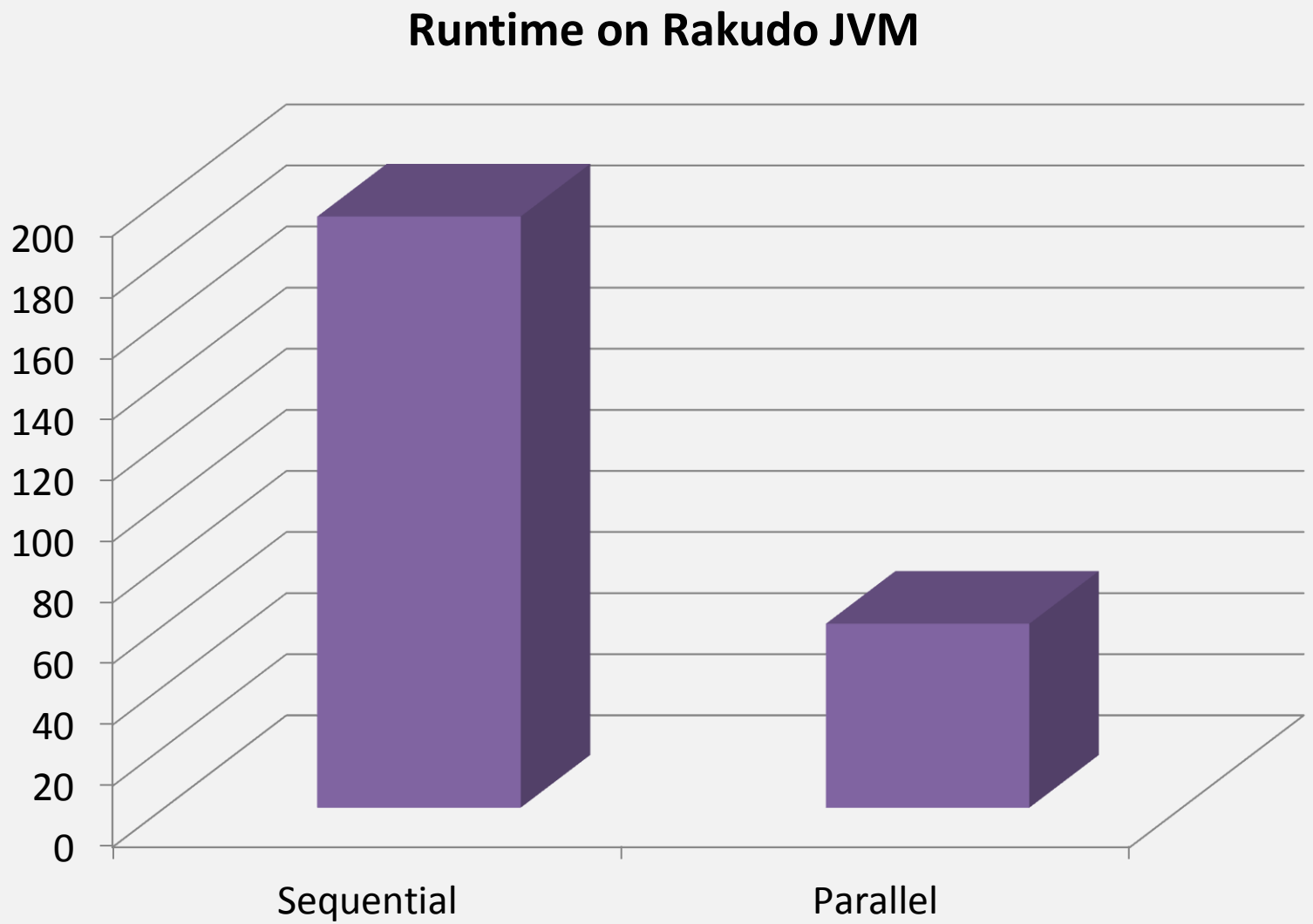
```
await do for @countries {
    next unless .years.elems > 1;
    start {
        spurt "{.name.lc}.svg", SVG.serialize(SVG::Plot.new(
            width  => 600,
            height => 350,
            x      => .years,
            values => [[.year_means], [.year_highs], [.year_lows]],
            title  => .name
        ).plot(:xy-lines));
        say "Wrote graph for {.name}";
    }
}
```

But does it use all the cores?

But does it use all the cores?



> 3 times speedup, on a quad-core box



So…

What is the state of Perl 6?

The good news

Perl 6 is a rich and expressive language

You can build stuff in it today - we just did

The community around Perl 6 is great

Feature wise, most of the way there; at some point soon, we'll work out what of the missing things are for > 6.0

Can run on different VMs, which gives us confidence that the executable specification (in the form of the test suite) is working out as hoped

The bad news

For many people's needs, it's not fast enough yet

Can be memory-hungry (MoarVM shows promise here, however there's still a need for improvement)

Concurrency only provided on JVM so far

Perl 5 interop (and therefore access to CPAN) still a work in progress, and the Perl 6 module collection is small

Documentation is not yet awesome; some progress on reference documentation, but there's a lack of tutorials

Reasons for optimism

Given we are most of the way there on features, the focus can now be on speed and memory use

We've already got a long list of optimizations to put in, rather than having no idea what to do in this area

Async I/O and concurrency support are expected to be in place on the MoarVM backend within a few months

A Perl 5 interoperability with Rakudo on MoarVM grant has been funded, which will provide access to CPAN and provide a migration strategy; v5 is also promising

Next FOSDEM, I want to be able tell you...

We've made things a lot faster

We've reduced memory usage

We've got working Perl 5 interop

Concurrency support is more evenly distributed

The documentation is more complete

The community is still as friendly as ever

Learning more

On the web:

www.perl6.org

On IRC:

#perl6 on freenode.org

The Perl 6 IRC channel is a helpful and welcoming place; do drop by some time! ☺

Give the Perl booth here at FOSDEM a visit

Thank you!

Questions?

(If time, otherwise see me at the booth)

If you want to contact me…

Email: jnthn@jnthn.net

Twitter: @jnthnwrthngtn