

Everyday Lessons from Rakudo Architecture

A scenic view of a coastal town at sunset. The sun is low on the horizon, casting a golden glow over the landscape. The town is built on a hillside overlooking the water, with several islands visible in the distance. The sky is a mix of orange and yellow, and the water reflects the sun's light.

Jonathan Worthington

What do I do?



**I teach our advanced
C#, Git and software
architecture courses**

**Sometimes a mentor at
various companies in
Sweden**



**Core developer on the
Rakudo implementation
of Perl 6**

**Focus on meta-object
protocol and the
gradual type system**

Why this session?

**While most developers are not building compilers,
they typically are building systems that...**

Are relatively complex

Need testing

Deal with languages in some form

**In these situations, I've found it valuable to draw
lessons from some of the things that have worked
well in dealing with such situations in Rakudo**



**Decouple around well-defined
data structures**

Rakudo at a high level

A series of strong isolated stages, passing well defined data structures between them

Text

Frontend (parse, build AST)

QAST

Optimizer

QAST

QAST -> VM AST

**JAST,
MAST,
PIRT**

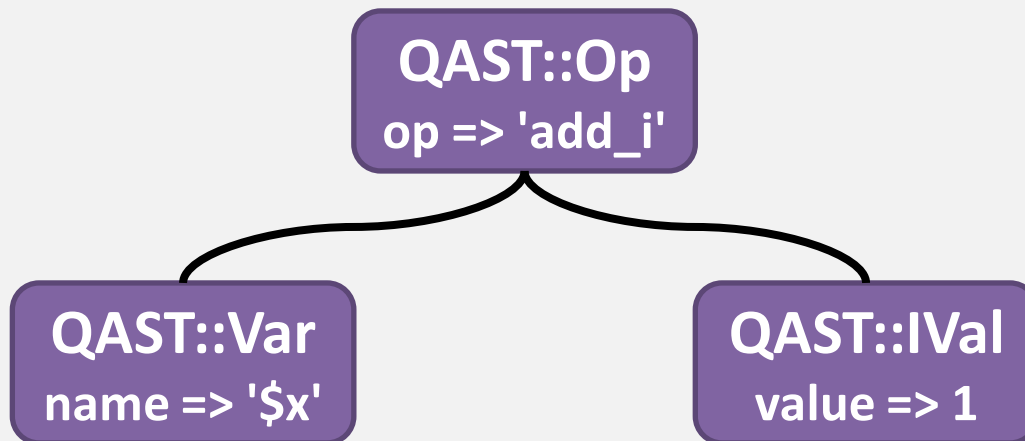
Bytecode

Assembler

QAST: making multi-backend sane

Between the frontend and the backend, we pass a QAST tree, representing the semantics of the program, abstract from any particular VM

This well-defined data structure is what enables a clean decoupling of frontend and backend



Another example: Git

Commit history is represented as a DAG of commit objects, each point to its parent(s)

Each commit object points to trees (representing a version of a directory), which in turn point to blobs (representing a version of a file)

Pretty much all the commands can be understood in terms of how they build, examine or mutate these various data structures

A quote from Linus Torvalds

“... git actually has a simple design, with stable and reasonably well-documented data structures. In fact, I'm a huge proponent of designing your code around the data, rather than the other way around, and I think it's one of the reasons git has been fairly successful ... I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”

So, what of OOP?

OO tells us to encapsulate data and expose behaviour. But here, we're looking at exposing data. That's what a data structure does!

Rakudo's stages use OOP quite heavily on the inside. You'll find examples of classes, inheritance, role composition and mix-ins all being put to good use.


But a small number of *well-defined, rarely mutated (or immutable)* data structures form an even looser coupling than behaviours on objects.



Testing is easiest when the system resembles a filter

Testing a compiler

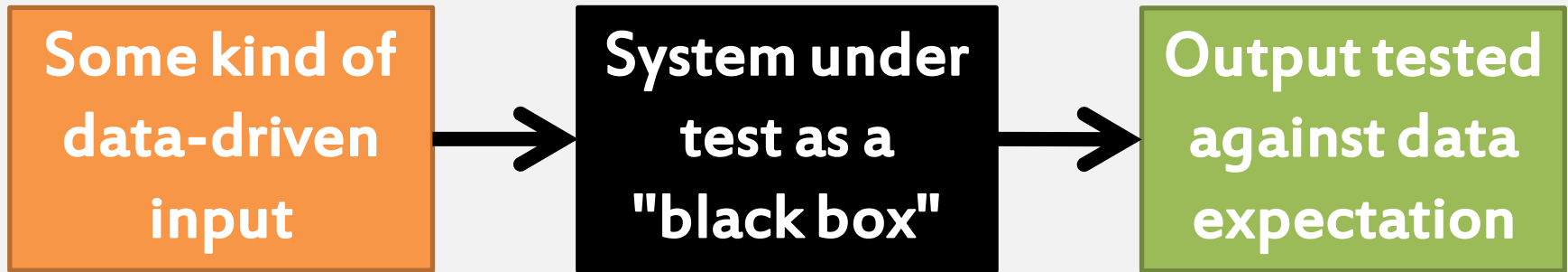
Compilers are amongst the easiest kinds of software to write automated tests for

- 
- **Write a small program that exercises a feature or feature interaction**
 - **Compile the program and execute it**
 - **See if the output is what was expected**

Observing the ease of testing, I pondered why, and how to apply it to other kinds of software...

Replicating the ease

The pattern seems to be something like...



Any time we can treat a system as being a black box that transforms one stream into another, we win.

But can we build business systems, that need to do persistence and stuff, like this?

The challenge of state

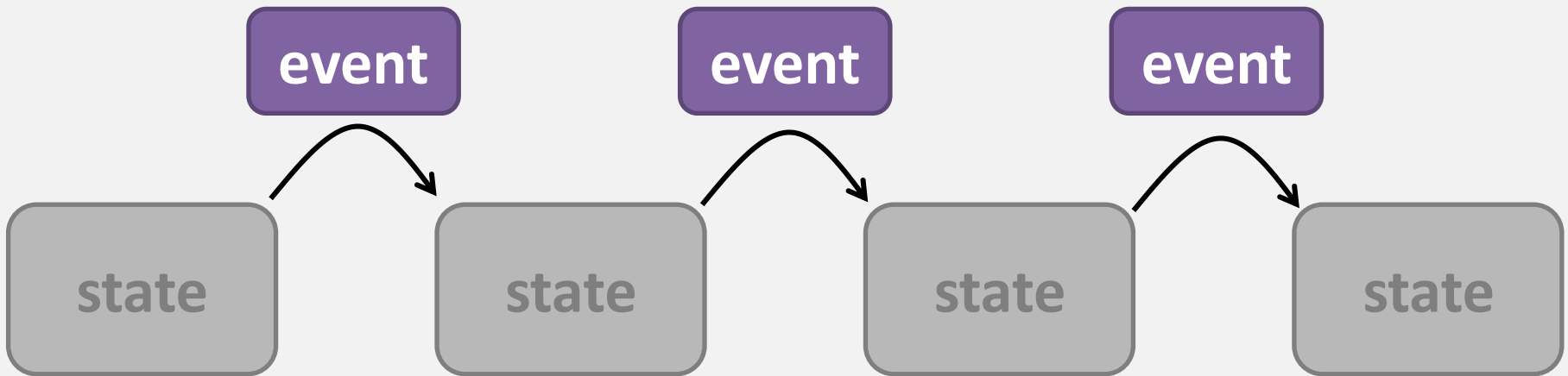
Compilers don't have to worry about prior state. Business logic almost always has to make decisions based on the current system state, however.

Setting objects up to represent an existing state makes retaining encapsulation hard, and mocking database access is not too fun either – and certainly doesn't feel filter-like!

Is there a way out?

Event sourcing

If we capture every decision made in the system as an event, and persist those events, we can always assemble the latest state from them



Better still, we can do this without breaking the encapsulation of objects; we just feed them events

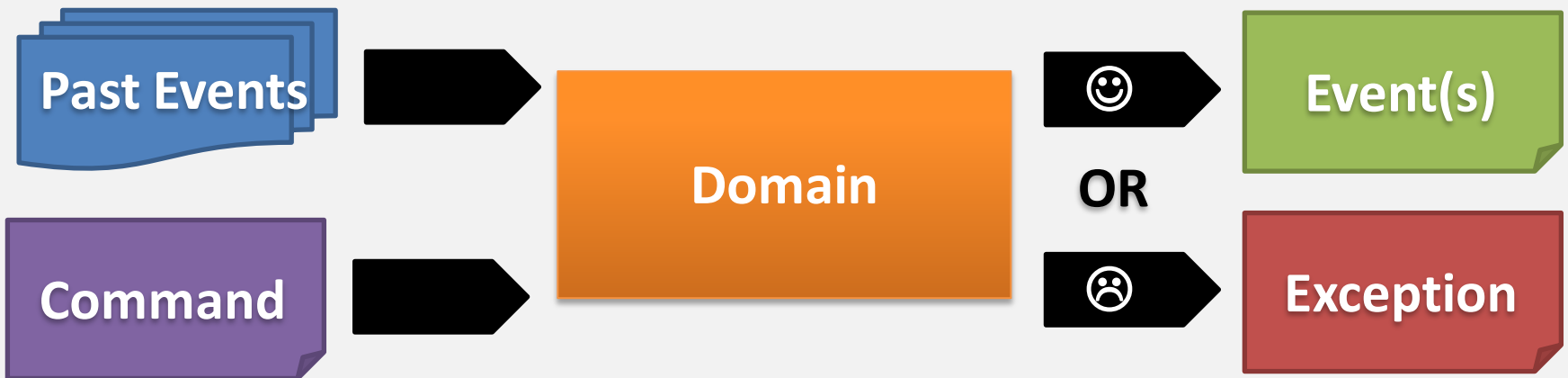
Commands and events

Events model decisions. We need two more things.

Commands to model requests

Exceptions to model refusing a request

With this, things look more filter-like!



BDD

This maps very naturally to the BDD approach to testing, which breaks a test into:

Given

(Zero or more past events)

When

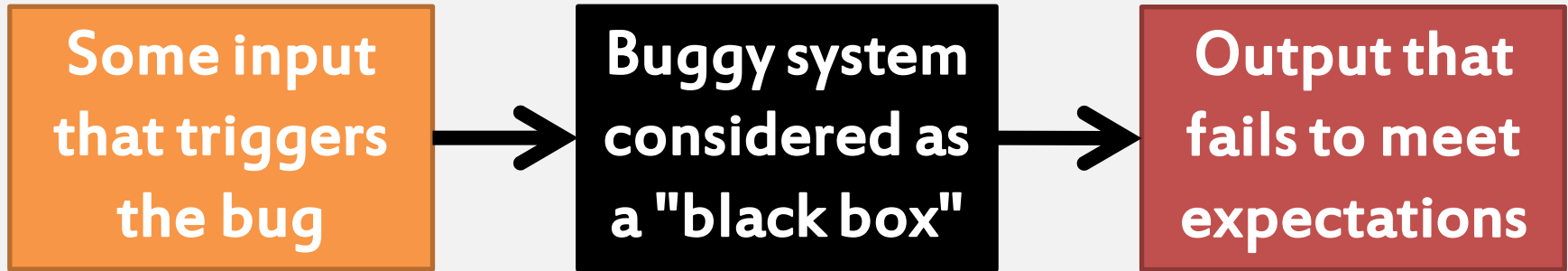
(A command is executed)

Then

(Zero or more new events, or an exception)

Bugs are tests we didn't write yet

The stream-transforming black box approach to testing extends very naturally to describing bugs



This approach optimizes for turning a bug report into a reproducible, automatable test case

Helps to produce more specific bug reports



**Languages are everywhere,
compilers skills can help**

Languages are everywhere

There's the things most people think of...

Perl 5, Perl 6, C, LOLCODE, etc.

SQL

HTML and XML

Then there's the things they don't...

INI file format

HTTP or email headers

Git commit message format

Languages are everywhere

There's the things most people think of...

Perl 5, Perl 6, C, LOLCODE, etc.

And we don't always
treat these as the
languages they are



SQL
HTML and XML

Then there's the things they don't...

INI file format

HTTP or email headers

Git commit message format

Strings attached

When we don't realize we're dealing with a language, we may be tempted to always treat things in that language as simple strings.

Compiler writers know that strings suck!

Strings attached

When we don't realize we're dealing with a language, we may be tempted to always treat things in that language as simple strings.

Compiler writers know that strings suck!

```
[branch "master"]  
  remote = origin  
  merge = refs/heads/master
```

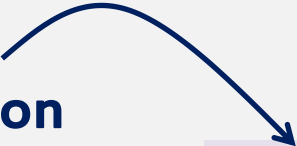
"It's just a string, right?"

Strings attached

When we don't realize we're dealing with a language, we may be tempted to always treat things in that language as simple strings.

Compiler writers know that strings suck!

Section
heading



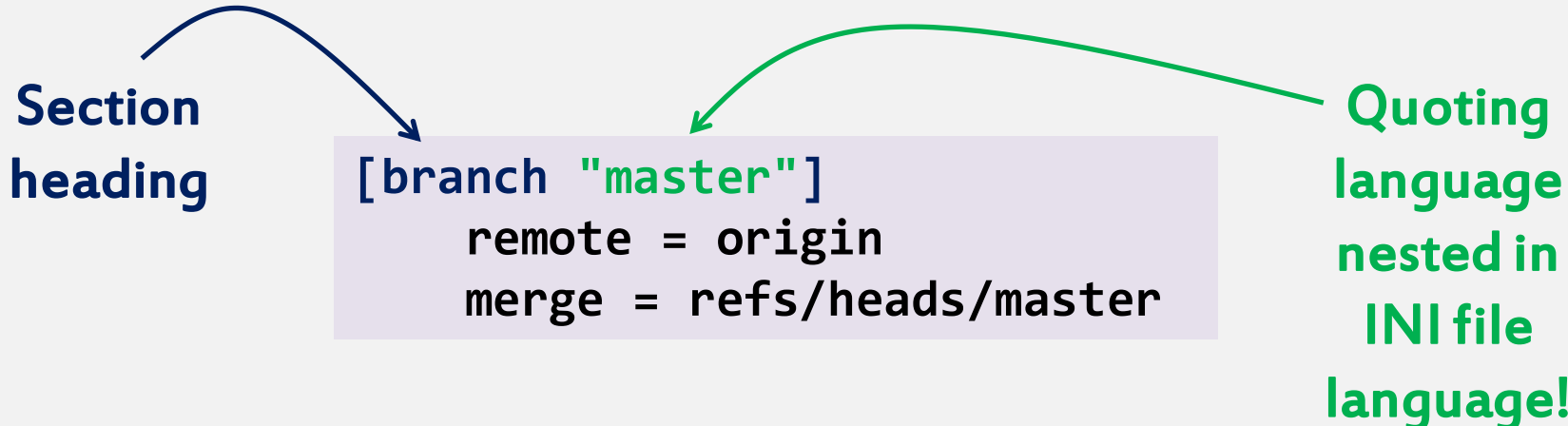
```
[branch "master"]  
  remote = origin  
  merge = refs/heads/master
```

Strings attached

When we don't realize we're dealing with a language, we may be tempted to always treat things in that language as simple strings.

Compiler writers know that strings suck!

Section
heading



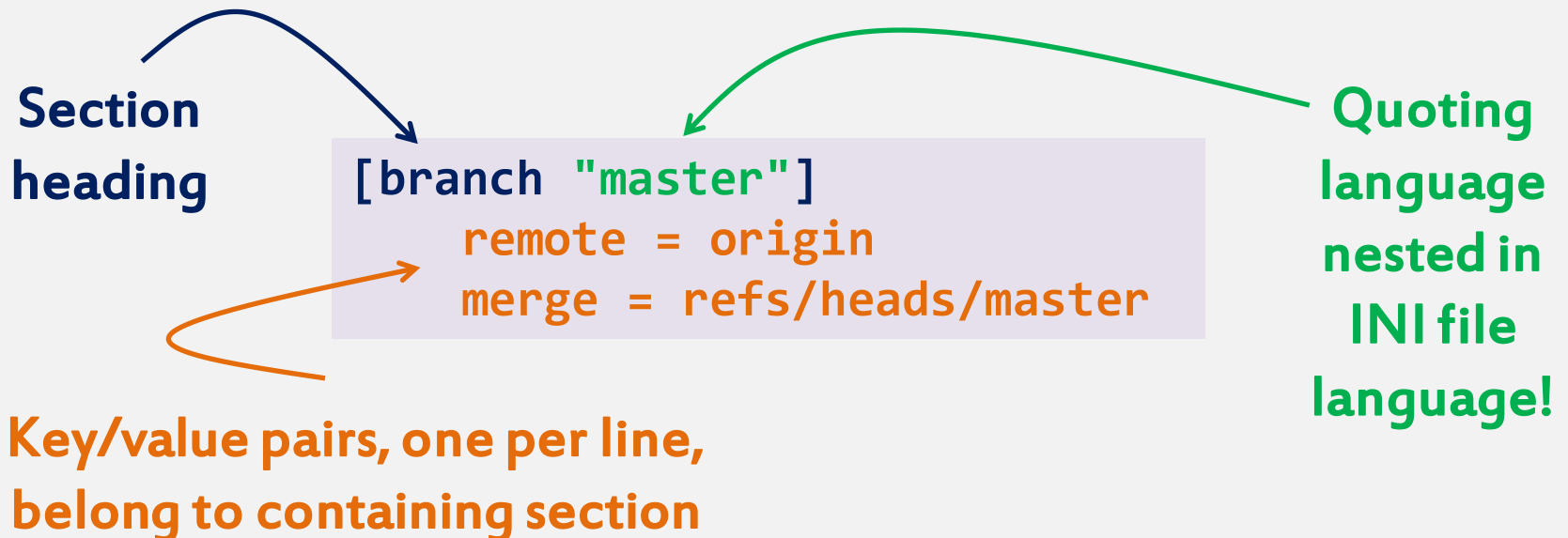
```
[branch "master"]
remote = origin
merge = refs/heads/master
```

Quoting
language
nested in
INI file
language!

Strings attached

When we don't realize we're dealing with a language, we may be tempted to always treat things in that language as simple strings.

Compiler writers know that strings suck!



SQL injection: mind your language

It's 2013 and SQL injection - the vulnerability of my web development childhood - is still everywhere!!!



SQL injection: mind your language

SQL injection happens when we treat building SQL - a rather complex language - as string manipulation.

```
$db.execute("
  SELECT Id, Name, AlcoholVolume, Description
  FROM Beers
  WHERE AlcoholVolume > $volume");
```

Works if we get a number as input. But what if `$volume` contains:

```
1; DROP TABLE Beers;
```

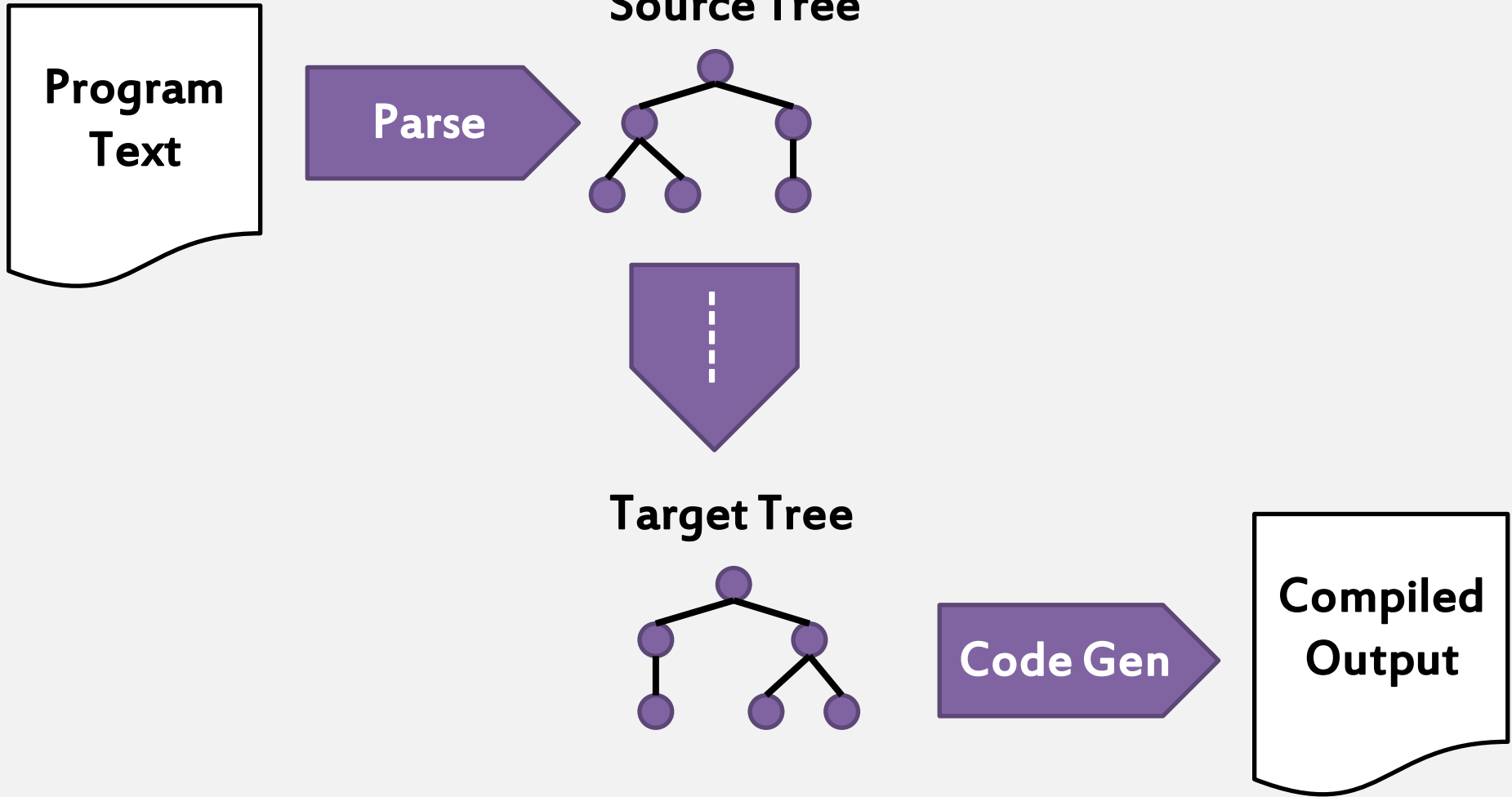
And there's more...

XSS (Cross-Site Scripting) happens when user input is incorporated into a web page without first escaping it to be treated as literal text by a browser

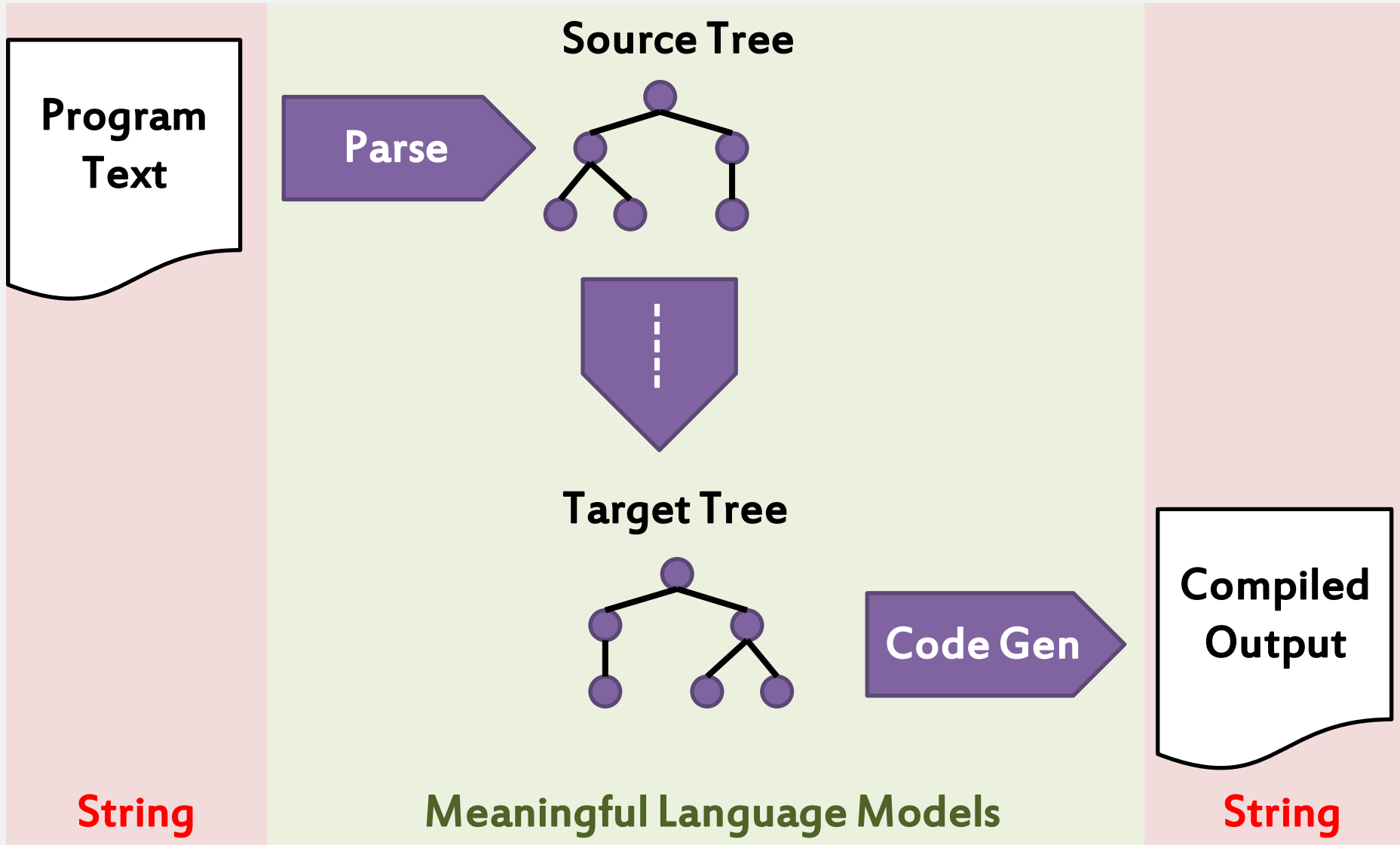
Many form mail scripts have been attacked by sending a subject containing a new line...then whatever extra headers the attacker likes

All injection vulnerabilities stem from failing to recognize languages for what they are, and instead working with them as strings.

What a compiler does

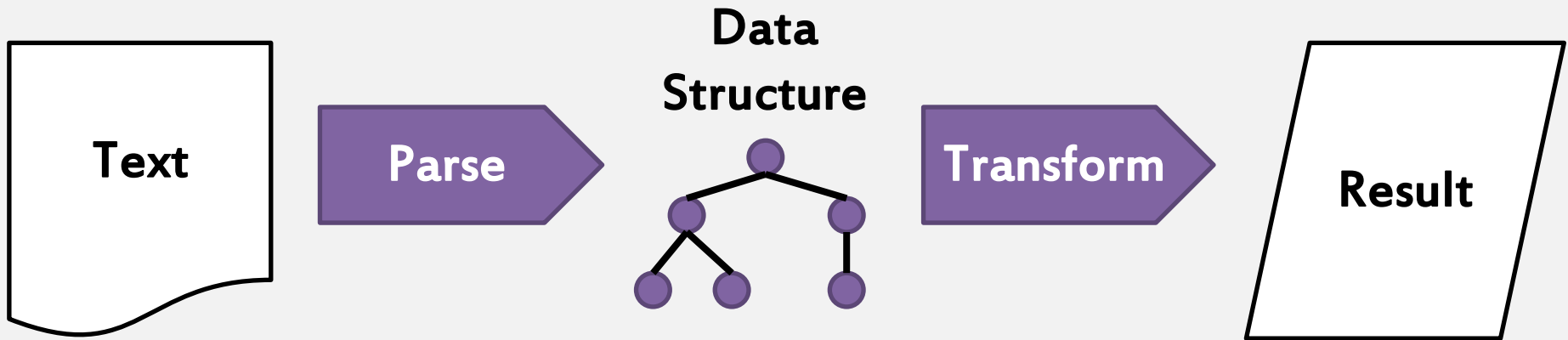


What a compiler does



Generalizing the pattern

Instead of trying to get straight from textual input to the desired result, compilers take care to separate parsing from transformation.



This is a pattern we can apply in a wide range of situations when dealing with textual input.

Story: porting hackathon

Fellow Perl 6 developer Carl Mäsak happened upon a bottle of port.

There was only one thing to do...

Port a module from some other language to Perl 6...while drinking the bottle of port!

Real "porting"... 😊



Story: porting JSON Path

We chose to port a module for handling JSONPath queries - a kind of XPath for JSON. We started off by examining how it was implemented in Perl 5...

Story: porting JSON Path

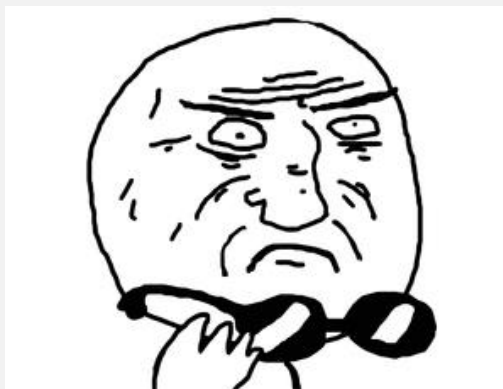
We chose to port a module for handling JSONPath queries - a kind of XPath for JSON. We started off by examining how it was implemented in Perl 5...

```
$x =~ s/[\['](\???(.*?\))[\]']/_callback_01($self,$1)/eg;  
$x =~ s/'?\.'?|\[ '?/;/g;  
$x =~ s/;;;|;;/;..;/g;  
$x =~ s/;\$|'?\]|'$/;/g;  
$x =~ s/#([0-9]+)/_callback_02($self,$1)/eg;
```

Story: porting JSON Path

We chose to port a module for handling JSONPath queries - a kind of XPath for JSON. We started off by examining how it was implemented in Perl 5...

```
$X =~ s/[\['](\??\(..*?\))[\']]/_callback_01($self,$1)/eg;  
$X =~ s/'?\.'?'|\['?/;/g;  
$X =~ s/;;;|;;/;..;/g;  
$X =~ s/;\$|'?\]|'$/g;  
$X =~ s/#([0-9]+)/_callback_02($self,$1)/eg;
```



It works. It has tests. It does kinda parse, but the parsing is tightly bound up with the expression evaluation. We could do better...

Story: porting JSON Path

Here are a few JSON Path examples:

JSON Path Query	What it does
<code>\$.store.book[*]</code>	Gets all book objects under the top store key
<code>\$.store.book[*].author</code>	Get the authors of all the books
<code>\$..book[-1:].author</code>	Get the author of the last book object found anywhere in the data

Story: porting JSON Path

For parsing, we turned to Perl 6 grammars.

```
my grammar JSONPathGrammar {
  token TOP {
    ^ <commandtree> [ $ || <giveup> ]
  }

  token commandtree {
    <command> <commandtree>?
  }

  proto token command { * }
  # Parsing of the commands come here

  method giveup() {
    die "Parse error near pos " ~ self.pos;
  }
}
```

Story: porting JSON Path

Here are the rules used for parsing some of the JSONPath commands.

```
token command:sym<$>    { <sym> }
token command:sym<.>    { <sym> <ident> }
token command:sym<[*]>  { '[' ~ ']' '*' }
token command:sym<..>  { <sym> <ident> }

token command:sym<[n]> {
  | '[' ~ ']' $<n>=[\d+]
  | "[" ~ "]" $<n>=[\d+]
}
```

Story: porting JSON Path

The grammar gets us as far as having a parse tree. In order to transform that, we can supply actions.

Actions are methods with names matching the rules in the grammar. When the grammar parses a rule successfully, it calls the action method.

The action methods can be used to build up a data structure based on the parse tree. That is, they offer a hook to do another step of transformation.

Story: porting JSON Path

At first we pondered transforming it into a tree and writing a little tree-walking interpreter. But, that felt like overkill. Then we realized: **we just want closures!**

```
method command:sym<$>($/) {
  make sub ($next, $current, @path) {
    $next($object, ['$']);
  }
}

method command:sym<.>($/) {
  my $key = ~$<ident>;
  make sub ($next, $current, @path) {
    $next($current{$key}, [@path, "['$key']"]);
  }
}
```


Story: porting JSON Path

Each command produces a closure. The first argument expects to be given a closure that executes the next command. The action for `commandtree` stitches it all together.

```
method commandtree($/) {
  make $<command>.ast.assuming(
    $<commandtree>
    ?? $<commandtree>[0].ast
    !! sub ($result, @path) {
      given $result_type {
        when ValueResult { take $result }
        when PathResult  { take @path.join('') }
      }
    });
}
```

Story: porting JSON Path

By separating parsing and transformation as we did the porting of the module, we had...

A cleaner solution

Concerns better separated, simpler code

A more efficient solution

We could cache the closure, not parse each time

A shorter solution

The original: 435 lines. Our port: 222 lines.

Takeaways

Many problems are easier to solve under a transform

In particular, solving problems involving all but the most trivial languages is greatly assisted by first transforming the text into a more suitable model

Strings are an anaemic model, and keep us from writing code that focuses on a language's concepts

Treat languages like they are languages!



tl;dr

**Decouple around well defined data
structures or events**

~

Tests that look streamy are good

~

Tests and bug reports can be unified

~

Don't treat languages as strings

Thanks for listening!

If you want to hunt me down online...

Email: jnthn@jnthn.net

Twitter: [@jnthnwrthngtn](https://twitter.com/jnthnwrthngtn)

Any questions?

