Story: Jose

- Why pipes?
- pipe_matching
- pipe_with
- Wrapping up

Why pipes?

José Valim

Why did you add Pipes to Elixir?

I stole them from F#.

Why pipes?

José Valim

Do you see them as an important part of the language?

It is puzzling. I haven't given much thought to it

Why pipes?

José Valim

Do you see them as an important part of the language?

It is puzzling. I haven't given much thought to it when I added to the language but many came to find it an essential tool to help thinking functionally!

Why pipes?

...

However, developers quickly embraced the operator, because it embodies one of the main ideas in function programming, which is the transformation of data, via multiple steps (functions).

Why pipes?

Dave Thomas

Programming Elixir
Functional |> Concurrent |> Pragmatic |> Fun

Story: Searching

Why pipes?

Chris McCord (Phoenix creator)

"Pipes and macros are why I am here"

Story: Searching

Why pipes?

Joe Armstrong

Actually, the Elixir version is easier to read:

```
:io_lib.format("~p", [x])
|> :lists.flatten
|> :erlang.list_to_binary
```

Just like the good ol' Unix pipe operator.

Why pipes?

Functional Programming

```
back(back(forward(step)))
```


Cliche one step forward two steps back

Why pipes?

Why pipes?

Functional Programming

```
forward(step)
```

Why pipes?

Functional Programming

```
step |> forward
```


Why pipes?

Functional Programming

```
back(back(forward(step)))
```

Why pipes?

Functional Programming

```
step |> forward |> back |> back
```


Why pipes?

Why pipes?

```
def request(conn) do
  conn |>
  enforce_ssl |>
  map_params |>
  route |>
  respond
end
```


- Why pipes?
- pipe_matching
- pipe_with
- Wrapping up

pipe_matching

Problem: Unreliable tasks

```
works |> works |> breaks |> works
```

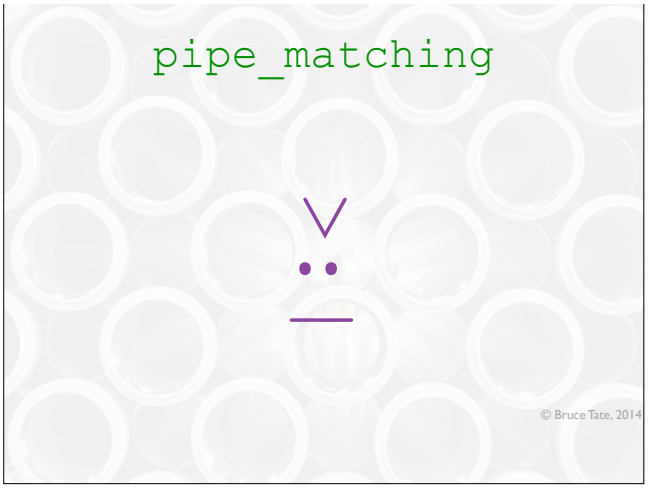

pipe_matching

```
defmodule RussianRoulette do
  def click(acc) do
    IO.puts "click..."
    {:ok, "click"}
  end

  def bang(acc) do
    IO.puts "BANG."
    {:error, "bang"}
  end
end

{:ok, ""} |> click |> click |> bang |> click
```

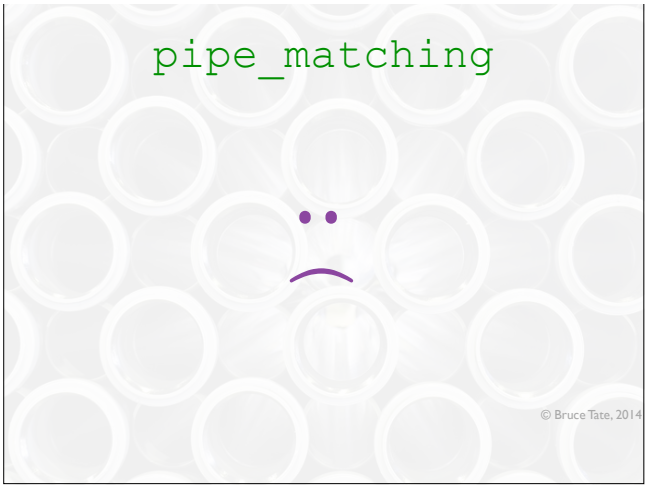
pipe_matching

```
mix run examples/return_codes.exs
click...
click...
BANG.
click...
```

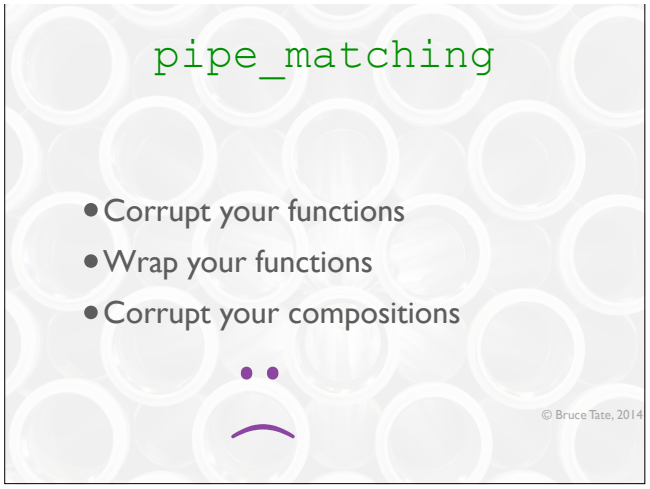

pipe_matching



May make you mad... unless you're willing to do something about it...

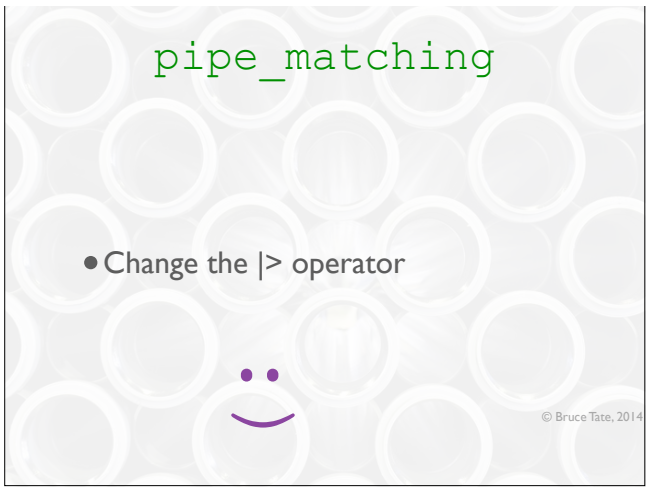


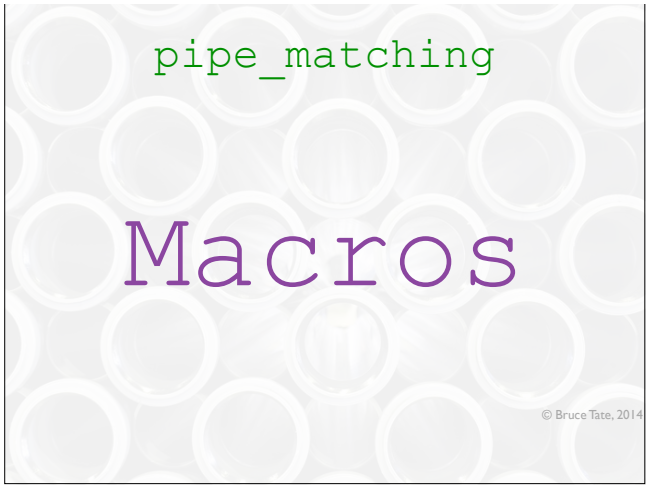
Our program is wrong and that makes us sad.
BECAUSE WE KNOW



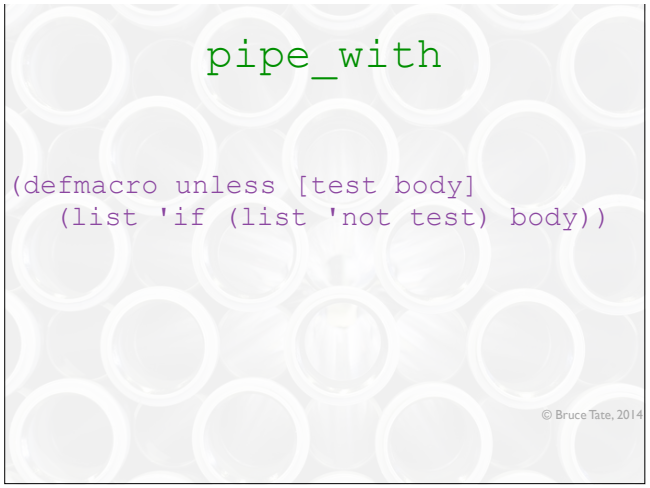
It's not dry. It's boiler plate.

When you reach for that cut and paste: UNDERSTAND WHY.



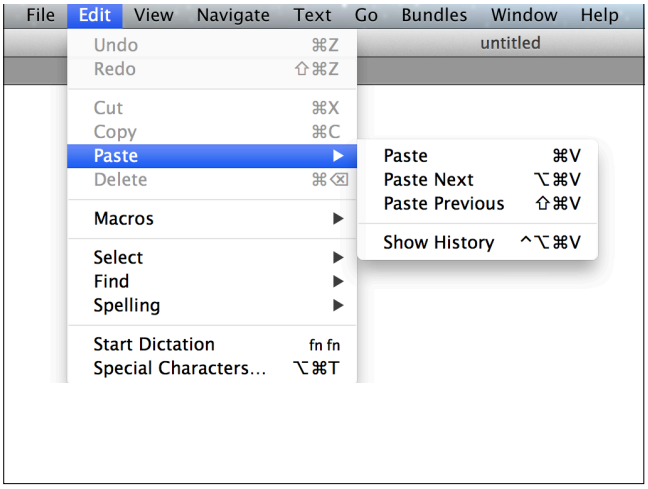


Macros in several languages

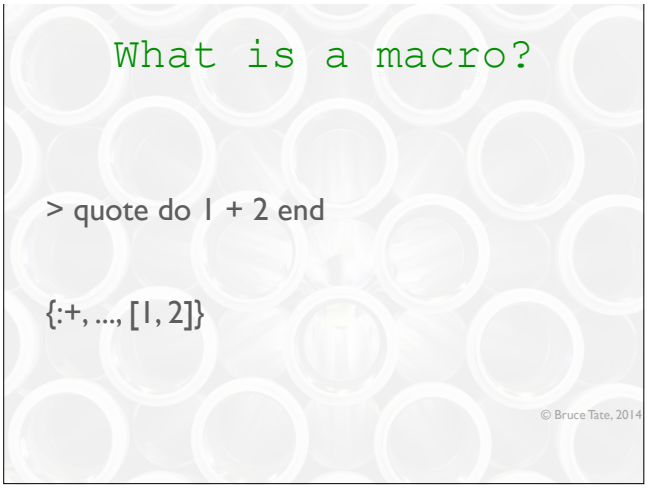


Macros in Clojure

I have a demo... Erlang macros



This is true of many languages. Means
My language isn't strong enough to do what I'm trying to do
In Ruby, open classes. In Java, byte code enhancement, Spring, ejb.



Every line of Elixir code is represented as a three-tuple. We don't deal with it in this form, though.
And you can change it..
we usually don't need to look at the raw tuples.

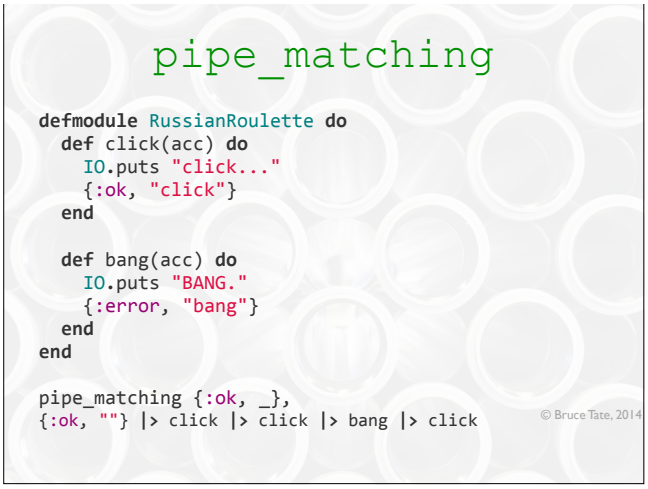
What is a macro?

```
defmacro unless(clause, expr) do
  quote do
    if(!unquote(clause), do: expr)
  end
end
```

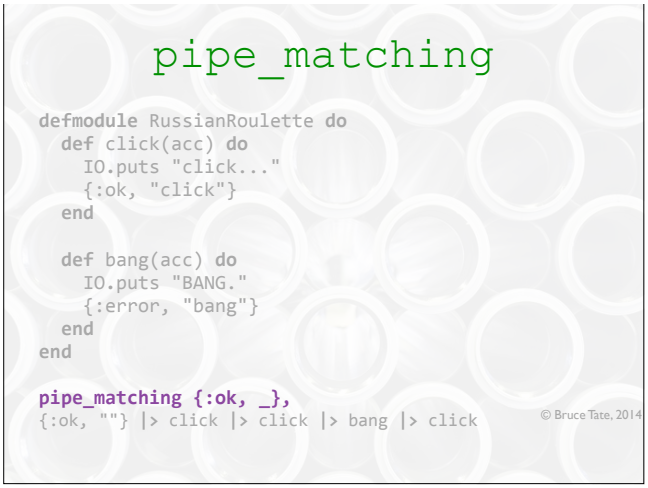

Every line of Elixir code is represented as a three-tuple. We don't deal with it in this form, though.
And you can change it..
we usually don't need to look at the raw tuples.

pipe_matching

```
defmodule RussianRoulette do
  def click(acc) do
    IO.puts "click..."
    {:ok, "click"}
  end

  def bang(acc) do
    IO.puts "BANG."
    {:error, "bang"}
  end
end

pipe_matching {:ok, _},
{:ok, ""} |> click |> click |> bang |> click
```



This is what we want: a single point of code that describes what the pipes are supposed to do



As long as the return code matches, we'll pipe.
Then, stop

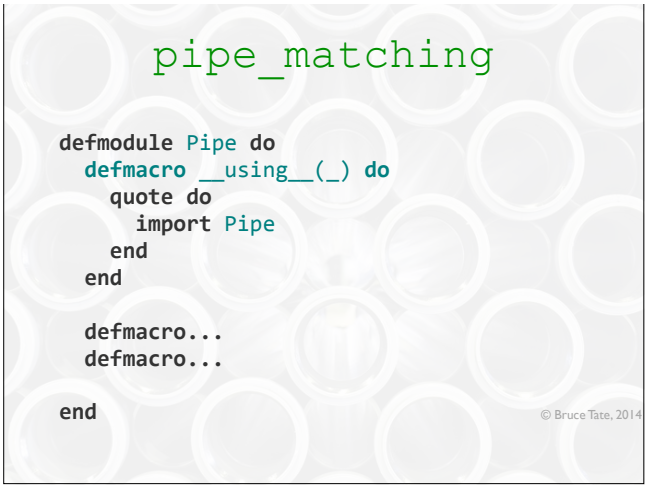
pipe_matching

- Preserve syntax
- Prevent execution
- Must be a macro

This code HAS TO BE A MACRO.
We don't want code prematurely executed

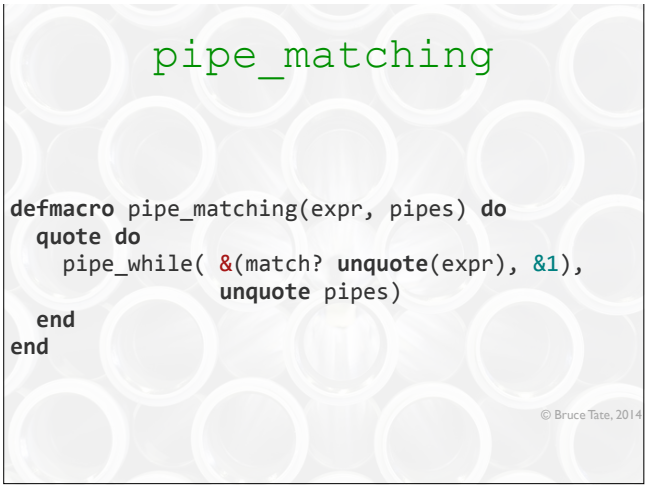
What is a macro?

- Executes at compile time
- Potentially changing the syntax tree
- Especially useful as code templates

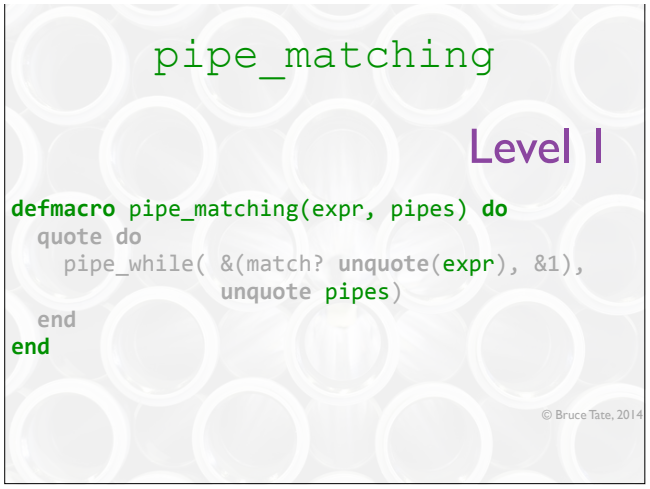


When the compiler encounters the use directive,

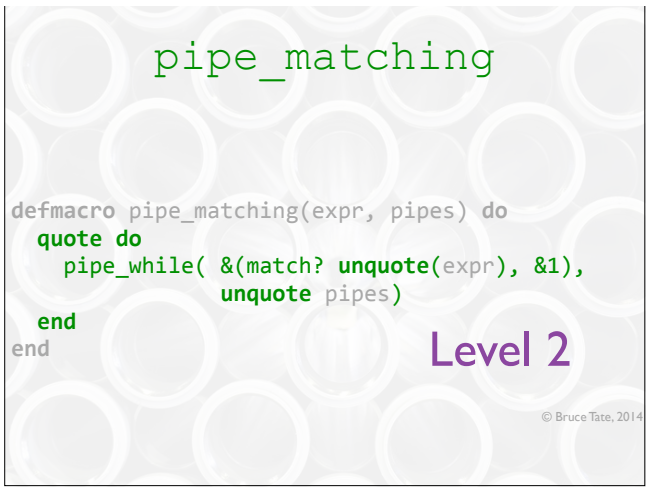
this code will execute, dropping the result into the codebase



Strategy: take pipes apart, and conditionally execute them with pipe_while
Think scope and visibility.
CRUISE SHIP Quoting goes up... ctime is 1; etime is 2



Here's what I mean:



pipe_matching

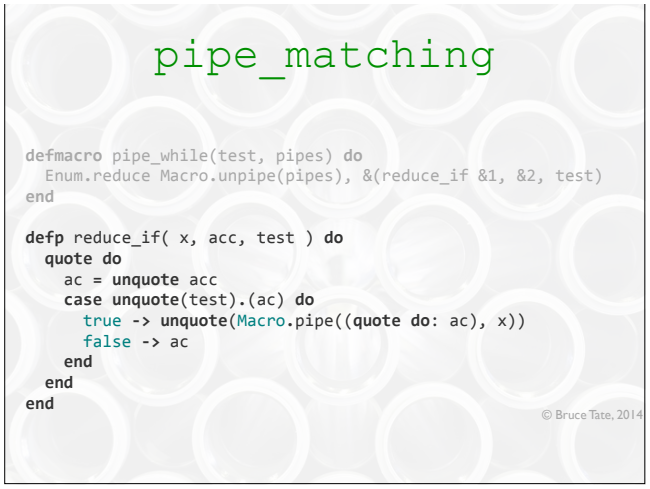
```
defmacro pipe_while(test, pipes) do
  Enum.reduce Macro.unpipe(pipes), &(reduce_if &1, &2, test)
end

defp reduce_if( x, acc, test ) do
  quote do
    ac = unquote acc
    case unquote(test).(ac) do
      true -> unquote(Macro.pipe((quote do: ac), x))
      false -> ac
    end
  end
end
```


pipe_matching

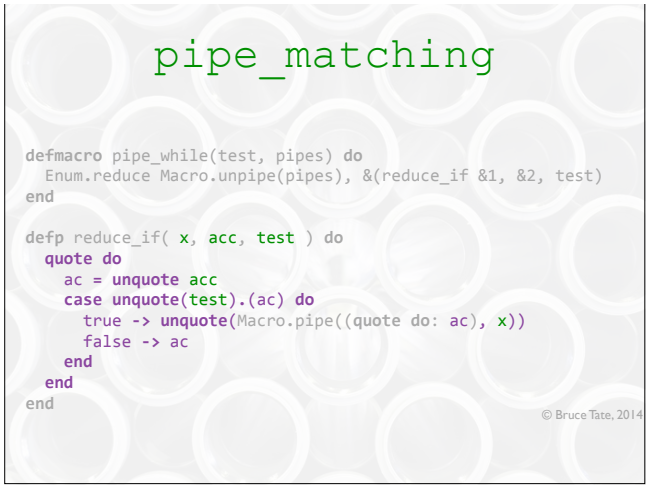
```
defmacro pipe_while(test, pipes) do
  Enum.reduce Macro.unpipe(pipes), &(reduce_if &1, &2, test)
end

defp reduce_if( x, acc, test ) do
  quote do
    ac = unquote acc
    case unquote(test).(ac) do
      true -> unquote(Macro.pipe((quote do: ac), x))
      false -> ac
    end
  end
end
```

© Bruce Tate, 2014

Take the pipes apart, and put them together. Reduce function puts each of the segments together while test is true
Reduce function: reduce_if, takes the unexecuted pipes, the accumulator (result so far), and a test function
We'll call test on the accumulator



Reduce: takes the pipe segments, the execution so far, and a test function
Combine pipes as long as test function is true



Reduce: takes the pipe segments, the execution so far, and a test function
Combine pipes as long as test function is true
Note: We only want to unquote code once

pipe_matching

```
defmacro pipe_while(test, pipes) do
  Enum.reduce Macro.unpipe(pipes), &(reduce_if &1, &2, test)
end

defp reduce_if( x, acc, test ) do
  quote do
    ac = unquote acc
    case unquote(test).(ac) do
      true -> unquote(Macro.pipe((quote do: ac), x))
      false -> ac
    end
  end
end
```


At run time, it will look something like this:

pipe_matching

```
mix run examples/return_codes.exs
click...
click...
BANG.
```

- Why pipes?
- pipe_matching
- pipe_with
- Wrapping up

pipe_with

Problem: Non-uniform exceptions

```
works |> works |> breaks |> works
```

pipe_with

Problem: Non-uniform exceptions

```
works |> works |> breaks |> works
```

Throws or {:error, ...}

Common in Erlang

pipe_with

```
defmodule Roulette do
  def start, do: :ok
  def click(acc) do
    IO.puts "oh yayz iz a liv #{inspect acc}"
  end

  def bang(_acc) do
    IO.puts "oh noz iz ded"
    raise "shotz"
  end
end

Roulette.start |>
Roulette.click |>
Roulette.click |>
Roulette.bang |>
Roulette.click
```

pipe_with

```
defmodule Roulette do
  def start, do: :ok
  def click(acc) do
    IO.puts "oh yayz iz a liv #{inspect acc}"
  end

  def bang(_acc) do
    IO.puts "oh noz iz ded"
    raise "shotz"
  end
end

Roulette.start |>
Roulette.click |>
Roulette.click |>
Roulette.bang |>
Roulette.click
```


pipe_with

```
defmodule ExceptionWrapper do
  def wrap({:error, e, acc}, _), do: {:error, e, acc}
  def wrap(acc, f) do
    f.(acc)
  rescue
    x in [RuntimeError] ->
      {:error, x, acc}
  end
end
```


1) Success
2) {:error...} form,
3) Exception form..... this is our g (or wrapper)

pipe_with

```
use Pipe
game = pipe_with &ExceptionWrapper.wrap/2,
         Roulette.start |>
         Roulette.click |>
         Roulette.click |>
         Roulette.bang |>
         Roulette.click
```


ExceptionWrapper.wrap/2 is our outer; click/bang are inner functions

pipe_with

Problem: Changing Semantics

```
matrix |> operator |> operator
```

pipe_with

```
list = [1, 2, 3]

list |>
Kernel.+(1) |>
Kernel.*(2)

matrix = [[1, 2], [2, 3], [0, 1]]
matrix |>
Kernel.+(1) |>
Kernel.*(2)
```


pipe_with

```
x   |>   f1   |>   f2
```


our strategy

pipe_with

```
x    |> g(f1)   |> g(f2)
```


pipe_with

```
defmacro pipe_with(fun, pipes) do
  Enum.reduce Macro.unpipe(pipes), &(reduce_with &1, &2, fun)
end

defp reduce_with( segment, acc, outer ) do
  quote do
    inner = fn(x) ->
      unquote Macro.pipe((quote do: x), segment)
    end

    unquote(outer).(unquote(acc), inner)
  end
end
```


Our goal: Unpipe to segments, and reduce with the outer function calling each pipe

pipe_with

```
defmacro pipe_with(fun, pipes) do
  Enum.reduce Macro.unpipe(pipes), &(reduce_with &1, &2, fun)
end

defp reduce_with( segment, acc, outer ) do
  quote do
    inner = fn(x) ->
      unquote Macro.pipe((quote do: x), segment)
    end

    unquote(outer).(unquote(acc), inner)
  end
end
```


pipe_with

```
defmacro pipe_with(fun, pipes) do
  Enum.reduce Macro.unpipe(pipes), &(reduce_with &1, &2, fun)
end

defp reduce_with( segment, acc, outer ) do
  quote do
    inner = fn(x) ->
      unquote Macro.pipe((quote do: x), segment)
    end

    unquote(outer).(unquote(acc), inner)
  end
end
```

© Bruce Tate, 2014

pipe_with

```
defmodule Matrix do
  def merge_list(x, f), do: Enum.map(x, f)
  def merge_lists(x, f), do: Enum.map(x, &Matrix.merge_list(&1, f))
end

use Pipe
list = [1, 2, 3]
pipe_with &Matrix.merge_list/2,
          list |> Kernel.+(1) |> Kernel.*(2)

matrix = [[1, 2], [2, 3], [0, 1]]
pipe_with &Matrix.merge_lists/2,
          matrix |> Kernel.+(1) |> Kernel.*(2)
```


- Why pipes?
- pipe_matching
- pipe_with
- Wrapping up

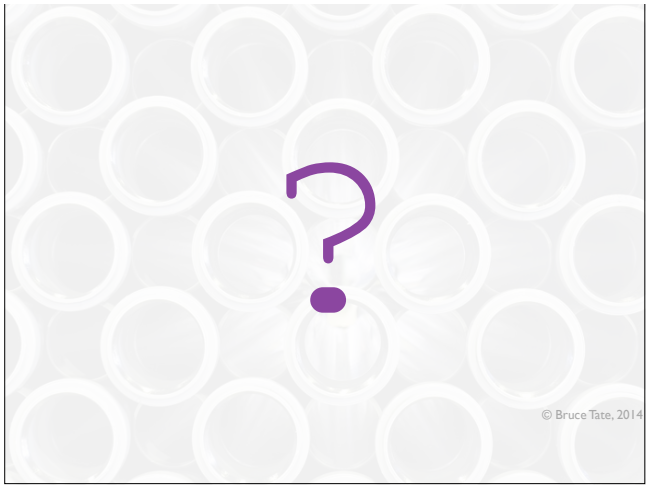
Programming is Thinking

Pipes Help Us Think

Macros make the Pipes Better

Resources

- Programming Elixir
  - http://pragprog.com/book/elixir/programming-elixir
- elixir-pipes
  - https://github.com/batate/elixir-pipes
- Joe Armstrong on pipes
  - http://joearms.github.io/2013/05/31/a-week-with-elixir.html


?
© Bruce Tate, 2014