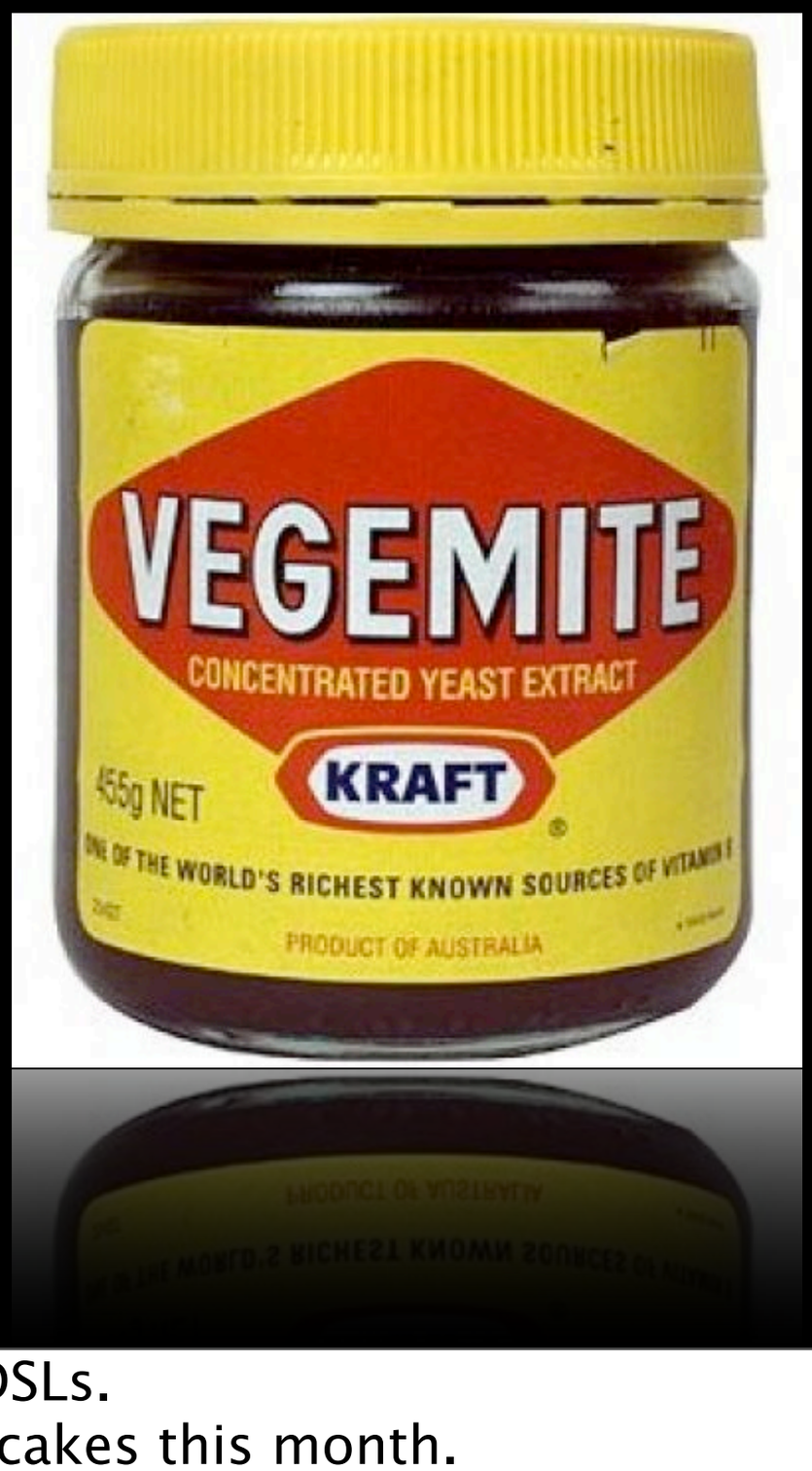
Do you speak-a my Language?



Rob asked my to speak about DSLs.
It's been popping up like mad-cakes this month.

I HATE CTT

Who's been on a surge project?
With the ada based projects, we could work around some of the short comings of CTT.
* With the advent of our testing C code, we lost the ability to stub.
* We had to implement sub-diagrams
* We found we needed to reuse a lot of code

I HATE CTT

HATE

Who's been on a surge project?
With the ada based projects, we could work around some of the short comings of CTT.
* With the advent of our testing C code, we lost the ability to stub.
* We had to implement sub-diagrams
* We found we needed to reuse a lot of code

I HATE CTT

HATE

HATE

Who's been on a surge project?
With the ada based projects, we could work around some of the short comings of CTT.
* With the advent of our testing C code, we lost the ability to stub.
* We had to implement sub-diagrams
* We found we needed to reuse a lot of code

I HATE CTT

HATE

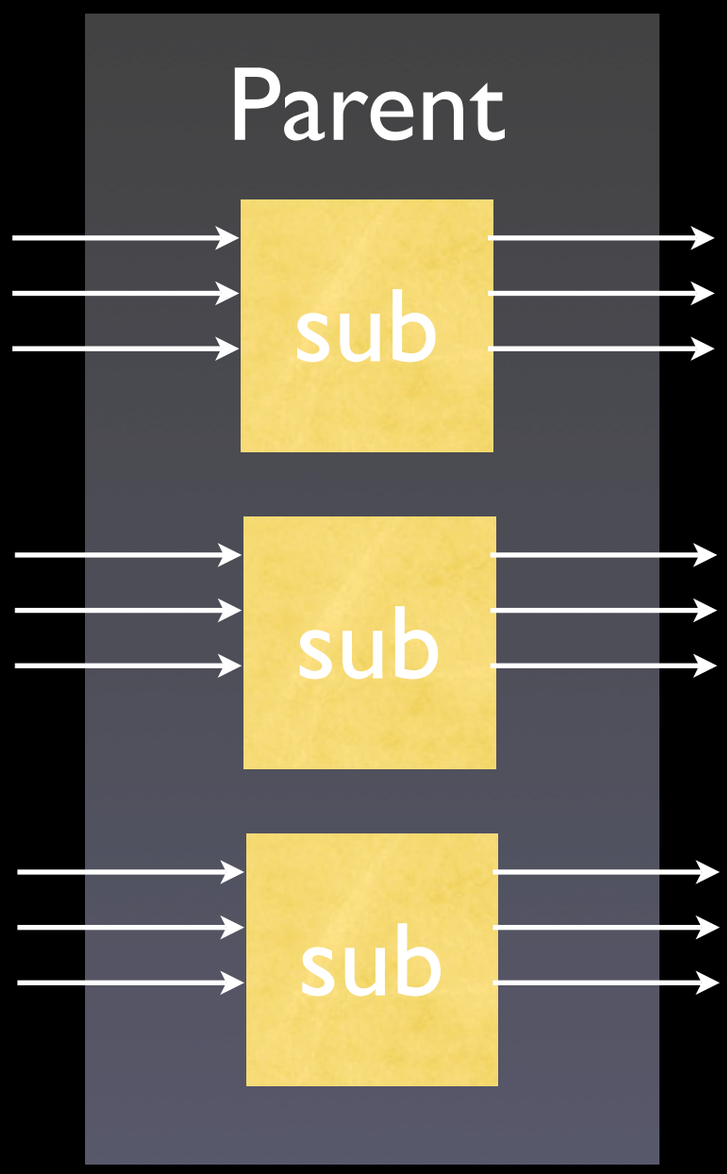
HATE

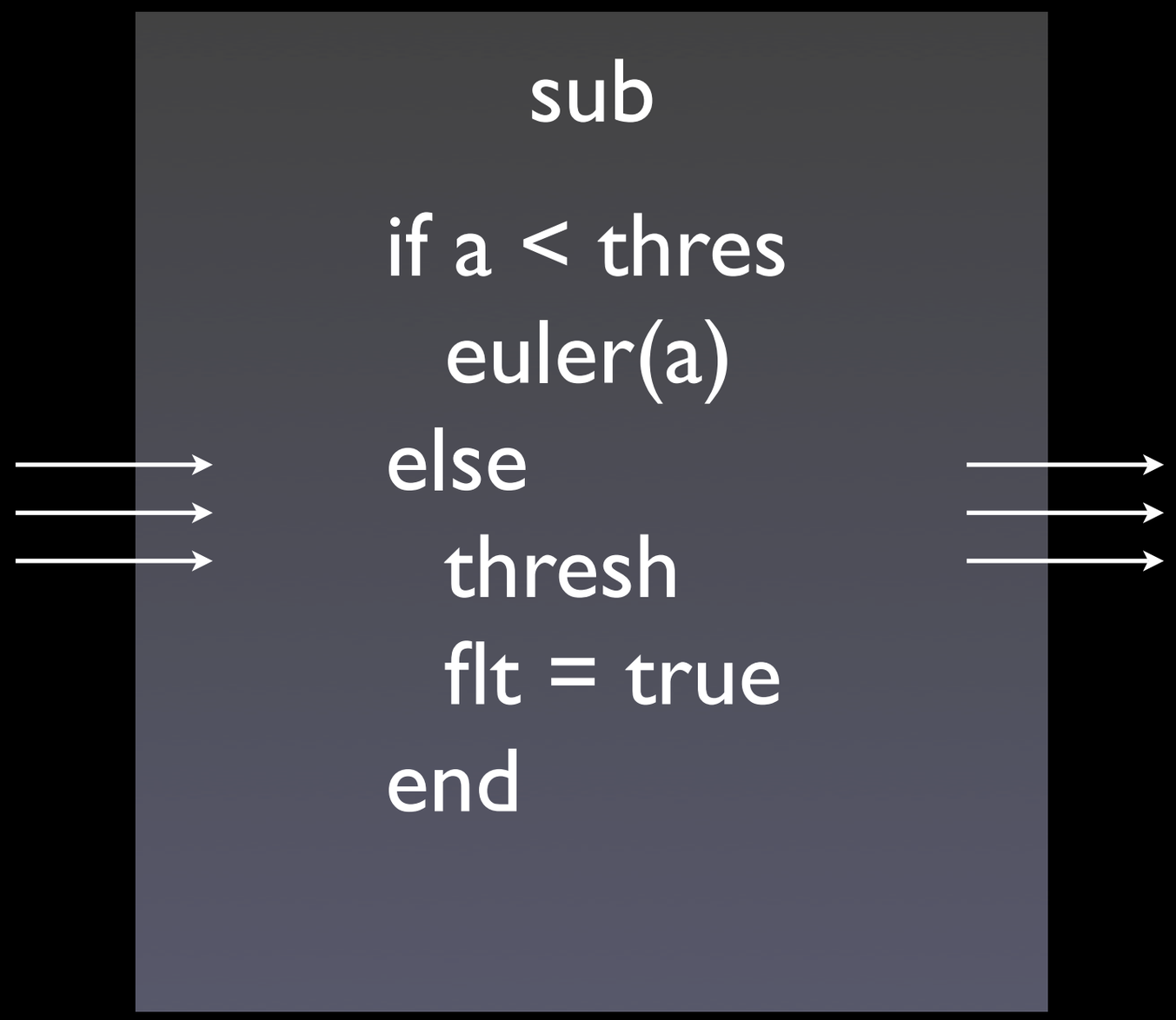
HATE

Who's been on a surge project?
With the ada based projects, we could work around some of the short comings of CTT.
* With the advent of our testing C code, we lost the ability to stub.
* We had to implement sub-diagrams
* We found we needed to reuse a lot of code

And we started to run into massive amounts of copy paste code and I could spend a whole day tracking down copy-paste errors.

http://www.flickr.com/photos/svenstorm/2442461886/

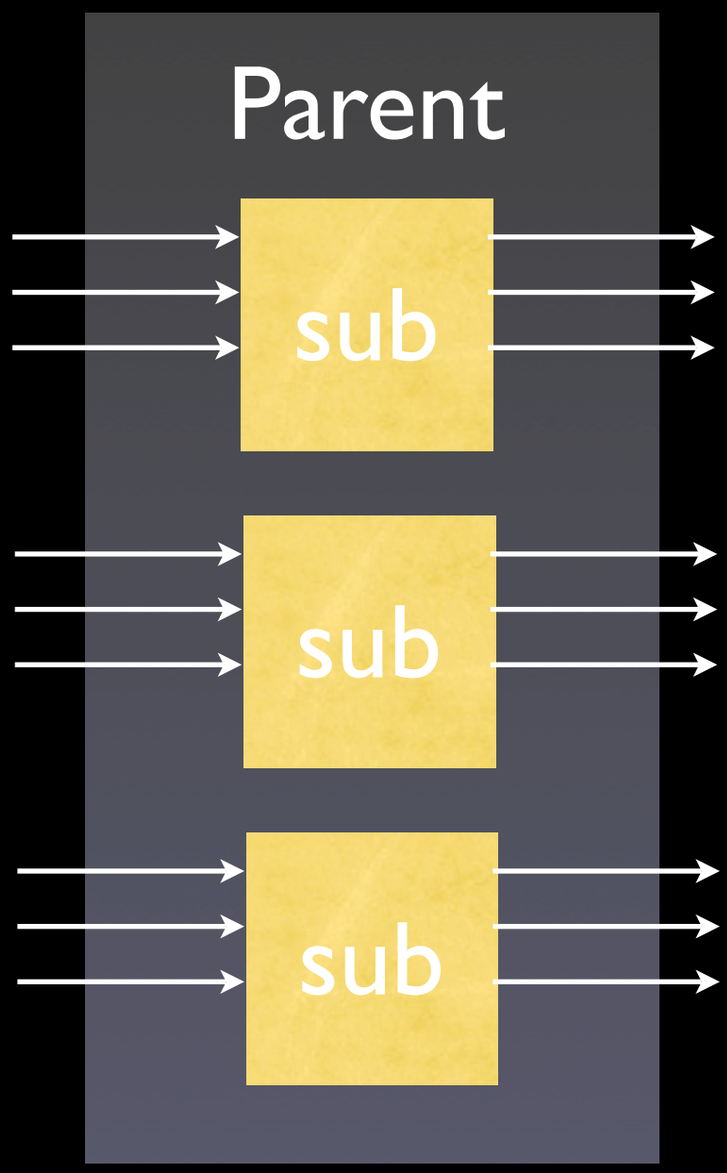


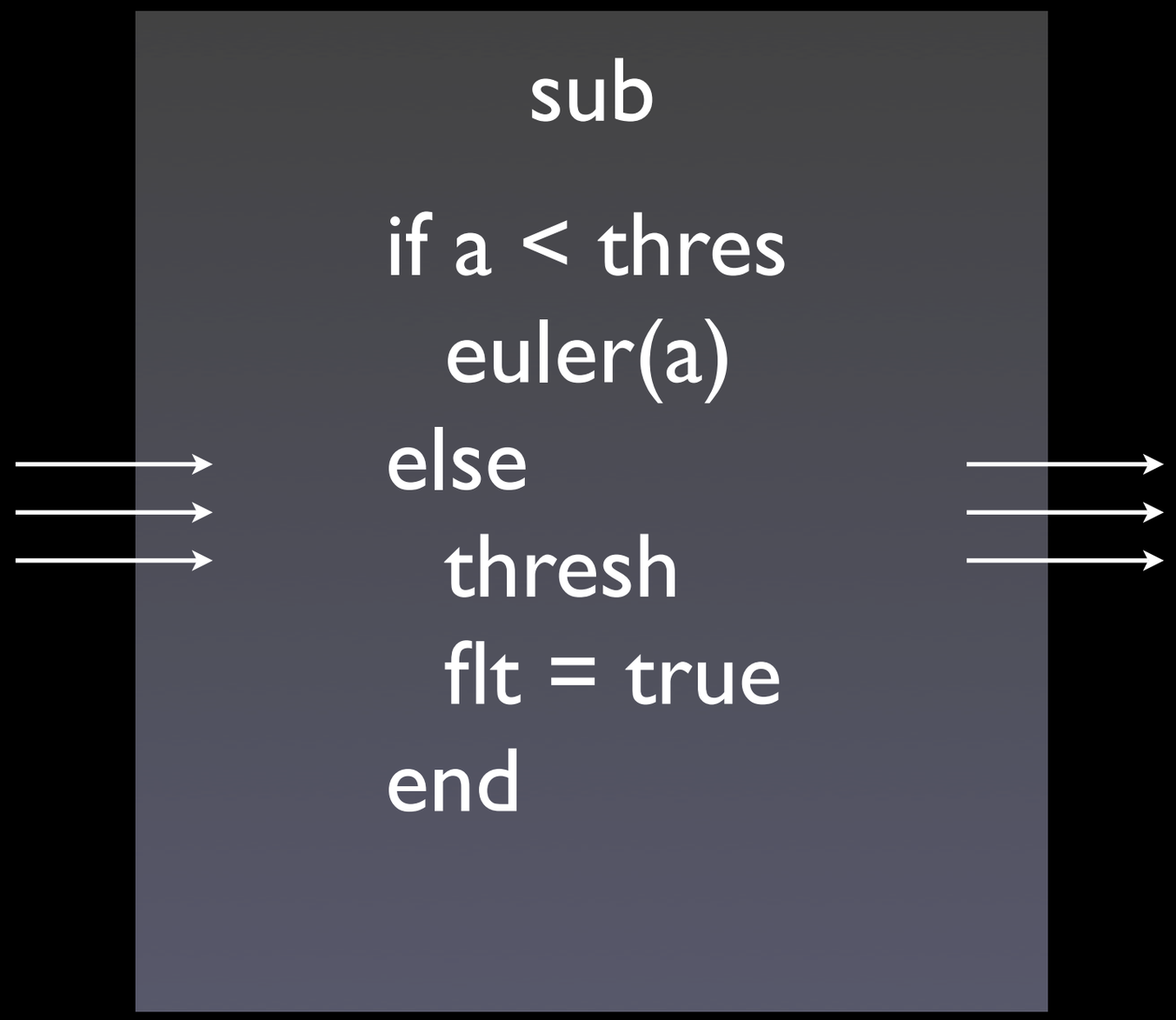


Here's a quick overview of the CTT problem that I had.
Except sub had 12 inputs, it's own very complex logic with reusable beacon objects
You can kinda see that in the euler function call

```
SYMBOL flt_in COMPUTED BOOLEAN := flt'IN;
SYMBOL flt_out COMPUTED BOOLEAN :=
  if a < thresh then
    flt_in
  else
    true
end;
SYMBOL pass COMPUTED BOOLEAN :=
  flt'OUT == flt_out;
```

This was all expressed like this....
So, I have to model a component, vary it's input (not shown) and compare the actual values with the expected values.





You can see here, there's a lot of duplicate code
This module is for one of 8 input sensors to determine which signals are to be trusted
and sub is the same for all of those, as well.
each script was over 400 lines of code that replicated the same functionality
waste / waste / waste. and annoying to write.

Now, we'd generated some scripts with ruby, but we didn't have a general solution to this problem that would let us nail these issues in a broad term.

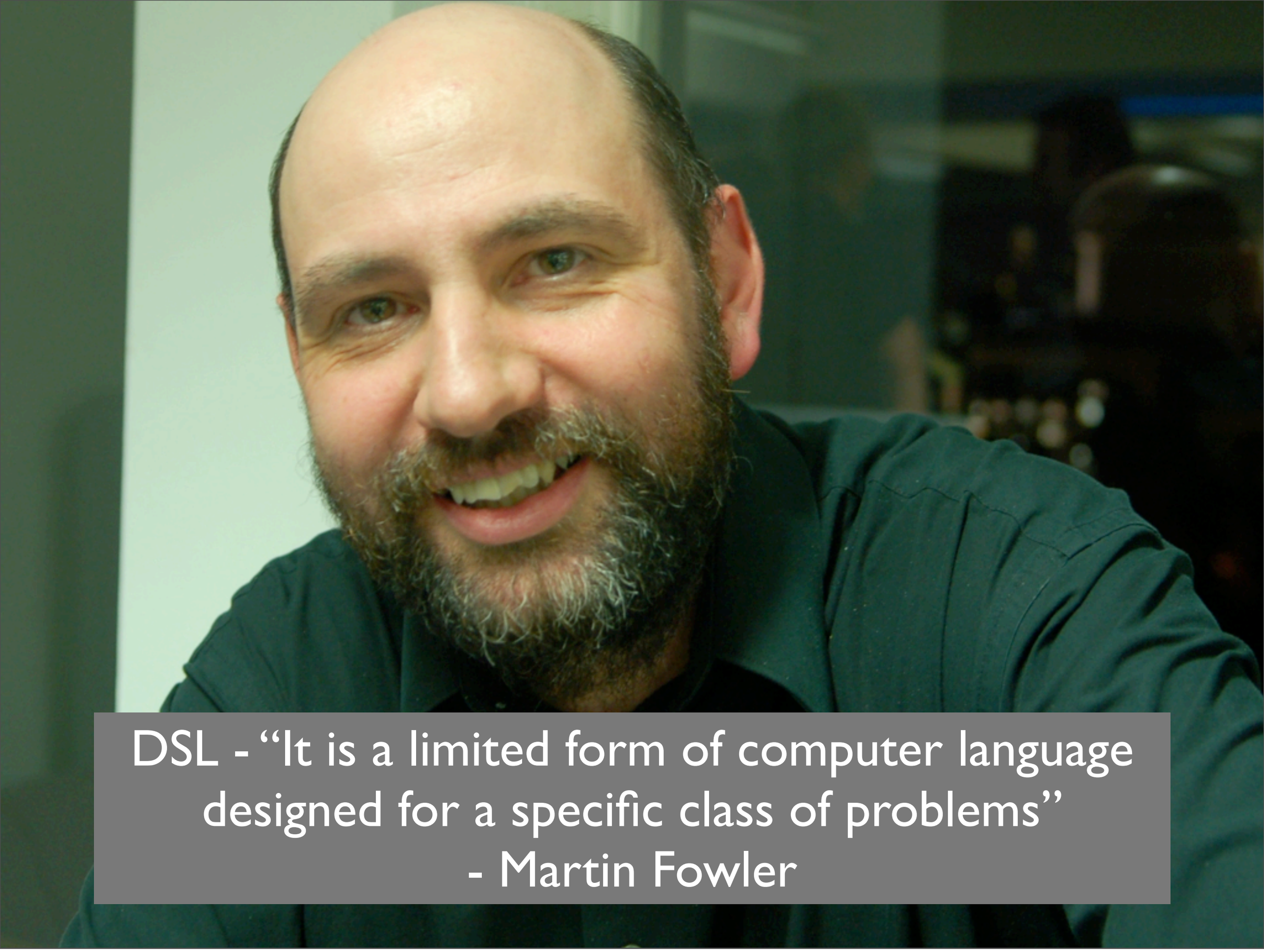
So, in one night, I banged out a DSL for handling 80 percent of the cases.
It was either that, or watch baskettball. Either way, it was full of win.
Or not, we were probably watching IU.

http://www.flickr.com/photos/bradjward/2327767871/

DSL - noun, Geeks geeking out and making it harder than it needs to be.

- your manager

DSLs are a new hammer, apply with caution.
Also, they have a reputation as being for “large” problems and for “eggheads” and a “rats nest of maintainability”. This ain’t so.



limited - NOT TURING COMPLETE
SPECIFIC PROBLEM - YAGNI is your guide. leave it out unless you do actually need it. if someone says “might”, hit them.
COMPUTER LANGUAGE - This ain’t english. Don’t every confuse the two

http://www.flickr.com/photos/adewale_oshineye/2933030620/

DSLs are a study in contrasts, they exist across several axes

http://www.flickr.com/photos/exfordy/387876592/

External vs Internal

An external DSL is parsed. An internal DSL is part of a host language.

External - Make
Internal - rSpec

http://www.flickr.com/photos/johnlinwood/372648413/

External

Updatable at runtime.
Can be a BNL

```
clean:
    rm -r *.o
    rm -r my_bin
```

This is make.
It tells you what commands need to be run at a given state.
It also can determine what state your project is in based on the depenencies

```
clean:
    rm -r *.o
    rm -r my_bin
```

```
app: *.c
    gcc -o my_bin app.c lib.c
```

This is make.
It tells you what commands need to be run at a given state.
It also can determine what state your project is in based on the depenencies

```
employee John Doe
compensate 500 dollars for each deal closed in
  the past 30 days
compensate 100 dollars for each active deal that closed
  more than 365 days ago
compensate 5 percent of gross profits if gross profits
  are greater than 1,000,000 dollars
compensate 3 percent of gross profits if gross profits
  are greater than 2,000,000 dollars
compensate 1 percent of gross profits if gross profits
  are greater than 3,000,000 dollars
```

Internal

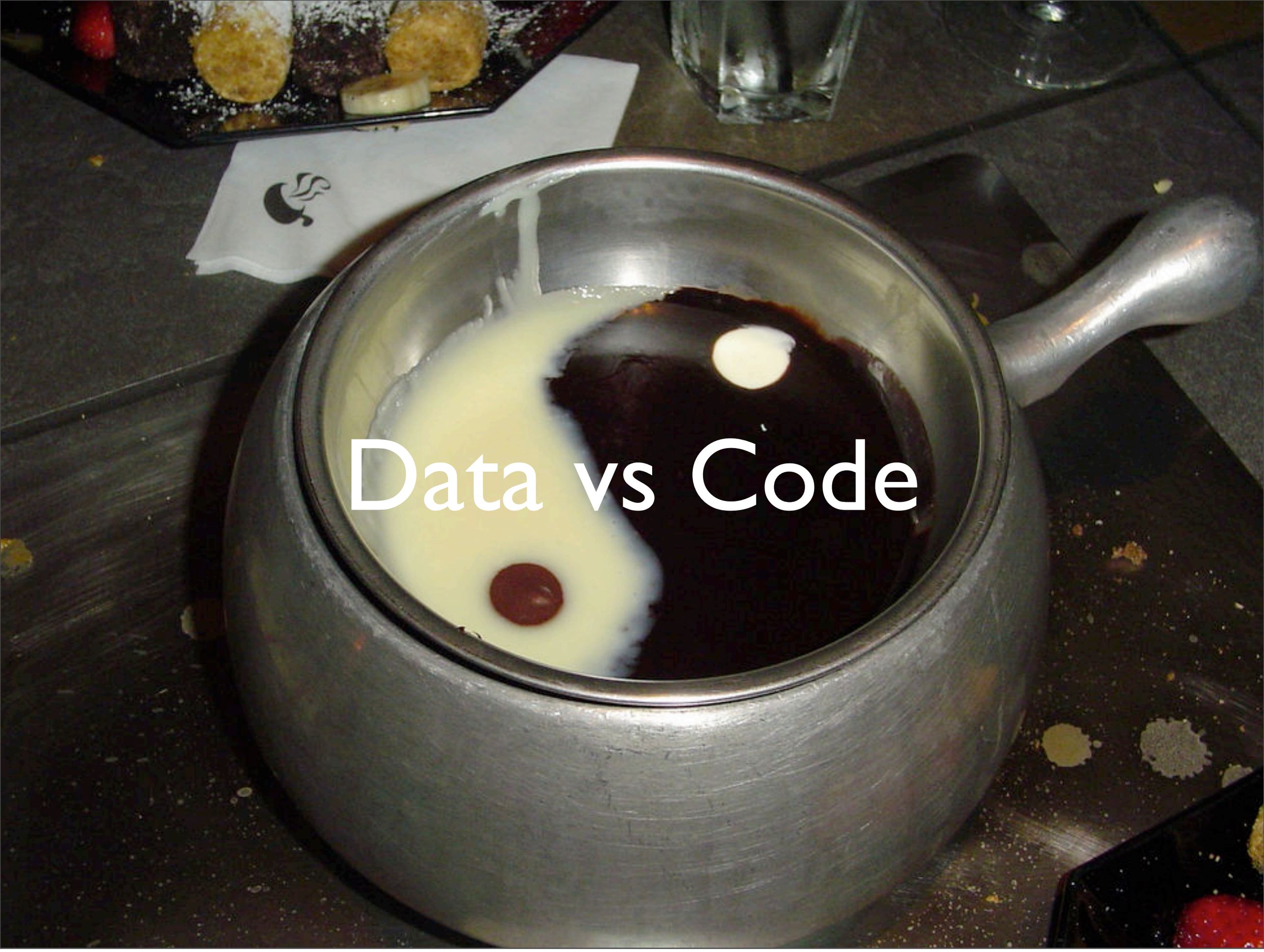
Compile time. or access to the programming language is important
Now, for dynamic languages, this means a lot less. CTT is actually an interal DSL, but we use it as through it were external.
Rake and rSpec are the same way.

```
Assert.That( value, Is.Not.Null );
```

This is part of NUnit. It's their syntactic helpers for the framework.
I want to point a few things out here. What the "That" method does is take the valuator generated and apply it to value. Here is also a common internal DSL pattern called method chaining. Not returns an object that will decorate a valuator with an inversion.

```
[<Scenario>]
let When_calculating_fac_5_it_should_equal_120() =
  Given 5
  |> When calculating factorial
  |> It should equal 120
  |> Verify
```

here’s an example in F#
It’s a behavioral driven testing language



Data vs Code

They generate executable code, or define input data
CODE - CTTwrapper
DATA - yaml

http://www.flickr.com/photos/lawley/6486116/

Code

Code is what you care about.
We needed to define the CTT output

```
grammar Expr;
prog:   stat+
stat:     expr NEWLINE
      |     ID '=' expr NEWLINE
      |     NEWLINE;
expr:   multExpr (('+' | '-') multExpr*;
multExpr:   atom ('*' atom)*;
atom:   INT
        |  ID
        |  '(' expr ')';

ID: ('a'..'z'|'A'..'Z')+;
INT: '0'..'9'+;
NEWLINE: '\r'? '\n';
WS: (' '|'\t'|'\n'|'\r')+ { skip(); };
```

This defines an abstract syntax tree.
It generates code based on this syntax that will go and parse the input.

I'll cover it some more later.

Data

Sometimes, all we need is a sensible way to define our problem and have the engine handle it.
Or just define data in general.

```
application: comic_site
version: 1
runtime: python
api_version: 1

handlers:
  - url: /
    script: welcome.py
  - url: /admin/.*
    script: admin.py
    login: admin
```

This is a fragment of yaml that defines my app engine site for comics.
I don’t want code, I just want to define some meta data for the underlying engine to be able to direct code where I want it.

Sometimes, they remove common boilerplate and let you focus on what's important
Sometimes, they add words to make explicit what's important

terse - rails routes
verbose - rspec

http://www.flickr.com/photos/naama/27544572/

Terse

```
map.connect 'articles/:year/:month/:day',
  :controller => 'articles',
  :action     => 'find_by_date',
  :year       => /\d{4}/,
  :month      => /\d{1,2}/,
  :day        => /\d{1,2}/
```

Rails routes abstract away the if/else block for figuring out what url goes to what handler.

Verbose

```
describe Bowling, "When scoring gutters" do
  it "should score 0 for gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }

    bowling.score.should == 0
  end
end
```

rspec is more wordy than a simple unit::test method might be, but it clarifies some aspects of the unit test with all that describe and it razzmatazz

"All problems in computer science can be solved by another level of indirection"

- David Wheeler

DSLs let us focus on what's important. We don't want to see what's under the hood, we just want to spec out a trip from here to Kalamazoo. In fact, why we want to go to Kalamazoo is someone else's problem.

"...except for the problem of too many layers of indirection."
- Kelvin Henney

But they aren't a silver bullet.

so, you're pumped. You're excited. You wanna do this for your code today!

How?

http://www.flickr.com/photos/maisonbisson/156901708/

Tools

http://www.flickr.com/photos/22280677@N07/2504310138/

Your Language

Extension methods
method chaining
lambda
LINQ abuse
Dynamic inderfaces

C#

```
Id( x => x.Id );
Map( x => x.Name);
Map( x => x.Price);
HasManyToMany( x => x.StoresStockedIn)
  .Cascade.All()
  .Inverse()
  .WithTableName("StoreProduct");
```

This is an example of FluentNHibernate.
Here, we make statements about the situation, and the logn

Ruby

```
plot = Plot.new
plot.add Circle.new [2,1], 5
plot.add Polygon.new [[2,1], [2,3], [1,1]]
```

This is where you can start.
Simple, basic API

```
Plot.new do
  @regions << Circle.new [2,1], 5
  @regions << Polygon.new [[2,1], [2,3], [1,1]]
end
```

Oh, but with instance eval, we can start to get somewhere.
The @regions variable belongs to the new plot. We're not closing over scope.

```
Plot.new do
  circle :center => [2, 1], :radius => 5
  polygon [[1,1], [2,1], [2,3]]
end
```

Because of that, we can lie. I can add methods to plot that wrap the object creation.
Then we can have this code live in an external file and our internal DSL is an external DSL

Treetop

A ruby lib for building external business languages.

```
circle [1,2] 5
polygon [[1,2], [3,4], [5,4]]
rectangle [3,3] width=5 height=1
```

I recently had to generate data for testing a plotting tool.
Do do so, I defined a bunch of regions of these three types.
Then I output events at each point of the plot field.
Here's an external language I wrote last monday, just to make treetop do something.
My hypothetical use case is "Lets imagine we had a non-technical user who needs to generate this data whenever they choose"

```
grammar Shapes
  rule shapes
    shape+
  end
  rule shape
    circle / rectangle / polygon
  end
  rule ws
    (' ')+
  end
  rule circle
    'circle' ws center:point ws 'radius=' radius:[0-9]+
  end
  rule rectangle
    'rectangle' ws start:point ws 'width=' width:[0-9]+ ws 'height=' height:[0-9]+
  end
  rule point
    '[' x:[0-9]+ ',' y:[0-9]+ ']'
  end
  rule polygon
    'polygon' ws '[' first_point:point other_points:(', ' a_point:point)* ']' {
      def points
        [first_point] + rest_points
      end
      def rest_points
        other_points.elements.map {|comma_and_point| comma_and_point.a_point }
      end
    }
  end
end
```

Here's the definition of the treetop grammar. It's a parser expression grammar.
I'm totally sold on this for Business Natural Lanugages

```
require 'rubygems'
require 'treetop'
require 'shapes'

parser = ShapesParser.new
thing = parser.parse("circle [1,2] radius=5").elements.first
puts thing
puts thing.center.x.text_value
puts thing.center.y.text_value
puts thing.radius.text_value

thing = parser.parse("rectangle [5,4] width=2 height=3").elements.first
puts thing
puts thing.start.x.text_value
puts thing.start.y.text_value
puts thing.width.text_value
puts thing.height.text_value

thing = parser.parse("polygon [[4,3], [2,3], [1,5]]").elements.first
puts thing
puts thing.points.map{ |e| e.text_value }
```

Irony

I found this while watching the Lang.NET conference talks that went up this month.
Treetop is sweet, because it's all ruby. But Irony is the same but C#.

Oh, and you'll notice if you're playing at home that irony uses an internal DSL to handle the grammar specification. Is VERRE NICE.

```
public ShapesGrammar()
{
    // Terminals
    var number = new NumberLiteral("number");

    // Non-Terminals
    var point           = new NonTerminal("point", CreatePointNode);
    var pointList       = new NonTerminal("pointList");
    var pointCollection = new NonTerminal("pointCollection", CreatePointCollection);
    var circle          = new NonTerminal("circle");
    var polygon         = new NonTerminal("polygon", CreatePolygon);
    var rectangle       = new NonTerminal("rectangle");
    var shape           = new NonTerminal("shape");
    var shapes          = new NonTerminal("shapes");

    // BNF Rules
    point.Rule           = "[" + number + "," + number + "]";
    pointList.Rule       = MakePlusRule(pointList, Symbol(",") , point);
    pointCollection.Rule = "[" + pointList + "]";
    polygon.Rule         = "polygon" + pointCollection;
    circle.Rule          = "circle" + point + "radius" + "=" + number;
    rectangle.Rule       = "rectangle" + point + "height" + "=" + number + "width" + "=" + number;
    shape.Rule           = rectangle | circle | polygon;
    shapes.Rule          = MakePlusRule(shapes, NewLine, shape);

    this.Root = shapes;

    RegisterPunctuation("[", "]", "=");
}
```

It’s beautiful!
You see here up by point, by passing in a delegate to the CreatePointNode method, I can massage the AST

You can see the operator overloading and how it makes sense to deal with types of BNF operations, concatenation and alteration.

Antlr

A Java framework for generating code from a language definition.
Will generate C#
What it's got going to for it are some tools that allow you to explore the grammar specification.

```
age = 4
myvar = 8 - 4 * (age + 3)
```

Here’s a simple arithmetic language

```
grammar Expr;
prog:   stat+
stat:     expr NEWLINE
      |     ID '=' expr NEWLINE
      |     NEWLINE;
expr:   multExpr (('+' | '-') multExpr*;
multExpr:   atom ('*' atom)*;
atom:   INT
        |  ID
        |  '(' expr ')';

ID: ('a'..'z'|'A'..'Z')+;
INT: '0'..'9'+;
NEWLINE: '\r'? '\n';
WS: (' '|'\t'|'\n'|'\r')+ { skip(); };
```

If you look at the bottom, you'll notice the squirrelly brackets and the skip function.
That's actually java code that's used to reduce the AST.
Now, while it looks a lot like java, it's actually not. It's a series of commands that you can use to modify the tree. We call this reducing.

MGrammar

Part of Oslo, microsoft's new data driven design framework. This allows you to define an external DSL that will create a data tree. Has a live designer that's pretty damn slick. You see what the output data looks like as you define the grammar.

Chris is 24 years old.
Pat is 32 years old.
Billy is 3 years old.
Granny is 98 years old.

This is the language we want to define. It will give us a list of person objects, those objects having a name and an age.

```
module LangNet {
  language Contacts {
    syntax Main = p:Person*
          => People { valuesof(p) };
    syntax Person =
          n:Name "is" a:Age "years" "old"
          => { Name = n, Age = a };
    token Age = '0'..'9'+;
    token Name = ('A'..'Z' | 'a'..'z')+;
    interleave Whitespace = ' ' | '\r' | '\n' | '\n\r';
  }
}
```

We name a syntax, define it's parts, then project it to a data object.

Meta Programming System

This is a product from JetBrains. I've not had time to dig into it, but Rob has when we where thinking about

Lex & Yacc

These are classic tools. I know there's a lex and yacc for fsharp. I've used lex in my fsharp talk. But I'm still not going to show you code, because it's historically seen as no fun.

Roll Your Own

You can use yaml, xml, raw text. What ever the hell you want.

Technique

http://www.flickr.com/photos/wysz/12222304/

Define Your Purpose

Why? What's the pain, or gain you hope to have?

Define Your Structure

You are going to transform the language to data objects. To do that, you need to have a solid representation.
Note something. TRANSFORM!
Don’t let the language drive the data, let the domain.

Define Your Language

How do you want to express the information you need?

Implement Your Language

Take the input text and make an ast, or... build up the APIs.

Implement Your Transformation

have the APIs generate the data you want.
Or massage the AST into real domain objects

Rinse and Repeat

Like all good things, this is an iterative design.
CTT was worked over a few example diagrams. We wrote what we wanted to work to identify the corner cases.

And now, for the main event... How I built Ctt Wrapper in two hours.
Okay, I built a functional prototype. A lot of corner cases didn't occur till after I left the project, but most of it happened relatively quickly.

Our DSL was internal (ruby), data (tree of objects) and code (printed code), terse (relatively)

http://www.flickr.com/photos/ktpupp/32738602/

Why?

I hate Ctt?
I wanted to allow reuse
simplify the modeling of the visual diagrams
there's another class of diagram that I've got an idea about, but it involves excel and is abased on a crazy language called subtextual. But we don't cover that diagram here.

Model

I cheated on this.
Since I'm targeting the CTT language, I have some parts that are similar. I've also added some new things that just make more sense to make explicit.

Schedule

Diagrams

Tests

Initial Conditions

Diagram Instances

Other Symbols

Explicit Assertions

A schedule is a script that drives a unit under test.
Diagrams should belong to the schedule. I'll probably reuse them in multiple tests
Tests are defied by their purpose and the intial conditions
I may not need an instance of all the diagrams in a test, and I may need additional variables, called symbols, to augment a diagram.
I want to be explicit on my pass-fail criteria.

Schedule

A schedule is the ctt file.
It defines the parameters of a test. The way we work is a schedule relates directly to a diagram under test.

```
@sched = Schedule.new do
    project :EngineProgram
    netlist :Module, :Function
    ...
end

puts @sched
```

The project is record keeping
The netlist defines the function we're actually going to call

Before we look inside, notice something. We're using a block of code that's being passed into the constructor.
The function project is defined inside the schedule class. This isn't a closure, because we're doing something strange with it. Any variables built outside the block aren't accessible

```
@sched = Schedule.new do |s|
    s.project :EngineProgram
    s.netlist :Module, :Function
    ...
end

puts @sched
```

We use instance eval so we clean up the syntax. We'll see this throughout.

```
class Schedule
  attr_writer :project
  def initialize(&block)
    instance_eval( &block )
  end
  def netlist(netlist, diagram)
    @netlist = netlist
    @diagram = diagram
  end
  def test(number, &block)
    @tests ||= []
    @tests << Test.new(number,&block)
  end
  def namespaces
    [@netlist, "Globals"].join(", ")
  end
  def to_s
"
Project #{@project};
Diagram #{@netlist}.#{@diagram};

Use #{namespaces};

#{@tests.join("\n")}

-- End Schedule
"
  end
end
```

Here's the code in question. We see the call to instance_eval up top in initialize. <Explain here>

We override to_s to generate a code.

We also see a test method here that creates a test object.

Test

We then break up a schedule into “tests” that represent the sets of inputs for the system.

```
test 10, "describe why this test exists" do
    inputs :var_a => 0,
           :var_b => [true, false]
    outputs :some_bit => true
    inouts :xVar_aState => [0,2]
    ...
end
```

So, we were wondering what to do about numbering tests. We could generate the numbers, but we didn't, just for control.
We are enforcing a description, which is a comment above the output code.

Here, we're setting up the initial conditions of the global variables that impact our unit under test. We're using an inplict hash to say "Var_a takes 0" and an array syntax for multiple values that the variable will iterate over.

CTT tries, but doesn't make a real distinction about these. We do, because the coding standard demands it, so we need this information so we generate the proper output. We actually wrap these values as "Variable" classed objects that understand how to to_string themselves.

```
float :max, "IF (a > b) THEN a ELSE b END"
```

we can create our computed symbols. This is our base functionality. With this, we can replicate everything else.

Another thing we have all over the place, we use a string to represent CTT expressions. I didn't want to try to make that happen. It might give us flexibility, but I couldn't find a way that wasn't too wordy. Also, we can munge just fine. I'll get to that.

This creates a typed "ComputedSymbol" object.

```
component :switch do
  input :selector, :BOOLEAN
  input :if_t, :FLOAT
  input :if_f, :FLOAT
  output :out, :FLOAT,
     "IF (selector) THEN if_t ELSE if_f END"
end
```

This is how we define a sub diagram. In this case, a switch.
It's simple, we have three inputs, the selector and the values we're choosing between. Then we have the output value.
The output and the names of the values need to be specific to each instance in the CTT. So, we munge the names.

```
vs = instance :switch, :valid_signal,
  :selector => :sig_fault,
  :if_t => :signal_default,
  :if_f => :signal_raw
```

So, here we create an instance of a diagram. We name it, look it up, and create an instance. the cool thing is, the outputs of this diagram can be accessed via method calls. And we'll see that when we verify our SUT actually passed.

```
assert :sig_out, vs.d
assert :sig_out, vs.d, "0.004"
```

The last part is how we define our passes.
CTT has us define a set of booleans that we and together to make sure our test passes, or not.
We just use an assert statement that will give us a variable called pass10_a and pass10_b and automatically make pass10 the and of all of them.

We can also set up a deadband for our equality.

But I left

I left without solving all of the problems. Some circular dependencies in diagrams where solved by Keith Marcum. He used procs to lazy bind the output.
I never got the ability to import library diagrams.
And several other features.

Wishful Thinking

DSLs are a type of wishful thinking.
We program outside in.
We start by defining what we want, and then find a way to make it real. This is orthogonal to how we usually do things, from the smallest part to the largest. We like the fiddly bits. DSLs want you to make the fiddly bits go away, to use convention or some other thing.
http://www.flickr.com/photos/andycastro/3301882877/