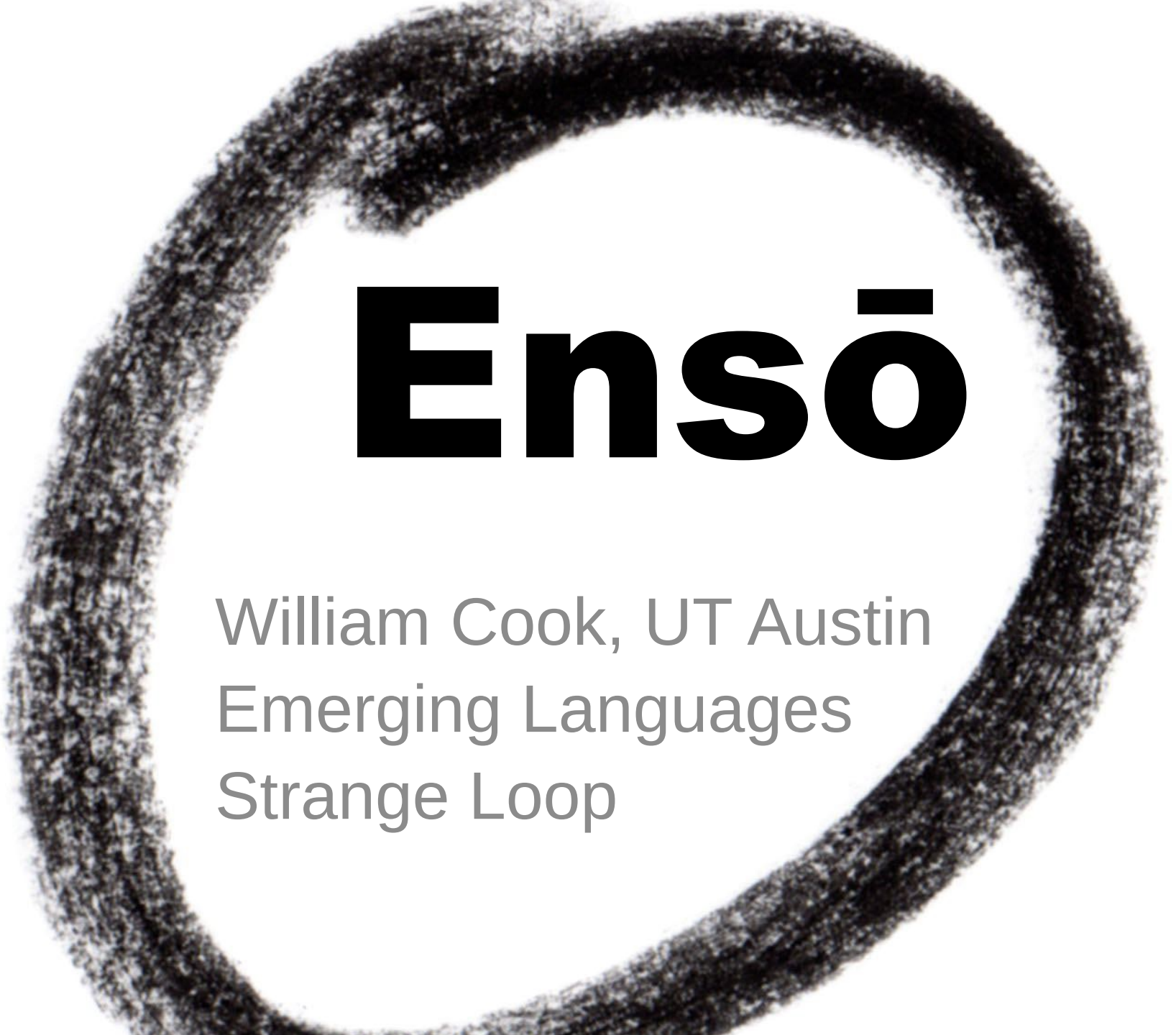


Ensō

William Cook, UT Austin

Emerging Languages

Strange Loop

Requirements (WHAT)

Code

Behavior

Requirements (WHAT)

Strategies (HOW)

Code

Behavior

Requirements

Technical Requirements

Models

Strategies

Behavior

Data Requirements

Technical Requirements

Data Model

Data Manager

Objects

Using Managed Data (Ruby)

- Description of data to be managed

```
Point = { x: Integer, y: Integer }
```

- Dynamic creation based on metadata

```
p = BasicRecord.new Point
p.x = 3
p.y = -10
print p.x + p.y
p.z = 3  # error!
```

Implementing Managed Data

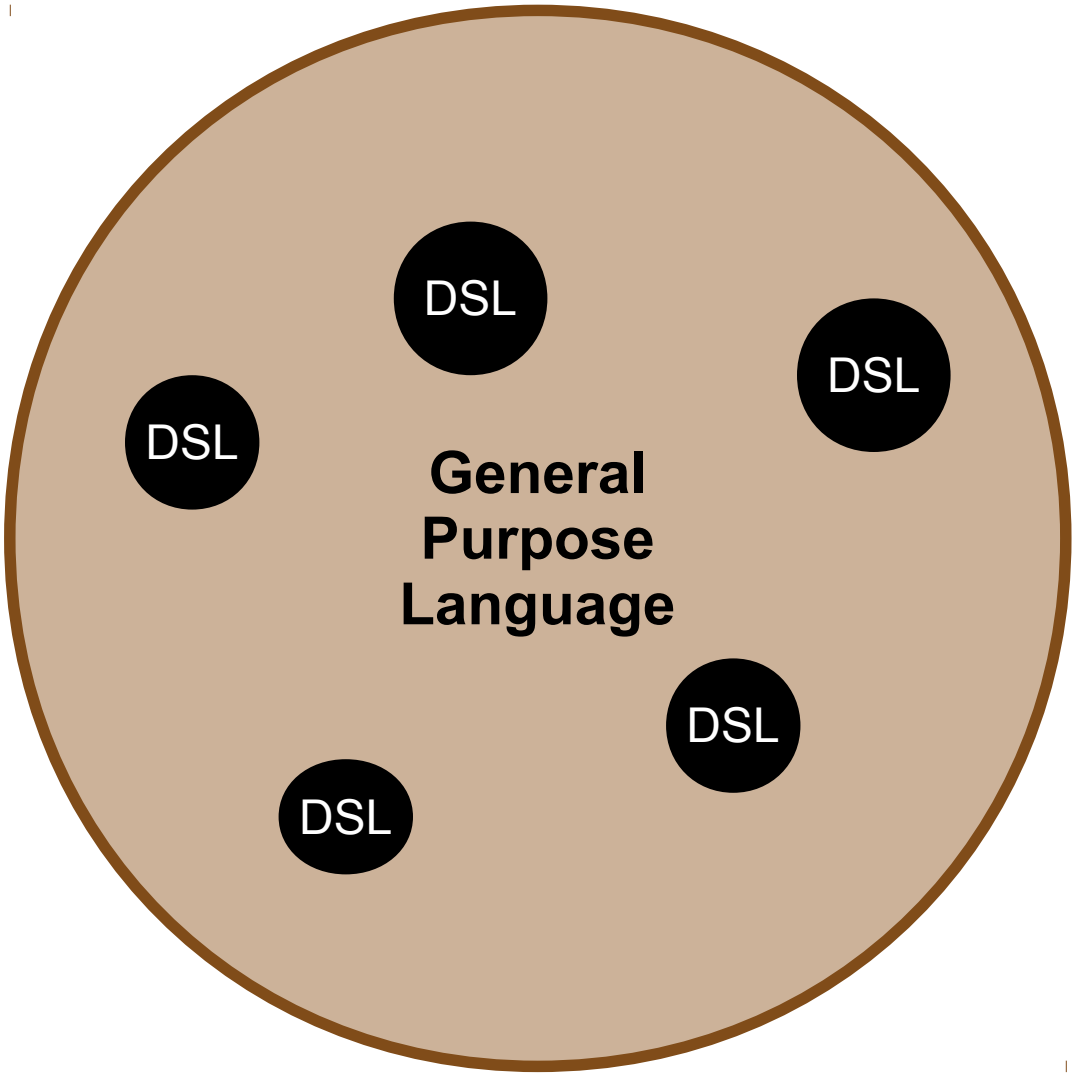
- Override the "dot operator" (p.x)
- Reflective handling of unknown methods
  - Ruby method_missing
  - Smalltalk: doesNotUnderstand
  - Also IDispatch, Python, Objective-C, Lua, CLOS
  - Martin Fowler calls it "Dynamic Reception"
  - Programmatic Method creation
    - E.g. Ruby define_method

Other Data Managers

- Mutability: control whether changes allowed
- Observable: posts notifications
- Constrained: checks multi-field invariants
- Derived: computed fields (reactive)
- Secure: checks authorization rules
- Graph: inverse fields (bidirectional)
- Persistence: store to database/external format
- General strategy for all accesses/updates
- Combine them for *modular strategies*

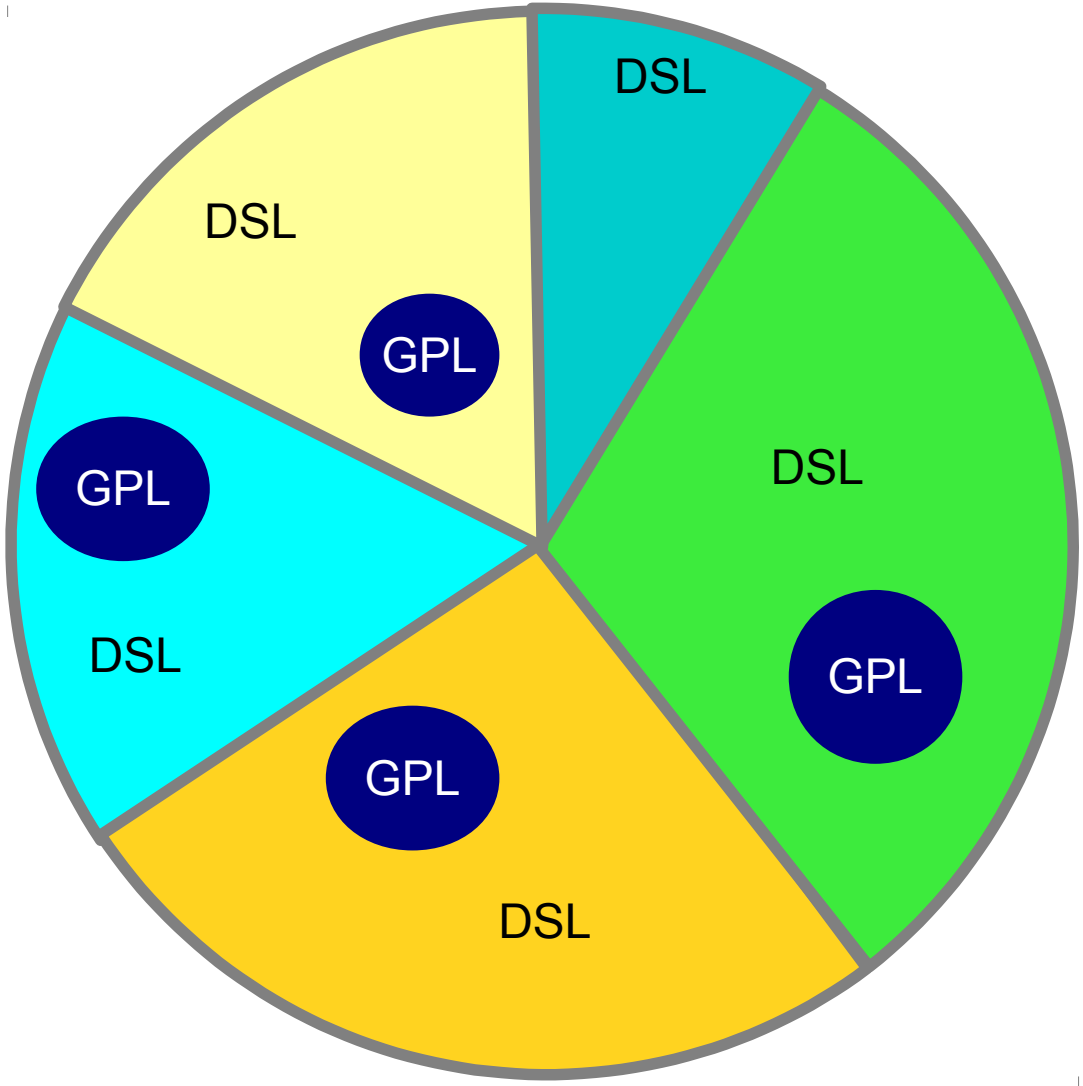
Current Practice

“Chocolate Chip Cookie”



Future Vision

“Blueberry Pie”



Grammars

- Mapping between *text* and *object graph*
- A *point* is written as (x, y)

Individual	***Grammar***
(3, 4)	P ::= [Point] "(" x:int "," y:int ")"

- Notes:
  - Direct reading, no abstract syntax tree (AST)
  - Bidirectional: can parse and pretty-print
  - GLL parsing, *interpreted!*

Expression Example

Sample Expression

3*(5+6)

Expression Grammar

```
E   ::= [Add] left:E "+" right:M   | M
M   ::= [Mul] left:M "*" right:P   | P
P   ::= [Num] val:int              | "(" E ")"
```

An *Expression* Interpreter

```
module Eval
  def eval(exp)
    dispatch(:eval, exp)
  end

  def eval_Num(val)
    val
  end

  def eval_Add(left, right)
    left.eval + right.eval
  end

  def eval_Mul(left, right)
    left.eval * right.eval
  end
end
```

Expression Schema

```
class Exp

class Num
  val : int

class Add
  left  : Exp
  right : Exp

class Mul
  left  : Exp
  right : Exp
```

State Machine Example

Door StateMachine

```
start Opened

state Opened
  on close go Closed

state Closed
  on open go Opened
  on lock go Locked

state Locked
  on unlock go
Closed
```

StateMachine Grammar

```
M::= [Machine] "start" \start:</states[it]> states:S*
S ::= [State] "state" name:sym   out:T*
T ::= [Trans] "on" event:sym "go" to:</states[it]>
```

A StateMachine Interpreter

```
def run_state_machine(m)
  current = m.start
  while gets
    puts "#{current.name}"
    input = $_.strip
    current.out.each do |trans|
      if trans.event == input
        current = trans.to
        break
      end
    end
  end
end
```

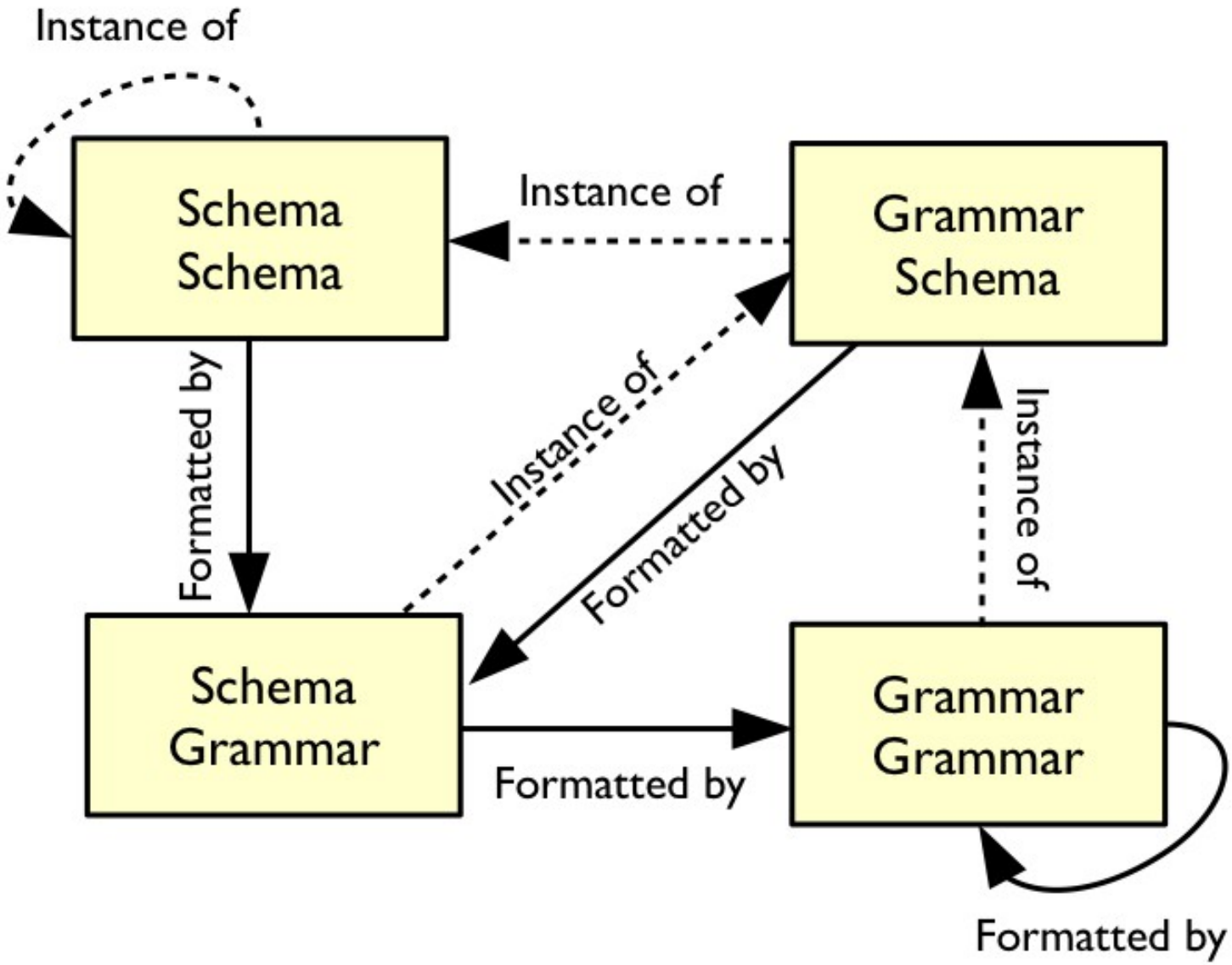
StateMachine Schema

```
class Machine
  start  : State
  states! State*

class State
  machine: Machine
  name #   str
  out    !   Trans*
  in       :   Trans*

class Trans
  event :  str
  from   :  State / out
  to       :  State / in
```

Grammar Grammar

```
start G
G   ::= [Grammar] "start"  start:</rules[it]> rules:R*
R   ::= [Rule]          name:sym "::=" arg:A
A   ::= [Alt]           alts:C+ @"|"
C   ::= [Create]        "[" name:sym "]" arg:S   |  S
S   ::= [Sequence] elements:F*
F   ::= [Field]         name:sym ":" arg:P       |  P
P   ::= [Lit]           value:str
      |  [Value]    kind:("int" | "str" | "real" | "sym")
      |  [Ref]      "<" path:Path ">"
      |  [Call]     rule:</rules[it]>
      |  [Code]     "{" code:Expr "}"
      |  [Regular] arg:P "*" Sep?     { optional && many }
      |  [Regular] arg:P "?"          { optional }
      |  "(" A ")"
Sep   ::= "@" sep:P
```

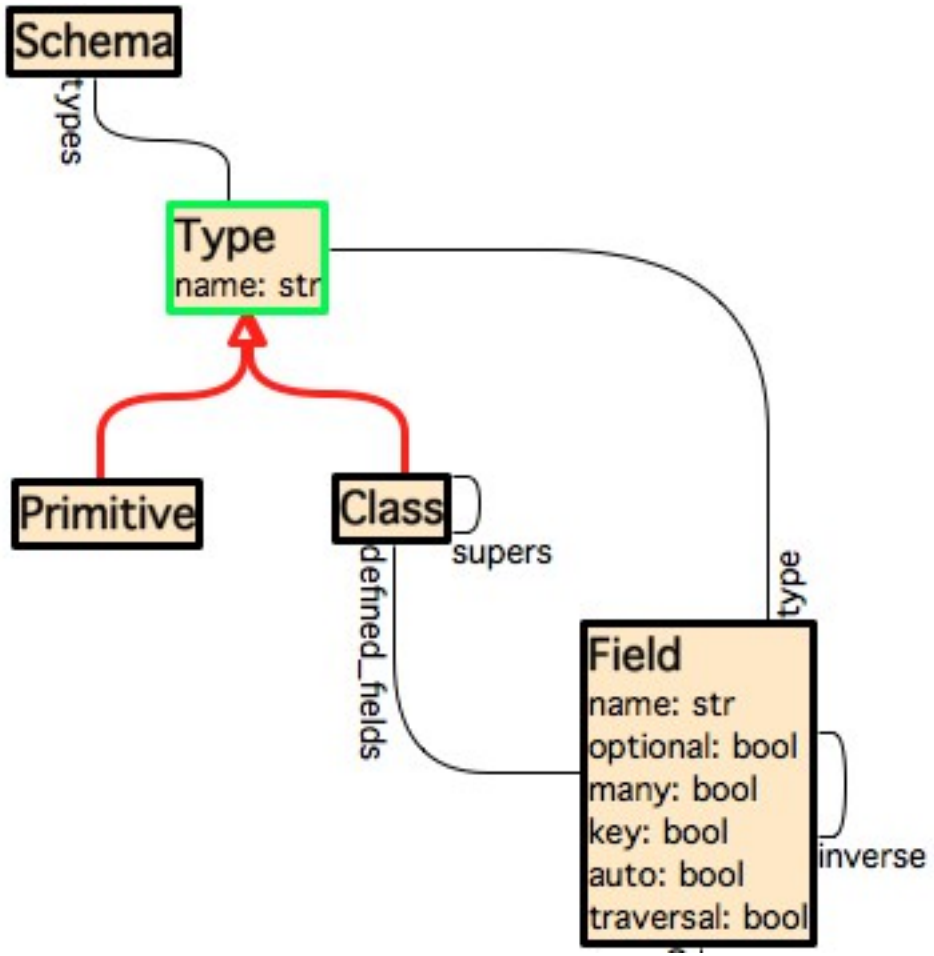
Quad-model


Instance of
Schema
Schema
Instance of
Grammar
Schema
Formatted by
Instance of
Formatted by
Instance of
Schema
Grammar
Formatted by
Grammar
Grammar
Formatted by

Diagrams

- Model
  - Shapes and connectors
- Interpreter
  - Diagram render/edit application
  - Basic constraint solver

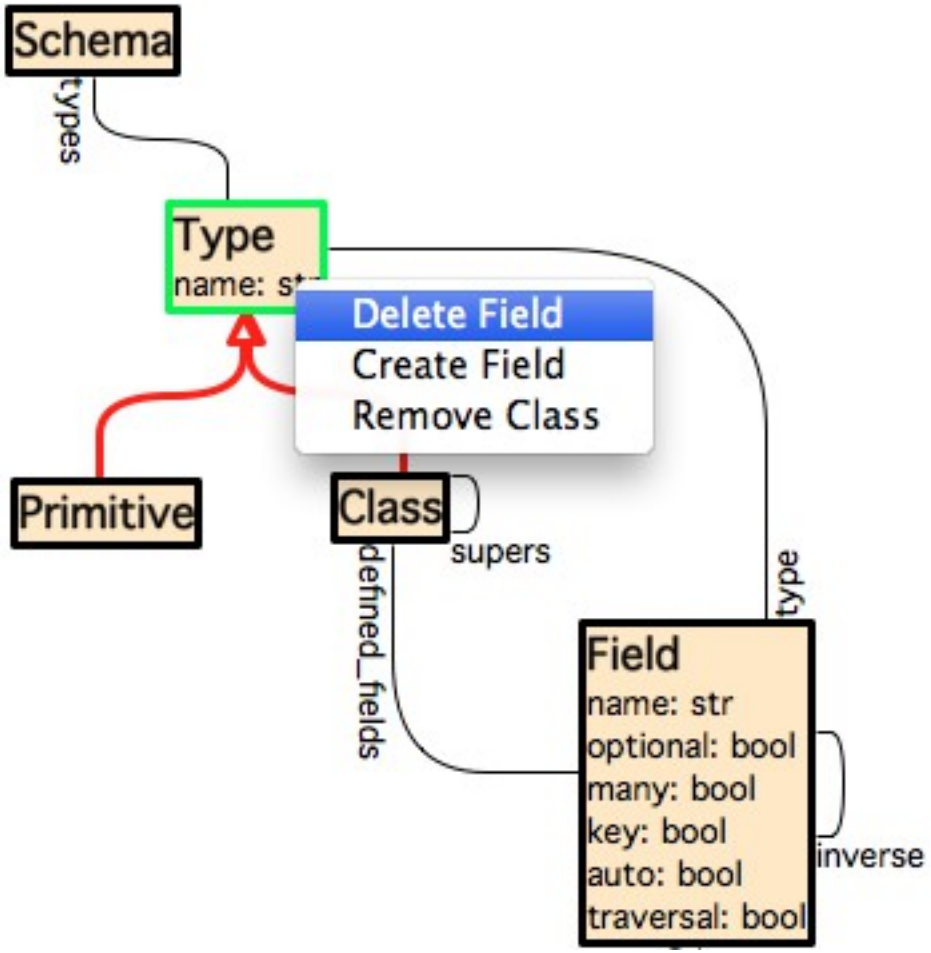
Schema Diagram



Stencils

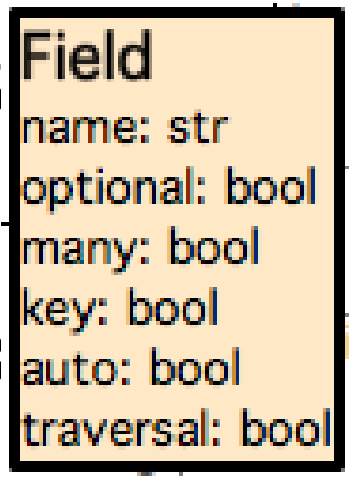
- Model: mapping object graph → diagram
- Interpreter
  - Inherits functionality of Diagram editor
  - Maps object graph to diagram
    - Update projection if objects change
  - Maps diagram *changes* back to object graph
  - Binding for data and collections
    - Strategy uses schema information
    - Relationships get drop-downs, etc
    - Collections get add/remove menus

Schema Diagram Editor



Schema Stencil

```
diagram(schema)
graph [font.size=12,fill.color=(255,255,255
menu "Class" for class : schema.classes
   label class
     box [line.width=3, fill.color=(255,228,18
       vertical {
         text [font.size=16,font.weight=700] class.name
         menu "Field" for field : class.defined_fields
           if (field.type is Primitive)
             horizontal {
               text field.name       // editable field name
               text ": "
               text field.type.name // drop-down for type
             }}}}
```



Requirements

Changed
Strategies
(HOW)

Very different
Code

Different
Behavior

Requirements

Technical Requirements

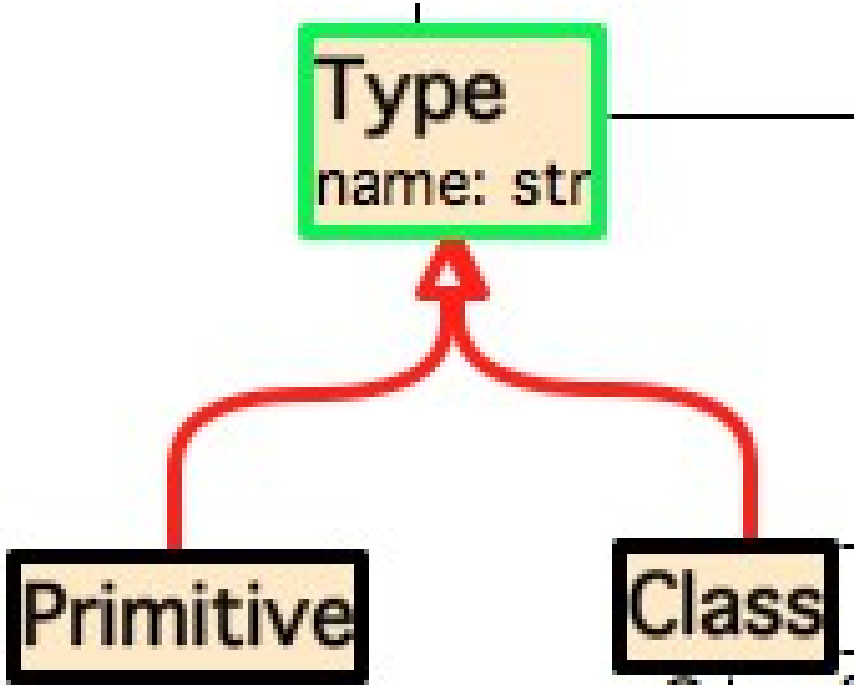
Models

Changed Strategies

Different Behavior

Schema Stencil: Connectors

```
 …
// create the subclass links
  for class : schema.classes
    menu  "Parent" for super : class.supers
      connector [line.width=3, line.color=(255,0,0)]
          (class --> super)
…
[also for relationships]
```



Requirements

Technical Requirements

Data, Security, User Interface, Workfow, etc...

Multiple Strategies

Behavior

Requirements

Technical Requirements

Models

Aspects + Strategies

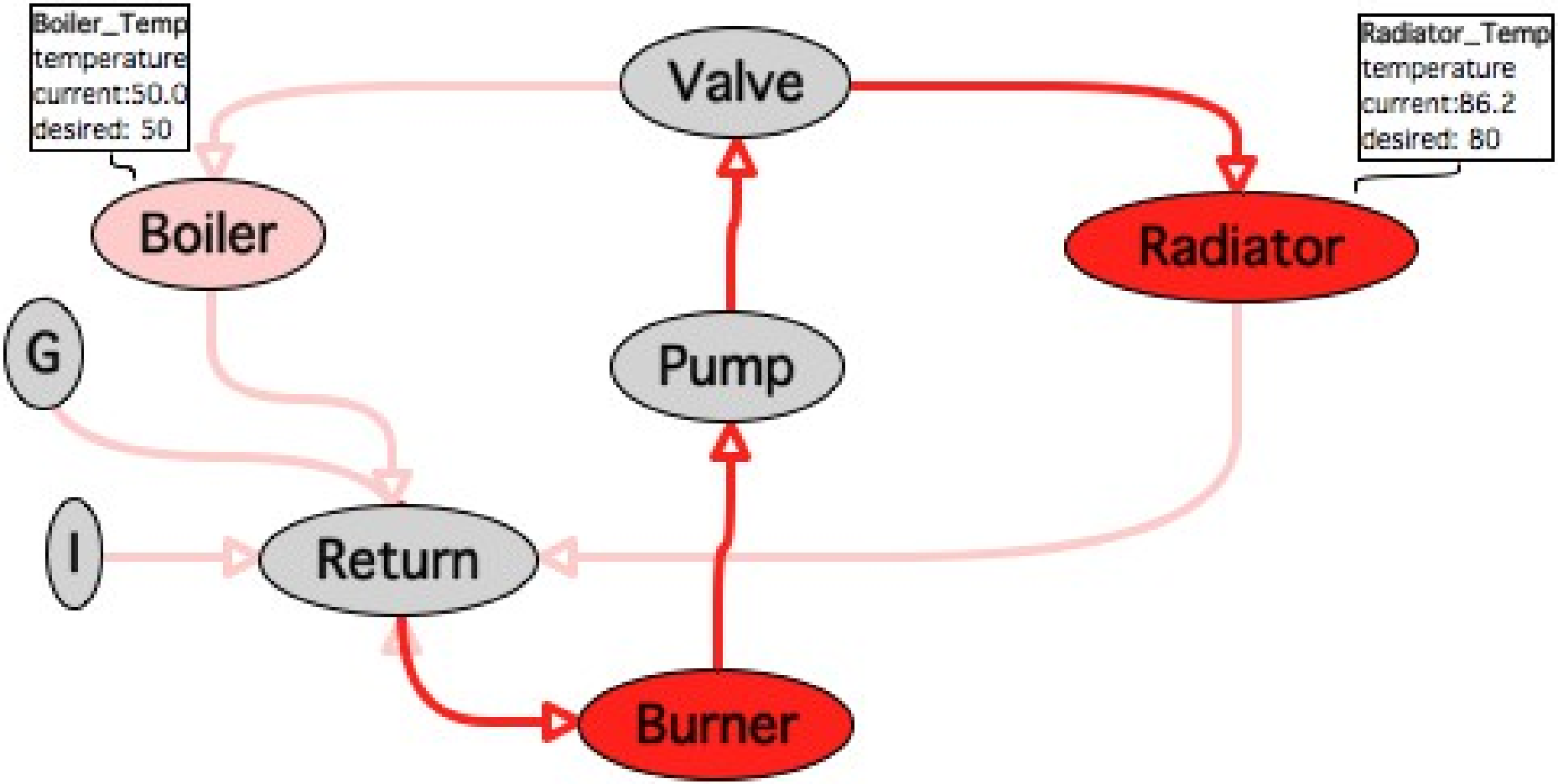
Debugging, Security, etc

Behavior

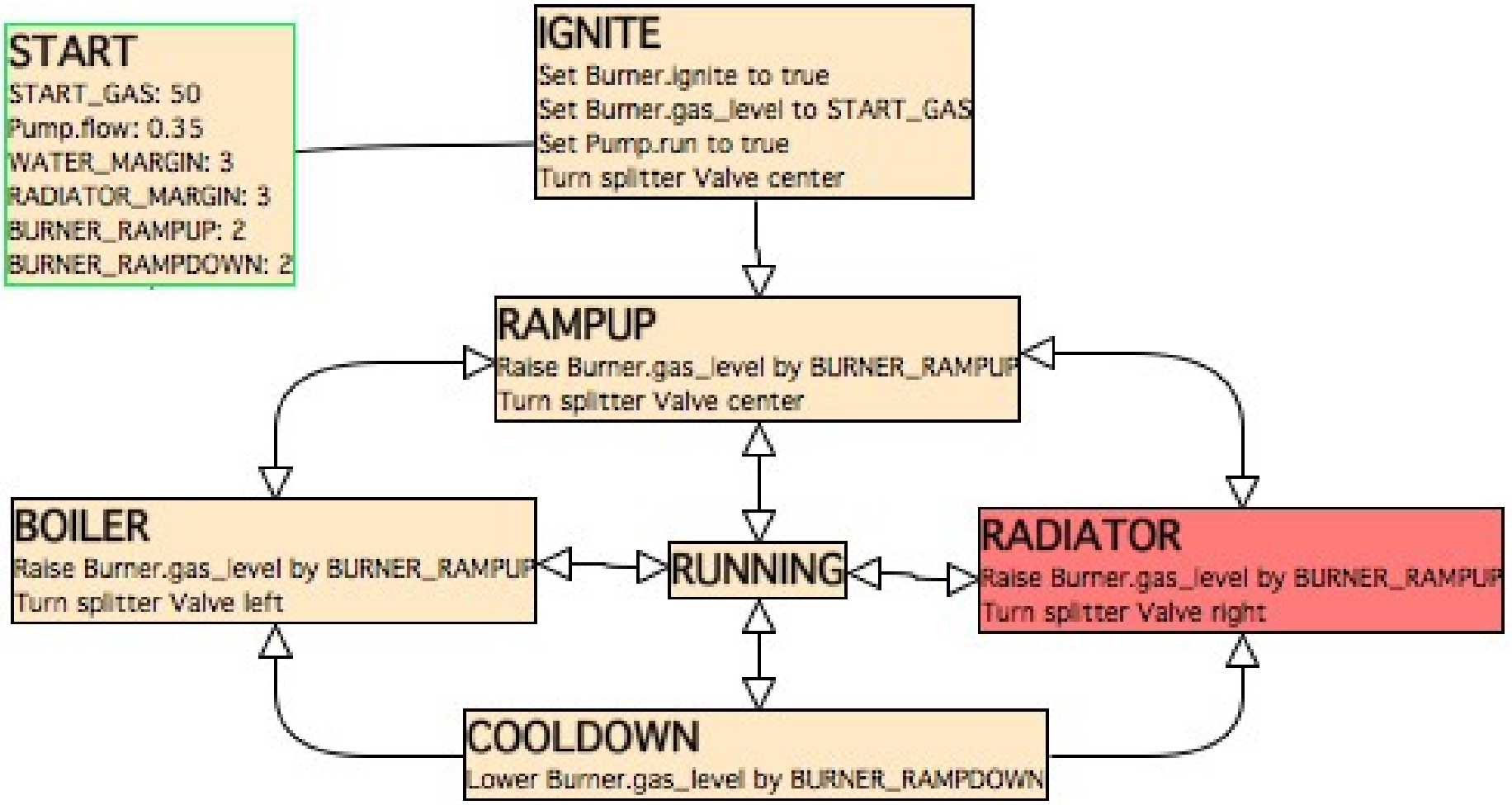
Language Workbench Challenge

- Models
  - Physical heating system
    - furnace, radiator, thermostat, etc
  - Controller for heating system
- Interpreter
  - Simulator for heating system
    - pressure, temperature
  - State machine interpreter
    - Events and actions

Physical Heating System Model



Piping Controller



Piping Details

- Simulation updates physical model
  - Change to physical model causes update to view
  - Observable Data Manager -> Presentation update
- State machine interpreter changes states
  - Presentation shows current state
- User can interact with physical model
  - Change thermostat
- User can edit diagram

Performance

- Ensō is currently slow but usable
  - Accessing a field involves two levels of meta-interpretation
  - My job is to give compiler people something to do
- Partial Evaluation of model interpreters

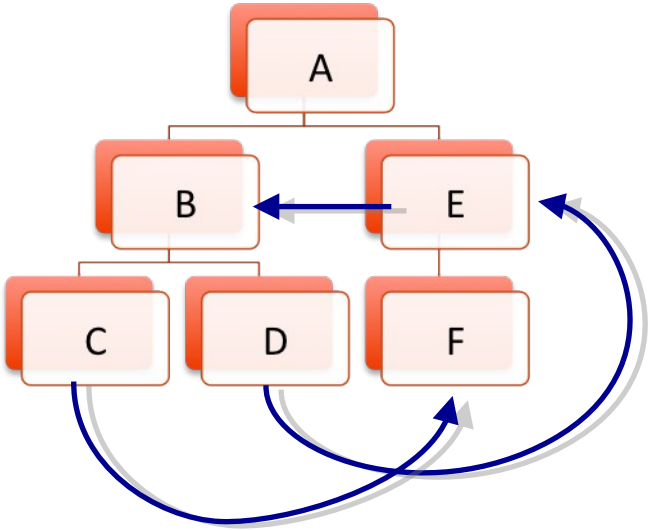
```
web(UI, Schema, db, request) : HTML
web[UI, Schema](db, request) : HTML
```

static

dynamic

Trees versus Graphs

- Trees (Algebraic types)
  - one problem: *dominant decomposition*



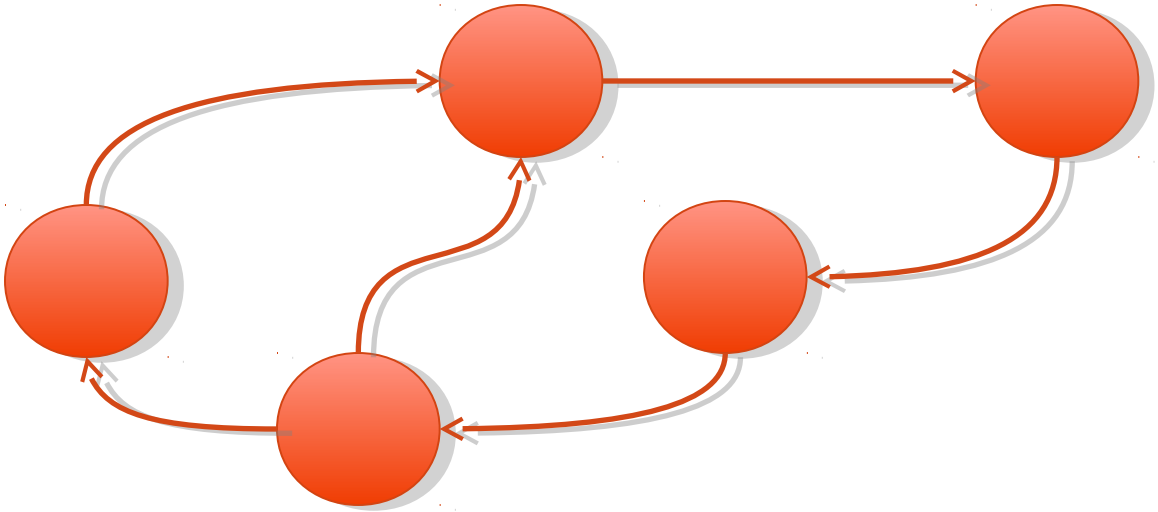
Cross links use names or identifiers

Similar to foreign key encoding
in relational databases

- *Graphs (Coalgebra?)*
  - *Potentially difficult to work with*

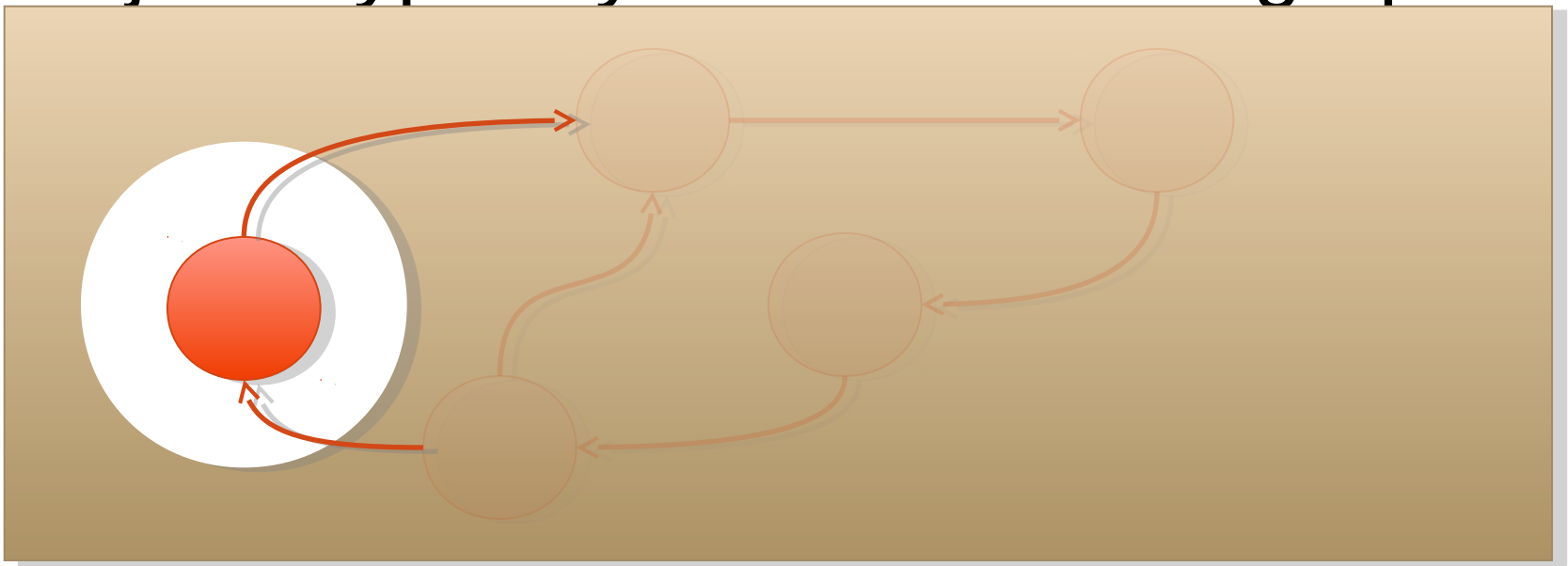
Objects versus Graphs

- Objects typically connected into graphs



Objects versus Graphs

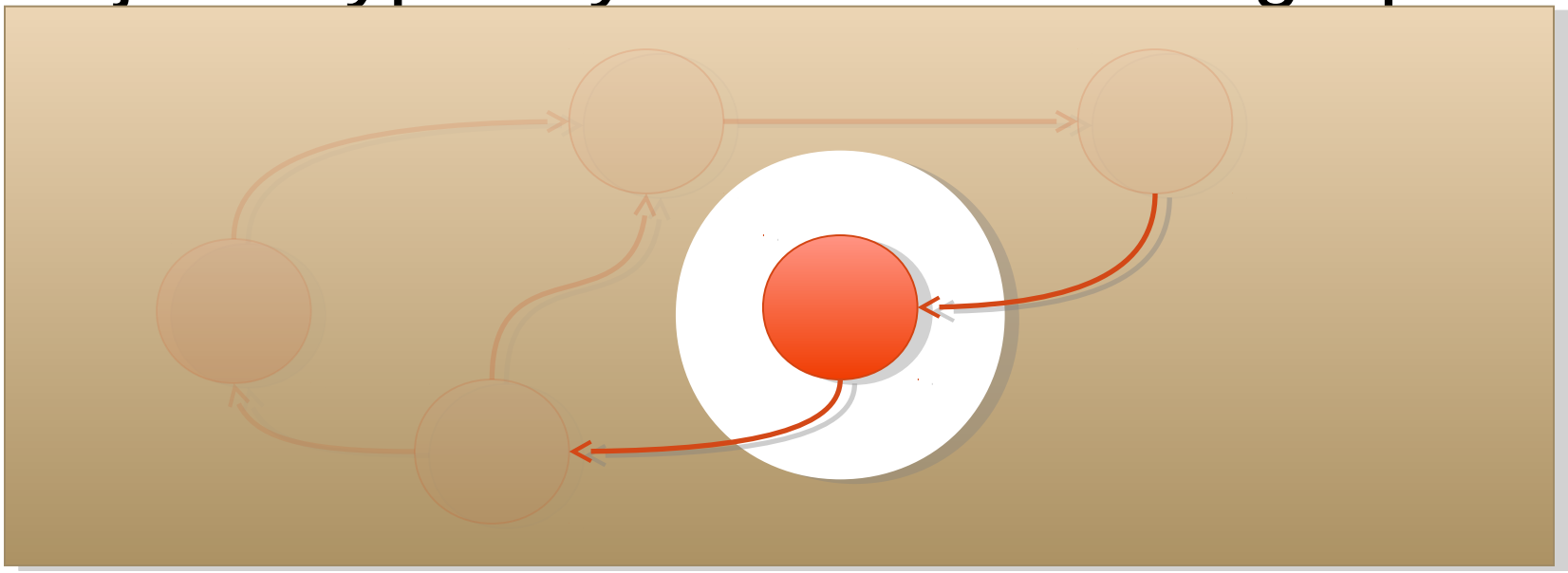
- Objects typically connected into graphs



- Focus on individual nodes
  - Only see the “trees” not the “forest”

Objects versus Graphs

- Objects typically connected into graphs



- Focus on individual nodes
  - Only see the “trees” not the “forest”

Aspect	Code SLOC	Model SLOC
Bootstrap	387	—
Utilities	256	—
Schemas	691	51
Grammar/Parse	885	106
Render	318	17
Web	932	305
Security	276	46
Diagram/Stencil	1389	176
Expressions	448	144
Core	5582	844
Piping	527	268

Ensō Summary

- Executable Specification Languages
  - Data, grammar, GUI, Web, Security, Queries, etc.
- External DSLs (not embeded)
- Interpreters (not compilers/model transform)
  - Multiple interpreters for each languages
- Composition of Languages/Interpreters
  - Reuse, extension, derivation (inheritance)
- Self-implemented (Ruby & JavaScript)
  - Partial evaluation for speed

Don't Design Your Programs...

Program
Your Designs

Ensō

enso-lang.org