The Reemergence of Datalog

Return of the Living Datalog


I like turtles

Data

Rectangles

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Pumpkin Spice	138	0.99	400,000	396,000
Dung Spice	666	2.00	null	null

RDL

```
create table COFFEE
   (COF_NAME varchar(32) NOT NULL,
   SUP_ID int NOT NULL,
   PRICE numeric(10,2) NOT NULL,
   SALES integer NOT NULL,
   TOTAL integer NOT NULL,
   PRIMARY KEY (COF_NAME),
   FOREIGN KEY (SUP_ID) REFERENCES SUPPLIERS (SUP_ID));
```

COFFEE
COF_NAME : varchar (PK)
SUP_ID : int (FK)
PRICE : numeric
SALES : integer
TOTAL : integer

Rectangulation

- Relationship between entities
- Sparse data
- Multi-valued attributes
- **PL**ace-**O**riented **P**rogramming


Java

```
public static void viewTable(Connection con) throws SQLException {
    Statement stmt = null;
    String query = "select COF_NAME, SUP_ID, " +
                   "PRICE, SALES, TOTAL from " +



    try {
        stmt = con.
        ResultSet r
        while (rs.n
            String
            int sup
            float p
            int sal
            int tot
            doPrint                                    , total);
        }
    } catch (SQLExc
        JDBCTutoria
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

Where's the Data?

Java

```java
public static void viewTable(Connection con) throws SQLException {
    Statement stmt = null;
    String query = "select COF_NAME, SUP_ID, " +
                   "PRICE, SALES, TOTAL from " +
                   dbName + ".COFFEE";

    try {
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            doPrint(coffeeName, supplierID, price, sales, total);
        }
    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

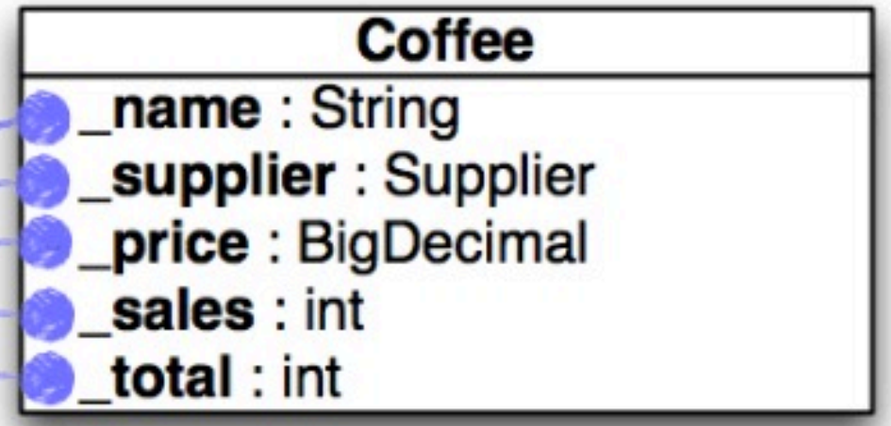
Java

```
public static void viewTable(Connection con) throws SQLException {

        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            doPrint(coffeeName, supplierID, price, sales, total);
        }


}
```

Java

RDL

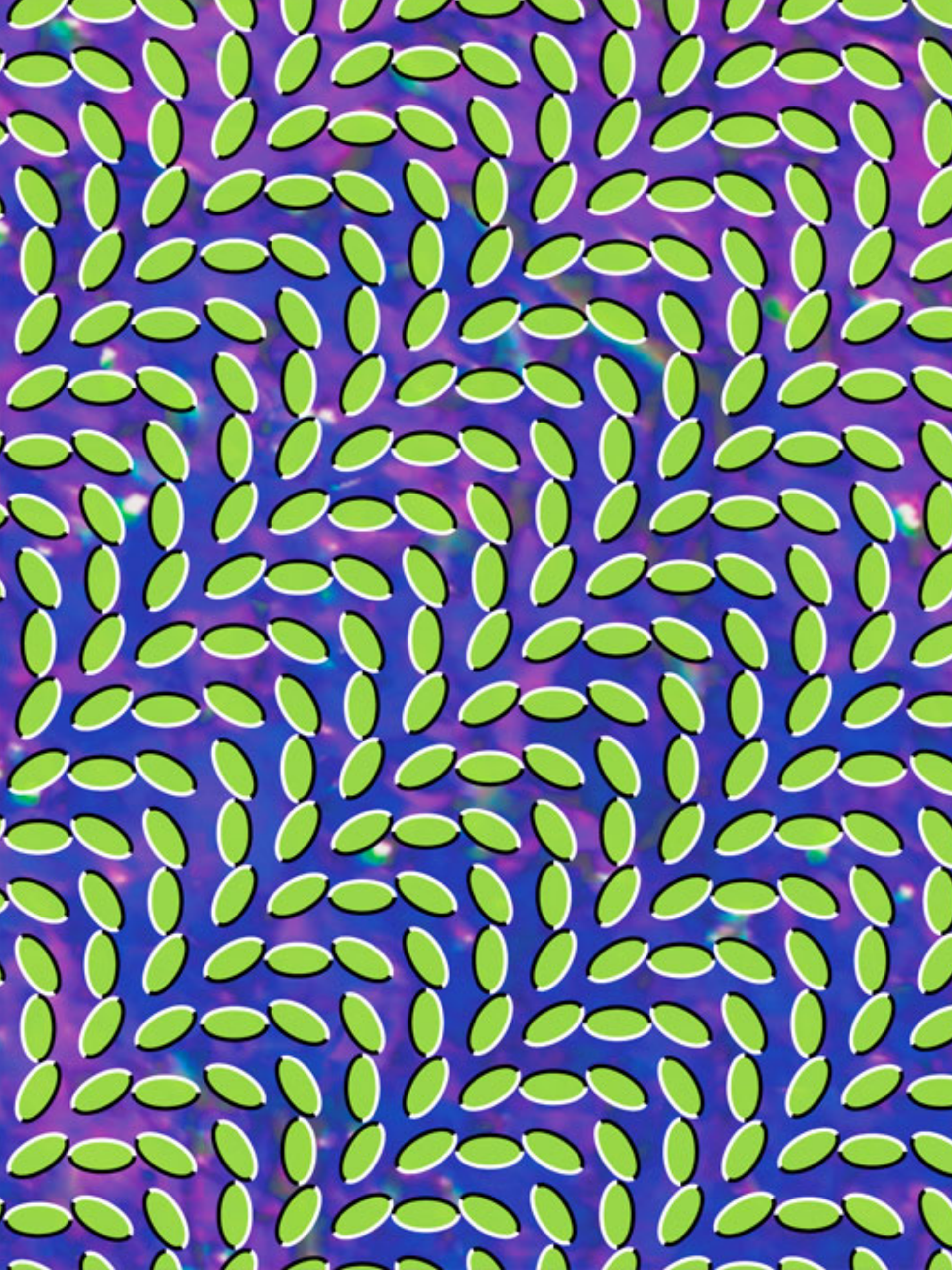
```
public static void viewTable(Connection con) throws SQLException {
    Statement stmt = null;
    String query = "select COF_NAME, SUP_ID, " +
                   "PRICE, SALES, TOTAL from " +
                   dbName + ".COFFEES";

    try {
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            doPrint(coffeeName, supplierID, price, sales, total);
        }
    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```




ORMG!

Code as Code.

Data as Data.



Cookies

User information

Protocols

Lisp

Schemas

Events

Chess moves

TX

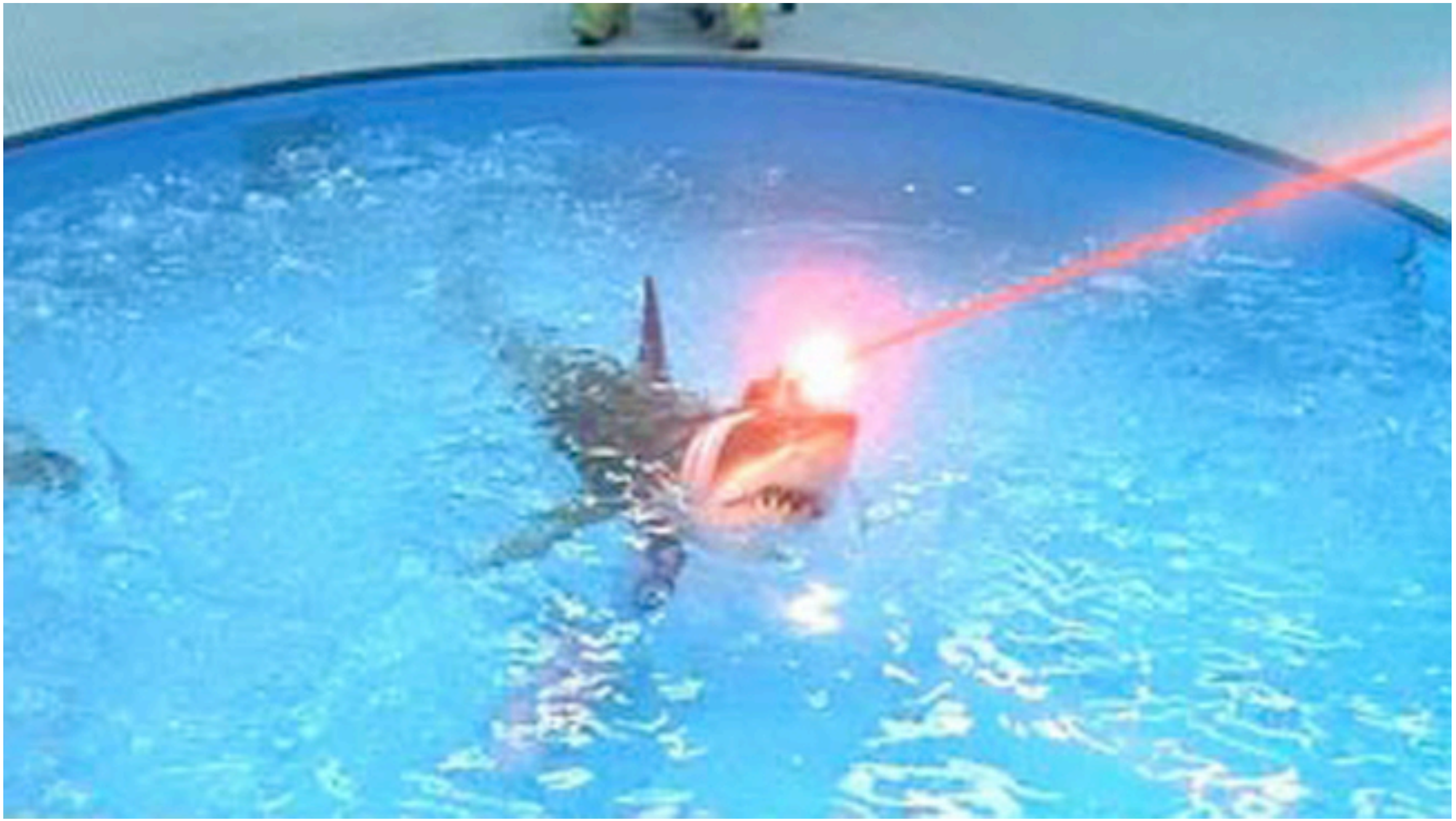
...

as Data.

Code as Code.

Data as Data.

Unification



Your data

```
{:first "Dario"
 :last  "Argento"
 :genre :giallo}
```

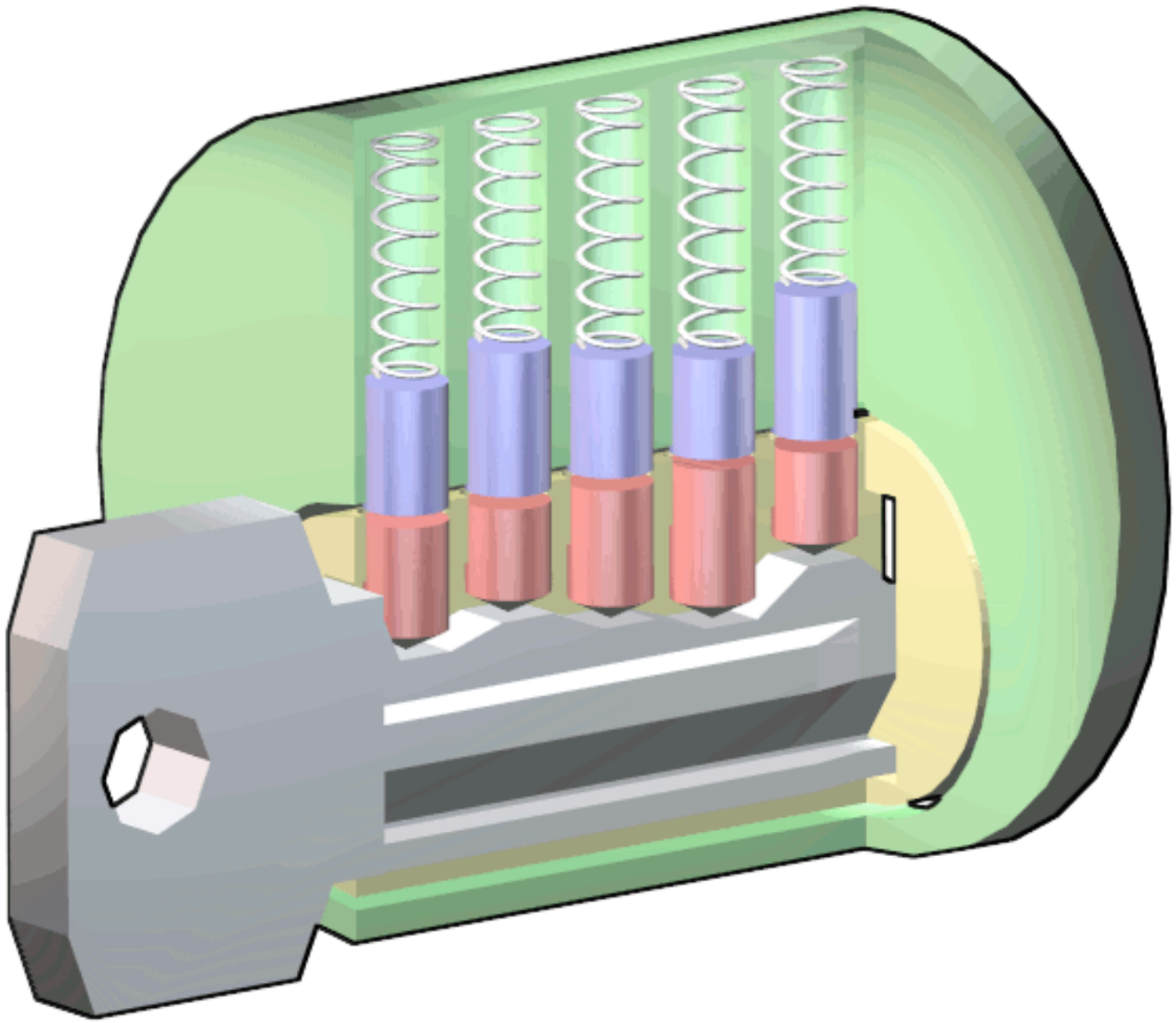
punching holes in

Your data

```
{:first ?
 :last  ?
 :genre :giallo}
```

fitting the holes in

Your data



Variables

```
{:first '?first
 :last  '?last
 :genre :giallo}
```

Deriving bindings

```
(unify
  {:first '?first
   :last  '?last
   :genre :giallo}

  {:first "Dario"
   :last  "Argento"
   :genre :giallo})

;=> {?first "Dario", ?last "Argento"}
```

Substitution

```
(subst
  ['?first '?last]              ;; TEMPLATE

  {'?first "Dario",             ;; BINDINGS
   '?last "Argento"})
```

```
;=> ["Dario" "Argento"]
```

Leaving variables

```
(subst '[1 2 ?x ?y]
       '{?x [3 4 ?y 6]})
;=> [1 2 [3 4 ?y 6] ?y]
```

Related variables

```
(subst '[1 2 ?x ?y]

       '{?x [3 4 ?y 6]
         ?y :BUB})

;=> [1 2 [3 4 :BUB 6] :BUB]
```

MGU

```
(unifier '[(?a * ?x | 2) + (?b * ?x) + ?c]
         '[?z + (4 * 5) + 3])

;=> [(?a * 5 | 2) + (4 * 5) + 3]
```

Prolog

Prolog data

type	example
atom	bill
atom	'hi there'
int	42
float	3.14
variable	X
variable	Name
variable	[L R]
functor	first(L).
list	[1,2]
operator	+ - / *

Prolog facts

```
spawned(bill, carl).
spawned(bill, dilbert).
spawned(dilbert, phil).

aged(bill,81).
aged(carl,33).
aged(dilbert, 40).
aged(phil,10).
```

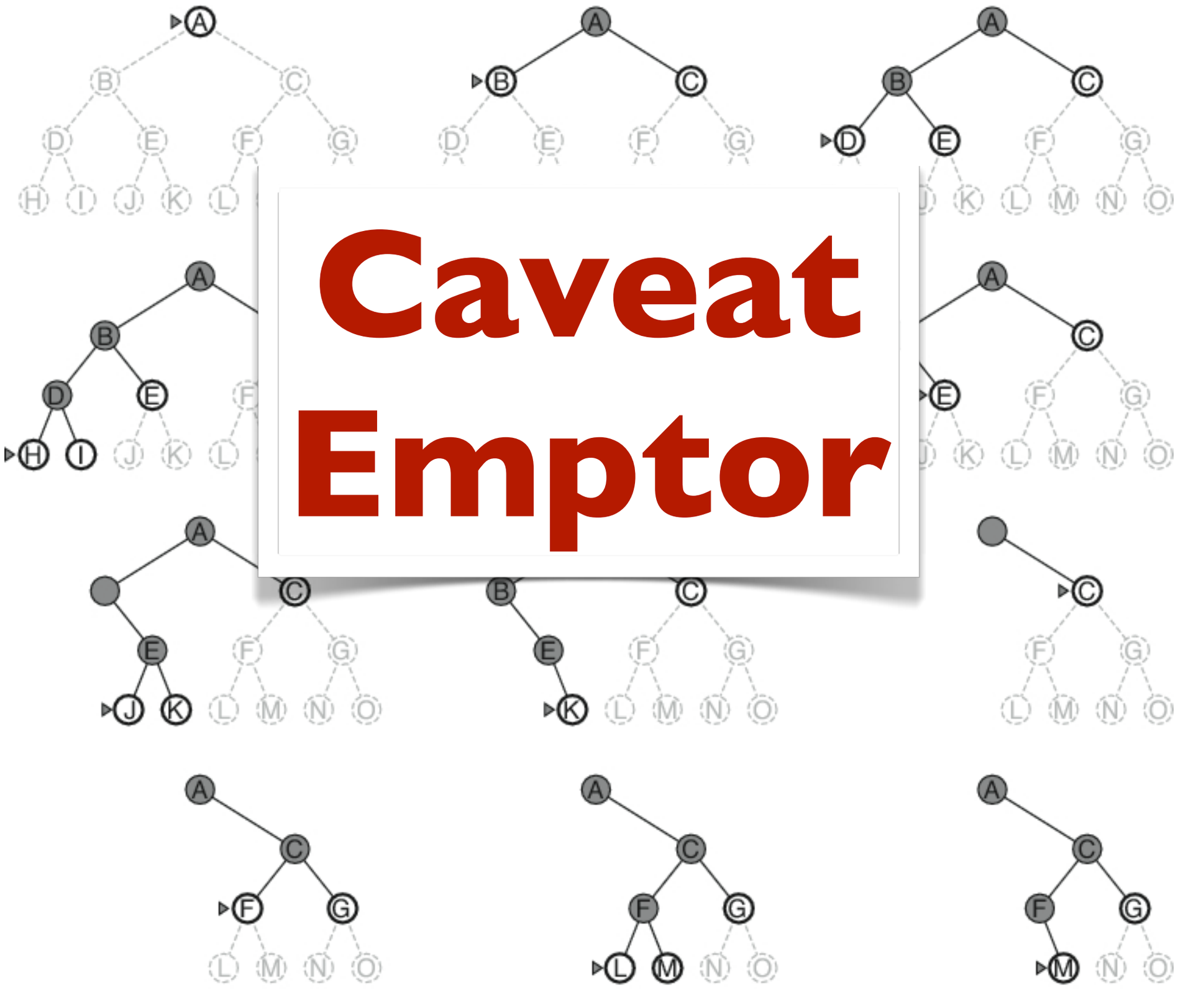
Prolog rules

```
descendant(X,Y) :- spawned(X,Y).
descendant(X,Y) :- descendant(X,Z), spawned(Z,Y).
```

Prolog query

```
descendant(bill,Kid),
aged(Kid,Age),
Age < 13.

% Kid = phil,
% Age = 10 .
```

Data as Code!



Caveat

Emptor

Caveats

- Clause-order dependence
- Non-termination
- Infection of imperative

Clause order dependence

```
descendant(X,Y) :- spawned(X,Y).
descendant(X,Y) :- descendant(X,Z), spawned(Z,Y).

descendant(bill,D).

% D = carl ;
% D = dilbert ;
% D = phil ;
% runs forever!!!
```

Clause order dependence

```
descendant(X,Y) :- spawned(X,Y).
descendant(X,Y) :- spawned(Z,Y), descendant(X,Z).

descendant(bill,D).

% D = carl ;
% D = dilbert ;
% D = phil ;
% false.
```

Non-termination

```
married(A,B) :- married(B,A).
```

Non-termination

```
married(edard,catelyn).
% never stops
```

Fixed?

```
are_married(A,B) :- married(A,B).
are_married(A,B) :- married(B,A).
married(edard, catelyn).

?- are_married(edard,catelyn).
true.

?- are_married(catelyn,edard).
true.
```

Imperative infection

- !
- fail

Cut

```
statement :- condition, !, then.
statement :- else.
```

```
zzz(T,A) :- T > 2100, !, A = sleep.
zzz(_,A) :- A = play_chess.

?- zzz(2000, A).
A = play_chess.

?- zzz(2300, A).
A = sleep.
```

fail

```
pparts(L) :- append(A, B, L),
             write(A), write(' '), write(B), nl,
             fail.

?- pparts([1,2]).
[] [1,2]
[1] [2]
[1,2] []
false.
```

Prolog is prelude

- Powerful
- Often beautiful
- Not as declarative as we'd like

Datalog

Datalog is...

- A query language
- Not Turing complete
- Explicit
- Simple
  - to use
    - ... and implement
      - ... kinda

History

- 1977: Gallaire and Minker's Symposium on Logic Data Bases
- 1980s: Nail, LDL, Coral
- 1995: Stonebraker and Hellerstein declare "no practical applications …"
- **The dark years...**

History

- 2002: Binder, a logic-based security language by DeTreville
- 2000s: Declarative networking, bddbddb, Orchestra CDSS, Doop, SecureBlox, Dedalus, more
- 2010: The Declarative Imperative by Hellerstein!
- Today: Bloom, Cascalog, Datomic, LogicBlox, more

Datalog is also

- Declarative logic programming with termination
- Recursive queries
- Implicit joins

EAV

Entity	Attribute	value
`999-99-9999`	`:person/age`	`42`

- “Entities” (objects?) - a grouping of tuples
- Make that efficient

Query elements

element	specifies	details
`:where`	constraints	patterns, predicates, functions
`:find`	return values	variable bindings
`:in`	data sources	databases, collections, objects

Patterns

pattern	binds type	input	`?a` binding
`?a`	scalar	42	42
`[?a ?b]`	tuple	[1 2]	1
`[?a ...]`	collection	[1 2]	1, 2
`[[?a ?b ?c] ]`	relation	[john :likes :pizza] [jane :likes :pasta]	john, jane

Simple query

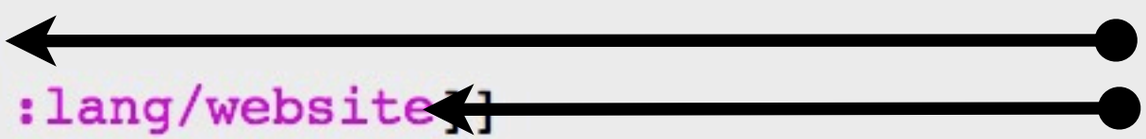
- Find all language entities with a website entry

```
[:find   ?language
 :where [?language :lang/website]]

;=> [[1234567890] [1234567891] ...]
```

Simple query

- Find all language <u>entities</u> with a website entry

```
[:find   ?language
 :where [?language :lang/website]]

;=> [[1234567890] [1234567891] ...]
```



entity
attribute

Binding query

- Find all language URLs with a website entry

```
[:find   ?url
 :where [?language :lang/website ?url]]

;=> [[#url "http://visi.io"] [#url "http://ioke.org"] ...]
```

entity

attribute

value

Join

```
[:find   ?name ?url
 :where [?language :lang/website ?url]
        [?language :lang/name    ?name]]

;=> [["Visi" #url "http://visi.io"]
;    ["Ioke" #url "http://ioke.org"]
;    ...]
```

- Repeating ?language indicates a join

Rules

```
[[(lang-anchor ?name ?url)
  [?language :lang/website ?url]
  [?language :lang/name    ?name]]]

[:find   ?name ?url
 :where (lang-anchor ?name ?url)]

;=> [["Visi" #url "http://visi.io"]
;    ["Ioke" #url "http://ioke.org"]
;    ...]
```

head
body

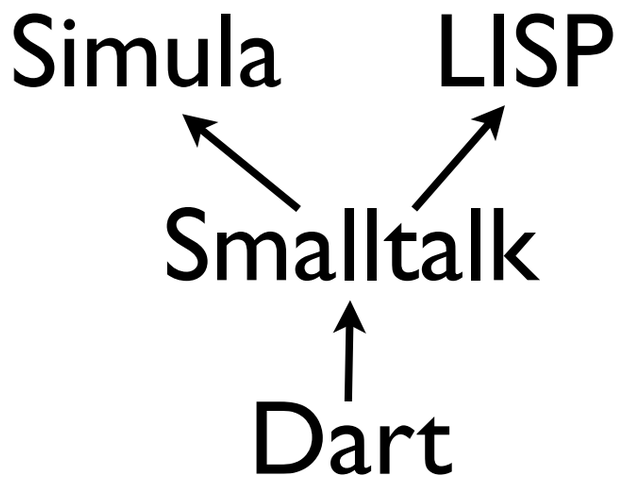
- All variables in the head, **must** appear in the body

Recursive rules

```
'[[(direct-influence ?old ?new)
   [?old :influenced ?new]]

  [(remote-influence ?old ?new)
   (direct-influence ?old ?new)]

  [(remote-influence ?old ?new)
   (direct-influence ?old ?intermediate)
   (remote-influence ?intermediate ?new)]]
```

Recursive rules

```
(q '[:find ?influence
     :in $ %
     :where
     [?new :lang "Dart"]
     (remote-influence ?old ?new)
     [?old :lang ?influence]]
   influences-db
   influces-rules)

;;=> #{["LISP"] ["Smalltalk"] ["Simula"]}
```



Datomic

Datalog plus

- No need for a database
- Time travel

Where's the DB?

```
public static void viewTable(Connection con) throws SQLException {
    Statement stmt = null;
    String query = "select COF_NAME, SUP_ID, " +
                   "PRICE, SALES, TOTAL from " +
                   dbName + ".COFFEE";

    try {
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            doPrint(coffeeName, supplierID, price, sales, total);
        }
    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

Where's the DB?

```
public static void viewTable(Connection con) throws SQLException {

        stmt = con.createStatement();

}
```

Where's the DB?

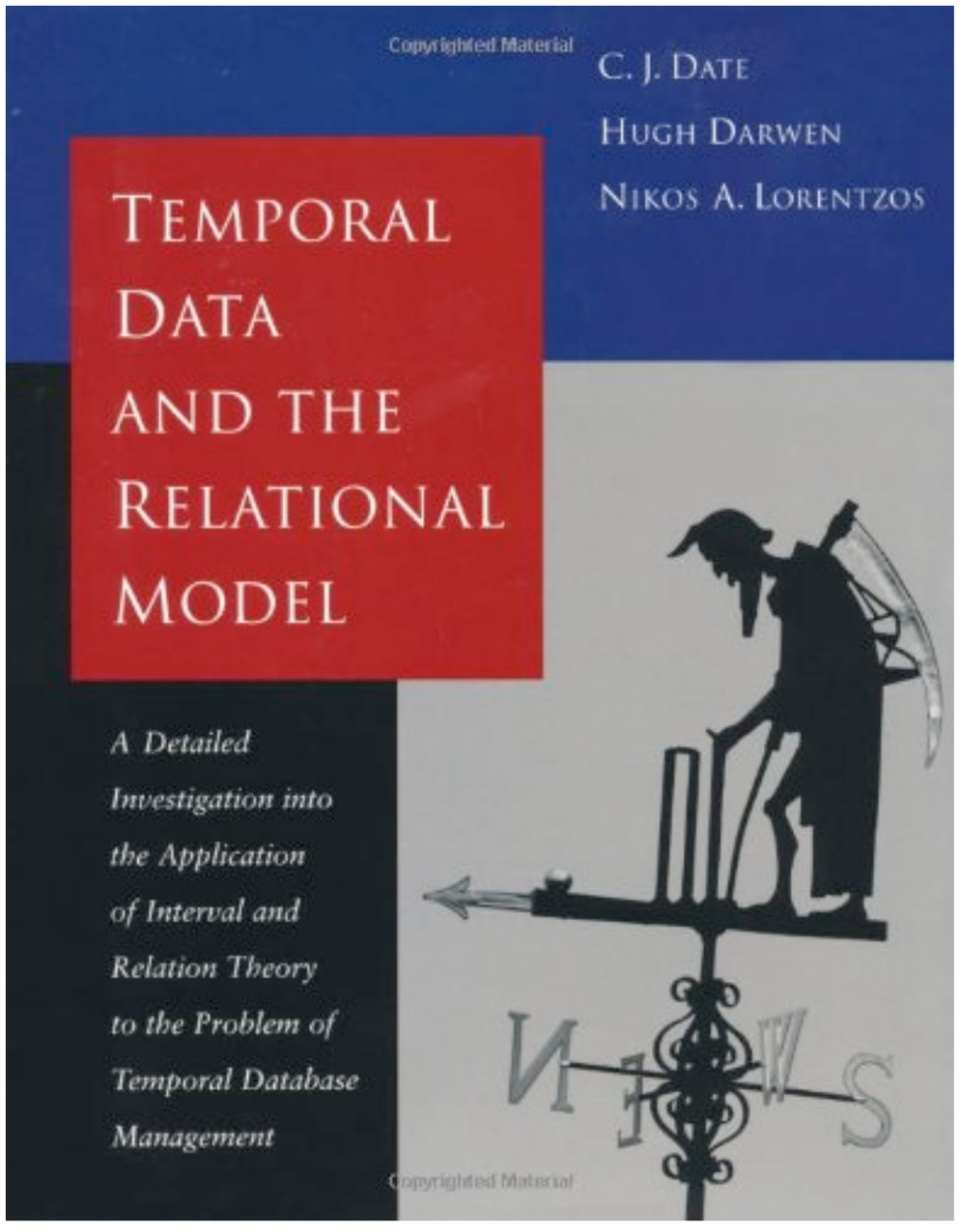
```
(datomic/q '[:find   ?name ?url
             :where (lang-anchor ?name ?url)]
           the-database                          ;; <- HINT: RIGHT HERE!
           the-rules)
```

Where's the DB?

```
(datomic/q '[:find   ?name ?url
             :where (lang-anchor ?name ?url)]
           [[-100 :language/name "Visi"]
            [-200 :language/name "Ioke"]
            [-3   :language/name "Frink"]
            [-4   :language/name "Roy"]
            [-100 :language/url  #url "http://visi.io"]
            [-200 :language/url  #url "http://ioke.org"]]
           the-rules)
```

now

- How do you keep a notion of time in a relational database?
- Ever write now()?



Always?

```
[john :favorite/food :pizza]
```

Time

```
[john :favorite/food :pizza <last week>]   ;; ASSERTION

[john :favorite/food :pho <now>]           ;; ANOTHER ASSERTION
```

- Total ordering of transactions
- Every datom retains a reference to its enclosing transaction
- Transactions are first-class entities, can have their own attributes

Time

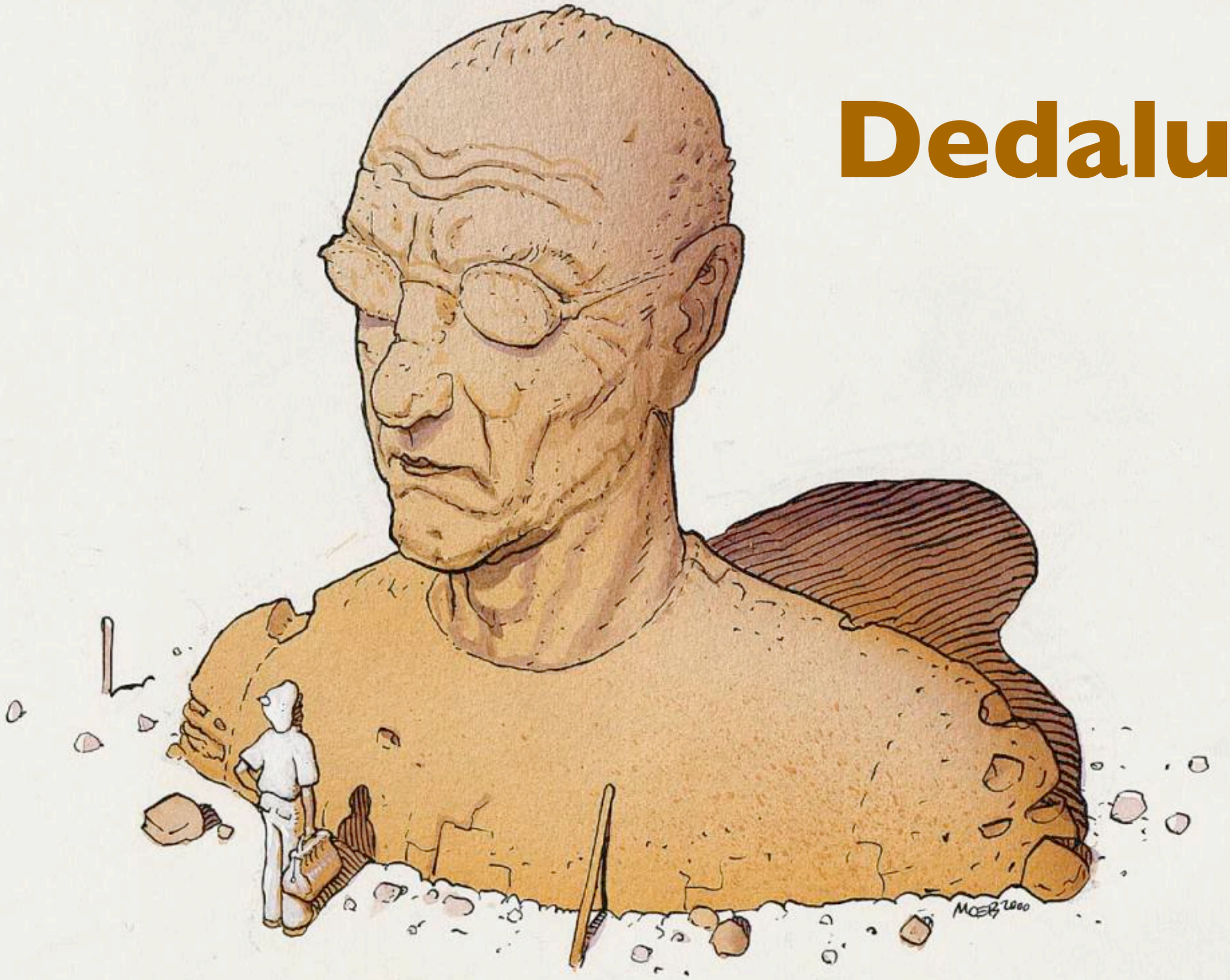
```
(q '[:find ?food
     :where [?person :person/name "John"]
            [?person :favorite/food ?food]]
   (d/since database #inst "2012-01-01T00:00:00.000-00:00"))
;=> [[:pho]]
```

- You can obtain the value of the db as-of, or since, a point in time, or both
  - without parameterizing your logic with a time argument
- You can also get the entire history of an entity!

Datomic is also

- Fully navigable lazy entity maps
- Query across databases
- Optimistic and pessimistic concurrency
- “Upsertting”
- http://datomic.com

Dedalus



Datalog plus

- Time
- State via rules

Time

```
loves(john, pizza, <last week>).
```

- Tick model
- Time is an element of the tuple

Deductive time

```
loves(john, pizza) :- order(john, Food).
```

- "Right now"
- All terms have the same time...

```
loves(john, pizza, <now>) :- order(john, Food, <now>),
<now> = <now>.
```

Inductive time

```
loves(john, pizza)@next :- order(john, Food).
```

- "Some other time"
- Next time tick

```
loves(john, pizza, T) :- order(john, Food, S).
successor(T, S).
```

Async time

```
loves(john, pizza)@async :- order(john, Food).
```

- Unreliable network

State

```
loves_pos(john, pizza).
```

- At time tick 0
- Facts

```
loves(john, pizza)@0;
```

Update

```
loves_pos(john, pho).
```

- At time tick 100
- Facts

```
loves(john, pizza)@0;
loves(john, pho)@100;
```

Update

```
loves_neg(john, pizza).
```

- At time tick 300
- Facts

```
loves(john, pizza)@0;
loves(john, pho)@100;
loves_neg(john, pizza)@300;
```

Mutable persistence rule

```
loves_pos(Person, Food) :- loves(Person, Food).
loves_pos(Person, Food)@next :-
      loves_pos(Person, Food),
     ~loves_neg(Person, Food).
```

- Update

```
loves(john, pizza)@<last year>;
loves(john, pho)@<now>;
loves_neg(john, pizza)@<now>;
```

Cascalog

Datalog plus

- Map/reduce processing
  - Order independence a win

Cascalog

```
(def people [["ben" 35]
             ["jerry" 41]])

(??- (<- [?name ?age]
         (people ?name ?age)
         (< ?age 40)))

;=> (["ben" 35])
```

Three stages

- Pre-aggregation
- Aggregation
- Post-aggregation

Pre-aggregation

- Joins aggregator functions
- Applies bindable functions and filters
- Dataflow-esque

Aggregation

- Partition result tuples along logic variables
- Execute aggregators for each logic var

Post-aggregation

- Execute the the remaining filters and functions on dependent aggregator output

Example

```
(<- [?word]
    ;; pre
    (sentence ?sentence)
    (split ?sentence :> ?word)
    ;; agg
    (c/count ?count)
    ;; post
    (> ?count 5))
```

Pre-aggregation

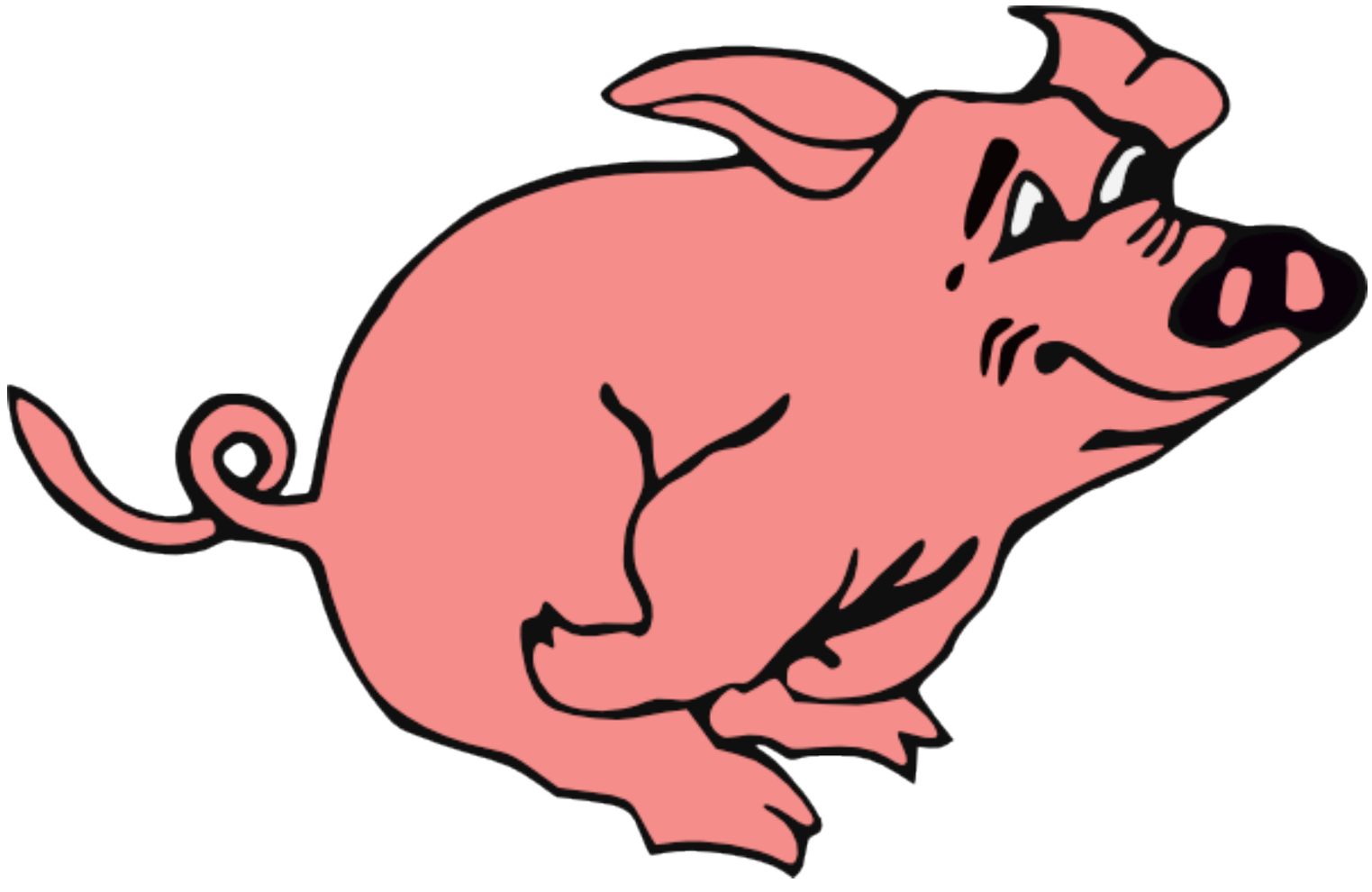
```
(<- [?word]
    ;; pre
    (sentence ?sentence)
    (split ?sentence :> ?word)
    ;; agg
    (c/count ?count)
    ;; post
    (> ?count 5))
```

Aggregation

```
(<- [?word]
    ;; pre
    (sentence ?sentence)
    (split ?sentence :> ?word)
    ;; agg
    (c/count ?count)
    ;; post
    (> ?count 5))
```

Post-aggregation

```
(<- [?word]
    ;; pre
    (sentence ?sentence)
    (split ?sentence :> ?word)
    ;; agg
    (c/count ?count)
    ;; post
    (> ?count 5))
```



Bacwn

Datalog plus

- Negation

Delicious!

Bacwn facts

```
(def mst3k-db
  (-> mst3k-schema
      (facts {:character/db.id 0 :character/name "Joel" :character/human? true}
             {:character/db.id 1 :character/name "Crow" :character/human? false}
             {:character/db.id 2 :character/name "TV's Frank" :character/human? true}
             {:location/db.id  0 :location/character 0 :location/name "SoL"}
             {:location/db.id  0 :location/character 1 :location/name "SoL"}
             {:location/db.id  1 :location/character 2 :location/name "Gizmonics"})))
```

Bacwn rules!

```
(def locate-rule
  (rules-set
   (<- (:stationed-at :location/name ?loc-name :character/name ?char-name)
       (:location  :name ?loc-name :character ?char)
       (:character :character/db.id ?char :name ?char-name))))
```

Bacwn query

```
(q (?- :stationed-at :location/name '??loc :character/name ?char-name)
   mst3k-db
   locate-rule
   {'??loc "SoL"})

;;=> ({:location/name "SoL", :character/name "Crow"}
;;    {:location/name "SoL", :character/name "Joel"})
```

Bacwn negation

```
(def non-human-locate-rule
  (rules-set
   (<- (:stationed-at :location/name ?loc-name :character/name ?char-name)
       (:location  :name ?loc-name :character ?char)
       (:character :character/db.id ?char :name ?char-name)
       (not! :character :character/db.id ?char :human? true))))  ;; ADDED NEGATION
```

Bacwn negation

```
(q (?- :stationed-at :location/name '??loc :character/name ?char-name)
   mst3k-db
   non-human-locate-rule
   {'??loc "SoL"})

;;=> ({:location/name "SoL", :character/name "Crow"})
```

- Find all non-humans at a location

Open questions

- Query plans
- Optimizations

Query plans

- The holy grail of DBs is that they **do the right thing**
- Query plans
- No runtime guarantees

Gaming the query

- /*+ Hinting */
- Prolog - Ordering for termination
- Datalog - Ordering for speed

Gaming the query

- /*+ Hinting */
- Prolog - Ordering for termination
- Datalog - Ordering for speed

Wut

```
[:find ?desc
 :where
 [?root :num 100]
 (descendant ?root ?desc)
 [?desc :num 4]]
```

- Find all descendants of root number #100 with value = 4
- SLLLLLLOOOOOOOWWWW

Wat

```
[:find ?desc
 :where
 [?root :num 100]
 (descendant ?root ?desc)
 [?desc :num ?n]                    ;; INDIRECTION
 [(= ?n 4)]]                        ;; CHECK 4-NESS
```

- Find all descendants of root number #100 with value = 4
- FAST!!

Wat

```
[:find ?desc
 :where
 [?root :num 100]
 [?desc :num 4]
 (descendant ?root ?desc)]
```

- Find all descendants of root number #100 with value = 4
- Most-bound

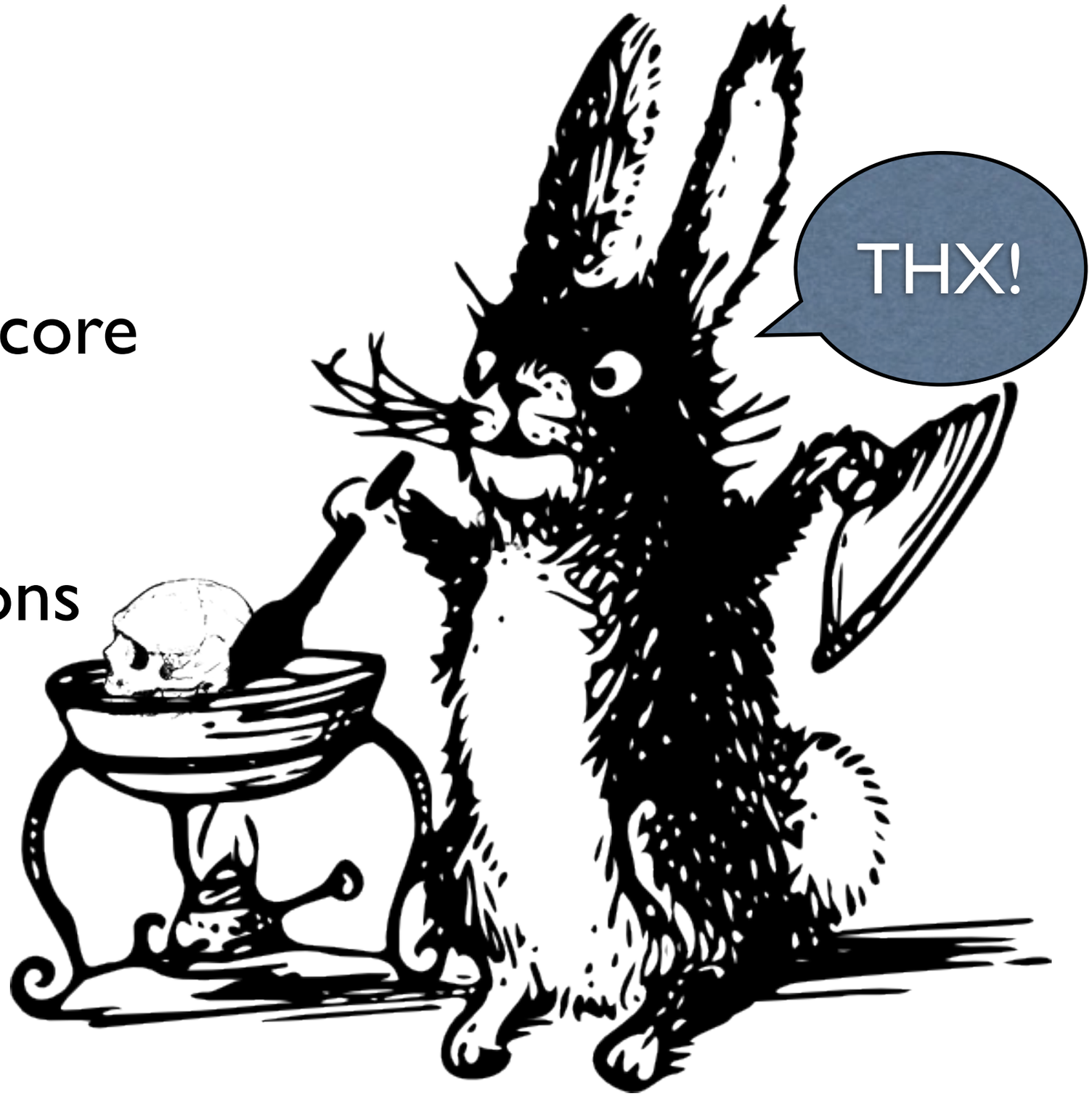
Order doesn't matter

Except when it does

Pluggable Optimizers

- Order agnostic is a win
- Some orders are better than others
- Plug in your own optimizer
  - That knows your data
- Will not affect other Datalog engine optimization techniques

- Rich Hickey, Stu Halloway, Clojure/core
- Clojure/dev
- Manning Publications
- The fam
- You



double-secret slides

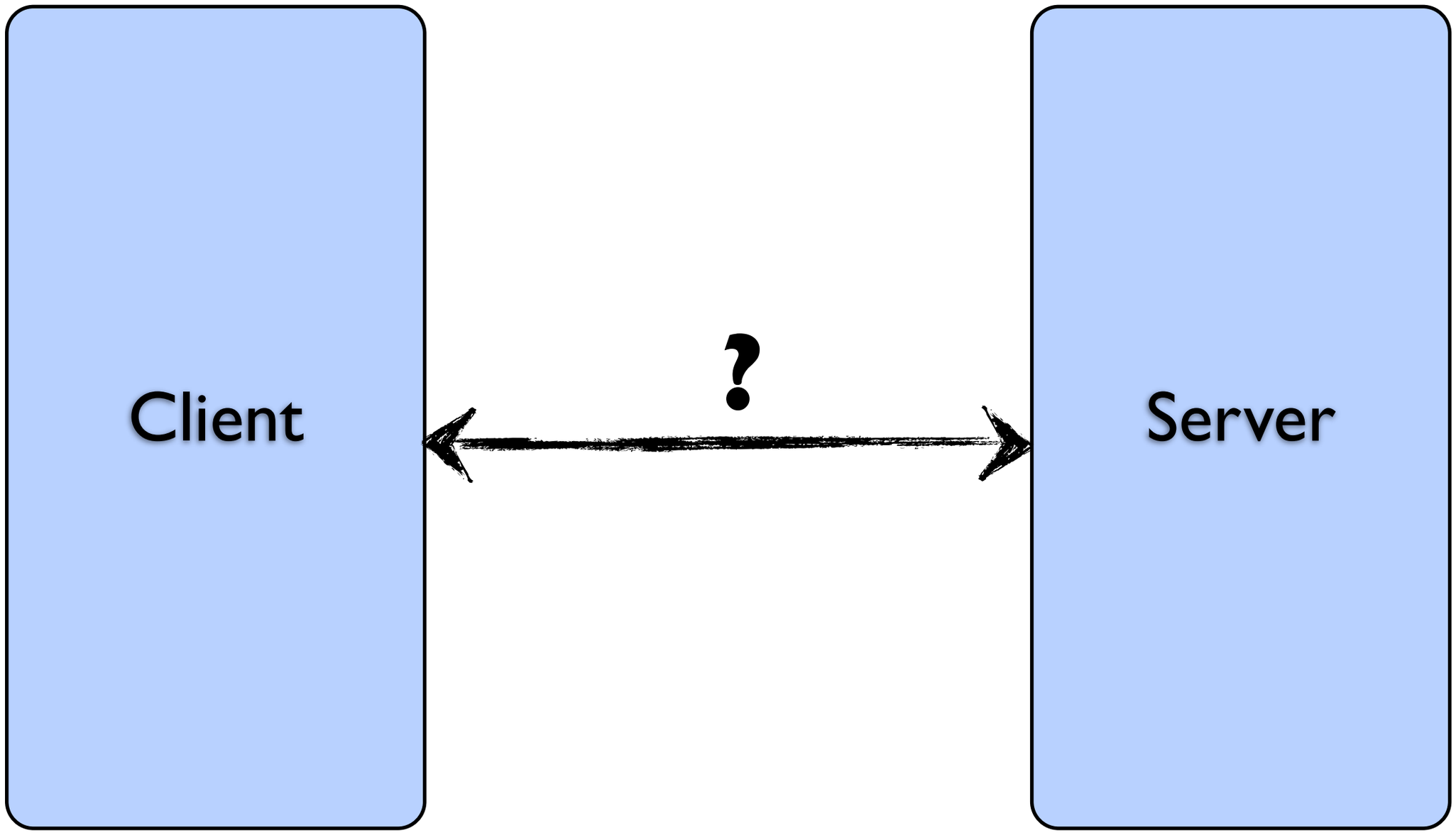
Data as Data.

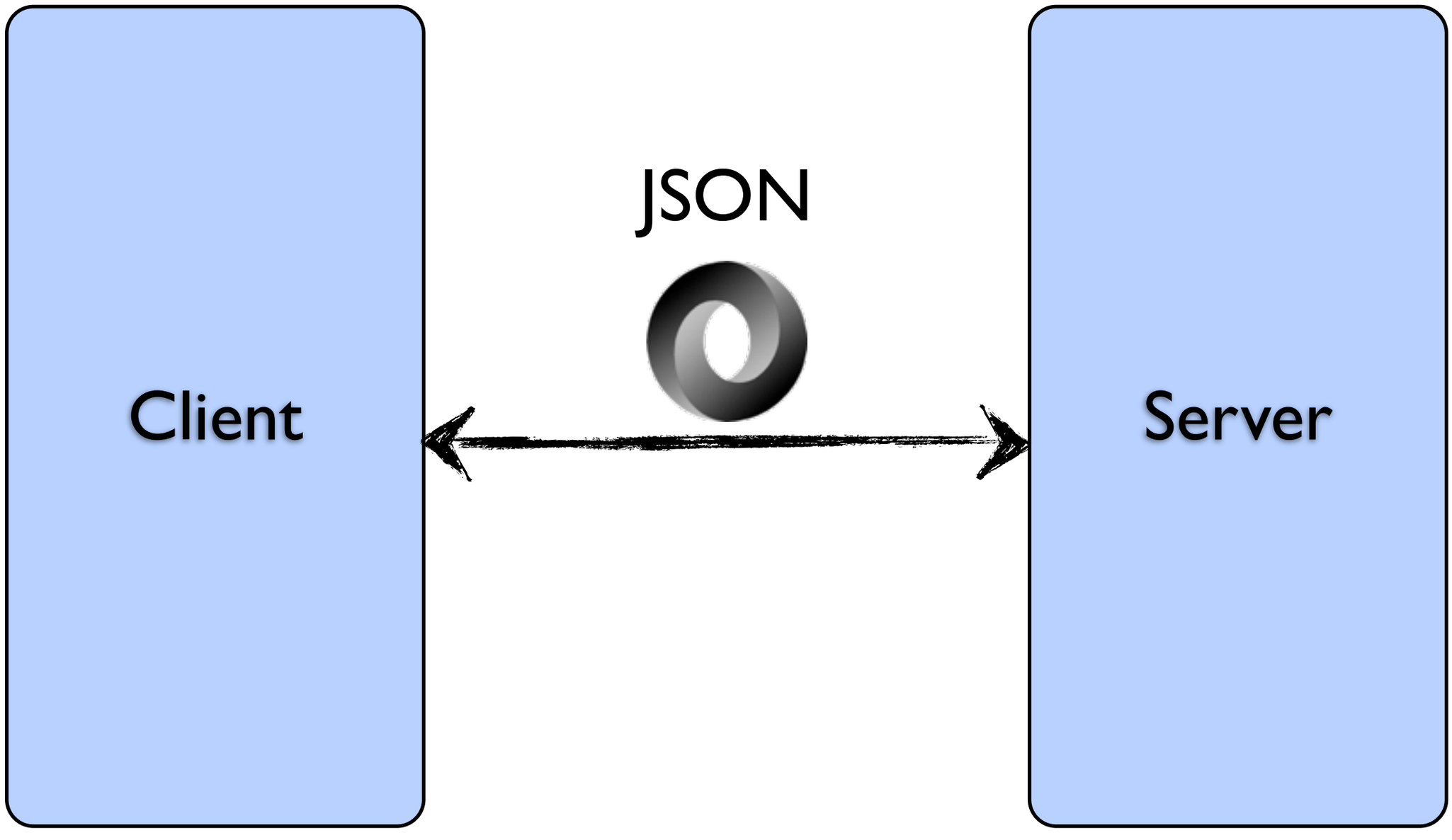
Scalar

type	in	example
string	_	"hi"
int	_	42
float	_	3.14159
char	Scheme	#\Z
boolean	Scheme	#t
nullity	Scala	null
nothingness	JavaScript	undefined
ratio	Clojure	22/7
symbol	Ruby	:hi
keyword	Clojure	:hi

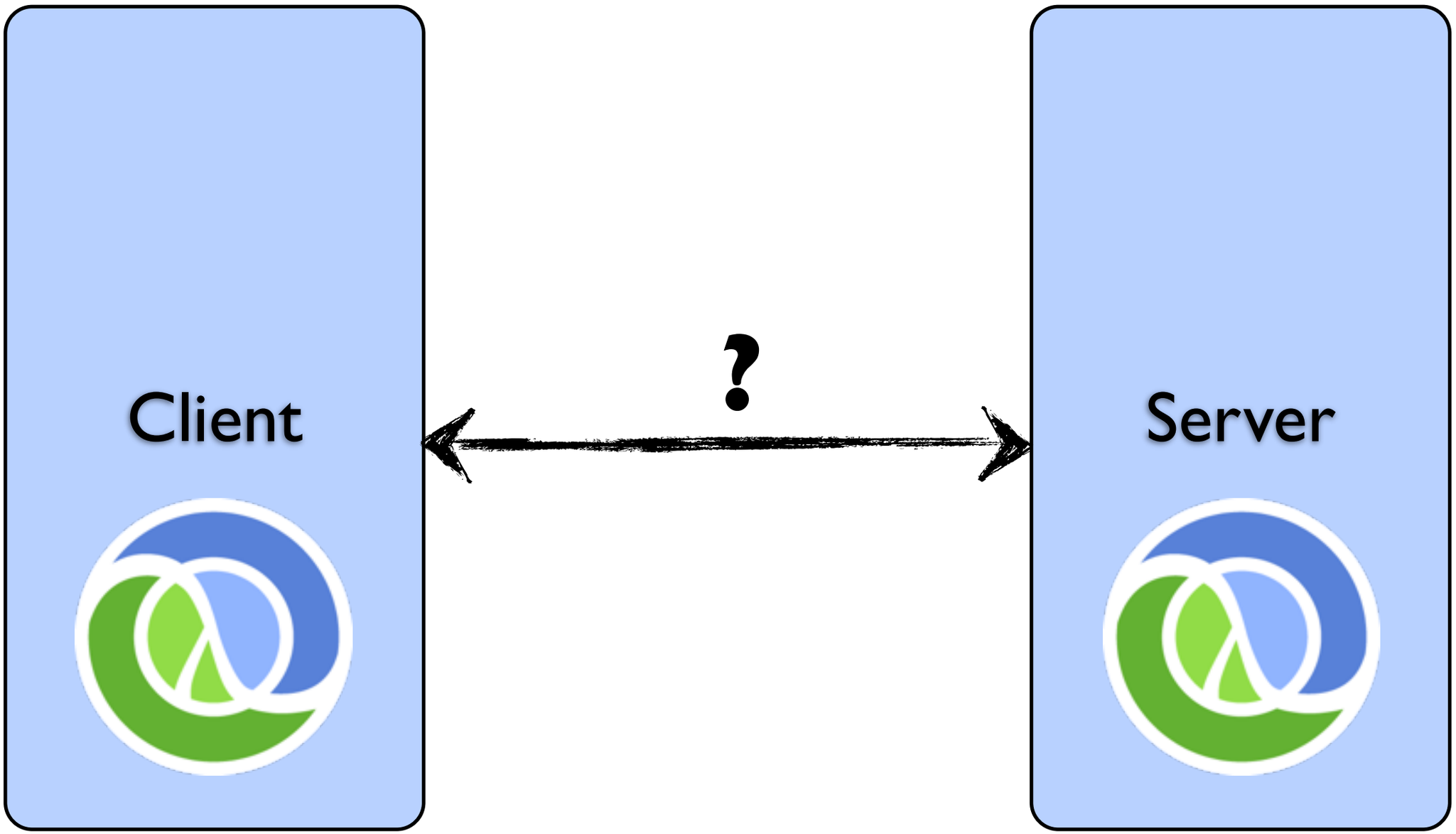
Scalar

type	in	example
list	Haskell	[1,2,3,4]
tuple	Erlang	{1,2,3,4}
vector	Clojure	[1 2 3 4]
dict	Python	{'a': 1, 'b': 2}
set	Clojure	#{1 2 3}
array	Pike	({ 1, 2, 3 })
table	J	2 2 $ 1 2 3 4
queue	Clojure	#queue [1 2 3]

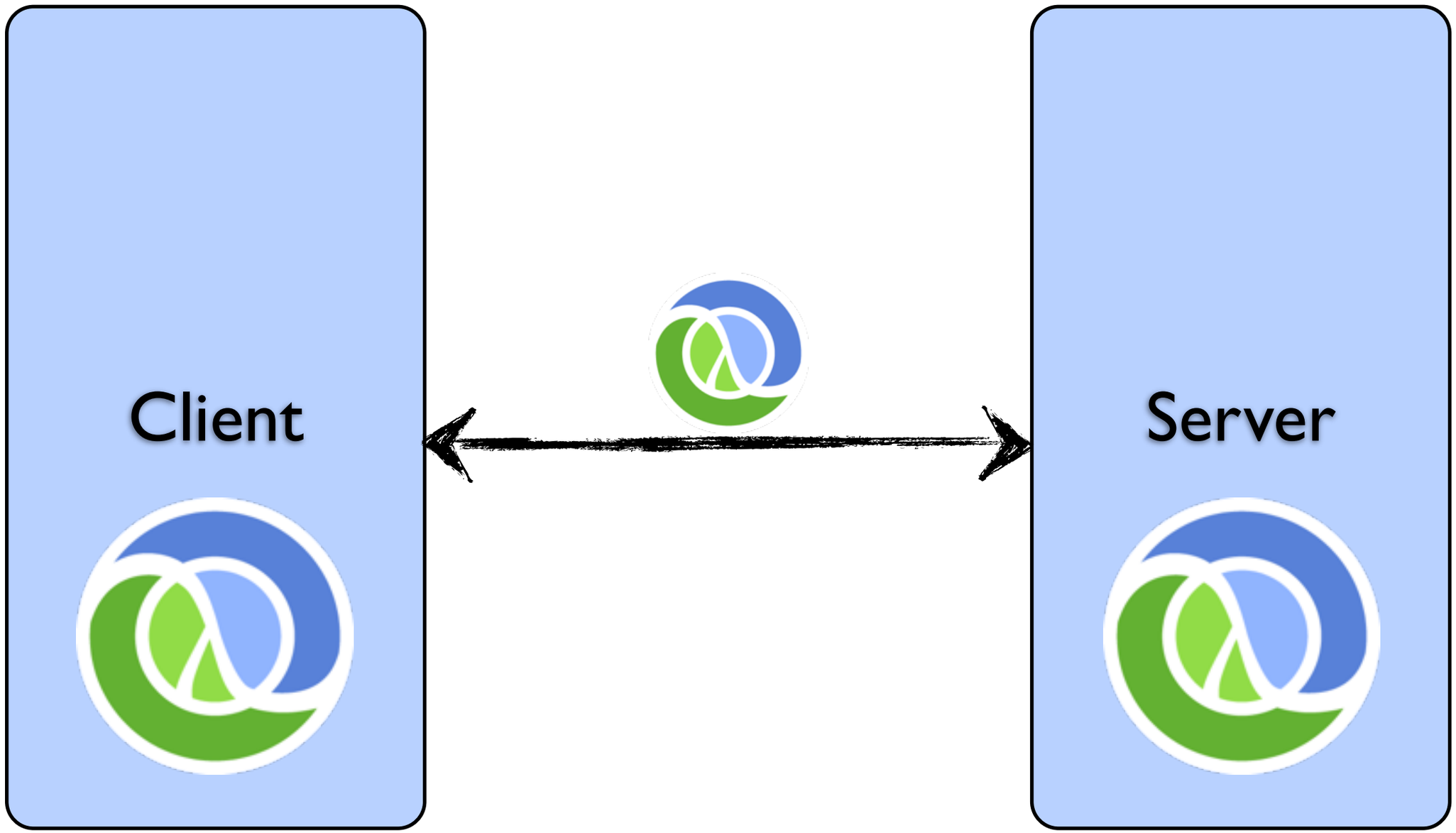

Client
?
Server



JSON as Data

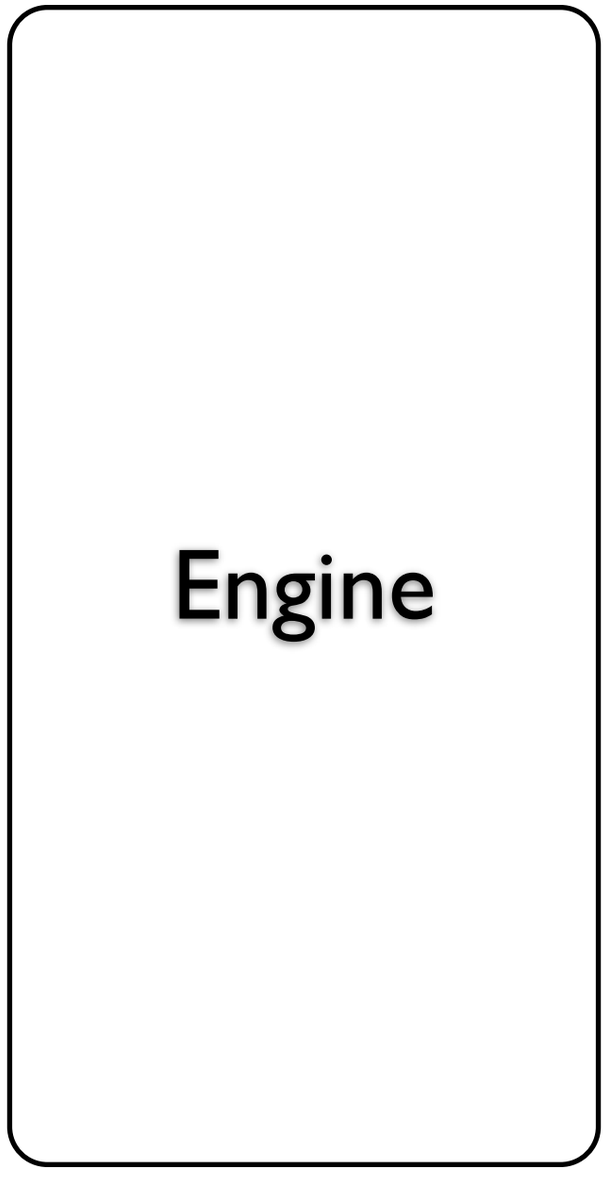


Clojure as Code



Clojure as Data

Data as Code.


Engine

Specification

Engine

Specification

Engine

Specification

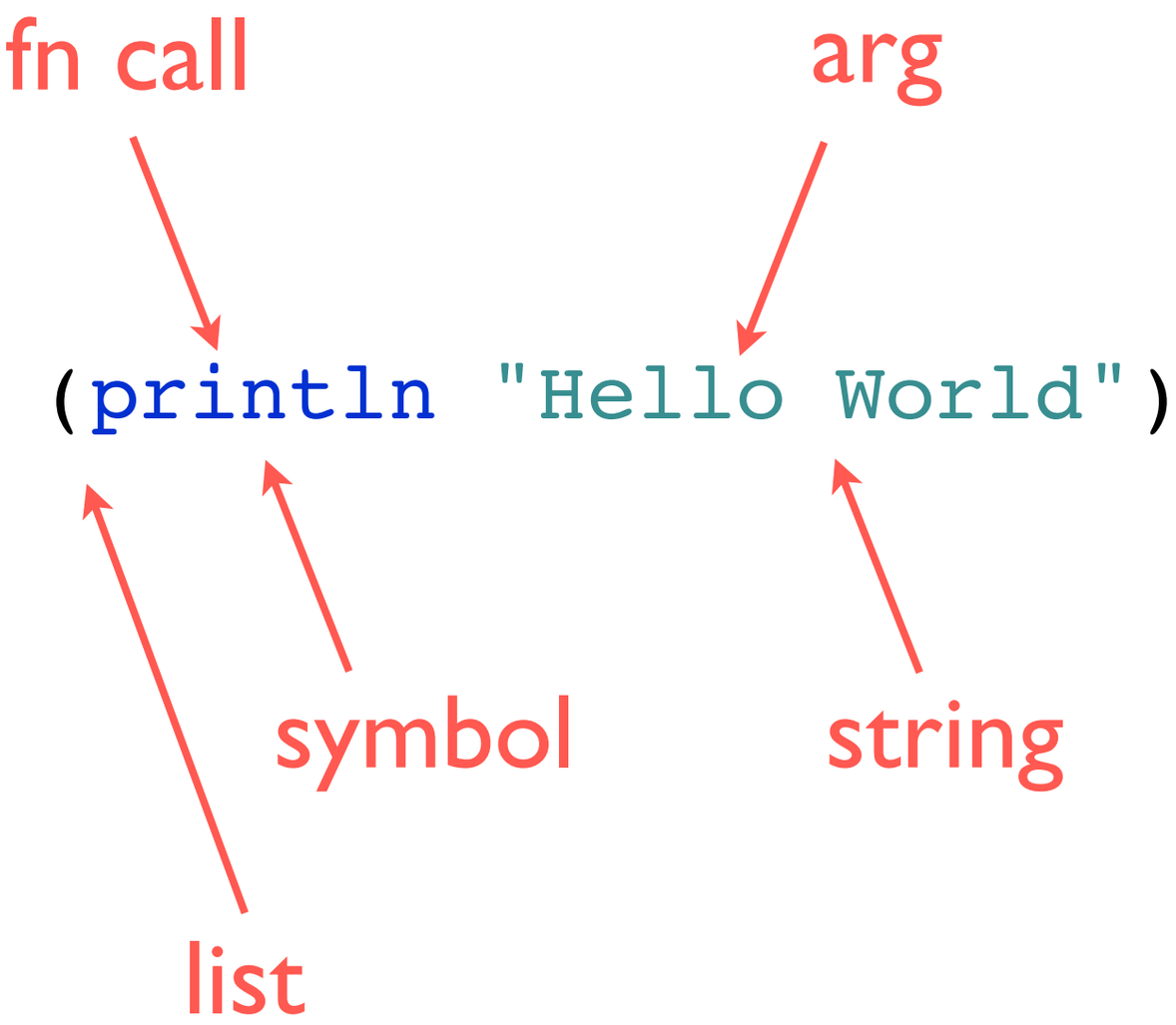
Engine

I'm really doing something cool now

Code as Data.

Function call

semantics:



structure:

Function definition

define a fn

fn name

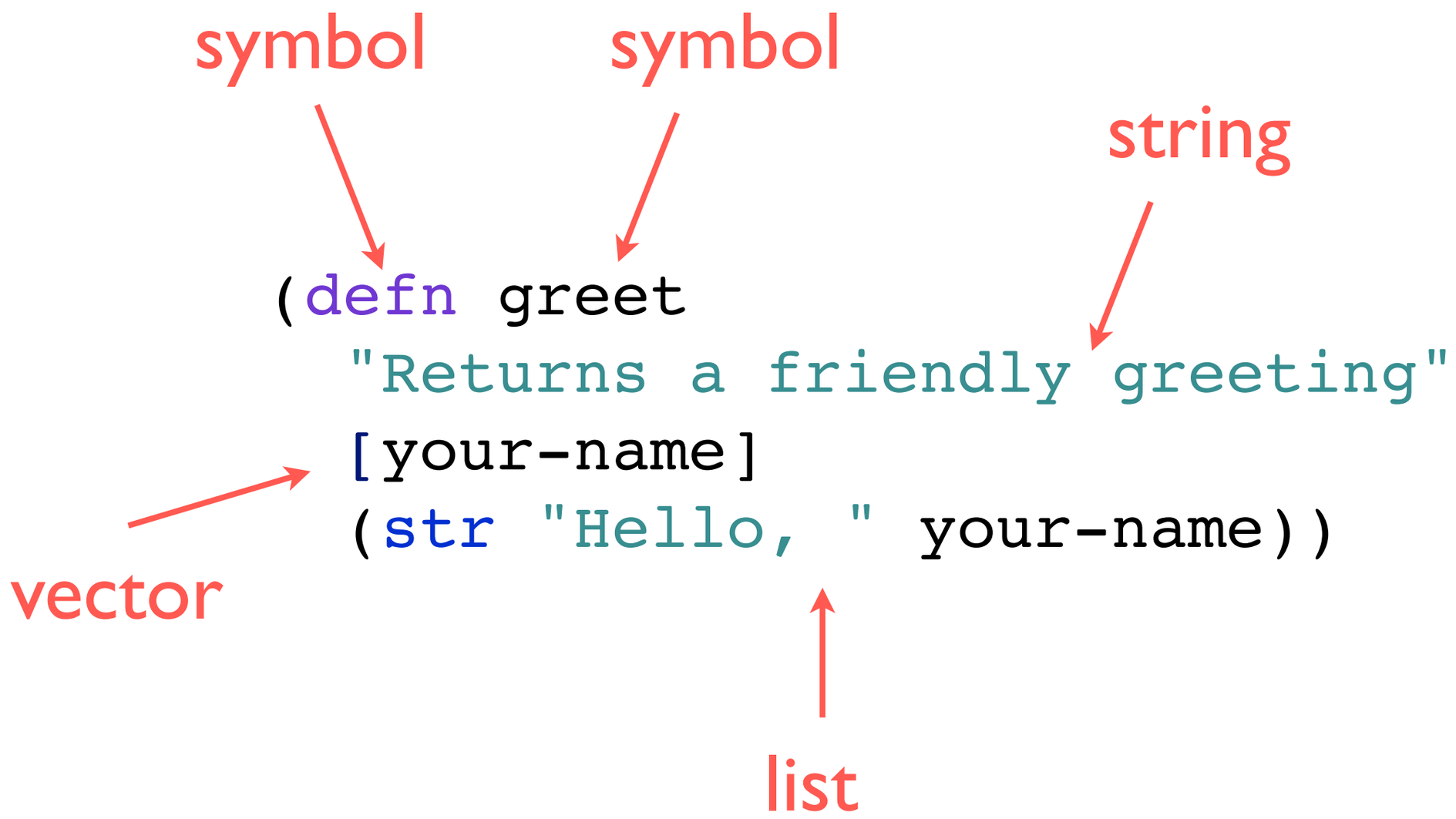
docstring

```
(defn greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
```

arguments

fn body

It's all data


symbol
symbol
string
(defn greet
"Returns a friendly greeting"
[your-name]
(str "Hello, " your-name))
vector
list