Get Logical with Datalog

@stuarthalloway

In computer science, **declarative programming** is a programming paradigm that expresses the logic of a computation without describing its control flow.[1] Many languages applying this style attempt to minimize or eliminate side effects by describing *what* the program should accomplish, rather than describing *how* to go about accomplishing it.

http://en.wikipedia.org/wiki/Declarative_programming

Why Logic?

"The promise of logic
programming is that programs
can be written relationally,
without distinguishing between
input and output arguments..."

http://pqdtopen.proquest.com/#abstract?dispub=3380156

Why Datalog?

"Unfortunately, writing relational programs is difficult."

http://pqdtopen.proquest.com/#abstract?dispub=3380156

Datomic

Query Anatomy

```
q([:find ...
   :in ...
   :where ...],
  input1,
  ...
  inputN);
```

Query Anatomy

constraints

```
q([:find ...
   :in ...
   :where ...],
  input1,
  ...
  inputN);
```

Query Anatomy

```
q([:find ...
   :in ...
   :where ...],
  input1,
  ...
  inputN);
```

inputs

Query Anatomy

```
q([:find ...
   :in ...
   :where ...],
  input1,
  ...
  inputN);
```

names for inputs

Query Anatomy

```
q([:find ...
   :in ...
   :where ...],
  input1,
  ...
  inputN);
```

variables to return

Variables

?customer

?product

?orderId

?email

Constants

```
42                                  :email

                  "john"

:order/id

                          #inst "2012-02-29"
```

Keywords

```
42                                  :email

                                  "john"

:order/id

                                  #inst "2012-02-29"
```

Namespaces

```
42                                  :email

                                    "john"

:order/id

                                    #inst "2012-02-29"
```

Extensible Reader

:email

42

“john”

:order/id

#inst "2012-02-29"

Example Database

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

Data Pattern

Constrains the results returned,
binds variables

```
[?customer :email ?email]
```

Data Pattern

Constrains the results returned,
binds variables

```
[?customer :email ?email]
```

Data Pattern

Constrains the results returned,
binds variables

constant

```
[?customer :email ?email]
```

Data Pattern

Constrains the results returned,
binds variables

variable variable

```
[?customer :email ?email]
```

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[?customer :email ?email]
```

Constants Anywhere

"Find a particular customer's email"

```
[42 :email ?email]
```

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[42 :email ?email]
```

Variables Anywhere

"What attributes does
customer 42 have?

```
[42 ?attribute]
```

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[42 ?attribute]
```

Variables Anywhere

"What attributes and values does customer 42 have?

```
[42 ?attribute ?value]
```

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[42 ?attribute ?value]
```

Where Clause

data pattern

```
[:find ?customer
 :where [?customer :email]]
```

Find Clause

variable to return

```
[:find ?customer
 :where [?customer :email]]
```

Implicit Join

"Find all the customers who
have placed orders."

```
[:find ?customer
 :where [?customer :email]
        [?customer :orders]]
```

API

```
import static datomic.Peer.q;

q("[:find ?customer
  :where [?customer :id]
         [?customer :orders]]",
db);
```

q

```
import static datomic.Peer.q;

q("[:find ?customer
  :where [?customer :id]
         [?customer :orders]]",
db);
```

Query

```
import static datomic.Peer.q;


q("[:find ?customer
  :where [?customer :id]
         [?customer :orders]]",
db);
```

Input(s)

```
import static datomic.Peer.q;


q("[:find ?customer
  :where [?customer :id]
         [?customer :orders]]",
db);
```

In Clause

Names inputs so you can refer to them elsewhere in the query

```
:in $database ?email
```

Parameterized Query

"Find a customer by email."

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
  db,
  "jdoe@example.com");
```

First Input

"Find a customer by email."

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
  db,
  "jdoe@example.com");
```

Second Input

"Find a customer by email."

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
  db,
  "jdoe@example.com");
```

Verbose?

"Find a customer by email."

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
  db,
  "jdoe@example.com");
```

Shortest Name Possible

"Find a customer by email."

```
q([:find ?customer
   :in $ ?email
   :where [$ ?customer :email ?email]],
  db,
  "jdoe@example.com");
```

Elide $ in Where

"Find a customer by email."

```
q([:find ?customer
   :in $ ?email
   :where [ ?customer :email ?email]],
  db,
  "jdoe@example.com");
```

Predicates

Functional constraints that can appear in a :where clause

```
[(< 50 ?price)]
```

Adding a Predicate

"Find the expensive items"

```
[:find ?item
 :where [?item :item/price ?price]
        [(< 50 ?price)]]
```

Functions

Take bound variables as inputs
and bind variables with output

```
[(shipping ?zip ?weight) ?cost]
```

Function Args

```
[(shipping ?zip ?weight) ?cost]
```

bound inputs

Function Returns

```
[(shipping ?zip ?weight) ?cost]
```

bind return values

Calling a Function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

Calling a Function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

navigate from customer to zip

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

Calling a Function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

get product facts needed *during query*

Calling a Function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

call web service to bind shipCost

BYO Functions

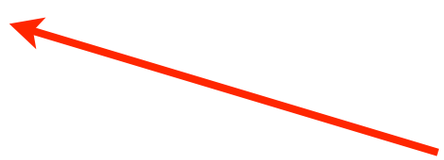
Functions can be plain JVM code.

```
public class Shipping {
  public static BigDecimal
  estimate(String zip1, int pounds);
}
```

Calling a Function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```



constrain price

Calling a Function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

return customer, product pairs

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

Why

Why Clojure?

- Data
  - good literals
  - immutable data
  - extensible reader
- Platform
  - extensibility
  - performance
- Lisp

Why Datalog?

- Equivalent to Relational Model + Recursion
- Better fit than Prolog for query
  - No clause order dependency
  - Guaranteed termination
- Pattern-matching style easy to learn

Problem: Rectangles

"People can belong to multiple clubs"

- join table
- person table
- club table

- id key in person table
- person key in join table
- club key in join table
- id key in club table

Structural Navigation

Structural Rigidity

Solution: Universal Relation

"People can belong to multiple clubs"

```
[?person :club ?club]
```

Did You Ever Want To...

- Make a column name variable?
- Make a table name variable?
- Treat metadata as first-class data?

First-Class Attributes

```
[?person ?attr ?value]
```

attribute slot
isn't special

Schema Made of Ordinary Data

```
[?e :db/valueType]
```

find all
attributes

Problem: Ambient DB

```
SELECT ID FROM CUSTOMERS;
```

in what db?

Solution: Explicit DB

```
q("[:find ?customer
    :where [?customer :id]
           [?customer :orders]]",
db);
```

in this db!

Benefit: Query Params

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
db,
"jdoe@example.com");
```

parameterized query
is not a separate feature

Benefit: BYO Data

What system properties are available?

```
(q '[:find ?k
     :in [[?k]]]
   (System/getProperties))
```

Benefit: BYO Data

What system properties are available?

bind first element of each tuple in a relation

```
(q '[:find ?k
     :in [[?k]]]
   (System/getProperties))
```

Binding Patterns

Pattern	Binds	Example Input	Binds ?a To
?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

BYO Data

Which system properties are path-related?

```
(q '[:find ?v
     :in [[?k ?v]]
     :where [(.endsWith ?k "path")]]
   (System/getProperties))
```

BYO Data

What path elements are mentioned in system properties?

```
(q '[:find ?pathElem
     :in [[?k ?v]]
     :where [(.endsWith ?k "path")]
            [(.split ?v ":") [?pathElem ...]]]
   (System/getProperties))
```

BYO Data

What JAR files are in my system property paths?

```
(q '[:find ?pathElem
     :in [[?k ?v]]
     :where [(.endsWith ?k "path")]
            [(.split ?v ":") [?pathElem ...]]
            [(.endsWith ?pathElem ".jar")]]
   (System/getProperties))
```

Benefit: Time Travel

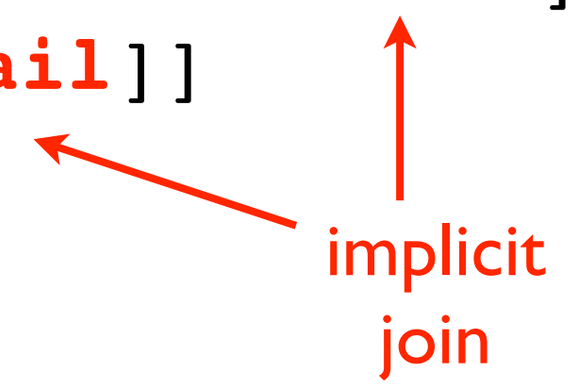
```
q("[:find ?customer
    :where [?customer :id]
           [?customer :orders]]",
db.asOf(lastMonth));
```

Benefit: Join Across DBs

"Find me the customers who are
also employees."

```
q(query, custDb, empDb);
```

```
[:find ?customer ?email
 :in $cust $emp
 :where [$cust ?customer :email ?email]
        [$emp _ :email ?email]]
```



Benefit: Join Across DBs

"Find me the customers who are
also employees."

```
q(query, custDb, empDb);
```

```
[:find ?customer ?email
 :in $cust $emp
 :where [$cust ?customer :email ?email]
        [$emp _ :email ?email]]
```

data patterns can be led
by database names

Problem: Better Views

- Good
  - abstraction
  - relational
- Bad
  - over there
  - rectangular
  - tool/language choices

Solution: Rules

"Products are related if they have a common category."

```
[(relatedProduct ?p1 ?p2)
 [?p1 :category ?c]
 [?p2 :category ?c]
 [(!= ?p1 ?p2)]]
```

Rule Head

"Products are related if they have a common category."

this is true...

```
[(relatedProduct ?p1 ?p2)
 [?p1 :category ?c]
 [?p2 :category ?c]
 [(!= ?p1 ?p2)]]
```

Rule Body

"Products are related if they have a common category."

```
[(relatedProduct ?p1 ?p2)
 [?p1 :category ?c]
 [?p2 :category ?c]
 [(!= ?p1 ?p2)]]
```

...if all these are true

Using Rules

"Find all products related to expensive chocolate."

```
q("[:find ?p2
    :in $ %
    :where (expensiveChocolate p1)
           (relatedProduct p1 p2)",
  db,
  rules)
```

rules are a kind of input

Using Rules

"Find all products related to expensive chocolate."

```
q("[:find ?p2
    :in $ %
    :where (expensiveChocolate p1)
           (relatedProduct p1 p2)",
  db,
  rules)
```

rule names begin with %

Using Rules

"Find all products related to expensive chocolate."

```
q("[:find ?p2
    :in $ %
    :where (expensiveChocolate p1)
           (relatedProduct p1 p2)",
  db,
  rules)
```

rule patterns can appear in :where clause

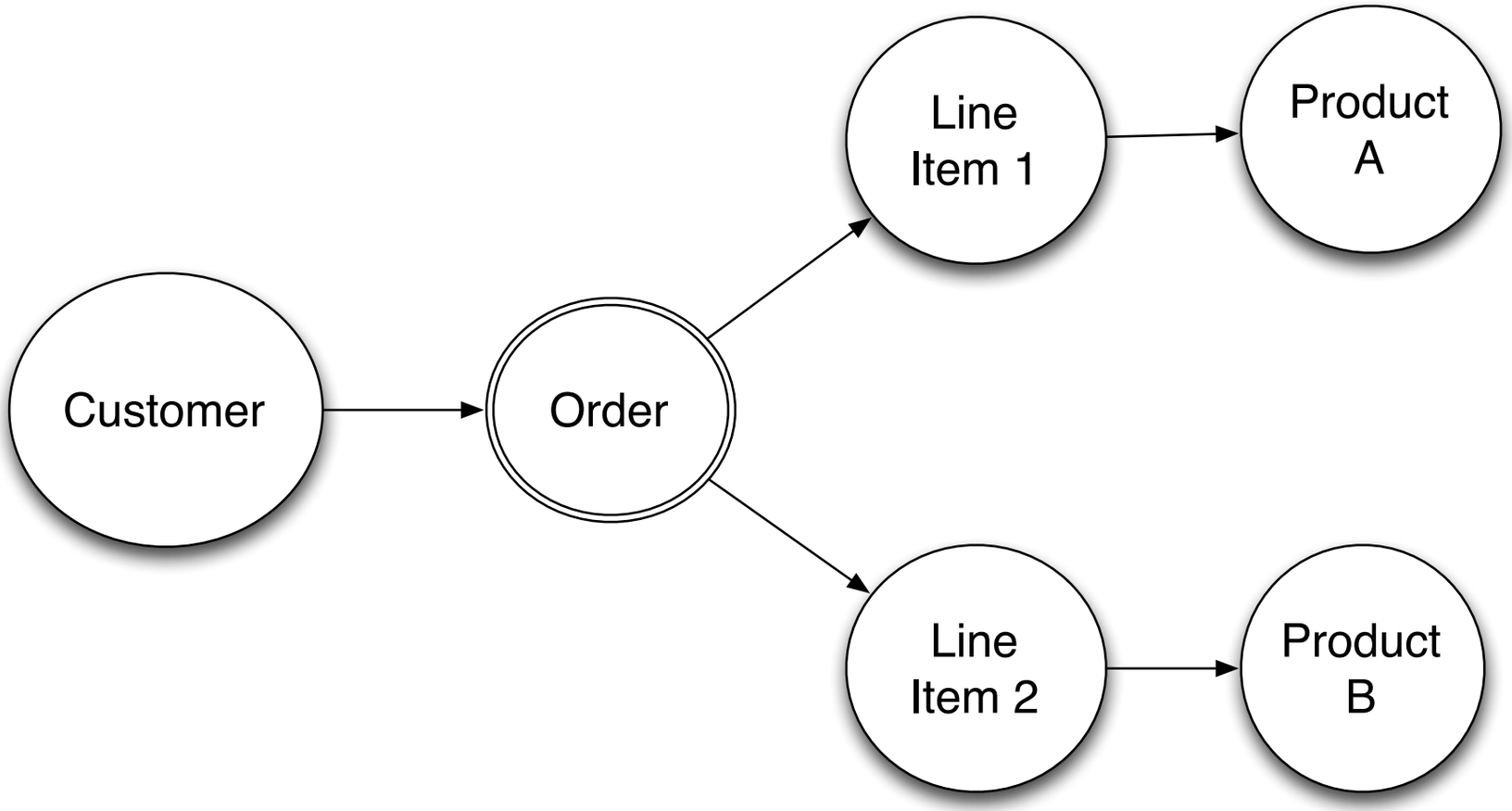
Implicit Or

"Products are related if they have the same category, or they have appeared in the same order."

```
[[(relatedProduct ?p1 ?p2)
  [?p1 :category ?c]
  [?p2 :category ?c]
  [(!= ?p1 ?p2)]]
 [(relatedProduct ?p1 ?p2)
  [?o :order/item ?item1]
  [?item1 :order/product ?p1]
  [?o :order/item ?item2]
  [?item2 :order/product ?p2]
  [(!= ?p1 ?p2)]]]
```

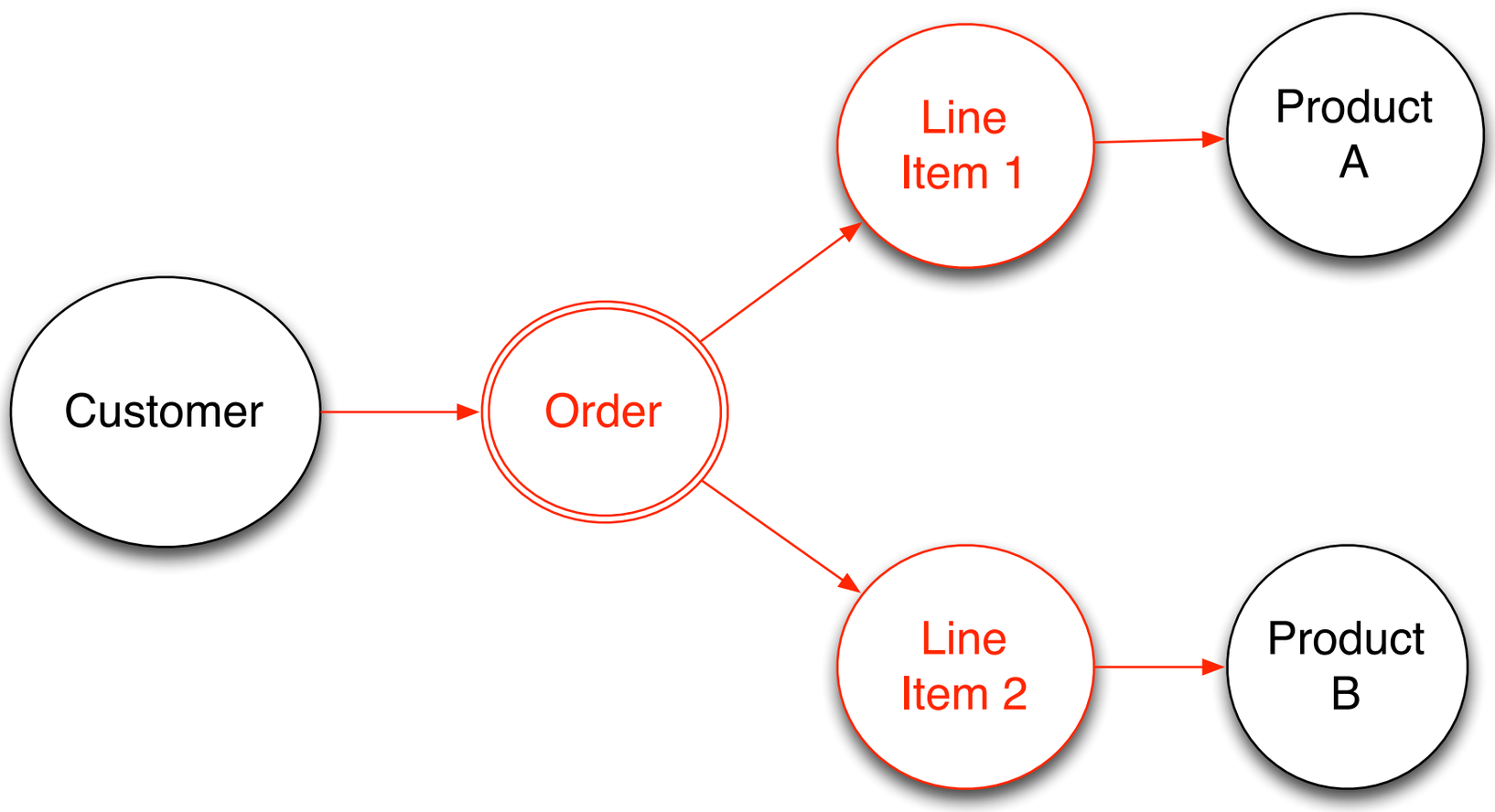
Problem: Extent

Get “the whole order”.



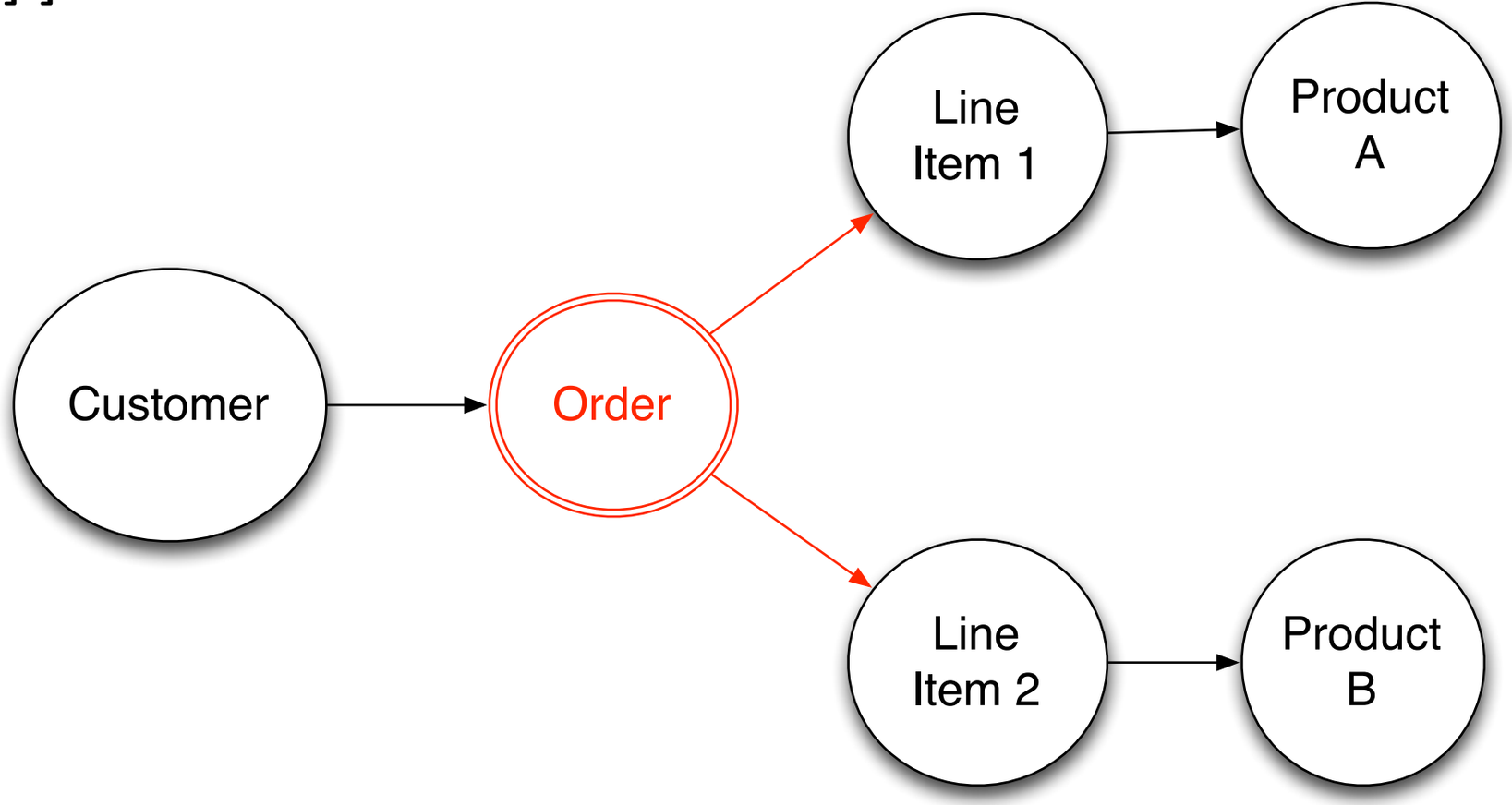
Problem: Extent

Get “the whole order”.



Find Values :x References

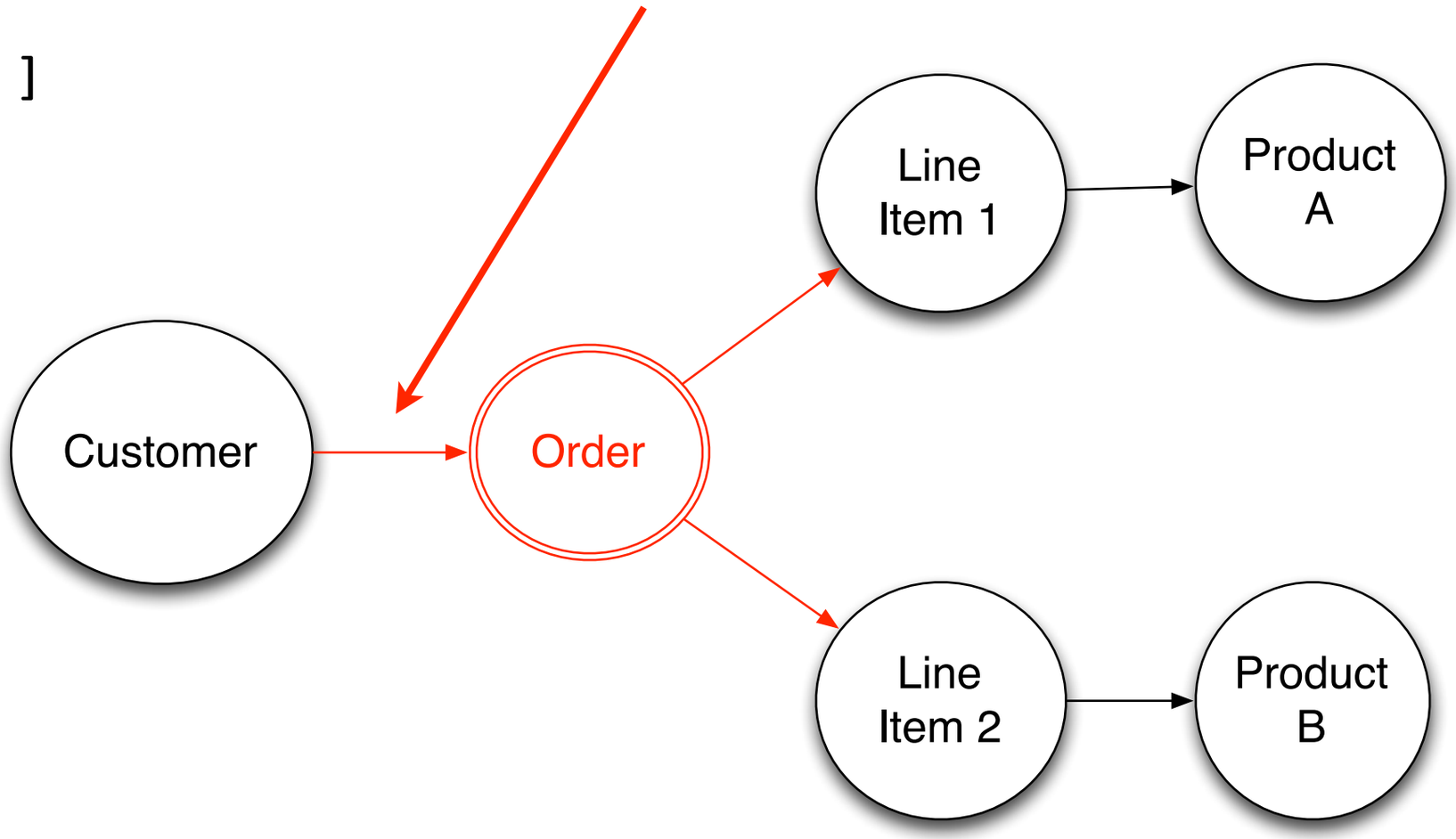
```
[(extent ?x ?e ?a ?v)
 (?e ?a ?v)
 (?x ?a ?v)
 [(= ?e ?x)]]
```



Finds Entities Referencing :x

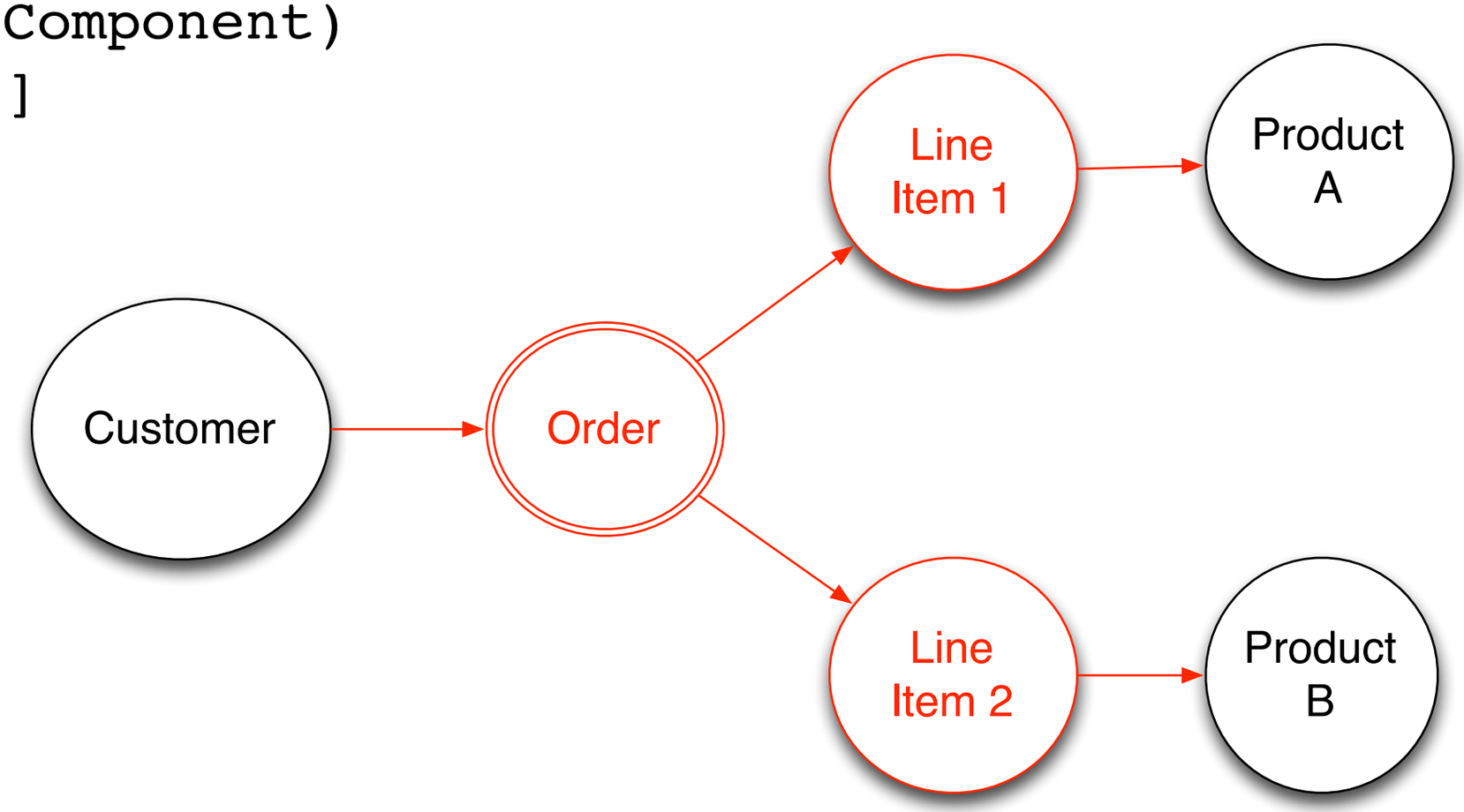
```
[(extent ?x ?e ?a ?v)
 (?e ?a ?v)
 (?e ?a ?x)
 [(= ?v ?x)]]
```

matches ref from customer,
not customer itself



Recurse Component Attributes

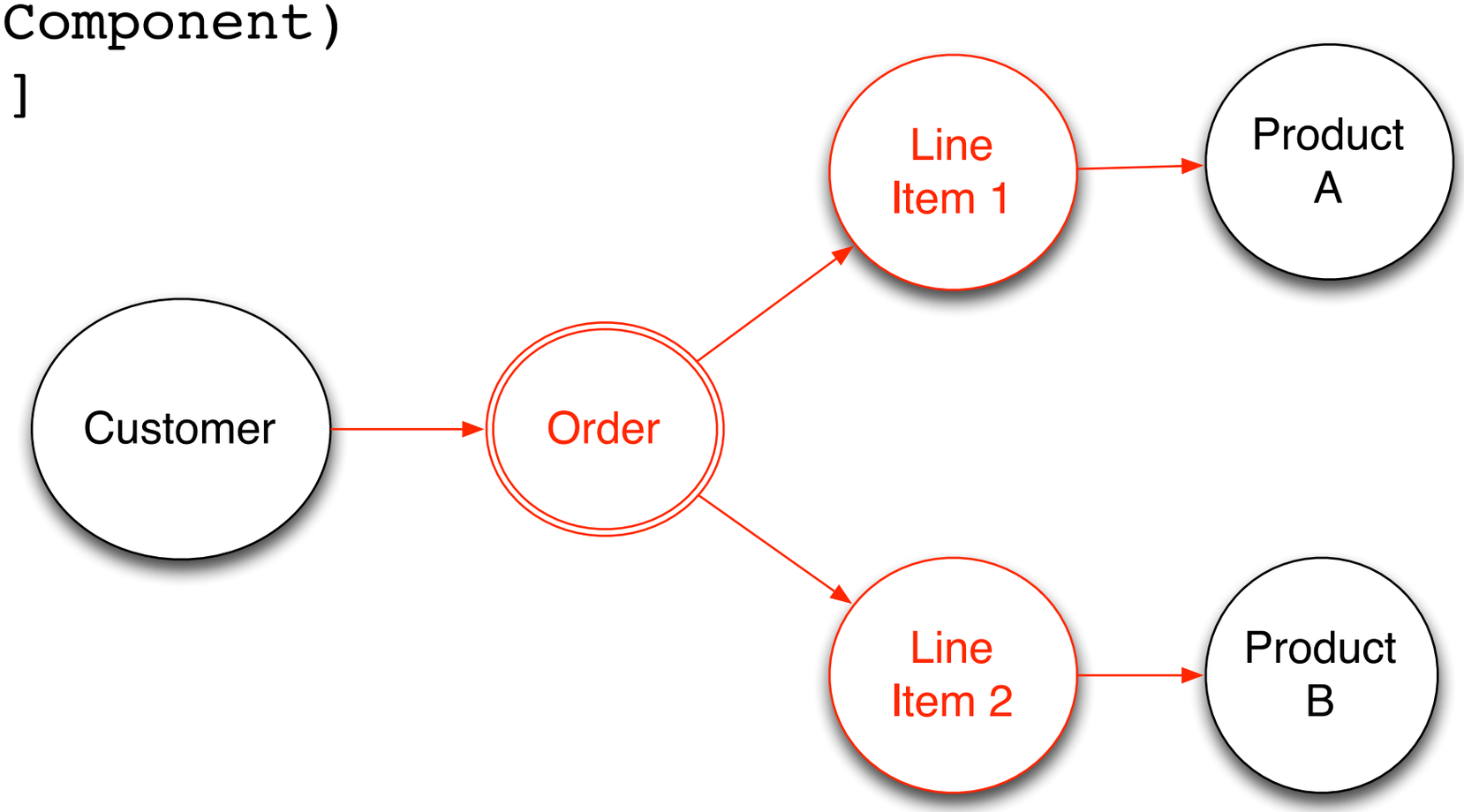
```
[(extent ?x ?e ?a ?v)
 (components ?x ?y)
 (extent ?y ?e ?a ?v)]
[(components ?p ?c)
 (?a :db/isComponent)
 (?p ?a ?c)]
```



Recurse Component Attributes

```
[(extent ?x ?e ?a ?v)
 (components ?x ?y)
 (extent ?y ?e ?a ?v)]
[(components ?p ?c)
 (?a :db/isComponent)
 (?p ?a ?c)]
```

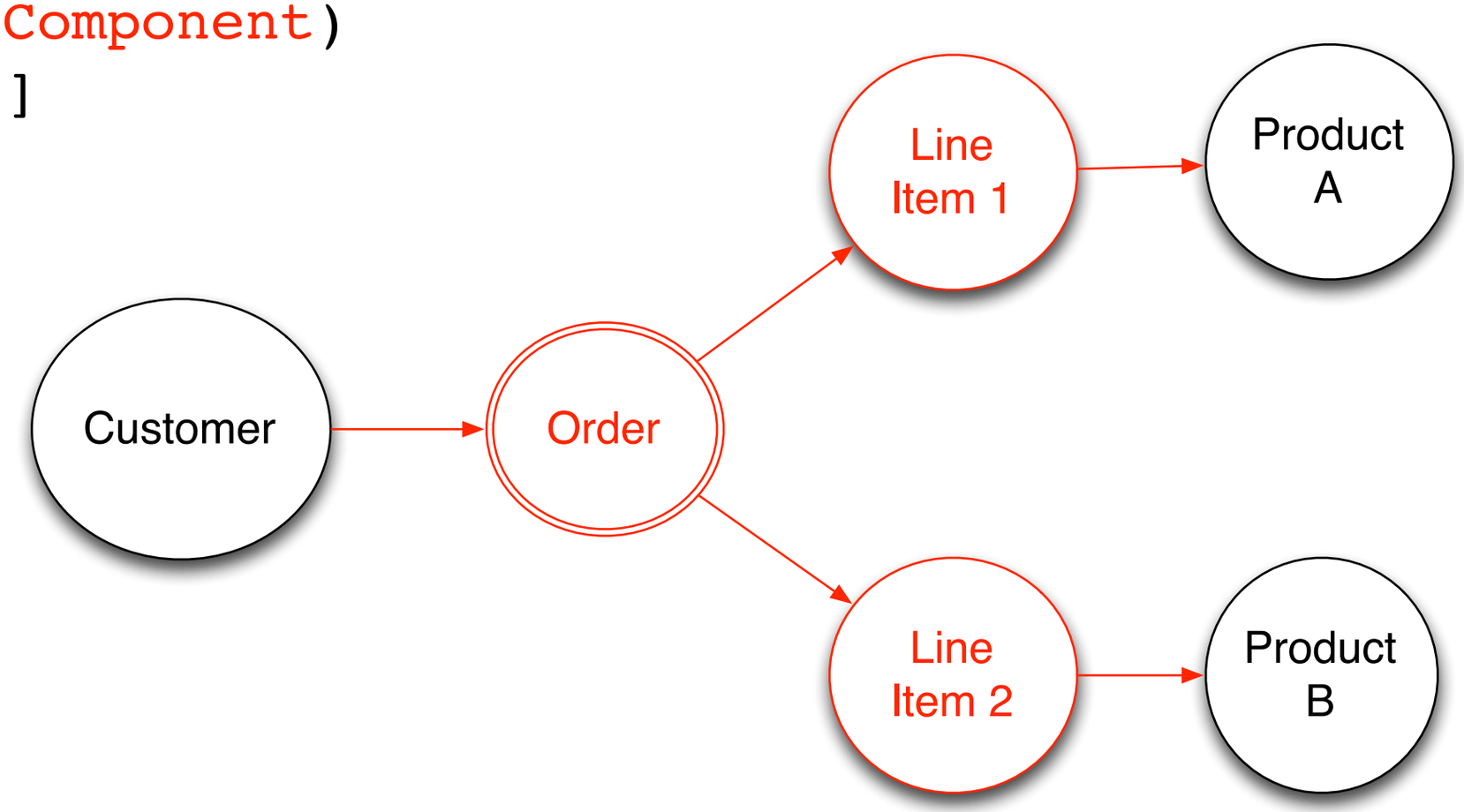
recursive definition



Recurse Component Attributes

```
[(extent ?x ?e ?a ?v)
 (components ?x ?y)
 (extent ?y ?e ?a ?v)]
[(components ?p ?c)
 (?a :db/isComponent)
 (?p ?a ?c)]
```

only recurse attributes
marked :db/isComponent



Resources

Datalog
http://www.amazon.com/Foundations-Databases-The-Logical-Level
http://www.datomic.com/
http://blog.datomic.com/2013/06/using-datomic-from-groovy-part-1.html
http://blog.datomic.com/2013/05/a-whirlwind-tour-of-datomic-query_16.html
https://github.com/datomic/day-of-datomic
https://github.com/datomic/datomic-groovy-examples

Stuart Halloway
https://github.com/stuarthalloway/presentations/wiki.
http://www.linkedin.com/pub/stu-halloway/0/110/543/
https://twitter.com/stuarthalloway
mailto:stu@thinkrelevance.com