

# Logic Programming

## *Datasets*

Michael Genesereth  
Computer Science Department  
Stanford University

# Datasets

**Dataset** - collection of simple facts about state of "world"

Facts in dataset are assumed to be true

Facts not in dataset are assumed to be false

*i.e. datasets are complete; there are no unknowns*

**Role #1 - Datasets as (trivial) logic programs**

used by themselves as standalone databases

used in combination with rules to define "virtual" relations

**Role #2 - Datasets as basis for semantics of logic programs**

# Basics

# Conceptualization

**Objects** - e.g. people, companies, cities

concrete (*person*) or abstract (*number, set, justice*)

primitive (*auto wheel*) or composite (*car*)

real (*earth*) or fictitious (*Sherlock Holmes*)

## **Relationships**

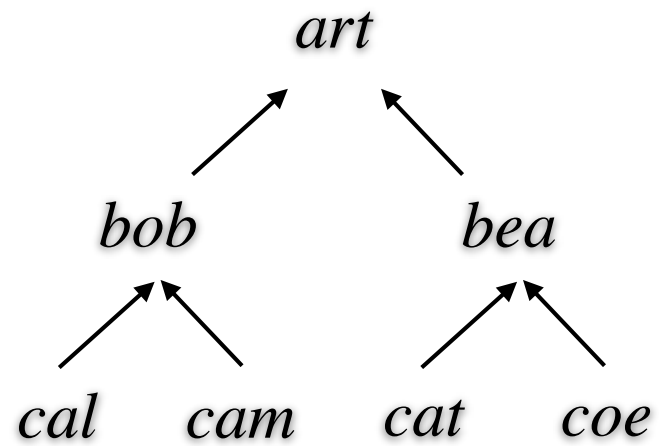
properties of objects or relationships among objects

e.g. *Joe is a person*

e.g. *Joe is the parent of Bill*

e.g. *Joe likes Bill more than Harry*

# Graphical Representation



# Tabular Representation

<b>parent</b>	
art	bob
art	bea
bob	cal
bob	cam
bea	cat
bea	coe

# Natural Language Representation

*Art is the parent of Bob.*

*Art is the parent of Bea.*

*Bob is the parent of Cal.*

*Bob is the parent of Cam.*

*Bea is the parent of Cat.*

*Bea is the parent of Coe.*

# Mathematical Notation

```
parent ( art , bob )  
parent ( art , bea )  
parent ( bob , cal )  
parent ( bob , cam )  
parent ( bea , cat )  
parent ( bea , coe )
```

# Constants

**Constants** are strings of lower case letters, digits, underscores, and periods *or* strings of arbitrary ascii characters within double quotes.

Examples:

```
joe, bill, cs151, 3.14159
```

```
person, worksfor, office.occupant
```

```
the_house_that_jack_built,
```

```
"Mind your p's & q's!"
```

Non-examples:

```
Art, p&q, the-house-that-jack-built
```

A set of constants is called a **vocabulary**.

# Types of Constants

**Symbols / object constants** represent objects.

joe, bill, harry, a23, 3.14159

the\_house\_that\_jack\_built

"Mind your p's & q's!"

**Constructors / function constants** represent functions.

cell, pair, triple, set

**Predicates / relation constants** represent relations.

person, parent, prefers

# Arity

The **arity** of a predicate is the number of arguments that can be associated with the predicate in writing sentences.

**Unary** predicate (1 argument): `person(joe)`

**Binary** predicate (2 arguments): `parent(art,bob)`

**Ternary** predicate (3 arguments): `prefers(art,bob,bea)`

In talking *about* vocabulary, we sometimes notate the arity of a predicate by annotating with a slash and the arity, e.g. `male/1`, `parent/2`, and `prefers/3`.

# Formality and Informality

In some logic programming languages (e.g. Prolog), types and arities determine syntactic legality; and they are enforced by interpreters and compilers.

In other languages (e.g. Epilog), types and arities suggest their intended use. However, they do not determine syntactic legality, and they are not enforced by interpreters and compilers.

In our examples, we use Epilog; but, in this course, we specify types and arities where appropriate and we *try* to adhere to them.

# Data / Factoids

A **datum** / **factoid** is an expression formed from an  $n$ -ary **predicate** and  $n$  symbols enclosed in parentheses and separated by commas.

Symbols:  $a, b$

Predicate:  $p/2, q/1$

Sample Datum:  $p(a, a)$

Sample Datum:  $p(a, b)$

Sample Datum:  $q(a)$

Sample Datum:  $q(b)$

# Herbrand Base

The **Herbrand base** for a vocabulary is the set of all factoids that can be formed from the vocabulary.

Symbols:  $a, b$

Predicate:  $p/2, q/1$

Herbrand Base:

$\{p(a, a), p(a, b), p(b, a), p(b, b), q(a), q(b)\}$

# Datasets

A **dataset** is any *set of factoids* that can be formed from a vocabulary, i.e. a subset of the Herbrand base.

Symbols:  $a, b$

Predicates:  $p/2, q/1$

Herbrand Base:

$\{p(a, a), p(a, b), p(b, a), p(b, b), q(a), q(b)\}$

Dataset:  $\{p(a, b), p(b, a), q(a)\}$

Dataset:  $\{\}$

Dataset:  $\{p(a, a), p(a, b), p(b, a), p(b, b), q(a), q(b)\}$

We use datasets to characterize states of the world. The *facts in a dataset are assumed to be true* and those that are *not in the dataset are assumed to be false*.

# Exercise

## Vocabulary

Symbols:  $a, b$

Predicates:  $p/2, q/1$

## Questions

How many symbols in our vocabulary?

How many elements in the Herbrand base?

How many possible datasets?

# Note on Spelling

Spelling carries no meaning in logic programming (except as informal documentation for programmers).

```
parent(art,bob)
```

```
parent(bob,cal)
```

```
p(a,b)
```

```
p(b,c)
```

```
coulish(widget,gadget)
```

```
coulish(gadget,framis)
```

*The meaning of a constant in logic programming is determined solely by the sentences that mention it.*

*Exception: numbers (23) and strings ("Like this!").*

# Note on Order of Arguments

The order of arguments in an instance of a relation is determined by one's understanding of the relation.

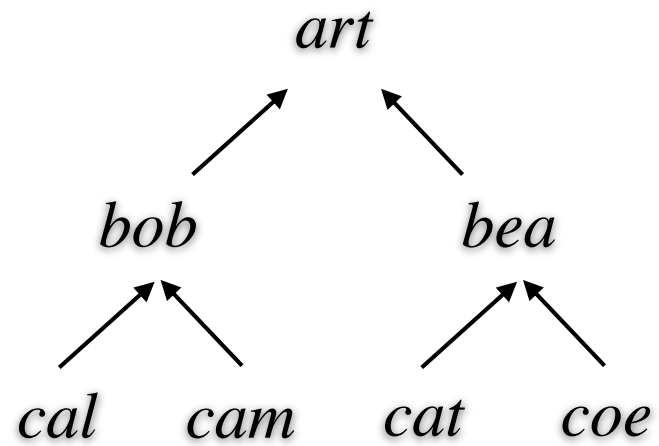
Example:

`prefers (art , bea , bob )`

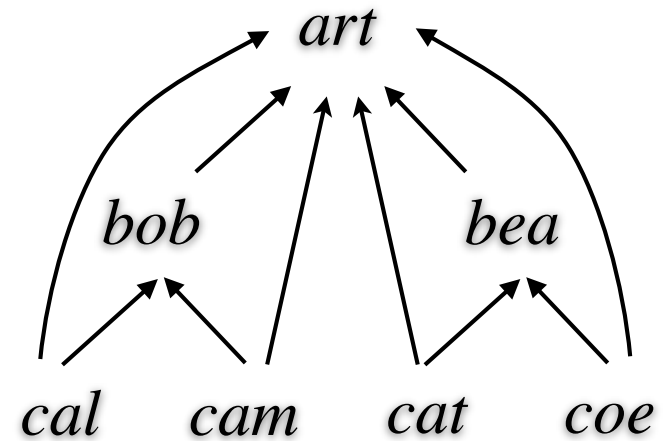
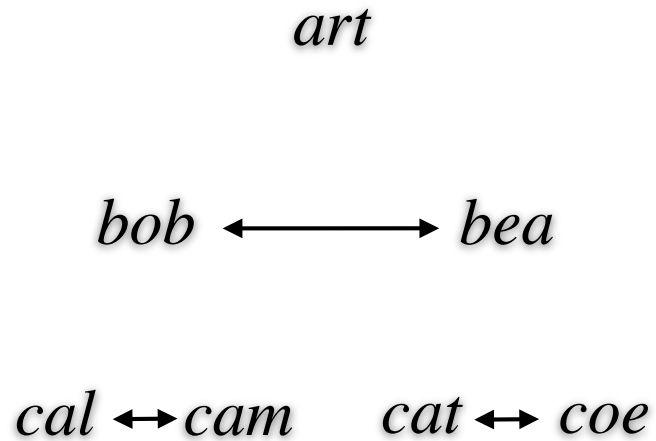
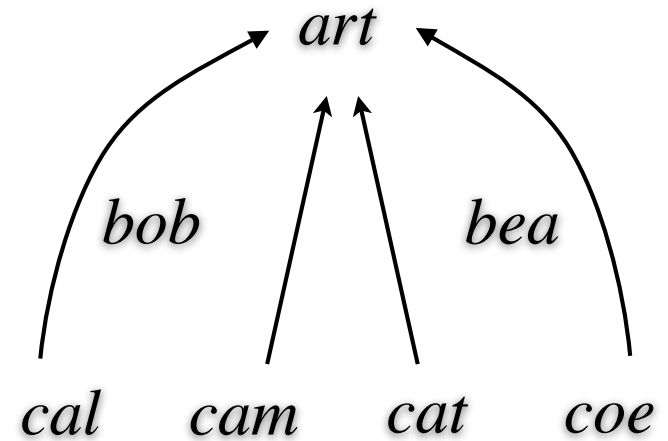
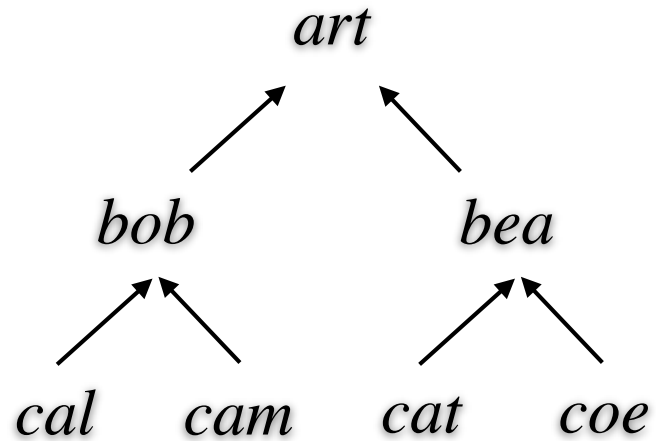
For me, this sentence means that Art prefers Bea to Bob. Other interpretations are possible; the important thing is to be consistent - once you choose, stick with it.

# Kinship

# Parentage



# Kinship Relations

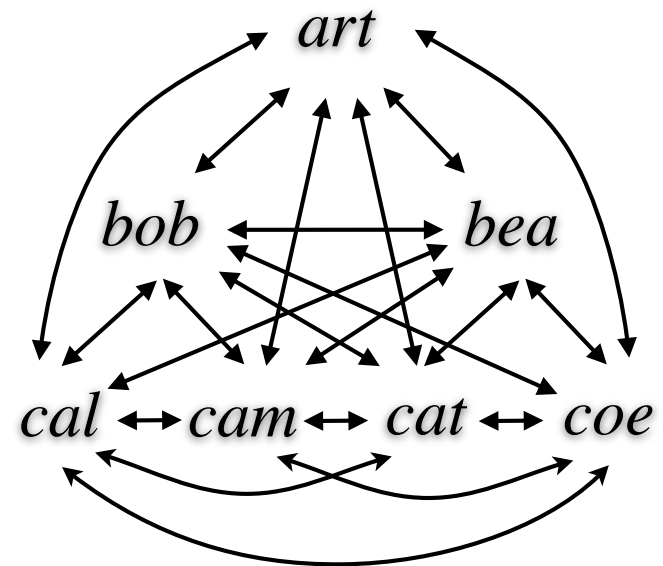


# Degenerate Relations

*art*

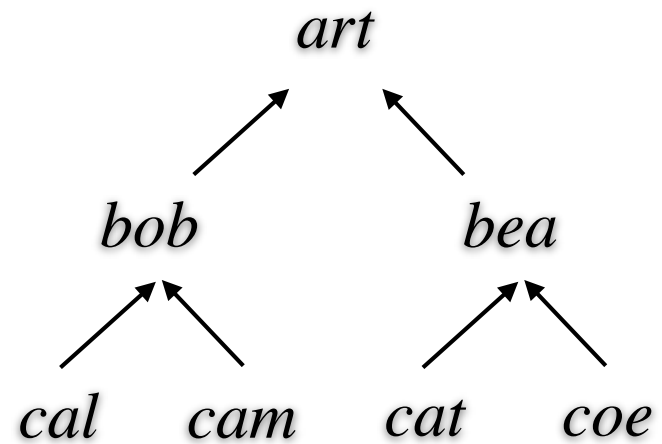
*bob*                      *bea*

*cal*   *cam*   *cat*   *coe*



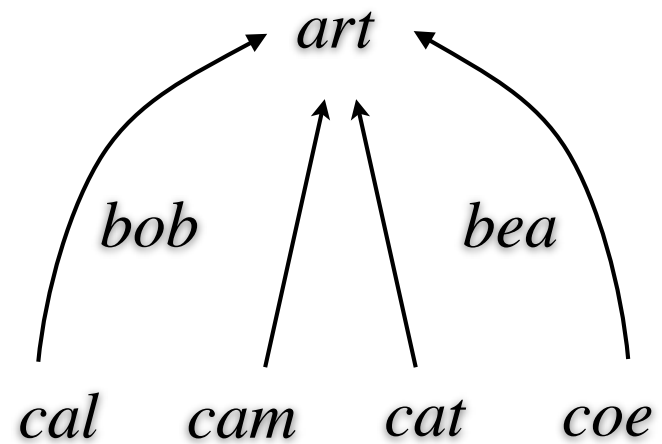
# Parent

```
parent(art,bob)  
parent(art,bud)  
parent(bob,cal)  
parent(bob,cam)  
parent(beat,cat)  
parent(beat,coe)
```



# Grandparent

```
grandparent(art,cal)  
grandparent(art,cam)  
grandparent(art,cat)  
grandparent(art,coe)
```



# Sibling

```
sibling(bob,bea)  
sibling(bea,bob)  
sibling(cal,cam)  
sibling(cam,cal)  
sibling(cat,coe)  
sibling(coe,cat)
```

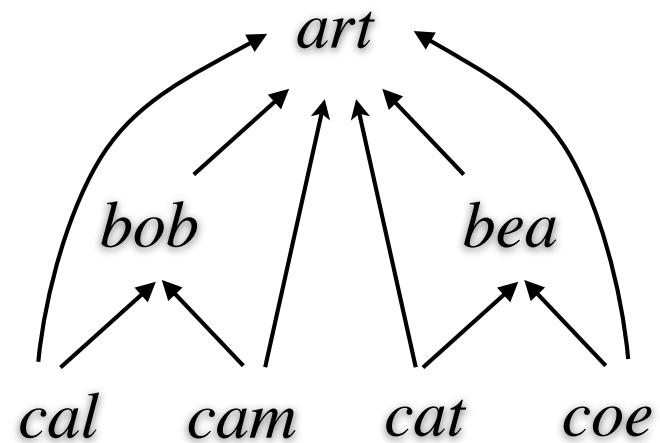
*art*

*bob* ↔ *bea*

*cal* ↔ *cam*    *cat* ↔ *coe*

# Ancestor

```
ancestor(art,bob)
ancestor(art,bea)
ancestor(art,cal)
ancestor(art,cam)
ancestor(art,cat)
ancestor(art,coe)
ancestor(bob,cal)
ancestor(bob,cam)
ancestor(bea,cat)
ancestor(bea,coe)
```



# Other Relations

## Unary Relations:

male(art)

male(bob)

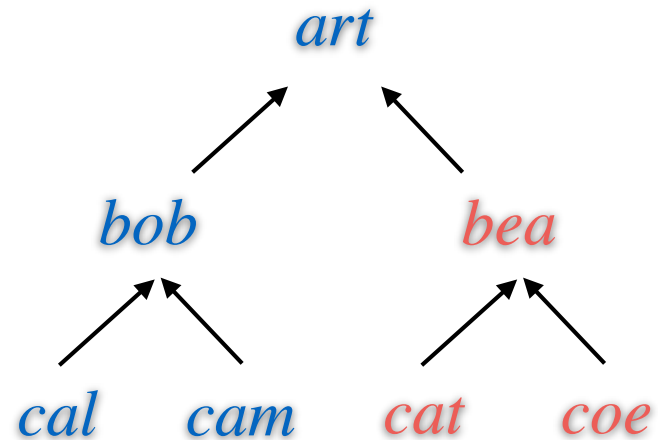
male(cal)

male(cam)

female(bea)

female(cat)

female(coe)



## Ternary Relations:

prefers(art,bob,bea)

prefers(bob,cam,cal)

prefers(bea,cat,coe)

?

# Comments

Some relations definable in terms of others

e.g. we can define grandparent in terms of parent

e.g. we can define sibling in terms of parent

e.g. we can define ancestor in terms of parent

e.g. we can define parent in terms of ancestor

See upcoming material on **view definitions**

Some combinations of arguments do not make sense

e.g. parent ( art , art )

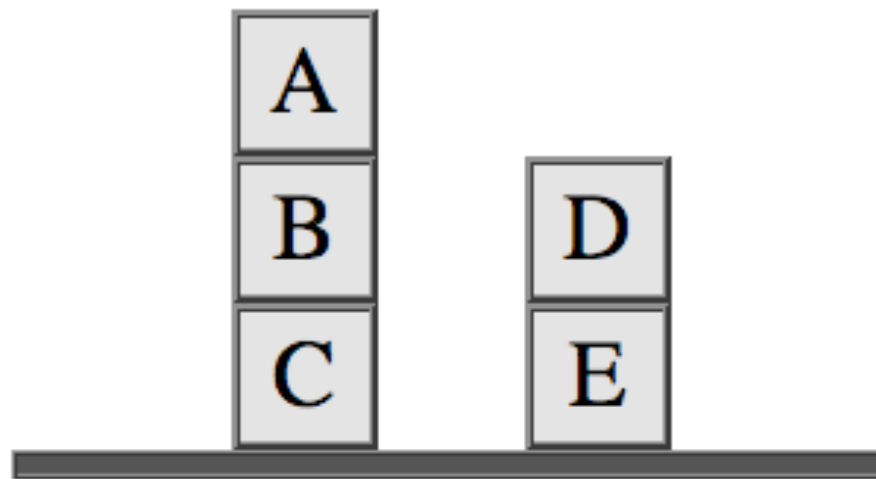
e.g. parent ( art , bob ) and parent ( bob , art )

e.g. old ( art ) and young ( art )

See upcoming material on **constraints**

# Blocks World

# Blocks World



# Vocabulary

Symbols: a, b, c, d, e

Unary Predicates:

`clear` - blocks with no blocks on top.

`table` - blocks on the table.

Binary Predicates:

`on` - pairs of blocks in which first is *on* the second.

`above` - pairs in which first block is *above* the second.

Ternary Predicates:

`stack` - triples of blocks arranged in a stack.

# Dataset

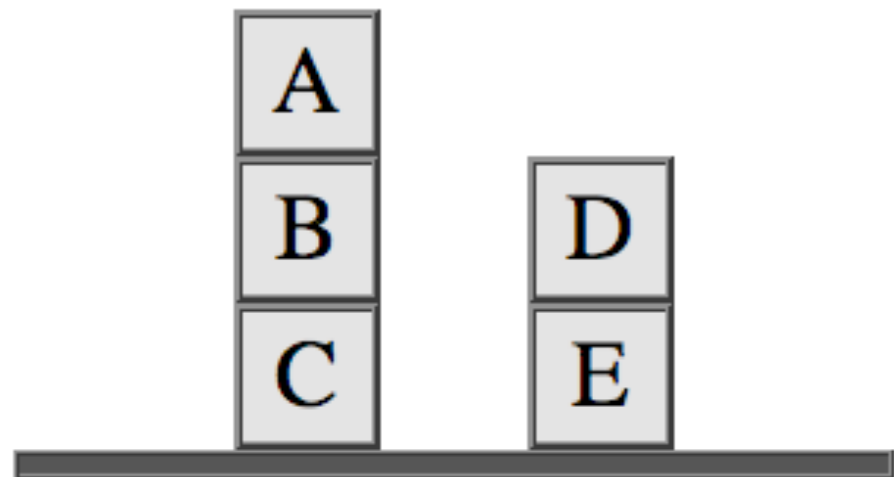
```
clear(a)  
clear(d)
```

```
table(c)  
table(e)
```

```
on(a,b)  
on(b,c)  
on(d,e)
```

```
above(a,b)  
above(b,c)  
above(a,c)  
above(d,e)
```

```
stack(a,b,c)
```



University

# University

## Students:

aaron  
belinda  
calvin  
george

## Departments:

architecture  
computers  
english  
physics

## Faculty:

alan  
cathy  
donna  
frank

## Years:

freshman  
sophomore  
junior  
senior

## Predicate:

student (Student, Department, Advisor, Year)

## Dataset:

student(aaron, architecture, alan, freshman)  
student(belinda, computers, cathy, sophomore)  
student(calvin, english, donna, junior)  
student(george, physics, frank, senior)

# Missing Values

Suppose a student has not declared a major.  
What if a student does not have an advisor?

Leave out fields (syntactically illegal):

```
student ( aaron , , , freshman )
```

Add suitable values to vocabulary (new symbol):

```
student ( aaron , undeclared , orphan , freshman )
```

Database nulls (new linguistic feature):

```
student ( aaron , null , null , freshman )
```

# Multiple Values

Suppose a student has *two* majors.

Multiple Rows (storage, update inconsistencies):

```
student(calvin,english,donna,junior)
```

```
student(calvin,physics,donna,junior)
```

Multiple fields (storage, extensibility?):

```
student(calvin,english,physics,donna,junior)
```

```
student(george,physics,physics,frank,senior)
```

Use compound symbols:

```
student(calvin,english_physics,donna,junior)
```

# Triples

Represent wide relations as collections of binary relations.

## Wide Relation:

```
student (Student, Department, Advisor, Year)
```

## Binary Relations:

```
student.major (Student, Department)
```

```
student.advisor (Student, Faculty)
```

```
student.year (Student, Year)
```

Always works when there is a field of the wide relation (called the **key**) that uniquely specifies the values of the other elements. If none exists, possible to create one.

# Triples

`student.major(aaron, architecture)`

`student.advisor(aaron, alan)`

`student.year(aaron, freshman)`

`student.year(belinda, sophomore)`

`student.major(calvin, english)`

`student.major(calvin, physics)`

`student.advisor(calvin, donna)`

`student.year(calvin, senior)`

`student.major(george, physics)`

`student.advisor(george, frank)`

`student.year(george, senior)`

# Terminology

## Classes

student, department, faculty, year

## Attributes (binary relations associated with a class):

student.major(Student, Department)

student.advisor(Student, Faculty)

student.year(Student, Year)

## Properties of Attributes:

**domain** is class of objects in first position (*arguments*)

**codomain** is class of objects in second position (*values*)

**unique** if *at most* one value for each argument

**total** if *at least* one value for each argument

# Subtlety

## **Missing information**

there is a value but we do not know it.

e.g. Aaron has an advisor but we do not know who it is.

## **Non-existent value**

there is no value

e.g. Aaron does not have an advisor.

*For now, in talking about datasets, we assume **full info**.*

*If a value is missing, it means that there is no value.*

**Sales**

# Sales Ledgers

- In 2015, Art sold Arborhouse to Bob for \$1000000.
- In 2016, Bob sold Pelicanpoint to Carl for \$2000000.
- In 2016, Carl sold Ravenswood to Dan in \$2000000.
- In 2017, Dan sold Ravenswood to Art for \$3000000.

# Real Estate Ledger

<b>People:</b>	<b>Properties:</b>	<b>Years:</b>	<b>Money:</b>
art	arborhouse	2015	1000000
bob	pelicanpoint	2016	2000000
carl	ravenswood	2017	3000000
dan	arborhouse		

## Relation Constant:

`sale(Year, Seller, Property, Buyer, Amount)`

## Dataset:

`sale(2015, art, arborhouse, bob, 1000000)`  
`sale(2016, art, pelicanpoint, bob, 2000000)`  
`sale(2016, carl, ravenswood, dan, 2000000)`  
`sale(2017, dan, arborhouse, art, 3000000)`

# Sales Ledgers

In 2015, Art sold Arborhouse to Bob for \$1000000.

In 2016, Bob sold Pelicanpoint to Carl for \$2000000.

In 2016, Carl sold Ravenswood to Dan in \$2000000.

In 2017, Dan sold Ravenswood to Art for \$3000000.

In 2015, Art sold Bob a widget for \$10.

In 2016, Art sold Bob a gadget for \$20.

In 2016, Art sold Bob a gadget for \$20.

In 2017, Art sold Bob a framis for \$30.

*Different sale!*

# Sales Ledger

<b>People:</b>	<b>Items:</b>	<b>Years:</b>	<b>Money:</b>
art	widget	2015	10
bob	gadget	2016	20
carl	framis	2017	30
dan			

## Relation Constant:

`sale(Year, Seller, Item, Buyer, Amount)`

## Dataset:

```
sale(2015, art, widget, bob, 10)
sale(2016, art, gadget, bob, 20)
sale(2016, art, gadget, bob, 20)
sale(2017, art, framis, bob, 30)
```

*Duplicate factoid!?*

# Sales Ledger

<b>Sales:</b>	<b>People:</b>	<b>Items:</b>	<b>Years:</b>	<b>Money:</b>
t1	art	widget	2015	10
t2	bob	gadget	2016	20
t3	carl	framis	2017	30
t4	dan			

## Relation Constant:

`sale(Sale, Year, Seller, Item, Buyer, Amount)`

## Dataset:

```
sale(t1, 2015, art, widget, bob, 10)
sale(t2, 2016, art, gadget, bob, 20)
sale(t3, 2016, art, gadget, bob, 20)
sale(t4, 2017, art, framis, bob, 30)
```

# Compound Names

# Problem

We sometimes want to talk about complex objects made up of simpler structures.

Examples:

the list of a, b, and c

the cell in row 2 and column 3

Alternative 1: Symbols (structure implicit):

`the_list_of_a_b_c`

`cell_2_3`

Alternative 2: Compound names (structure explicit):

`[a,b,c]`

`cell(2,3)`

# Types of Constants

**Symbols / object constants** represent objects.

`joe, bill, harry, a23, 3.14159`

`the_house_that_jack_built`

`"Mind your p's & q's!"`

**Constructors / function constants**

`cell, pair, triple, set`

**Predicates / relation constants** represent relations.

`person, parent, prefers`

# Types of Constants

**Symbols / object constants** represent objects.

joe, bill, harry, a23, 3.14159

the\_house\_that\_jack\_built

"Mind your p's & q's!"

**Constructors / function constants**

cell, pair, triple, set

**Predicates / relation constants** represent relations.

person, parent, prefers

# Arity

The **arity** of a predicate is the number of arguments that can be associated with the predicate in writing sentences.

**Unary** predicate (1 argument): `person(joe)`

**Binary** predicate (2 arguments): `parent(art,bob)`

**Ternary** predicate (3 arguments): `prefers(art,bob,bea)`

In talking about vocabulary, we sometimes notate the arity of a predicate by annotating with a slash and the arity, e.g. `male/1`, `parent/2`, and `prefers/3`.

# Arity

The **arity** of a **constructor** or a predicate is the number of arguments that can be associated with the **constructor** or predicate in writing complex expressions in the language.

**Unary** constructor (1 argument): `successor(1)`

**Binary** constructor (2 arguments): `pair(1,2)`

**Ternary** constructor (3 arguments): `triple(1,2,3)`

**Unary** predicate (1 argument): `person(joe)`

**Binary** predicate (2 arguments): `parent(art,bob)`

**Ternary** predicate (3 arguments): `prefers(art,bob,bea)`

In talking about vocabulary, we sometimes notate the arity of a **constructor** or predicate by annotating with a slash and the arity, e.g. `successor/1`, `pair/2`, `triple/3`, `male/1`, `parent/2`, and `prefers/3`.

# Compound Names (version 1)

A **compound name** is an expression formed from an  $n$ -ary *constructor* and  $n$  *symbols* enclosed in parentheses and separated by commas.

Symbols:  $a, b$

Constructor:  $f/2, g/1$

Compound Names:  $f(a, b), f(b, a), g(a), g(b)$

*This allows us to refer to complex objects made up of simple objects. How do we refer to complex objects made up of other complex objects?*

# Compound Names (version 2)

A **compound name** is an expression formed from an  $n$ -ary *constructor* and  $n$  *symbols* **or** *compound names* enclosed in parentheses and separated by commas.

Symbols:  $a, b$

Constructor:  $f/2, g/1$

Compound Names:  $f(a, b), f(b, a), g(a), g(b)$

Compound Names:  $f(g(a), b), g(f(a, b))$

Compound Names:  $g(g(a)), g(f(g(a), g(b)))$

Compound Names:  $g(g(g(a)))$

# Ground Terms

A **ground term** is either a *symbol* or a *compound name*.

The adjective *ground* here means that the term does not contain any *variables* (which we discuss in later lessons).

# Herbrand Universe

The **Herbrand universe** for a vocabulary is the set of **all** *ground terms* that can be formed from the *symbols* and *constructors* in the vocabulary.

# Data / Factoids

A **datum / factoid** is an expression formed from an  $n$ -ary predicate and  $n$  *ground terms* enclosed in parentheses and separated by commas.

Symbols:  $a, b$

Constructor:  $f/2, g/1$

Predicate:  $p/2$

Sample Datum:  $p(a, g(a))$

Sample Datum:  $p(f(a, b), g(b))$

# Other Notions

The **Herbrand universe** for a vocabulary is the set of all *ground terms* that can be formed from the *symbols* and *constructors* in the vocabulary.

The **Herbrand base** for a vocabulary is the set of all *factoids* that can be formed from the vocabulary.

A **dataset** is any *set of factoids* that can be formed from a vocabulary, i.e. a subset of the Herbrand base.

# Exercise

## Vocabulary

Symbols:  $a, b$

Predicates:  $p/2, q/1$

## Questions

How many symbols in the Herbrand universe?

How many elements in the Herbrand base?

How many possible datasets?

# Exercise

## Vocabulary

Symbols:  $a, b$

Constructor:  $f/1, g/1$

Predicates:  $p/2, q/1$

## Questions

How many elements in the Herbrand universe?

How many elements in the Herbrand base?

How many possible datasets?

Sierra

# Sierra

**Sierra** is a collection of related browser-based applications of use in developing rulesets and datasets encoded in Epilog.

[Sierrabase](#) provides elementary tools for viewing and editing Epilog datasets. It includes tools for evaluating relational and functional queries on data and applying transformations to that data.

[Sierralite](#) provides elementary tools for viewing and editing rulesets as well as datasets. It includes tools for computing defined relations, evaluating defined functions, and applying defined operations.

[Sierraplus](#), [Sierraplan](#), [Sierraplay](#), [Sierrarule](#), [Sierrameta](#) - experimental

<http://epilog.stanford.edu/homepage/sierra.php>

# Assignment

# Assignment 1.1 - Movies

Consider a vocabulary that includes the following relations.

`movie.instance(x)` means that  $x$  is a movie.

`actor.instance(x)` means that  $x$  is an actor.

`director.instance(x)` means that  $x$  is a director.

`year.instance(x)` means that  $x$  is a year.

`title.instance(x)` means that  $x$  is a title.

`movie.star(x,y)` means that movie  $x$  stars actor  $y$ .

`movie.director(x,y)` means that movie  $x$  was directed by  $y$ .

`movie.year(x,y)` means that movie  $x$  was released in year  $y$ .

`movie.title(x,y)` means that movie  $x$  has the title  $y$ .

Choose symbols for a few movies, actors, directors, years, and titles, and encode the relevant data about these entities using this vocabulary.

# Assignment 1.2 - Metadata

Consider a vocabulary that includes the following relations.

`type.instance(x)` means that  $x$  is a type.

`type.predicate(x,y)` means that type  $x$  has predicate  $y$ .

`type.attribute(x,y)` means that type  $x$  has attribute  $y$ .

`predicate.instance(x)` means that  $x$  is a predicate.

`predicate.domain(x,y)` means that predicate  $x$  has domain  $y$ .

`attribute.instance(x)` means that  $x$  is an attribute.

`attribute.domain(x,y)` means that attribute  $x$  has domain  $y$ .

`attribute.codomain(x,y)` means that  $x$  has codomain  $y$ .

`attribute.total(x,yes)` whether  $x$  has at least one value.

`attribute.unique(x,yes)` whether  $x$  has at most one value.

Use this vocabulary to encode types and relations in movie vocabulary.

# Assignment 1.3 - Escher

Use the vocabulary in Assignment 1.2 to describe itself.

Factoids describing type are shown below. Your job is to do other types, predicates, attributes, and booleans.

```
type.instance(type)
type.predicate(type,type.instance)
type.attribute(type,type.predicate)
type.attribute(type,type.attribute)
...
```

Yes, the *predicates* in our vocabulary are *symbols* in this vocabulary as well as *predicates*!



