Databases

A database is a collection of records (tuples) and an interface for adding, editing, and retrieving records.

The Structured Query Language (SQL) is perhaps the most widely used programming language on Earth.

**SELECT * FROM toy_info WHERE color='yellow';**

<table>
<thead>
<tr>
<th>toy_id</th>
<th>toy</th>
<th>color</th>
<th>cost</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>whiffleball</td>
<td>yellow</td>
<td>2.20</td>
<td>0.40</td>
</tr>
<tr>
<td>5</td>
<td>frisbee</td>
<td>yellow</td>
<td>1.50</td>
<td>0.20</td>
</tr>
<tr>
<td>10</td>
<td>yoyo</td>
<td>yellow</td>
<td>1.50</td>
<td>0.20</td>
</tr>
</tbody>
</table>

SQL is an example of a declarative programming language.

It separates _what_ to compute from _how_ it is computed.

The language interpreter is free to compute the result in any way it deems appropriate.

http://www.headfirstlabs.com/sql_hands_on/
Logical/Declarative Programming

The principal characteristics of declarative languages:
• A "program" is a description of the desired solution.
• The interpreter figures out how to generate such a solution.

By contrast, in imperative languages such as Python & Scheme:
• A "program" is a description of procedures.
• The interpreter carries out execution/evaluation rules.

Building a universal problem solver is a difficult task.

Declarative programming languages compromise by solving only a subset of problems.

They typically trade off data scale for problem complexity.
The *Logic* Language

The *Logic* language is invented for this course.

- Based on the Scheme project + ideas from Prolog
- Expressions are facts or queries, which contain relations.
- Expressions and relations are Scheme lists.
- For example, *(likes John dogs)* is a relation.
- Implementation fits on a single sheet of paper (next lecture)

Today's theme:
Simple Facts

A simple fact expression in the Logic language declares a relation to be true.

Let's say I want to track my many dogs' ancestry.

Language Syntax:
• A relation is a Scheme list.
• A fact expression is a Scheme list of relations.

```scheme
logic> (fact (parent delano herbert))
logic> (fact (parent abraham barack))
logic> (fact (parent abraham clinton))
logic> (fact (parent fillmore abraham))
logic> (fact (parent fillmore delano))
logic> (fact (parent fillmore grover))
logic> (fact (parent eisenhower fillmore))
```
In Logic, a relation is not a call expression.

- In Scheme, we write (abs -3) to call abs on -3. It returns 3.
- In Logic, (abs -3 3) asserts that the abs of -3 is 3.

For example, if we wanted to assert that 1 + 2 = 3,

\[(\text{add} \ 1 \ 2 \ 3)\]

Why declare knowledge in this way? It will allow us to solve problems in two directions:

\[(\text{add} \ 1 \ 2 \ _)\]

\[(\text{add} \ _ \ 2 \ 3)\]

\[(\text{add} \ 1 \ _ \ 3)\]
Queries

A query contains one or more relations. The Logic interpreter returns whether (& how) they are all simultaneously satisfied.

Queries may contain variables: symbols starting with ?

logic> (fact (parent abraham barack))
logic> (fact (parent abraham clinton))
logic> (fact (parent delano herbert))
logic> (fact (parent fillmore abraham))
logic> (fact (parent fillmore delano))
logic> (fact (parent fillmore grover))
logic> (fact (parent eisenhower fillmore))

logic> (query (parent abraham ?child))
Success!
child: barack
child: clinton
Compound Facts

A fact can include multiple relations and variables as well.

(fact <conclusion> <hypothesis₀> <hypothesis₁> ... <hypothesisₙ>)

Means <conclusion> is true if all <hypothesisᵢ> are true.


logic> (query (child herbert delano))
Success!

logic> (query (child eisenhower clinton))
Failure.

logic> (query (child ?child fillmore))
Success!
child: abraham
child: delano
child: grover
Recursive Facts

A fact is recursive if the same relation is mentioned in a hypothesis and the conclusion.

```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (query (ancestor ?a herbert))
Success!
a: delano
a: fillmore
a: eisenhower

logic> (query (ancestor ?a barack)
             (ancestor ?a herbert))
Success!
a: fillmore
a: eisenhower
```

Demo
Searching to Satisfy Queries

The Logiq interpreter performs a search in the space of relations for each query to find a satisfying assignment.

\[
\text{logic} > (\text{query (ancestor ?a herbert)}) \\
\text{Success!} \\
a: \text{delano} \\
a: \text{fillmore} \\
a: \text{eisenhower}
\]

\[
\text{logic} > (\text{fact (parent delano herbert)}) \\
\text{logic} > (\text{fact (parent fillmore delano)})
\]

\[
\text{logic} > (\text{fact (ancestor ?a ?y) (parent ?a ?y)}) \\
\text{logic} > (\text{fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y)})
\]

\[
(\text{parent delano herbert}) \quad ; \ (1), \text{ a simple fact} \\
(\text{ancestor delano herbert}) \quad ; \ (2), \text{ from (1) and the 1st ancestor fact} \\
(\text{parent fillmore delano}) \quad ; \ (3), \text{ a simple fact} \\
(\text{ancestor fillmore herbert}) \quad ; \ (4), \text{ from (2), (3), & the 2nd ancestor fact}
\]
Hierarchical Facts

Relations can contain relations in addition to atoms.

logic> (fact (dog (name abraham) (color white)))
logic> (fact (dog (name barack) (color tan)))
logic> (fact (dog (name clinton) (color white)))
logic> (fact (dog (name delano) (color white)))
logic> (fact (dog (name eisenhower) (color tan)))
logic> (fact (dog (name fillmore) (color brown)))
logic> (fact (dog (name grover) (color tan)))
logic> (fact (dog (name herbert) (color brown)))

Variables can refer to atoms or relations.

logic> (query (dog (name clinton) (color ?color)))
Success!
color: white

logic> (query (dog (name clinton) ?info))
Success!
info: (color white)
Example: Combining Multiple Data Sources

Which dogs have an ancestor of the same color?

```
logic> (query (dog (name ?name) (color ?color))
  (ancestor ?ancestor ?name)
  (dog (name ?ancestor) (color ?color)))

Success!

name: barack     color: tan     ancestor: eisenhower
name: clinton    color: white   ancestor: abraham
name: grover     color: tan     ancestor: eisenhower
name: herbert    color: brown   ancestor: fillmore
```
Example: Appending Lists

Two lists append to form a third list if:
• The first list is empty and the second and third are the same
• The rest of 1 and 2 append to form the rest of 3

logic> (fact (append-to-form () ?x ?x))

logic> (fact (append-to-form (?a . ?r) ?y (?a . ?z))
 (append-to-form ?r ?y ?z))

Demo