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

http://www.headfirstlabs.com/sql_hands_on/
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.

http://www.headfirstlabs.com/sql_hands_on/
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.
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.

```sql
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
Logical/Declarative Programming

The principal characteristics of declarative languages:
Logical/Declarative Programming

The principal characteristics of declarative languages:
• A "program" is a description of the desired solution.
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.
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:
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.
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.
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.
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.
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

The *Logic* language is invented for this course.
The *Logic* Language

The *Logic* language is invented for this course.

- Based on the Scheme project + ideas from Prolog
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.
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.
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.
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)
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:
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
Simple Facts

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

```
logic> (fact (parent delano herbert))
```
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.

logic> (fact (parent delano herbert))
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.

```
logic> (fact (parent delano herbert))
logic> (fact (parent abraham barack))
```

D
Herbert
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.

```
logic> (fact (parent delano herbert))
logic> (fact (parent abraham barack))
```
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.

```
logic> (fact (parent delano herbert))
logic> (fact (parent abraham barack))
logic> (fact (parent abraham clinton))
```

A

B

D

Herbert
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.

logic> (fact (parent delano herbert))
logic> (fact (parent abraham barack))
logic> (fact (parent abraham clinton))

A

B

C

D

Herbert
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.

```
logic> (fact (parent delano herbert))
logic> (fact (parent abraham barack))
logic> (fact (parent abraham clinton))
logic> (fact (parent fillmore abraham))
```
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.

```
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))
```

![Ancestry Diagram]

- F
- A
- D
- B
- C
- Herbert
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.

```
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))
```
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.

```
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))
```
Relations are Not Procedure Calls
Relations are Not Procedure Calls

In *Logic*, a relation is not a call expression.
Relations are Not Procedure Calls

In Logic, a relation is not a call expression.
• In Scheme, we write (abs -3) to call abs on -3. It returns 3.
Relations are Not Procedure Calls

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.
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,
Relations are Not Procedure Calls

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 $1 + 2 = 3$,

$$(\text{add } 1 \ 2 \ 3)$$
Relations are Not Procedure Calls

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 $1 + 2 = 3$,

```
(add 1 2 3)
```

Why declare knowledge in this way? It will allow us to solve problems in two directions:
Relations are Not Procedure Calls

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`,

```
(add 1 2 3)
```

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

```
(add 1 2 _)
```
Relations are Not Procedure Calls

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 \ _)$$

$(\text{add } _ \ 2 \ 3)$
Relations are Not Procedure Calls

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:

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

A

B

C

D

G

H

F

E

A

D

G

B

C

Herbert
Queries

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

```plaintext
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))
```
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))
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!
```
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
```
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
```
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.
Compound Facts

A fact can include multiple relations and variables as well.

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

A fact can include multiple relations and variables as well.

(fact <conclusion> <hypothesis_0> <hypothesis_1> ... <hypothesis_N>)

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

Compound Facts

A fact can include multiple relations and variables as well.

(fact <conclusion> <hypothesis_0> <hypothesis_1> ... <hypothesis_N>)

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

logic> (query (child herbert delano))
Success!
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.
Compound Facts

A fact can include multiple relations and variables as well.

(fact <conclusion> <hypothesis_0> <hypothesis_1> ... <hypothesis_N>)

Means <conclusion> is true if all <hypothesis_i> 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

E

F

A

D

G

B C Herbert

A

D

G

B C Herbert

A

D

G

B C Herbert
Recursive Facts
Recursive Facts

A fact is recursive if the same relation is mentioned in a hypothesis and the conclusion.
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))
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
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
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
```
Searching to Satisfy Queries
Searching to Satisfy Queries

The *Logic* interpreter performs a search in the space of relations for each query to find a satisfying assignment.
Searching to Satisfy Queries

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

```logic
logic> (query (ancestor ?a herbert))
Success!
a: delano
a: fillmore
a: eisenhower
```
The *Logic* interpreter performs a search in the space of relations for each query to find a satisfying assignment.

```sql
logic> (query (ancestor ?a herbert))
Success!
a: delano
a: fillmore
a: eisenhower
```
Searching to Satisfy Queries

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

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

\[
\text{logic} > (\text{fact (parent delano herbert)})
\]
\[
\text{logic} > (\text{fact (parent fillmore delano)})
\]
Searching to Satisfy Queries

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

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

```
logic> (fact (parent delano herbert))
logic> (fact (parent fillmore delano))
```

```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
```
Searching to Satisfy Queries

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

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

```
logic> (fact (parent delano herbert))
logic> (fact (parent fillmore delano))
```

```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
```

```
(parent delano herbert) ; (1), a simple fact
```
Searching to Satisfy Queries

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

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

logic> (fact (parent delano herbert))
logic> (fact (parent fillmore delano))

logic> (fact (ancestor ?a ?y) (parent ?a ?y))

(parent delano herbert) ; (1), a simple fact
(ancestor delano herbert) ; (2), from (1) and the 1st ancestor fact
Searching to Satisfy Queries

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

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

(logic> (fact (parent delano herbert)))
(logic> (fact (parent fillmore delano)))

(logic> (fact (ancestor ?a ?y) (parent ?a ?y)))
(logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y)))
```

(1), a simple fact
(2), from (1) and the 1st ancestor fact
(3), a simple fact
Searching to Satisfy Queries

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

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

logic> (fact (parent delano herbert))
logic> (fact (parent fillmore delano))

logic> (fact (ancestor ?a ?y) (parent ?a ?y))
```

<table>
<thead>
<tr>
<th>(parent delano herbert)</th>
<th>(1), a simple fact</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ancestor delano herbert)</td>
<td>(2), from (1) and the 1st ancestor fact</td>
</tr>
<tr>
<td>(parent fillmore delano)</td>
<td>(3), a simple fact</td>
</tr>
<tr>
<td>(ancestor fillmore herbert)</td>
<td>(4), from (2), (3), &amp; the 2nd ancestor fact</td>
</tr>
</tbody>
</table>
Hierarchical Facts
Hierarchical Facts

Relations can contain relations in addition to atoms.
Hierarchical Facts

Relations can contain relations in addition to atoms.

logic> (fact (dog (name abraham) (color white)))
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)))
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)))
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.
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
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)
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?
Example: Combining Multiple Data Sources

Which dogs have an ancestor of the same color?

```
logic> (query (dog (name ?name) (color ?color))
```
Example: Combining Multiple Data Sources

Which dogs have an ancestor of the same color?

logic> (query (dog (name ?name) (color ?color))
    (ancestor ?ancestor ?name)
Example: Combining Multiple Data Sources

Which dogs have an ancestor of the same color?

\[
\text{logic}\> \text{(query (dog (name ?name) (color ?color))}
\]
\[
\text{ (ancestor ?ancestor ?name)}
\]
\[
\text{ (dog (name ?ancestor) (color ?color)))}
\]
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
Example: Appending Lists

Two lists append to form a third list if:
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
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
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))
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))
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