Announcements

- HW12 due Wednesday
- Scheme project, contest due next Monday
Databases
A database is a collection of records (tuples) and an interface for adding, editing, and retrieving records.
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.
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';
```
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>
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.

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.

```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.

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.

```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.

Declarative Programming
Declarative Programming

The main characteristics of declarative languages:
The main characteristics of declarative languages:

• A "program" is a description of the desired solution
Declarative Programming

The main characteristics of declarative languages:

• A "program" is a description of the desired solution
• The interpreter figures out how to generate such a solution
Declarative Programming

The main 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 procedural languages such as Python & Scheme:
Declarative Programming

The main 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 procedural languages such as Python & Scheme:

• A "program" is a description of procedures
The main 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 procedural languages such as Python & Scheme:

- A "program" is a description of procedures
- The interpreter carries out execution/evaluation rules
Declarative Programming

The main 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 procedural 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

The main 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 procedural 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 all problems
Declarative Programming

The main 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 procedural 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 all 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 both Scheme lists
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 both Scheme lists
- For example, (likes Amir 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 both Scheme lists
- For example, *(likes Amir dogs)* is a relation
- Implementation fits on a single sheet of paper (next lecture)
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 both Scheme lists
- For example, *(likes Amir 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 both Scheme lists
- For example, *(likes Amir dogs)* is a relation
- Implementation fits on a single sheet of paper (next lecture)

Today’s theme:

http://awhimsicalbohemian.typepad.com/.a/6a00e5538b84f3883301538dfa8f19970b-800wi
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 containing fact followed by one or more relations
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 containing `fact` followed by one or more relations.

```plaintext
logic> (fact (parent delano 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 containing `fact` followed by one or more 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 containing **fact** followed by one or more relations.

```logic
(fact (parent delano herbert))
(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 containing *fact* followed by one or more relations.

```
logic> (fact (parent delano herbert))
logic> (fact (parent abraham barack))
```
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 containing `fact` followed by one or more 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 containing `fact` followed by one or more relations.

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

![Diagram showing the ancestry of dogs with relations between A, B, C, D, and 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 containing `fact` followed by one or more relations.

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

```
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 containing `fact` followed by one or more 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))
```

![Diagram of 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:

- A relation is a Scheme list.
- A fact expression is a Scheme list containing `fact` followed by one or more 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))
```

Diagram:

```
  F
 /|
/  |
A  D  G
 /|
/  |
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 containing `fact` followed by one or more 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 \((\text{abs} \ -3)\) to call \text{abs} on \(-3\)
Relations are Not Procedure Calls

In *Logic*, a relation is not a call expression

- In Scheme, we write \((\text{abs} \ -3)\) to call \text{abs} on \(-3\)
- In *Logic*, \((\text{abs} \ -3 \ 3)\) asserts that the \text{abs} of \(-3\) is \(3\)
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`
- 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`
- 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)
```
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`
• 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`
- 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`
- 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 _)
(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`
- 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 _)
(add _ 2 3)
(add 1 _ 3)
```
Queries

E
  ↓
F
  ↓
A  D  G
  ↓  ↓
B  C  Herbert
A query contains one or more relations. The Logic interpreter returns whether (and how) they are all simultaneously satisfied.
Queries

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

Queries may contain variables: symbols starting with ?

```
E
  ↓
F
  ↓
A  D  G
  ↓  ↓  ↓
B  C  Herbert
```
Queries

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

Queries may contain variables: symbols starting with `?`:

```prolog
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))
```

Diagram of relations:

```
  E
   ↓
  F
  / \
A   D   G
  ↓   ↓   ↓
B   C   Herbert
```
 Queries

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

Queries may contain variables: symbols starting with ?

```
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))
logic> (query (parent abraham ?child))
```
Queries

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

Queries may contain variables: symbols starting with `?`

```
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))
logic> (query (parent abraham ?child))
```

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

Queries may contain variables: symbols starting with `?`

```
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))

logic> (query (parent abraham ?child))
Success!
child: barack
```
Queries

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

Queries may contain variables: symbols starting with ?

```
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))
logic> (query (parent abraham ?child))
Success!
child: barack
child: clinton
```
Queries

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

Queries may contain variables: symbols starting with ?

```
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))
logic> (query (parent abraham ?child))
Success!
child: barack
child: clinton
```
Queries

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

Queries may contain variables: symbols starting with ?

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

Queries may contain variables: symbols starting with `?`.

```
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))
logic> (query (parent ?who barack))
```
Queries

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

Queries may contain variables: symbols starting with ?

```logic
(fact (parent delano herbert))
(fact (parent abraham barack))
(fact (parent abraham clinton))
(fact (parent fillmore abraham))
(fact (parent fillmore delano))
(fact (parent fillmore grover))
(fact (parent eisenhower fillmore))
(query (parent ?who barack)
  (parent ?who clinton))
```
Queries

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

Queries may contain variables: symbols starting with ?

```
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))
logic> (query (parent ?who barack)
              (parent ?who clinton))
Success!
```
A query contains one or more relations. The Logic interpreter returns whether (and how) they are all simultaneously satisfied.

Queries may contain variables: symbols starting with ?

```
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))
logic> (query (parent ?who barack) (parent ?who clinton))
```

Success!

who: abraham
Compound Facts

\[ \text{E} \rightarrow \text{F} \]

\[ \text{A} \rightarrow \text{D} \rightarrow \text{G} \]

\[ \text{B} \rightarrow \text{C} \rightarrow \text{Herbert} \]
Compound Facts

A fact can include multiple relations and variables as well.

Diagram:

```
      E
     /\  
    F   
   / \  /
  A   D G
 /    /  \
B    C    Herbert
```
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> <hypothesis₀> <hypothesis₁> ... <hypothesisₙ>)

Means <conclusion> is true if all <hypothesisₖ> 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 <hypothesisₖ> are true


Diagram:

- E
- F
  - A
  - D
  - G
    - B
    - C
    - Herbert
Compound Facts

A fact can include multiple relations and variables as well

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

Means <conclusion> is true if all <hypothesisₖ> are true

\[ \text{logic} \] (fact (child ?c ?p) (parent ?p ?c))

\[ \text{logic} \] (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 <hypothesisₖ> are true

Eisenhower


Success!

logic> (query (child herbert delano))

Failure.

logic> (query (child eisenhower clinton))
A fact can include multiple relations and variables as well

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

Means <conclusion> is true if all <hypothesisₖ> are true

Success!

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
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

\[
\text{logic}\text{> } (\text{fact } (\text{ancestor } ?a ?y) (\text{parent } ?a ?y)) \\
\text{logic}\text{> } (\text{fact } (\text{ancestor } ?a ?y) (\text{parent } ?a ?z) (\text{ancestor } ?z ?y)) \\
\text{logic}\text{> } (\text{query } (\text{ancestor } ?a \text{ herbert})) \\
\text{Success!}
\]
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
```
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> (query (ancestor ?a herbert))
Success!
\[a: \text{delano}\]
\[a: \text{fillmore}\]
\[a: \text{eisenhower}\]
```
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
```
Searching to Satisfy Queries

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

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

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

(p-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))
```

(\text{parent delano herbert}) \quad ; \quad (1), \text{ a simple fact}

(\text{ancestor delano herbert}) \quad ; \quad (2), \text{ from (1) and the } 1\text{st} \text{ 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
(query (ancestor ?a herbert))
Success!
```

```logic
(a: delano)
(a: fillmore)
(a: eisenhower)
```

```logic
(fact (parent delano herbert))
(fact (parent fillmore delano))
(fact (ancestor ?a ?y) (parent ?a ?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))

(parent delano herbert)  ; (1), a simple fact
(ancestor delano herbert)  ; (2), from (1) and the 1st ancestor fact
(parent fillmore delano)  ; (3), a simple fact
(ancestor fillmore herbert) ; (4), from (2), (3), & the 2nd ancestor fact
```
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!
infol (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)
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?

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

  () (a b c) (a b c)
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

\((\,)(a\ b\ c)(a\ b\ c)\)

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
  (a b c) (a b c)

• Both of the following hold:

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

\[ () (a\ b\ c) (a\ b\ c) \]

• Both of the following hold:
  • List 1 and 3 have the same first element

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
  
  ( ) (a b c) (a b c)

- Both of the following hold:
  - List 1 and 3 have the same first element
  
  (a b c) (d e f) (a b c d e f)

\[
\text{logic}\> (\text{fact (append-to-form ( ) ?x ?x)})
\]
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
  
  () (a b c) (a b c)

• Both of the following hold:
  • List 1 and 3 have the same first element

  (a b c) (d e f) (a b c d e f)

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

  () (a b c) (a b c)

• Both of the following hold:
  • List 1 and 3 have the same first element

  (a b c) (d e f) (a b c d e f)

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

  () (a b c) (a b c)

• Both of the following hold:
  • List 1 and 3 have the same first element

  (a b c) (d e f) (a b c d e f)

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

  () (a b c) (a b c)

• Both of the following hold:
  • List 1 and 3 have the same first element
  • The rest of list 1 and all of list 2 append to form the rest of list 3

  (a b c) (d e f) (a b c d e f)

logic> (fact (append-to-form () ?x ?x))
logic> (fact (append-to-form (?a . ?r) ?y (?a . ?z)))
Example: Append Lists

Two lists append to form a third list if:

• The first list is empty and the second and third are the same
  
  ( ) (a b c) (a b c)

• Both of the following hold:
  • List 1 and 3 have the same first element
  • The rest of list 1 and all of list 2 append to form the rest of list 3
  
  (a b c) (d e f) (a b c d e f)

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

  () (a b c) (a b c)

• Both of the following hold:
  • List 1 and 3 have the same first element
  • The rest of list 1 and all of list 2 append to form the rest of list 3

  (a b c) (d e f) (a b c d e f)

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

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

  () (a b c) (a b c)

• Both of the following hold:
  • List 1 and 3 have the same first element
  • The rest of list 1 and all of list 2 append to form the rest of list 3

  (a b c) (d e f) (a b c d e f)

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

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

\((\text{empty}) (a \ b \ c) (a \ b \ c)\)

• Both of the following hold:
  • List 1 and 3 have the same first element
  • The rest of list 1 and all of list 2 append to form the rest of list 3

\((a \ b \ c) (d \ e \ f) (a \ b \ c \ d \ e \ f)\)

\(\text{logic}\> (\text{fact} \ (\text{append-to-form} \ (\text{empty}) \ ?x \ ?x))\)

\(\text{logic}\> (\text{fact} \ (\text{append-to-form} \ (?a \ . \ ?r) \ ?y \ (?a \ . \ ?z)) \ (\text{append-to-form} \ ?r \ ?y \ ?z))\)