# 61A Extra Lecture 11

# Announcements

# The Logic Language

# The Logic Language

## The Logic Language

The *Logic* language was invented for Structure and Interpretation of Computer Programs

## The Logic Language

The *Logic* language was invented for Structure and Interpretation of Computer Programs

- Based on Prolog (1972)

## The Logic Language

The *Logic* language was invented for Structure and Interpretation of Computer Programs

- Based on Prolog (1972)
- Expressions are facts or queries, which contain relations

## The Logic Language

The *Logic* language was invented for Structure and Interpretation of Computer Programs

- Based on Prolog (1972)
- Expressions are facts or queries, which contain relations
- Expressions and relations are Scheme lists

## The Logic Language

The *Logic* language was invented for Structure and Interpretation of Computer Programs

- Based on Prolog (1972)
- Expressions are facts or queries, which contain relations
- Expressions and relations are Scheme lists
- For example, **(likes john dogs)** is a relation

# 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 the heredity of a pack of dogs

## Simple Facts

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

Let's say I want to track the heredity of a pack of dogs

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 the heredity of a pack of dogs

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 the heredity of a pack of dogs

Language Syntax:

- A relation is a Scheme list
- A fact expression is a Scheme list of relations

## Simple Facts

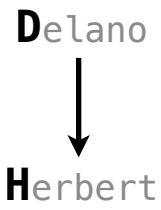
A simple fact expression in the Logic language declares a relation to be true

Let's say I want to track the heredity of a pack of dogs

Language Syntax:

- A relation is a Scheme list
- A fact expression is a Scheme list of relations

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



## Simple Facts

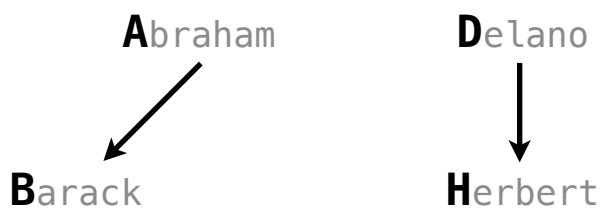
A simple fact expression in the Logic language declares a relation to be true

Let's say I want to track the heredity of a pack of dogs

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

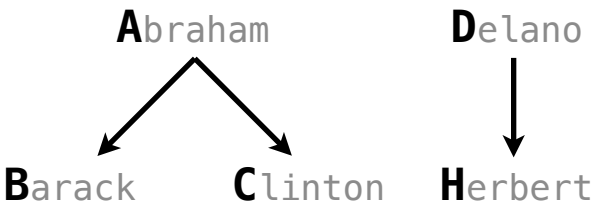
A simple fact expression in the Logic language declares a relation to be true

Let's say I want to track the heredity of a pack of dogs

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))
```



## Simple Facts

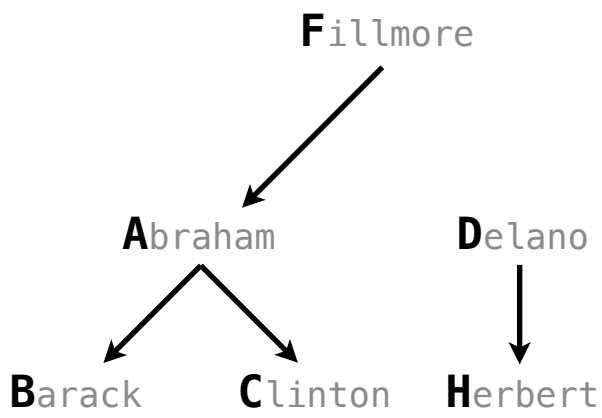
A simple fact expression in the Logic language declares a relation to be true

Let's say I want to track the heredity of a pack of dogs

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

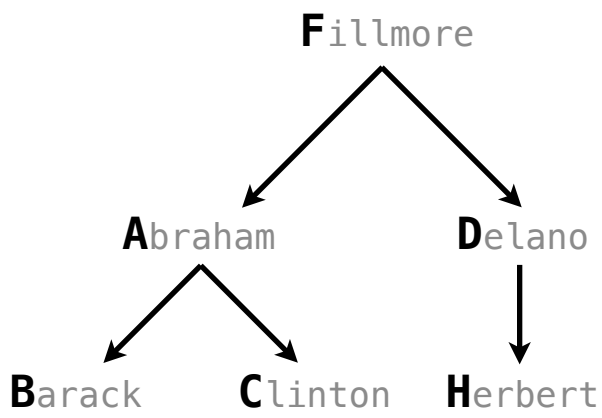
A simple fact expression in the Logic language declares a relation to be true

Let's say I want to track the heredity of a pack of dogs

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))
```



## Simple Facts

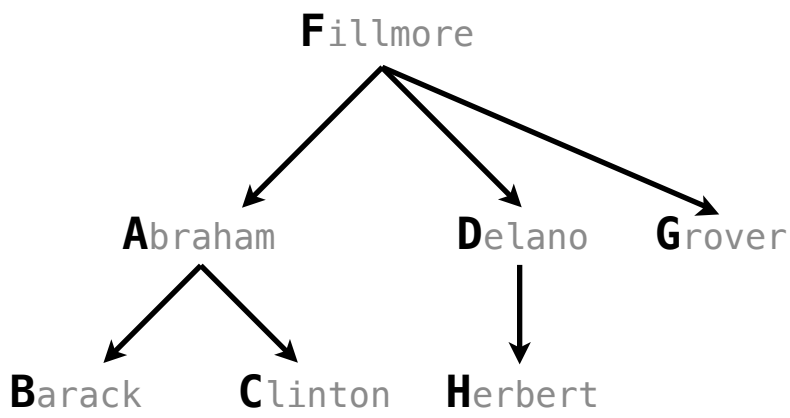
A simple fact expression in the Logic language declares a relation to be true

Let's say I want to track the heredity of a pack of dogs

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

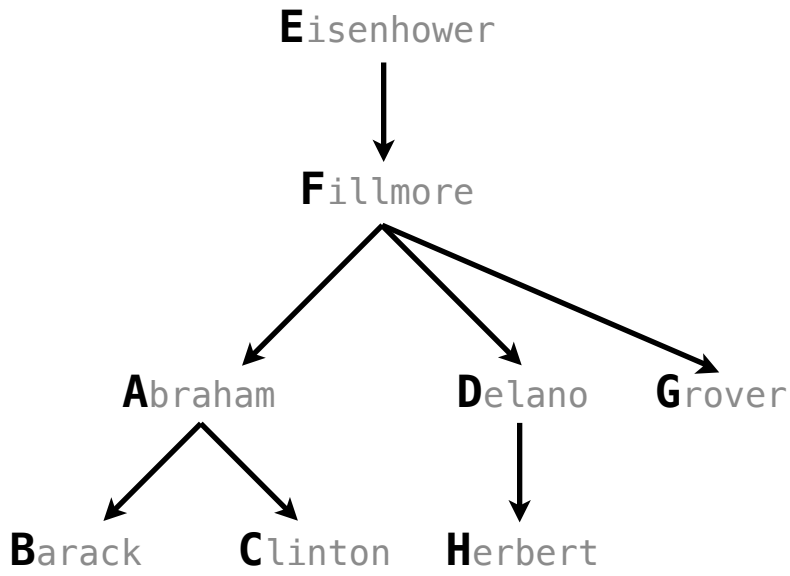
A simple fact expression in the Logic language declares a relation to be true

Let's say I want to track the heredity of a pack of dogs

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.

- *Scheme*: the expression **(abs -3)** calls *abs* on -3. It returns 3.

## Relations are Not Procedure Calls

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

- *Scheme*: the expression **(abs -3)** calls *abs* on -3. It returns 3.
- *Logic*: **(abs -3 3)** asserts that *abs* of -3 is 3.

## Relations are Not Procedure Calls

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

- *Scheme*: the expression **(abs -3)** calls *abs* on -3. It returns 3.
- *Logic*: **(abs -3 3)** asserts that *abs* of -3 is 3.

To assert that 1 + 2 = 3, we use a relation: **(add 1 2 3)**

## Relations are Not Procedure Calls

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

- *Scheme*: the expression **(abs -3)** calls *abs* on -3. It returns 3.
- *Logic*: **(abs -3 3)** asserts that *abs* of -3 is 3.

To assert that 1 + 2 = 3, we use a relation: **(add 1 2 3)**

We can ask the Logic interpreter to complete relations based on known facts.

## Relations are Not Procedure Calls

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

- *Scheme*: the expression **(abs -3)** calls *abs* on -3. It returns 3.
- *Logic*: **(abs -3 3)** asserts that *abs* of -3 is 3.

To assert that 1 + 2 = 3, we use a relation: **(add 1 2 3)**

We can ask the Logic interpreter to complete relations based on known facts.

**(add ? 2 3)**

## Relations are Not Procedure Calls

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

- *Scheme*: the expression **(abs -3)** calls *abs* on -3. It returns 3.
- *Logic*: **(abs -3 3)** asserts that *abs* of -3 is 3.

To assert that 1 + 2 = 3, we use a relation: **(add 1 2 3)**

We can ask the Logic interpreter to complete relations based on known facts.

**(add ? 2 3)** **1**

## Relations are Not Procedure Calls

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

- *Scheme*: the expression **(abs -3)** calls *abs* on -3. It returns 3.
- *Logic*: **(abs -3 3)** asserts that *abs* of -3 is 3.

To assert that 1 + 2 = 3, we use a relation: **(add 1 2 3)**

We can ask the Logic interpreter to complete relations based on known facts.

**(add ? 2 3)** **1**

**(add 1 ? 3)**

## Relations are Not Procedure Calls

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

- *Scheme*: the expression **(abs -3)** calls *abs* on -3. It returns 3.
- *Logic*: **(abs -3 3)** asserts that *abs* of -3 is 3.

To assert that 1 + 2 = 3, we use a relation: **(add 1 2 3)**

We can ask the Logic interpreter to complete relations based on known facts.

**(add ? 2 3)** **1**

**(add 1 ? 3)** **2**

## Relations are Not Procedure Calls

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

- *Scheme*: the expression **(abs -3)** calls *abs* on -3. It returns 3.
- *Logic*: **(abs -3 3)** asserts that *abs* of -3 is 3.

To assert that 1 + 2 = 3, we use a relation: **(add 1 2 3)**

We can ask the Logic interpreter to complete relations based on known facts.

```
(add ? 2 3)     1

(add 1 ? 3)     2

(add 1 2 ?)
```

## Relations are Not Procedure Calls

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

- *Scheme*: the expression **(abs -3)** calls *abs* on -3. It returns 3.
- *Logic*: **(abs -3 3)** asserts that *abs* of -3 is 3.

To assert that 1 + 2 = 3, we use a relation: **(add 1 2 3)**

We can ask the Logic interpreter to complete relations based on known facts.

| | |
|---|---|
| **(add ? 2 3)** | **1** |
| **(add 1 ? 3)** | **2** |
| **(add 1 2 ?)** | **3** |

## Relations are Not Procedure Calls

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

- *Scheme*: the expression **(abs -3)** calls *abs* on -3. It returns 3.
- *Logic*: **(abs -3 3)** asserts that *abs* of -3 is 3.

To assert that 1 + 2 = 3, we use a relation: **(add 1 2 3)**

We can ask the Logic interpreter to complete relations based on known facts.

| | |
|---|---|
| **(add ? 2 3)** | **1** |
| **(add 1 ? 3)** | **2** |
| **(add 1 2 ?)** | **3** |
| **( ? 1 2 3)** | |

## Relations are Not Procedure Calls

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

- *Scheme*: the expression **(abs -3)** calls *abs* on -3. It returns 3.
- *Logic*: **(abs -3 3)** asserts that *abs* of -3 is 3.

To assert that 1 + 2 = 3, we use a relation: **(add 1 2 3)**

We can ask the Logic interpreter to complete relations based on known facts.

| | |
|---|---|
| **(add ? 2 3)** | **1** |
| **(add 1 ? 3)** | **2** |
| **(add 1 2 ?)** | **3** |
| **( ? 1 2 3)** | **add** |

# Queries

# Queries

## Queries

A *query* contains one or more relations that may contain variables.

## Queries

A *query* contains one or more relations that may contain variables.

Variables are symbols starting with **?**

## Queries

A *query* contains one or more relations that may contain variables.

Variables are symbols starting with **?**

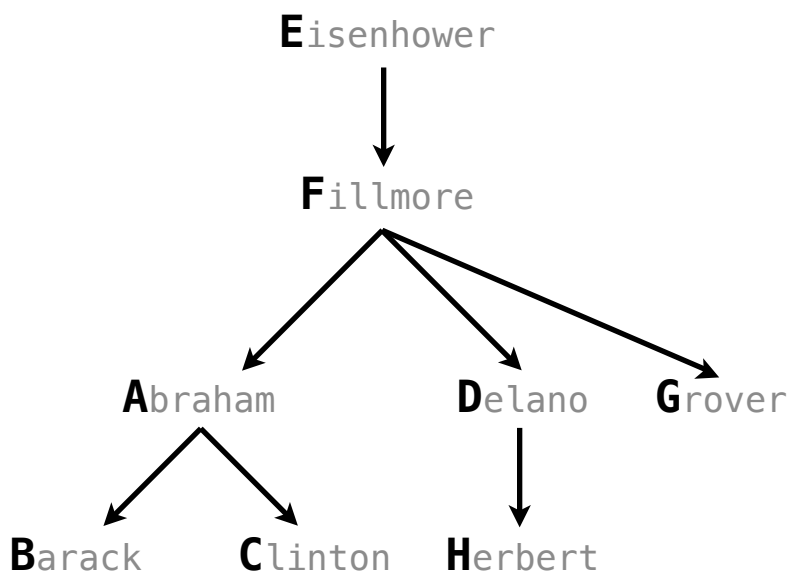
```
logic> (fact (parent delano herbert))
logic> (fact (parent abraham barack))
logic> (fact (parent abraham clinton))
logic> (fact (parent fillmore abraham))
logic> (fact (parent fillmore delano))
logic> (fact (parent fillmore grover))
logic> (fact (parent eisenhower fillmore))
```

## Queries

A *query* contains one or more relations that may contain variables.

Variables are 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))
```

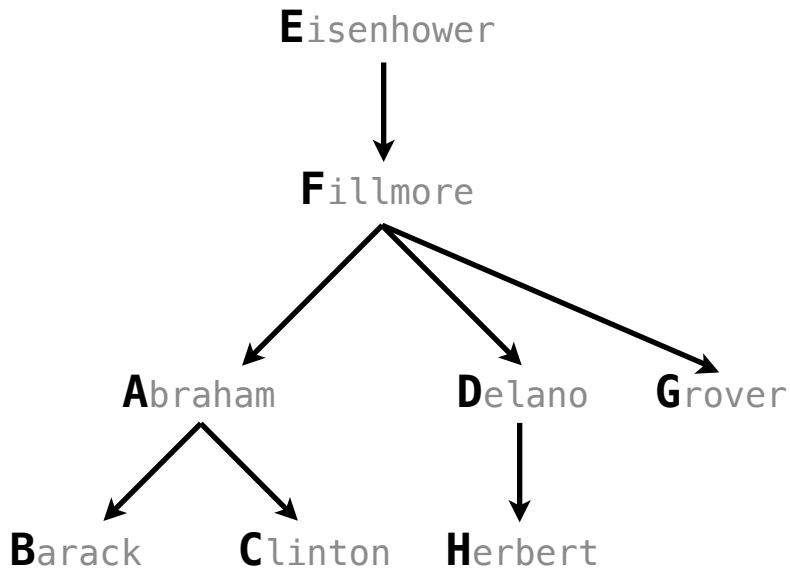


## Queries

A *query* contains one or more relations that may contain variables.

Variables are 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 ?puppy))
```



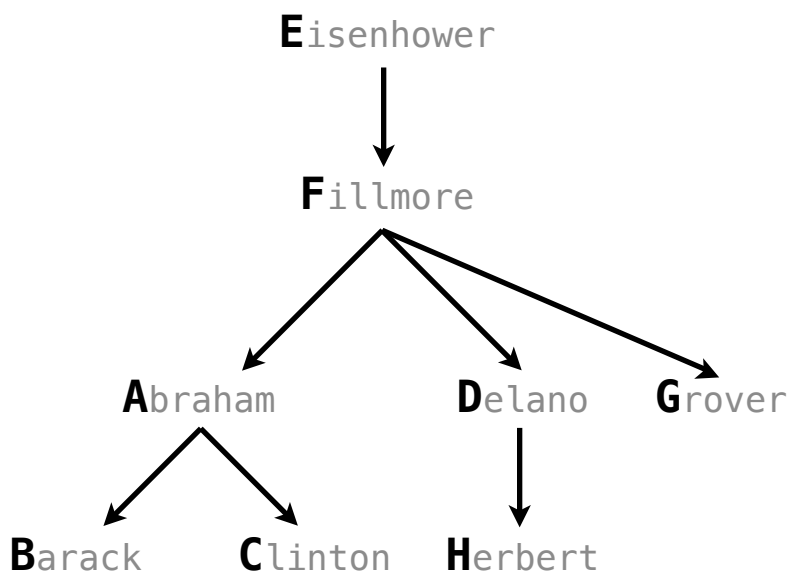
## Queries

A *query* contains one or more relations that may contain variables.

Variables are 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 ?puppy))
```

A variable can have any name



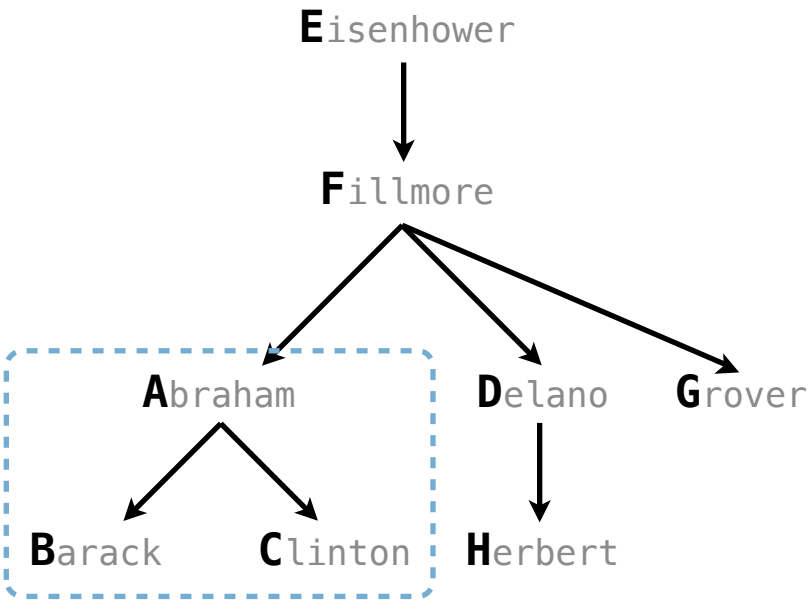
## Queries

A *query* contains one or more relations that may contain variables.

Variables are 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 ?puppy))
```

A variable can have any name



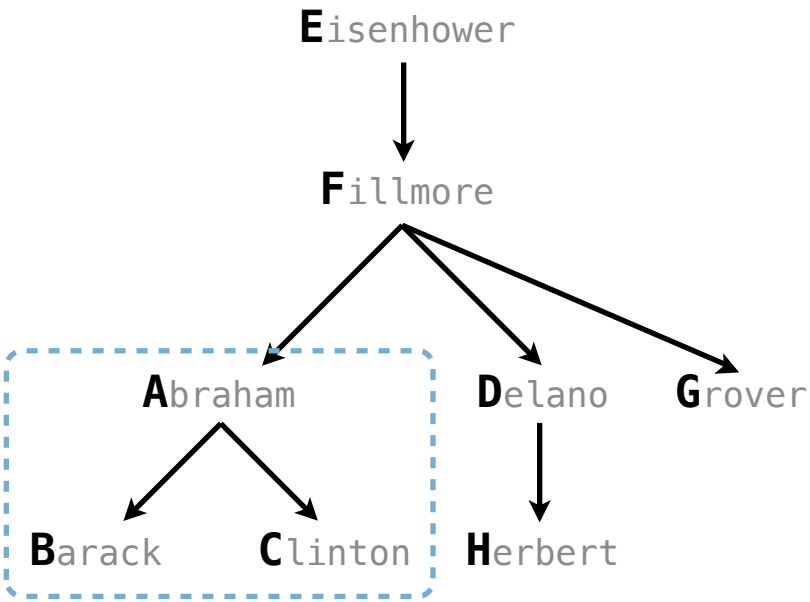
## Queries

A *query* contains one or more relations that may contain variables.

Variables are 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 ?puppy))
Success!
```

A variable can have any name



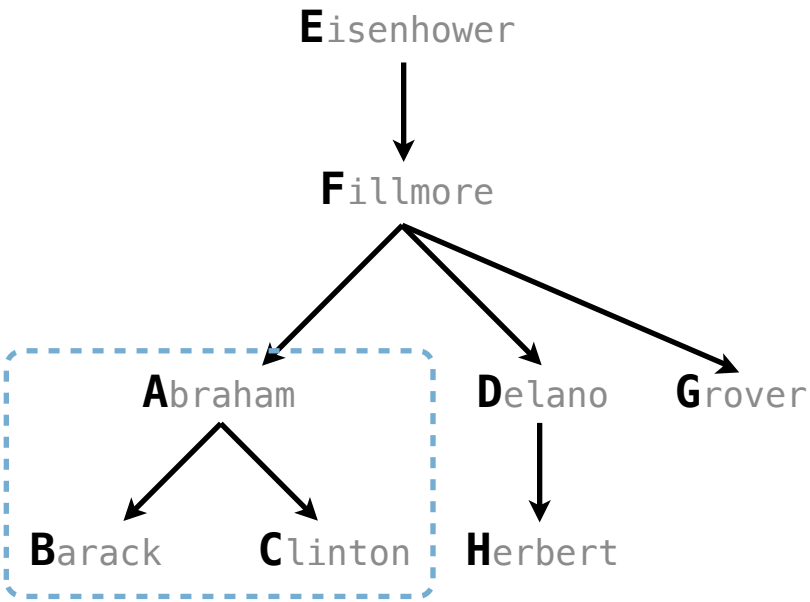
## Queries

A *query* contains one or more relations that may contain variables.

Variables are 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 ?puppy))
Success!
puppy: barack
```

A variable can have any name



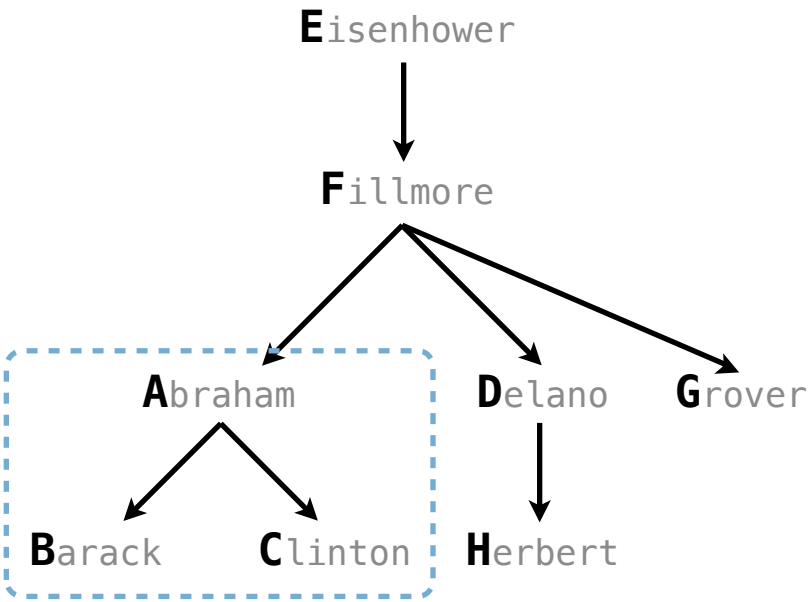
## Queries

A *query* contains one or more relations that may contain variables.

Variables are 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 ?puppy))
Success!
puppy: barack
puppy: clinton
```

A variable can have any name



## Queries

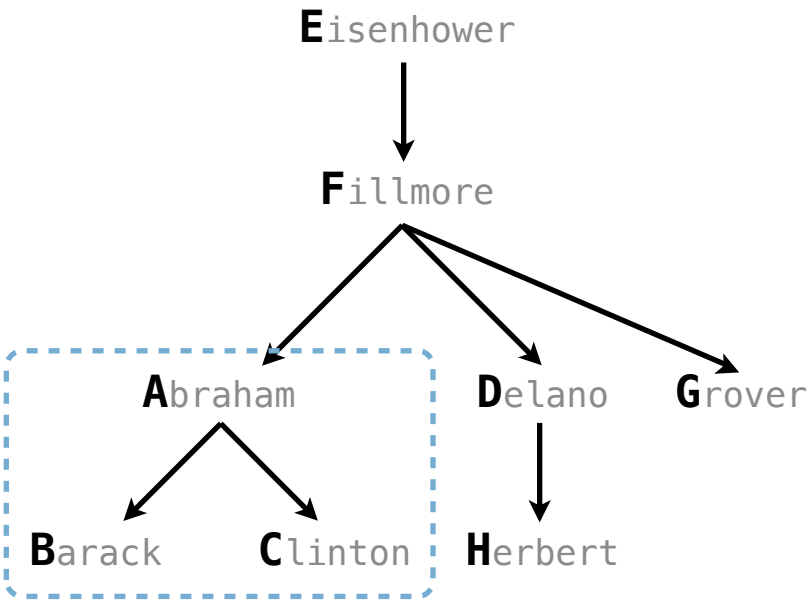
A *query* contains one or more relations that may contain variables.

Variables are 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 ?puppy))
Success!
puppy: barack
puppy: clinton
```

A variable can have any name

Each line is an assignment of variables to values



## Queries

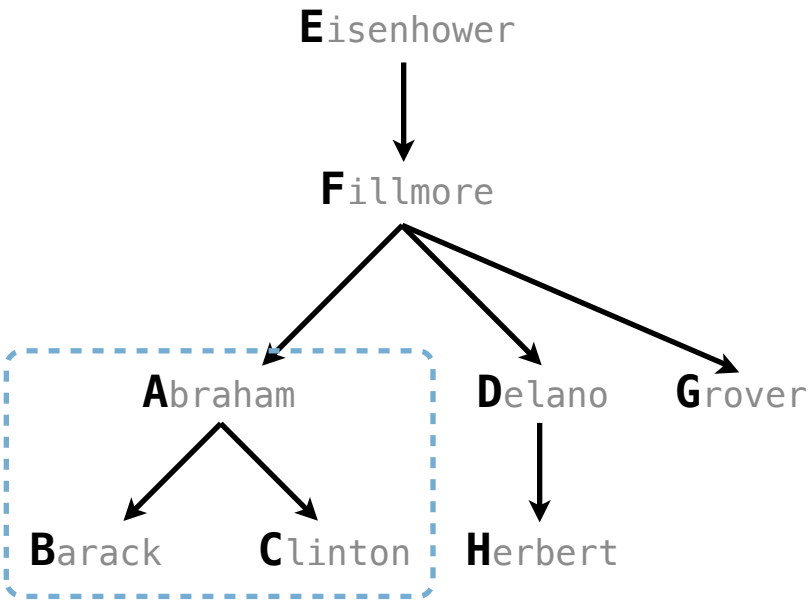
A *query* contains one or more relations that may contain variables.

Variables are 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 ?puppy))
Success!
puppy: barack
puppy: clinton
```

A variable can have any name

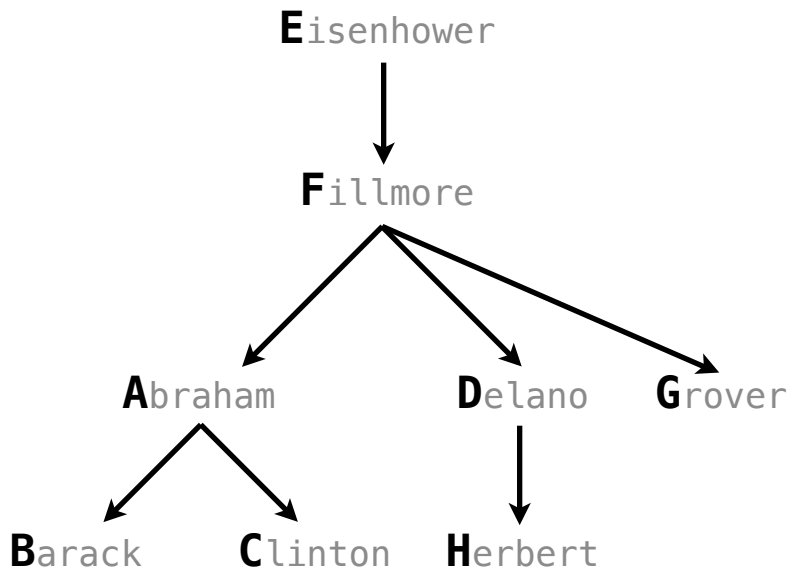
Each line is an assignment of variables to values




(Demo)

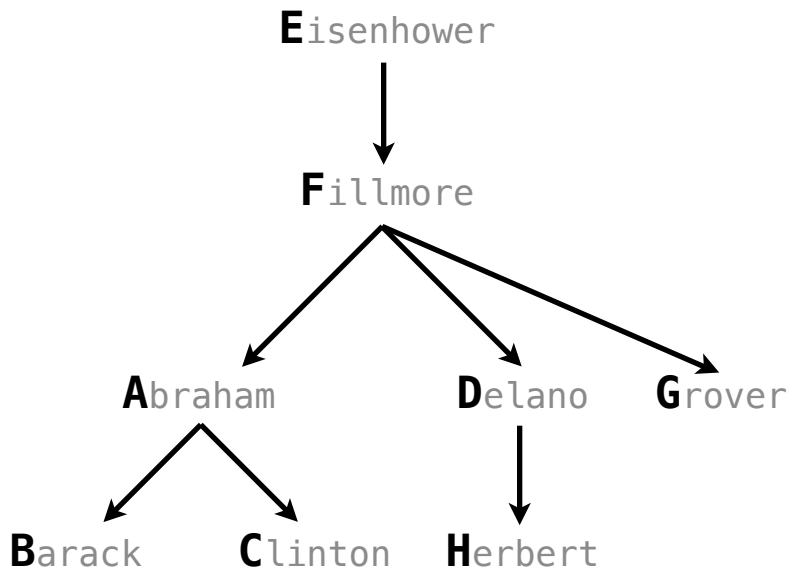
# Compound Facts and Queries

# Compound Facts



## Compound Facts

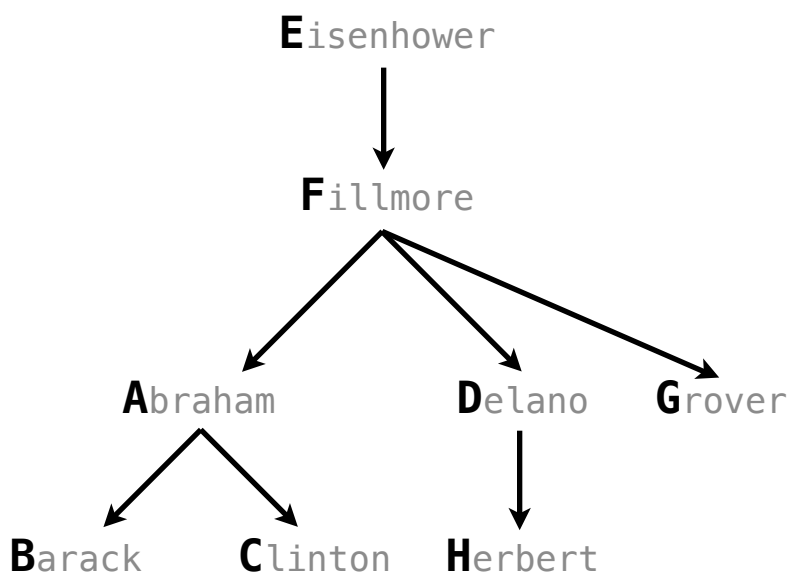
A fact can include multiple relations and variables as well.



## Compound Facts

A fact can include multiple relations and variables as well.

```
(fact <conclusion> <hypothesis0> <hypothesis1> ... <hypothesisN>)
```

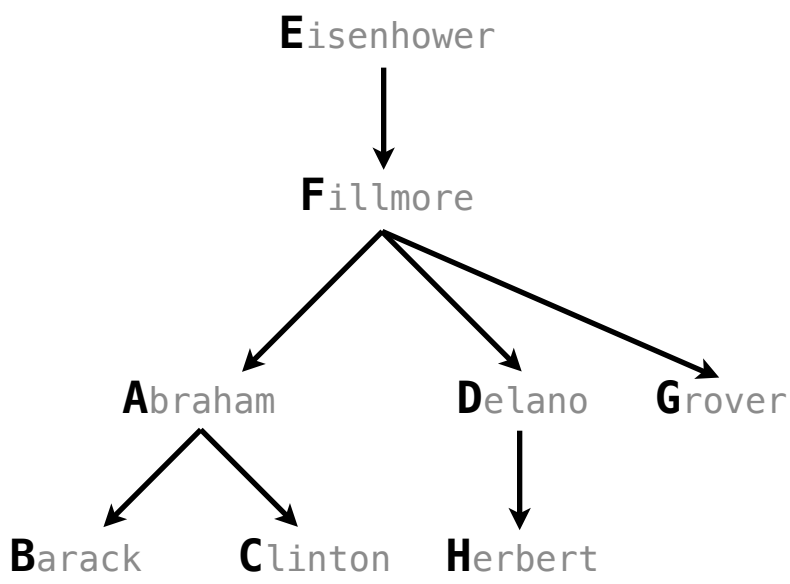


## Compound Facts

A fact can include multiple relations and variables as well.

(fact <conclusion> <hypothesis$_0$> <hypothesis$_1$> ... <hypothesis$_N$>)

Means <conclusion> is true if all the <hypothesis$_K$> are true.



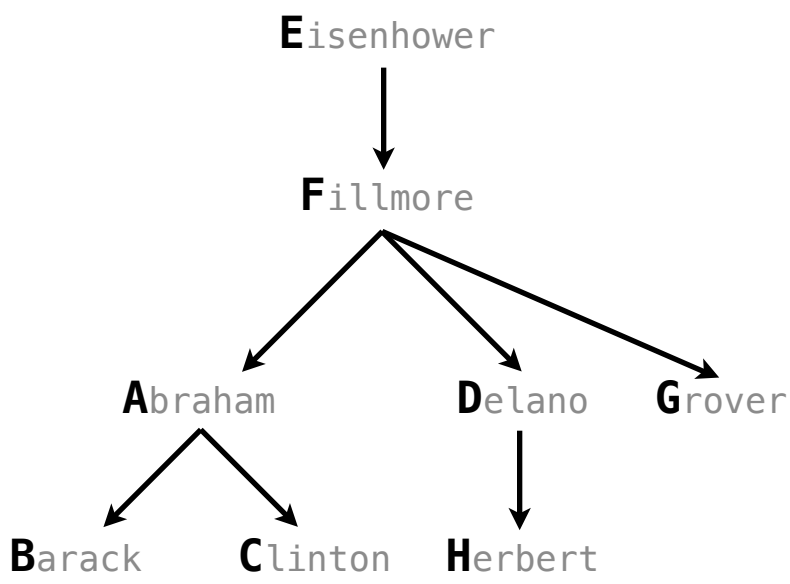
## Compound Facts

A fact can include multiple relations and variables as well.

(fact <conclusion> <hypothesis$_0$> <hypothesis$_1$> ... <hypothesis$_N$>)

Means <conclusion> is true if all the <hypothesis$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))
```



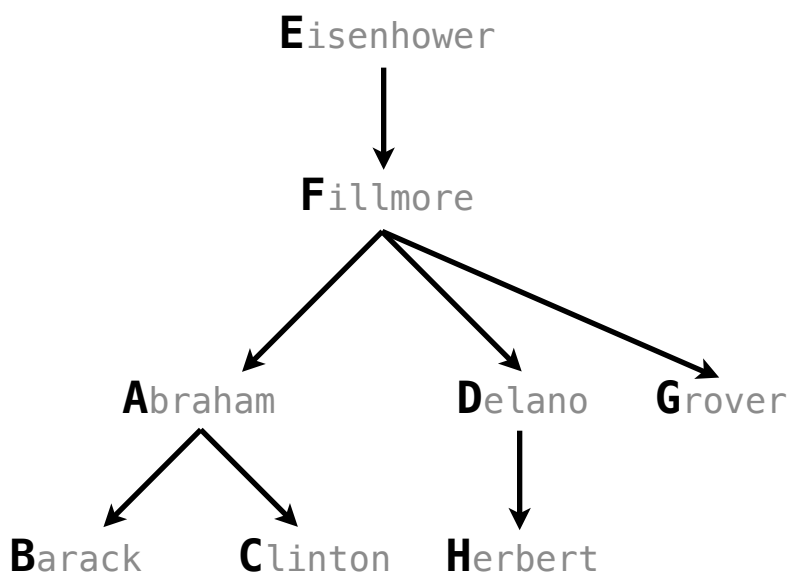
## Compound Facts

A fact can include multiple relations and variables as well.

(fact <conclusion> <hypothesis$_0$> <hypothesis$_1$> ... <hypothesis$_N$>)

Means <conclusion> is true if all the <hypothesis$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))
```



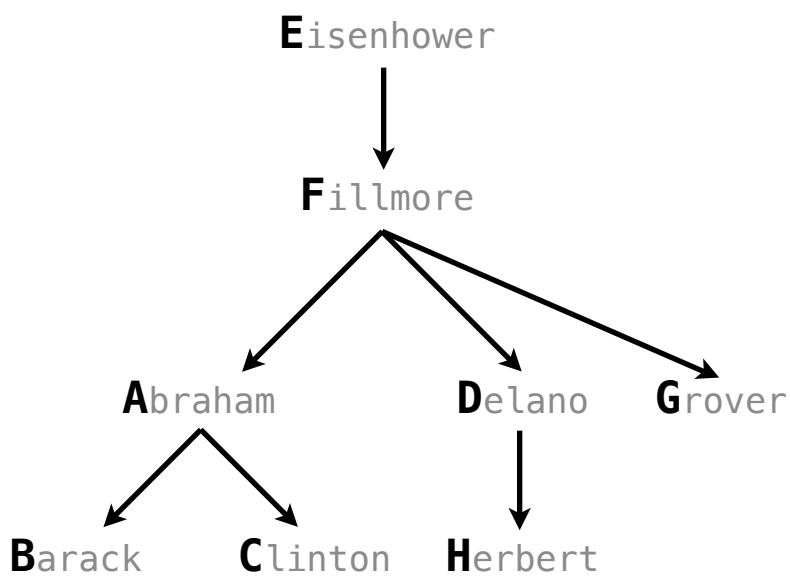
## Compound Facts

A fact can include multiple relations and variables as well.

(fact <conclusion> <hypothesis$_0$> <hypothesis$_1$> ... <hypothesis$_N$>)

Means <conclusion> is true if all the <hypothesis$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

logic> (query (child herbert delano))
```



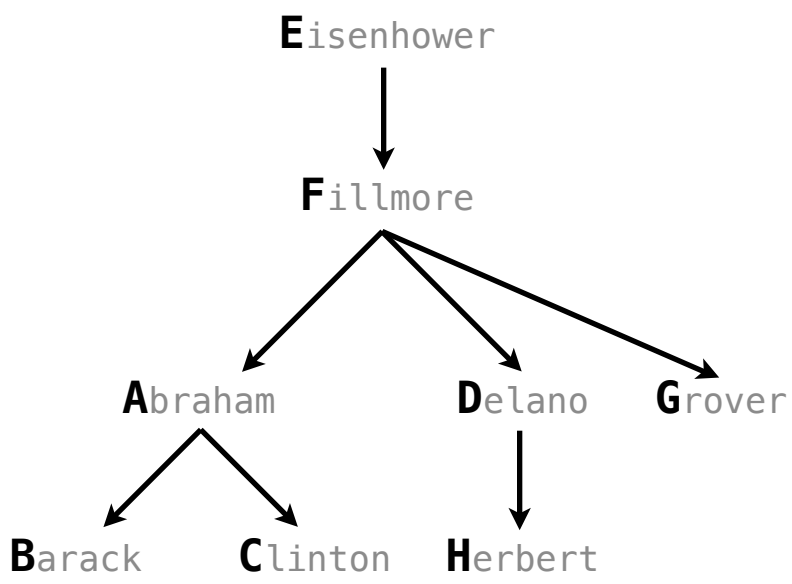
## Compound Facts

A fact can include multiple relations and variables as well.

(fact <conclusion> <hypothesis$_0$> <hypothesis$_1$> ... <hypothesis$_N$>)

Means <conclusion> is true if all the <hypothesis$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

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



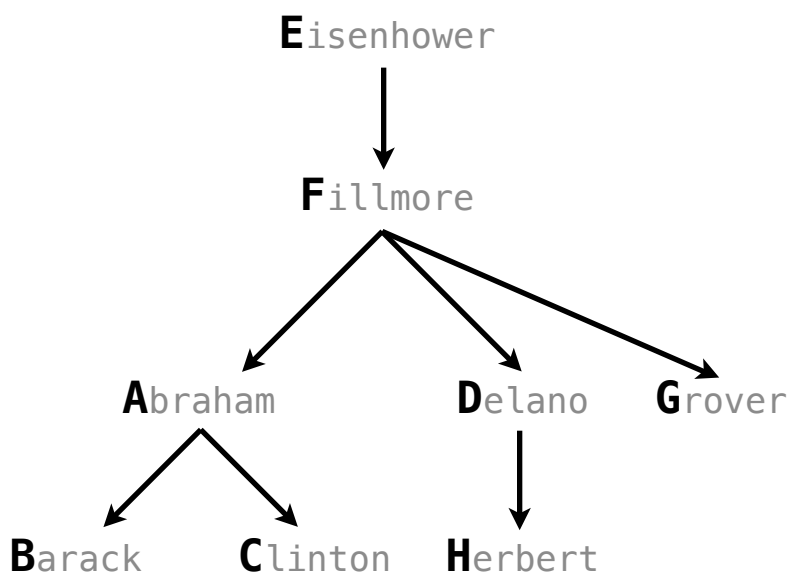
## Compound Facts

A fact can include multiple relations and variables as well.

(fact <conclusion> <hypothesis$_0$> <hypothesis$_1$> ... <hypothesis$_N$>)

Means <conclusion> is true if all the <hypothesis$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

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



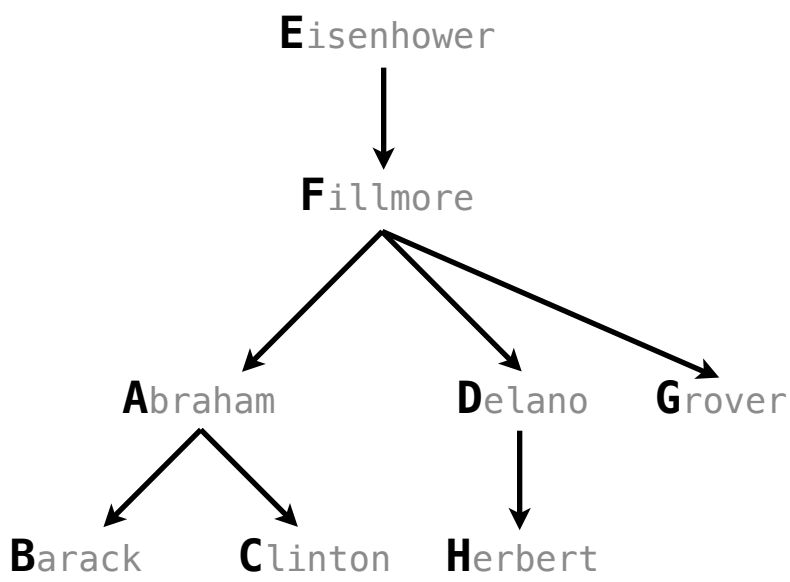
## Compound Facts

A fact can include multiple relations and variables as well.

(fact <conclusion> <hypothesis$_0$> <hypothesis$_1$> ... <hypothesis$_N$>)

Means <conclusion> is true if all the <hypothesis$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

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

logic> (query (child eisenhower clinton))
```



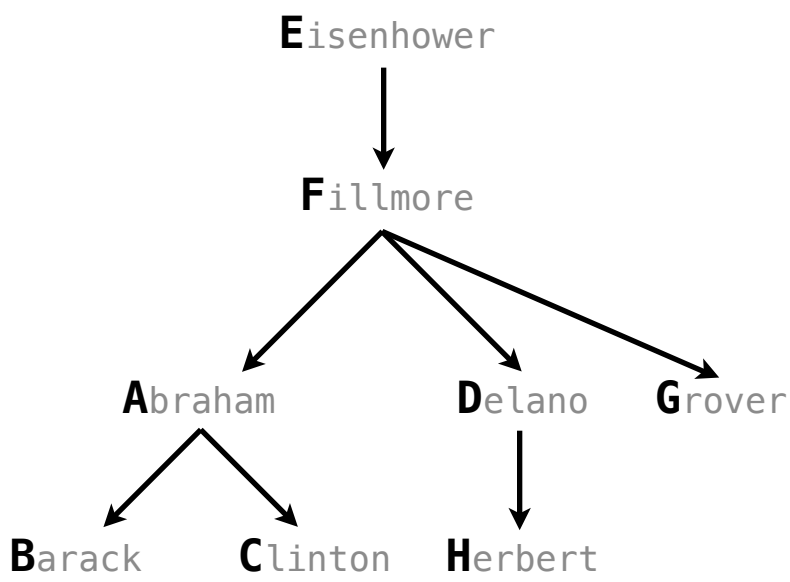
## Compound Facts

A fact can include multiple relations and variables as well.

(fact <conclusion> <hypothesis$_0$> <hypothesis$_1$> ... <hypothesis$_N$>)

Means <conclusion> is true if all the <hypothesis$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

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

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



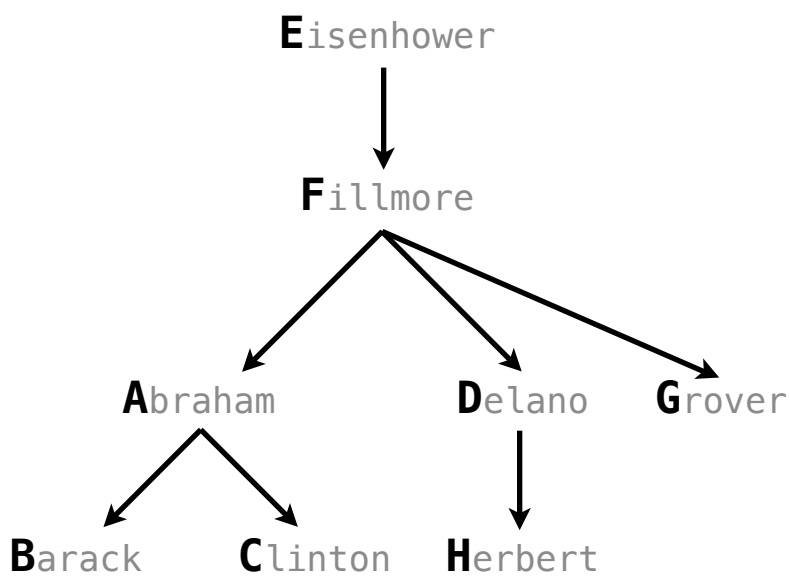
## Compound Facts

A fact can include multiple relations and variables as well.

(fact <conclusion> <hypothesis$_0$> <hypothesis$_1$> ... <hypothesis$_N$>)

Means <conclusion> is true if all the <hypothesis$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

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

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

logic> (query (child ?kid fillmore))
```



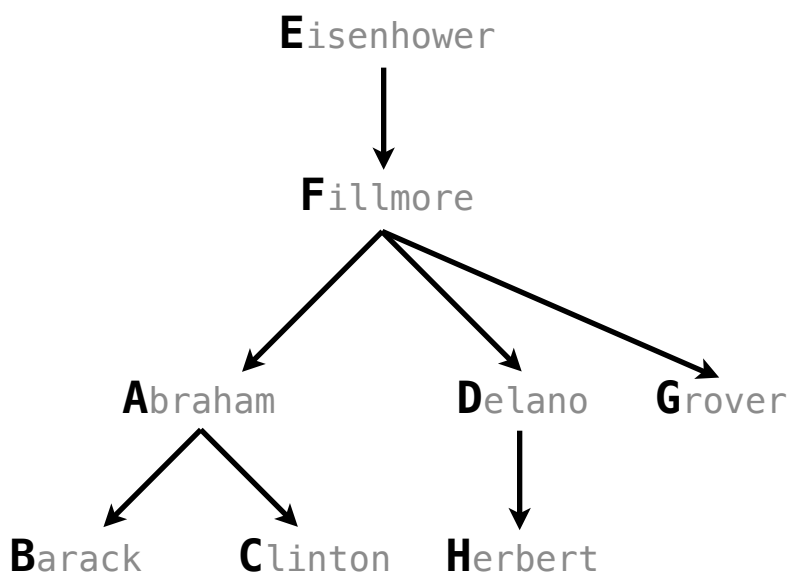
## Compound Facts

A fact can include multiple relations and variables as well.

(fact <conclusion> <hypothesis$_0$> <hypothesis$_1$> ... <hypothesis$_N$>)

Means <conclusion> is true if all the <hypothesis$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

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

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

logic> (query (child ?kid fillmore))
Success!
```



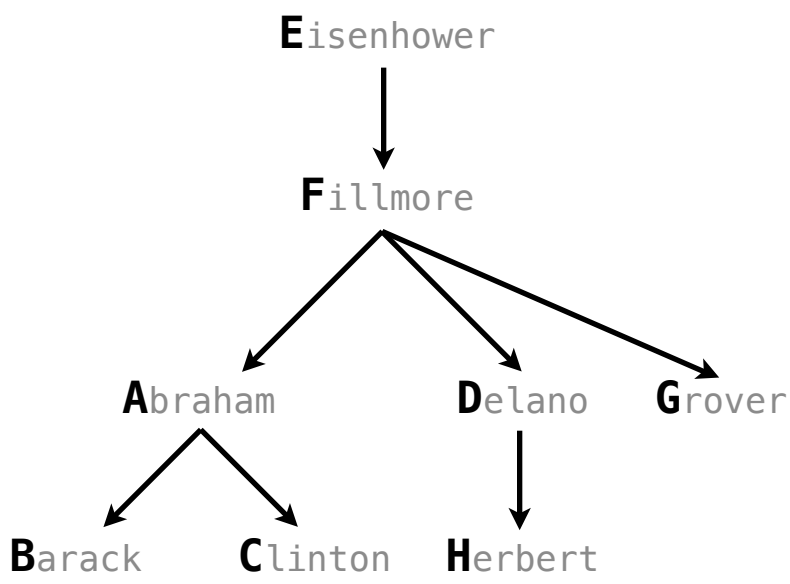
## Compound Facts

A fact can include multiple relations and variables as well.

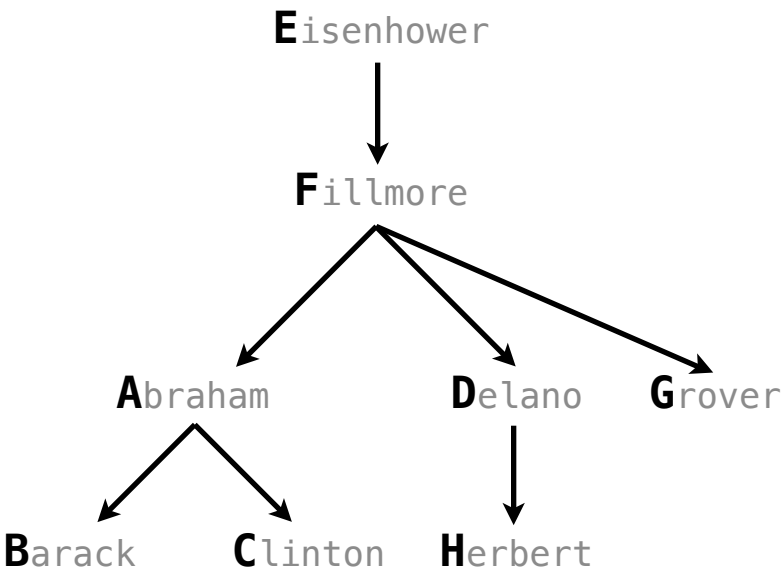
(fact <conclusion> <hypothesis$_0$> <hypothesis$_1$> ... <hypothesis$_N$>)

Means <conclusion> is true if all the <hypothesis$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

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

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

logic> (query (child ?kid fillmore))
Success!
kid: abraham
```

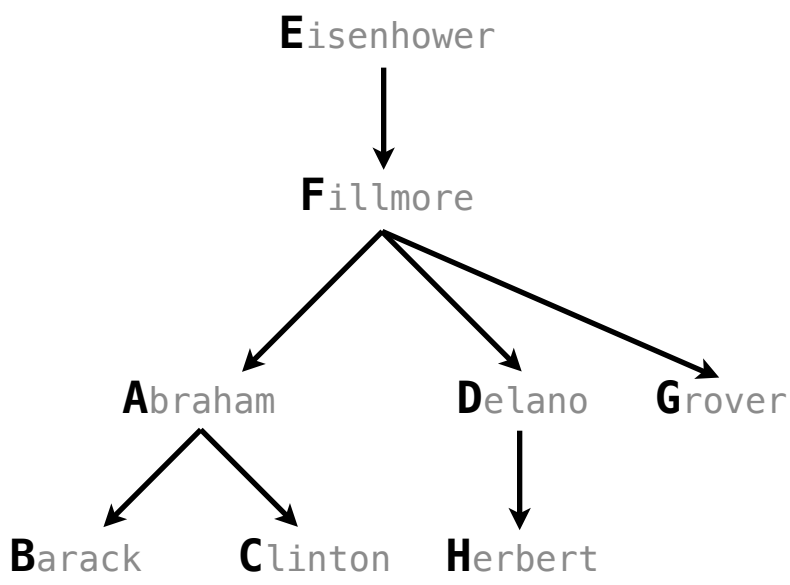


# Compound Queries



## Compound Queries

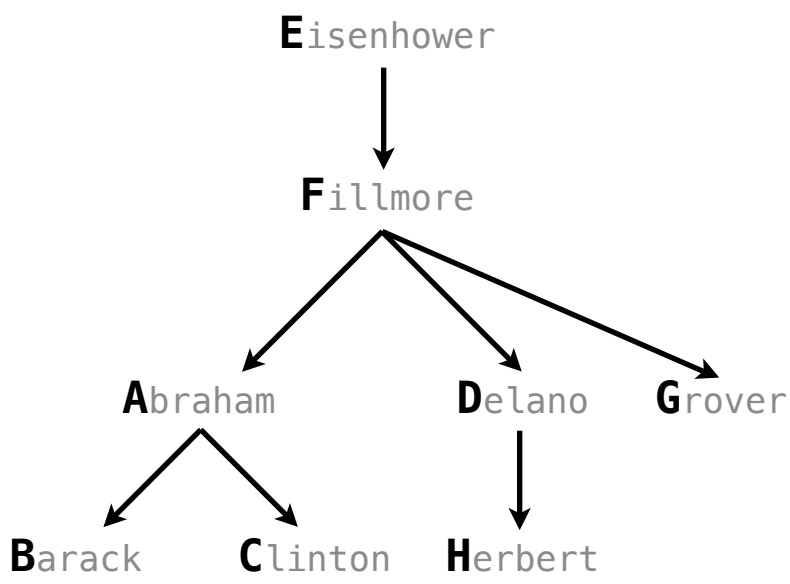
An assignment must satisfy all relations in a query.



## Compound Queries

An assignment must satisfy all relations in a query.

(query <relation$_0$> <relation$_1$> ... <relation$_N$>)

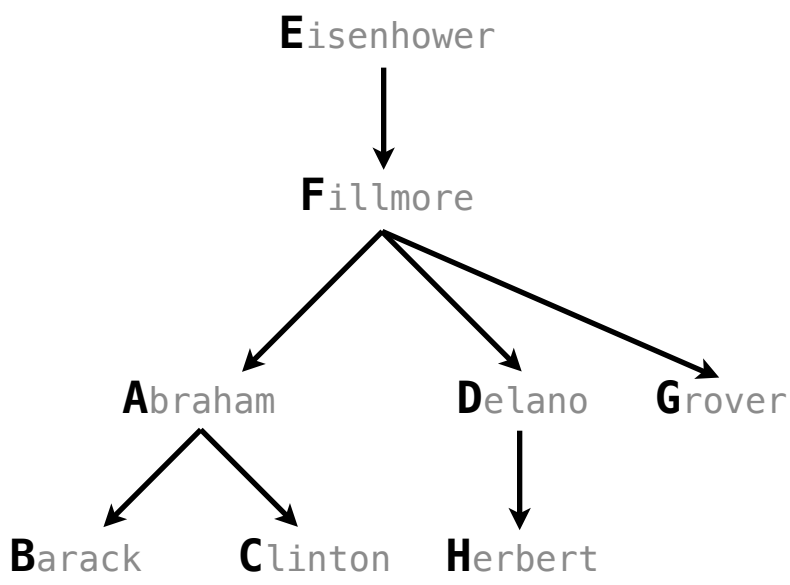


## Compound Queries

An assignment must satisfy all relations in a query.

(query <relation$_0$> <relation$_1$> ... <relation$_N$>)

is satisfied if all the <relation$_K$> are true.



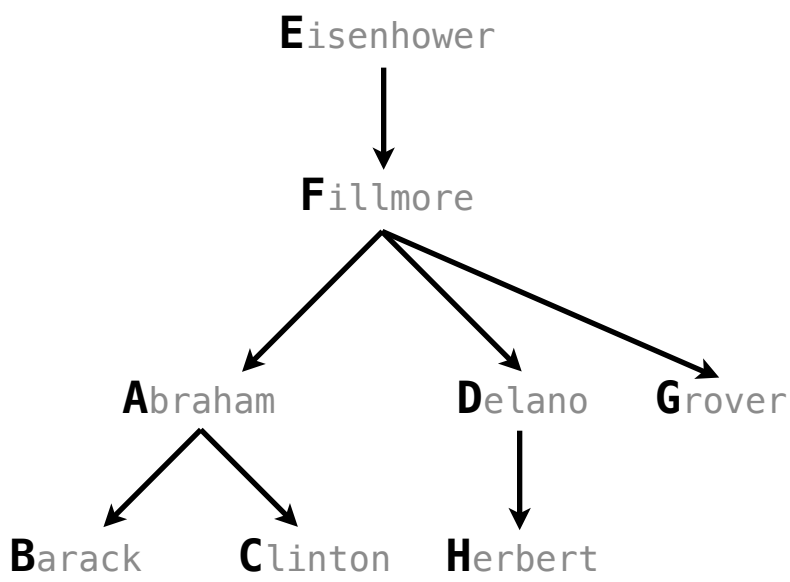
## Compound Queries

An assignment must satisfy all relations in a query.

(query <relation$_0$> <relation$_1$> ... <relation$_N$>)

is satisfied if all the <relation$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))
```



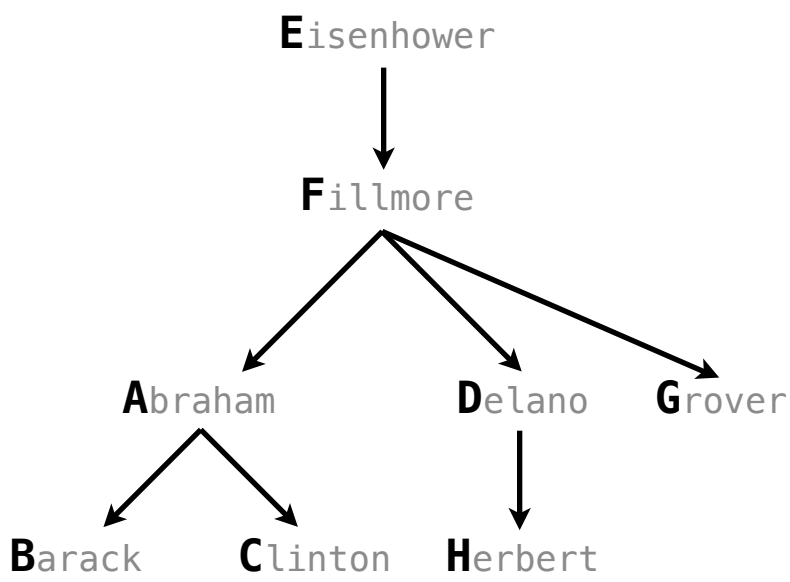
## Compound Queries

An assignment must satisfy all relations in a query.

(query <relation$_0$> <relation$_1$> ... <relation$_N$>)

is satisfied if all the <relation$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))
```



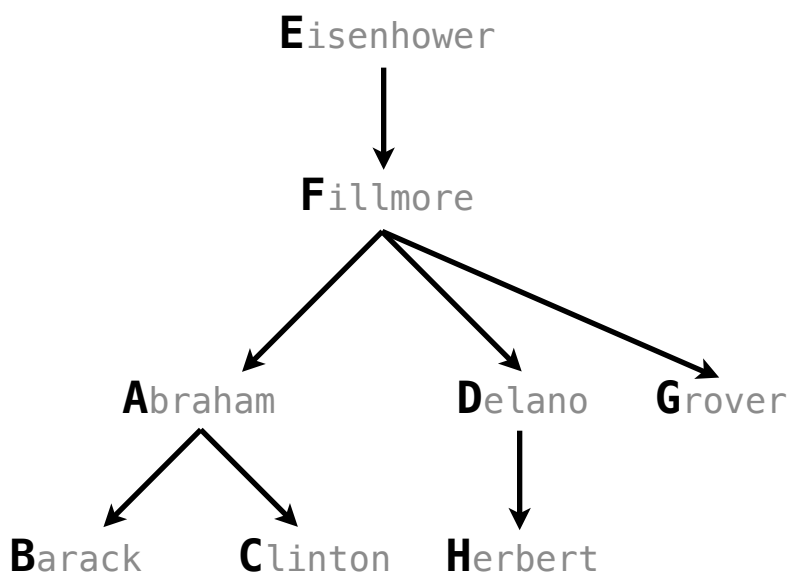
## Compound Queries

An assignment must satisfy all relations in a query.

(query <relation$_0$> <relation$_1$> ... <relation$_N$>)

is satisfied if all the <relation$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

logic> (query (parent ?grampa ?kid)
```



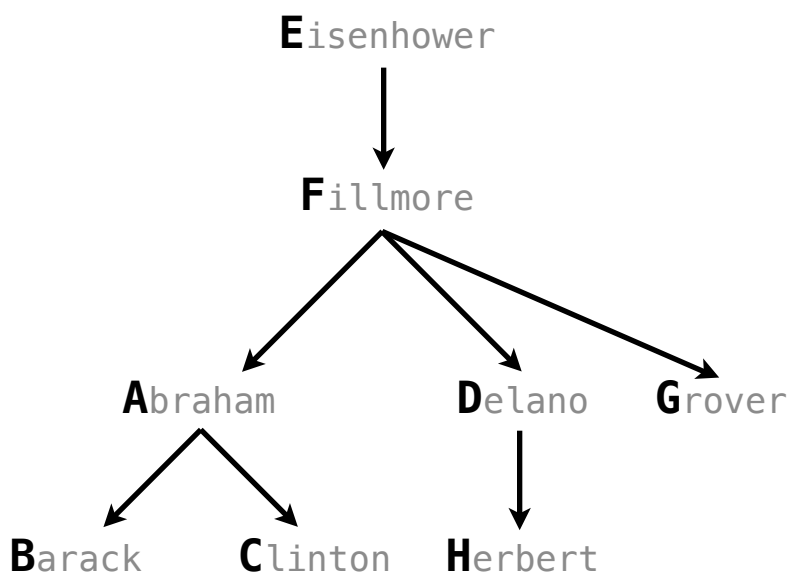
## Compound Queries

An assignment must satisfy all relations in a query.

(query <relation$_0$> <relation$_1$> ... <relation$_N$>)

is satisfied if all the <relation$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

logic> (query (parent ?grampa ?kid)
              (child clinton ?kid))
```



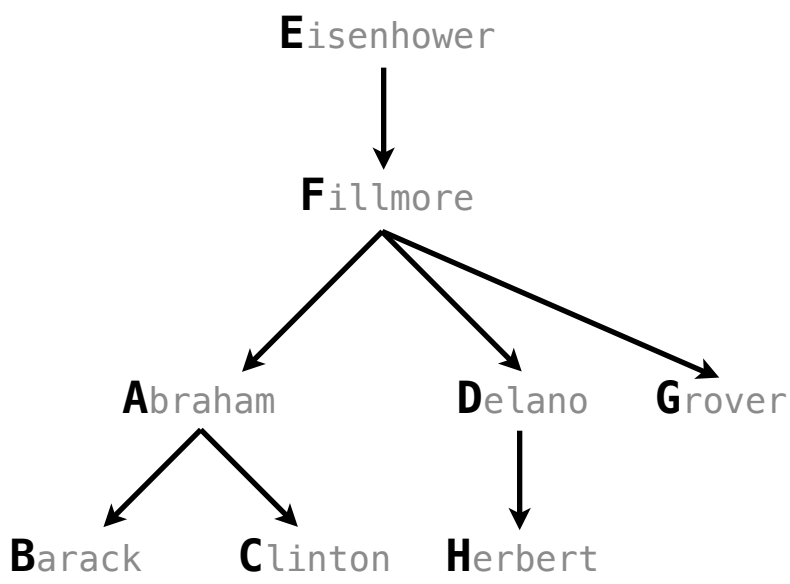
## Compound Queries

An assignment must satisfy all relations in a query.

(query <relation$_0$> <relation$_1$> ... <relation$_N$>)

is satisfied if all the <relation$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

logic> (query (parent ?grampa ?kid)
              (child clinton ?kid))
Success!
```



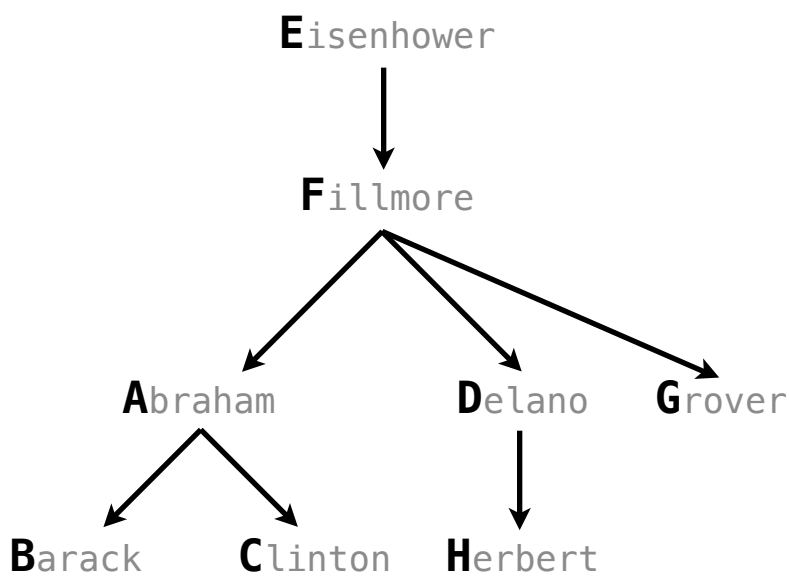
## Compound Queries

An assignment must satisfy all relations in a query.

(query <relation$_0$> <relation$_1$> ... <relation$_N$>)

is satisfied if all the <relation$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

logic> (query (parent ?grampa ?kid)
              (child clinton ?kid))
Success!
grampa: fillmore    kid: abraham
```



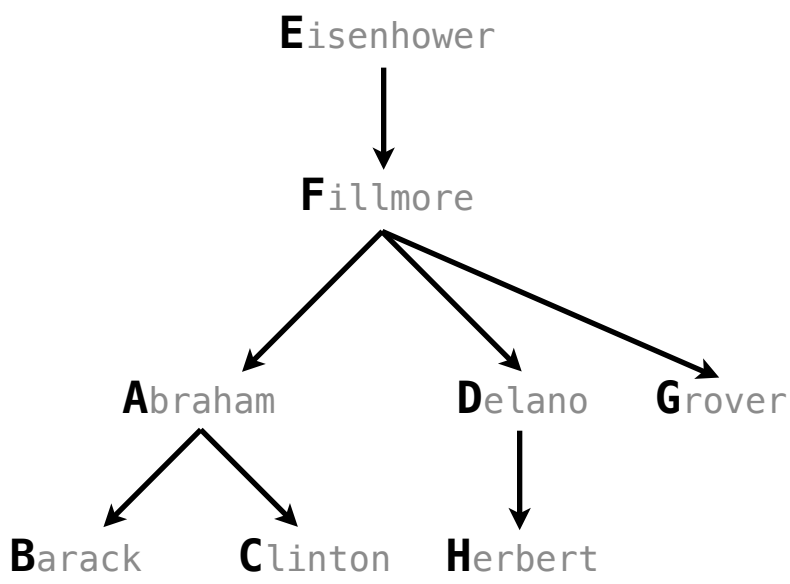
## Compound Queries

An assignment must satisfy all relations in a query.

(query <relation$_0$> <relation$_1$> ... <relation$_N$>)

is satisfied if all the <relation$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

logic> (query (parent ?grampa ?kid)
              (child clinton ?kid))
Success!
grampa: fillmore    kid: abraham
```



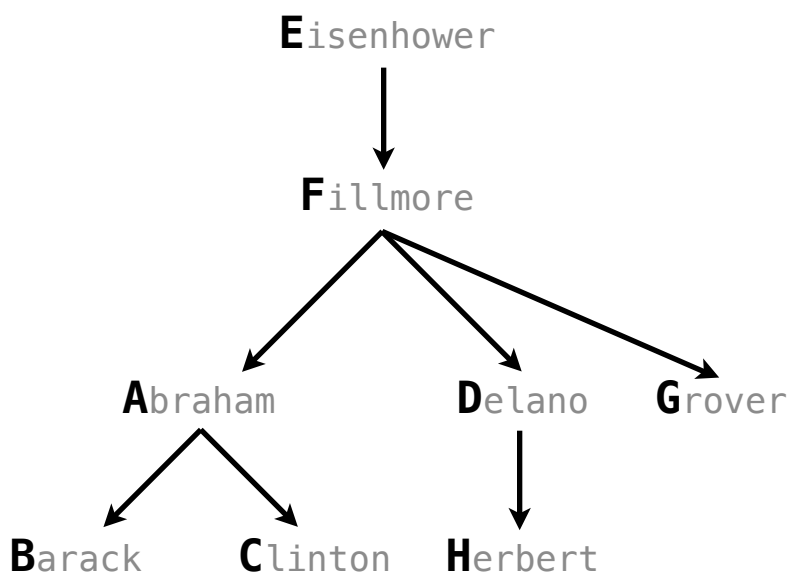
## Compound Queries

An assignment must satisfy all relations in a query.

(query <relation$_0$> <relation$_1$> ... <relation$_N$>)

is satisfied if all the <relation$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

logic> (query (parent ?grampa ?kid)
              (child clinton ?kid))
Success!
grampa: fillmore    kid: abraham

logic> (query (child ?y ?x)
```



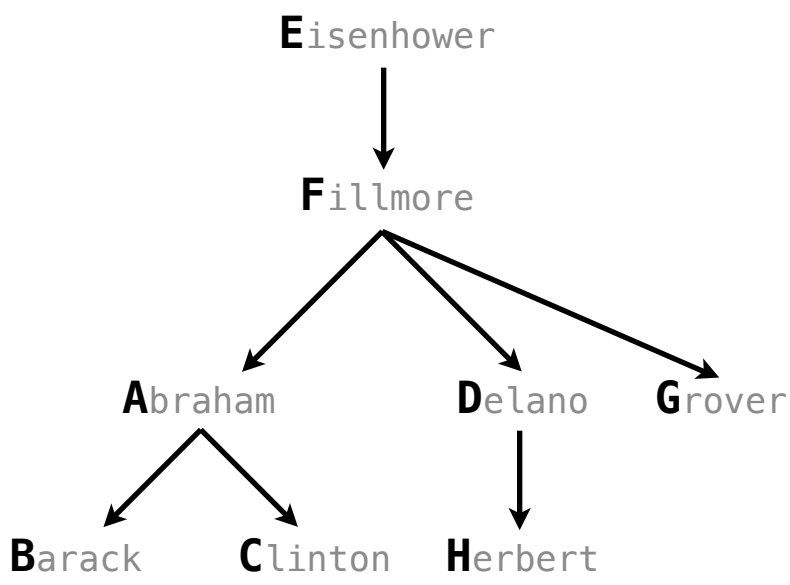
## Compound Queries

An assignment must satisfy all relations in a query.

(query <relation$_0$> <relation$_1$> ... <relation$_N$>)

is satisfied if all the <relation$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

logic> (query (parent ?grampa ?kid)
              (child clinton ?kid))
Success!
grampa: fillmore    kid: abraham

logic> (query (child ?y ?x)
              (child ?x eisenhower))
```



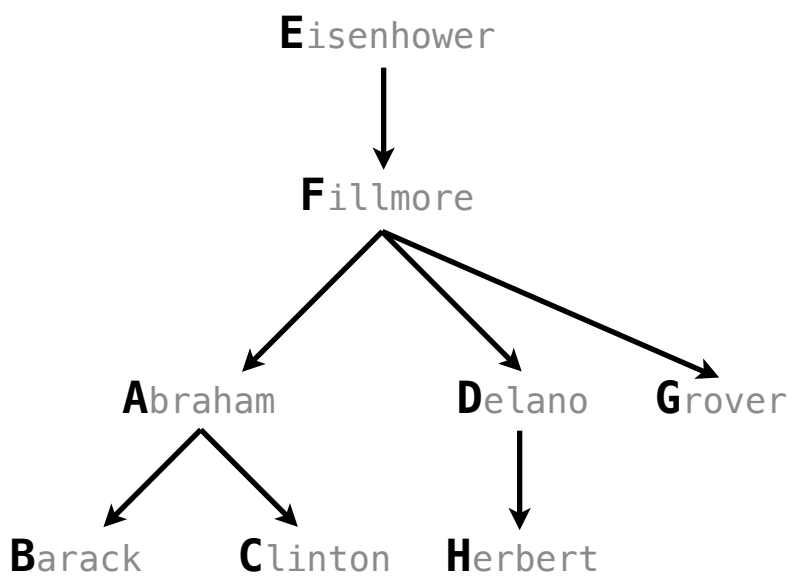
## Compound Queries

An assignment must satisfy all relations in a query.

(query <relation$_0$> <relation$_1$> ... <relation$_N$>)

is satisfied if all the <relation$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

logic> (query (parent ?grampa ?kid)
              (child clinton ?kid))
Success!
grampa: fillmore    kid: abraham

logic> (query (child ?y ?x)
              (child ?x eisenhower))
Success!
```



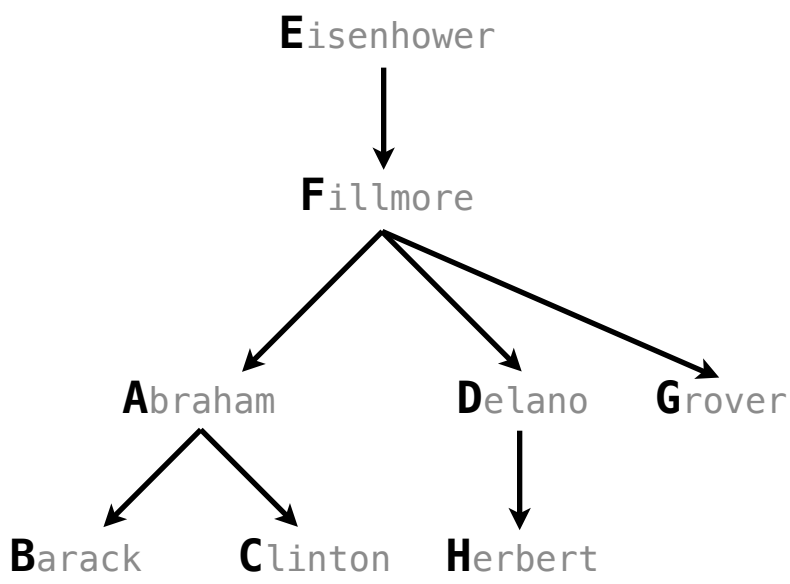
## Compound Queries

An assignment must satisfy all relations in a query.

(query <relation$_0$> <relation$_1$> ... <relation$_N$>)

is satisfied if all the <relation$_K$> are true.

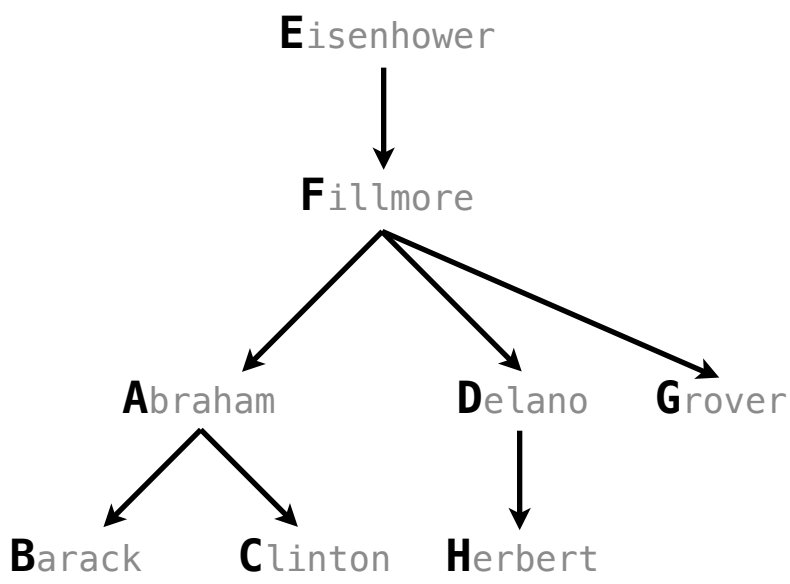
```
logic> (fact (child ?c ?p) (parent ?p ?c))

logic> (query (parent ?grampa ?kid)
              (child clinton ?kid))
Success!
grampa: fillmore    kid: abraham

logic> (query (child ?y ?x)
              (child ?x eisenhower))
Success!
y: abraham    x: fillmore
```



# Recursive Facts

## Recursive Facts

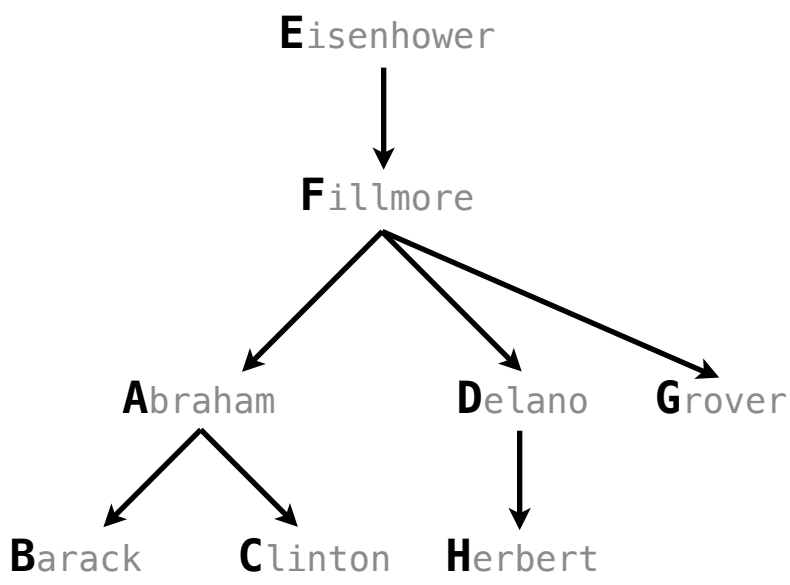
A fact is recursive if the same relation is mentioned in a hypothesis and the conclusion.



## Recursive Facts

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

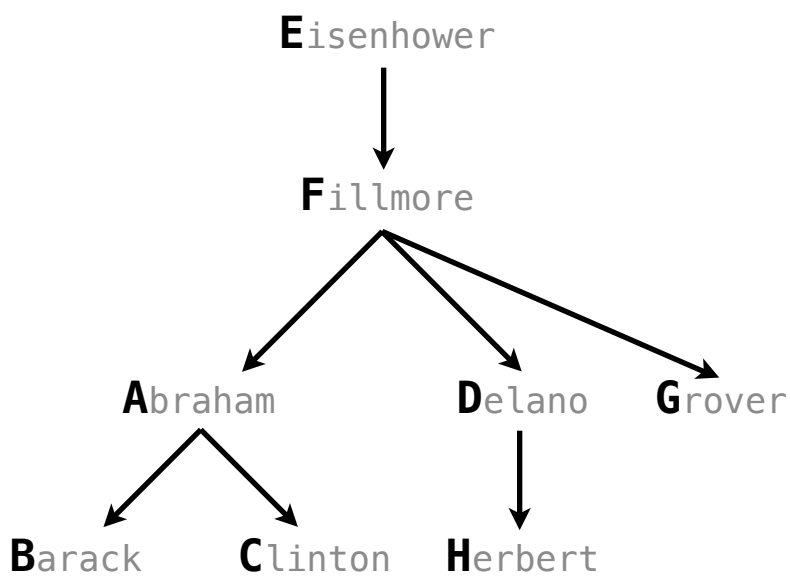
```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
```



## Recursive Facts

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

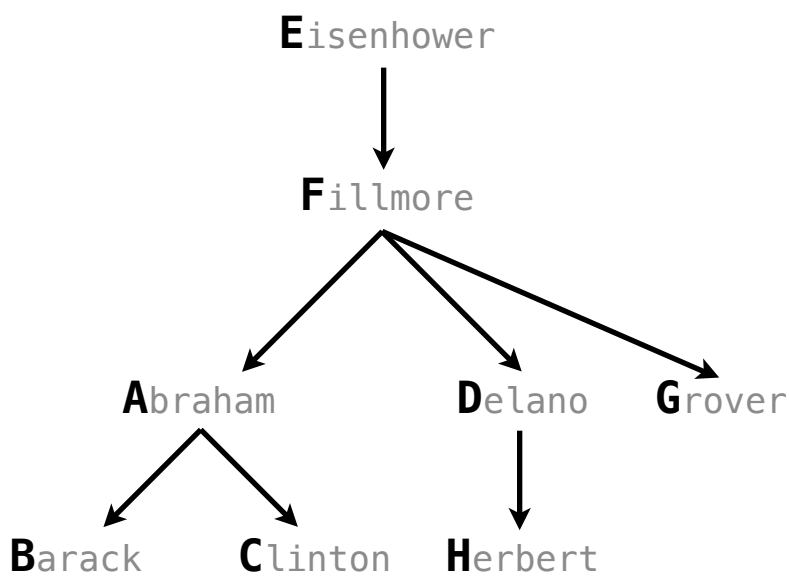
```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))
```



## Recursive Facts

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

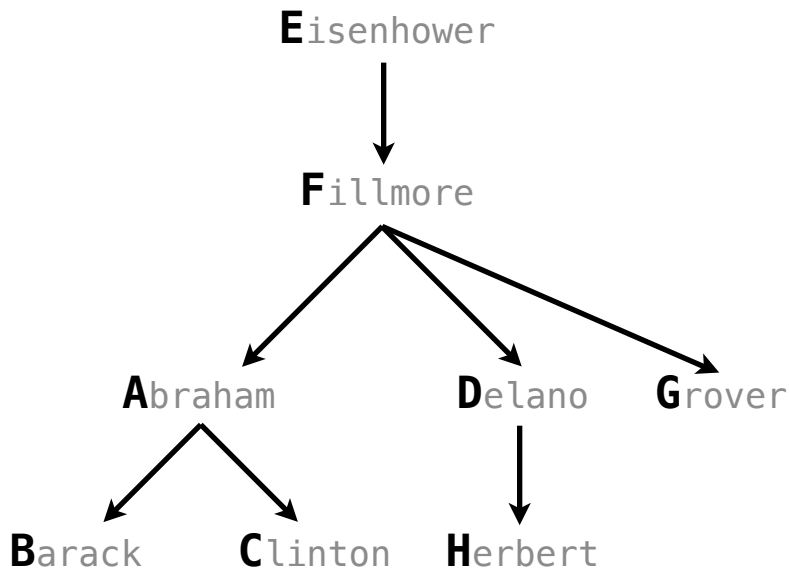
```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))
```



## Recursive Facts

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

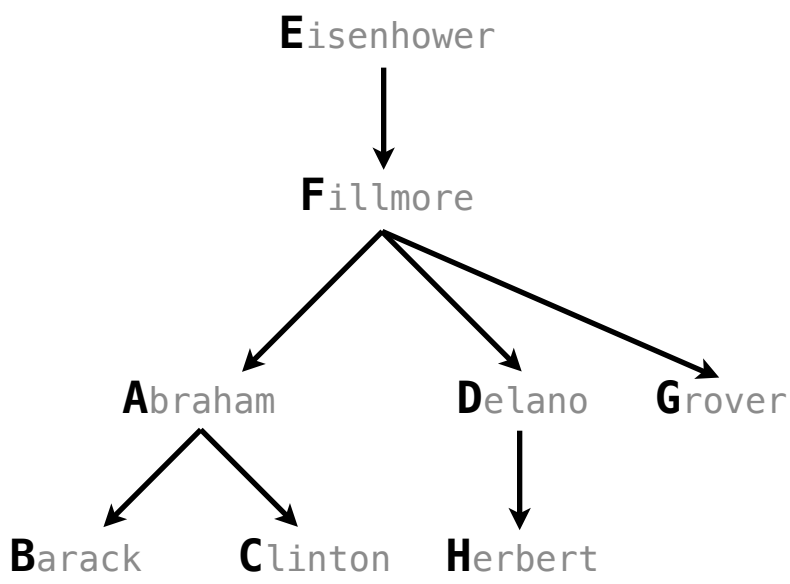
```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))

logic> (query (ancestor ?a herbert))
```



## Recursive Facts

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

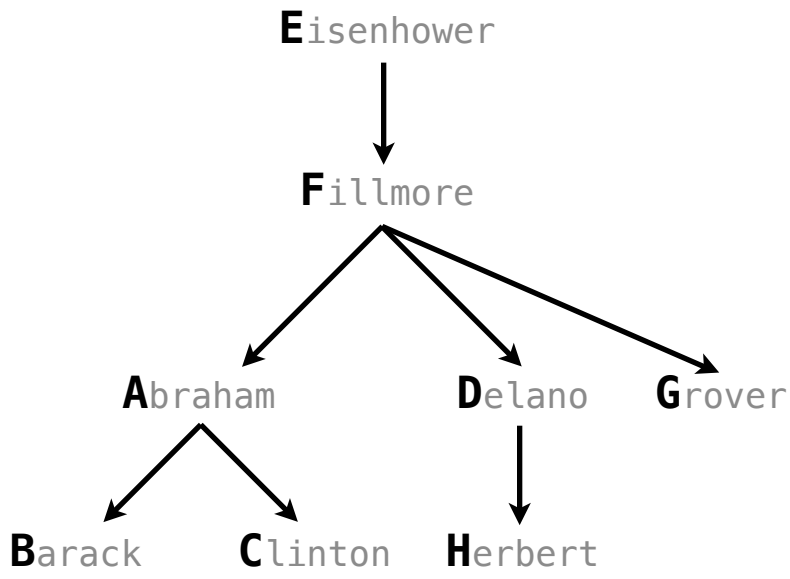
```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))

logic> (query (ancestor ?a herbert))
Success!
```



## Recursive Facts

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

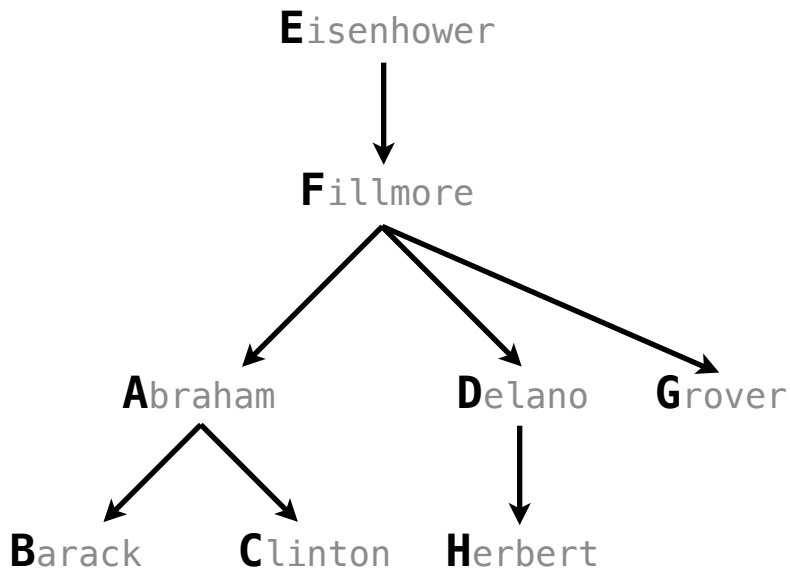
```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))

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



## Recursive Facts

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

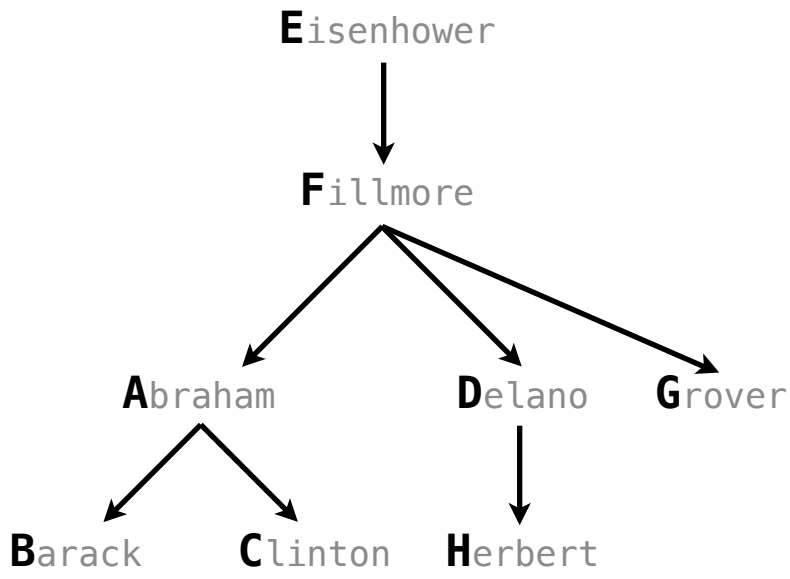
```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))

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



## Recursive Facts

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

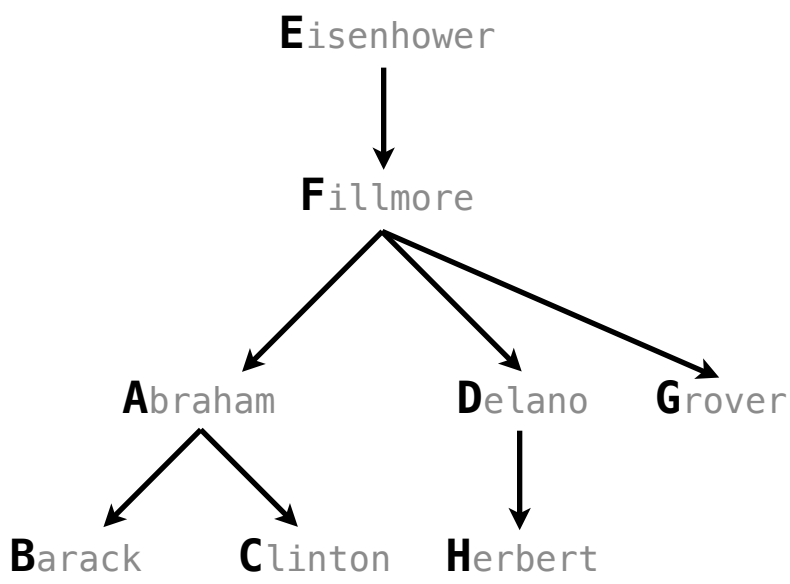
```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))

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


logic> (query (ancestor ?a barack)
```



## Recursive Facts

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

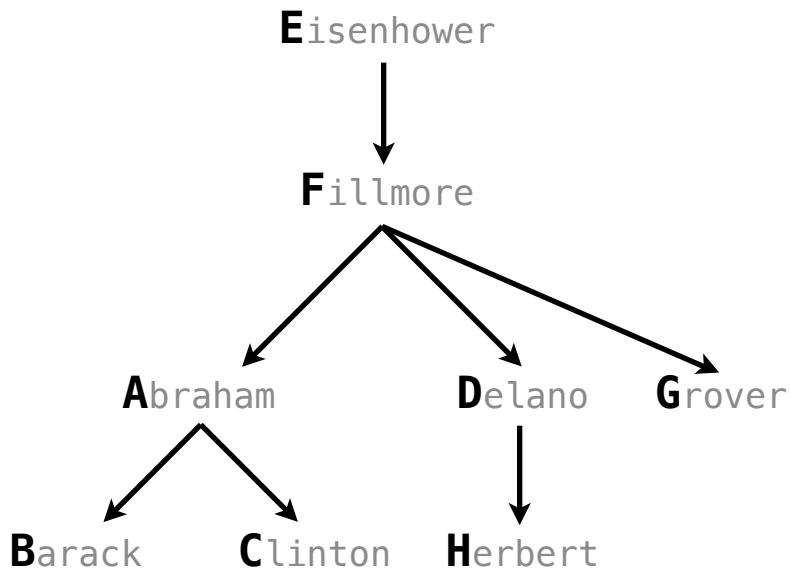
```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))

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

logic> (query (ancestor ?a barack)
              (ancestor ?a herbert))
```



## Recursive Facts

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

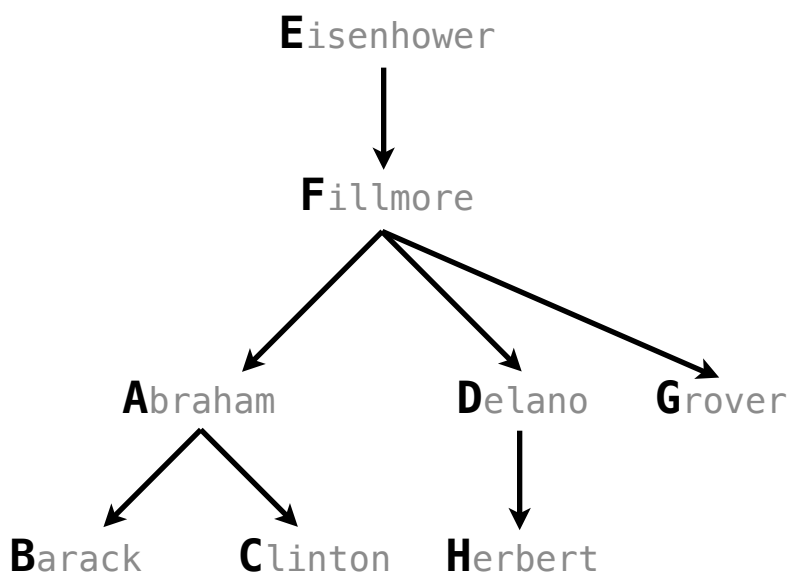
```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))

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


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



## Recursive Facts

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

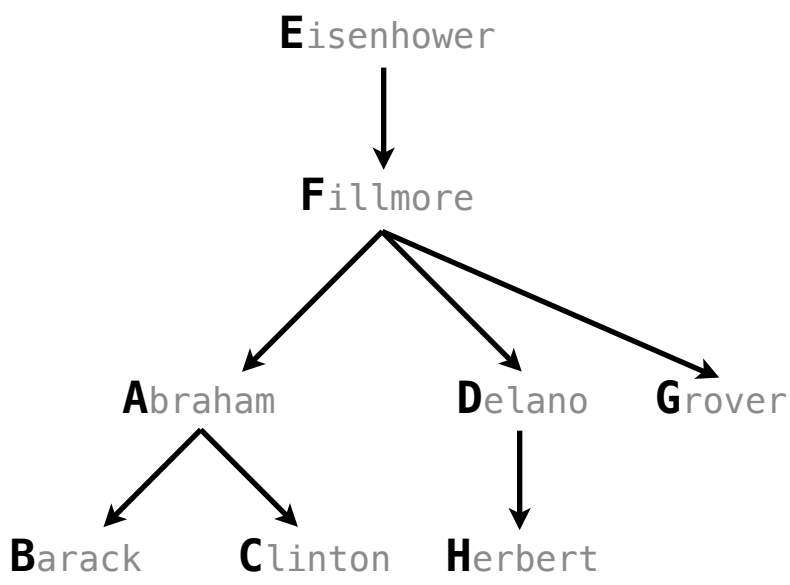
```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))

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


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



## 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> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))

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


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 satisfying assignments.

## Searching to Satisfy Queries

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

```
logic> (query (ancestor ?a herbert))
```

## Searching to Satisfy Queries

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

```
logic> (query (ancestor ?a herbert))
Success!
```

## Searching to Satisfy Queries

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

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

## Searching to Satisfy Queries

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

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

## Searching to Satisfy Queries

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

```
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 satisfying assignments.

```
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 satisfying assignments.

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

## Searching to Satisfy Queries

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

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

```
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 satisfying assignments.

```
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))
```

## Searching to Satisfy Queries

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

```
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))

(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 satisfying assignments.

```
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))
```

```
(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 satisfying assignments.

```
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))
```

```
(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
```

## Searching to Satisfy Queries

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

```
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))
```

```
(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

# Hierarchical Facts

## Hierarchical Facts

```
Relations can contain relations in addition to symbols.
```

## Hierarchical Facts

Relations can contain relations in addition to symbols.

```
logic> (fact (dog (name abraham) (fur long)))
```

## Hierarchical Facts

Relations can contain relations in addition to symbols.

```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
```

## Hierarchical Facts

Relations can contain relations in addition to symbols.

```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
logic> (fact (dog (name clinton) (fur long)))
```

## Hierarchical Facts

Relations can contain relations in addition to symbols.

```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
logic> (fact (dog (name clinton) (fur long)))
logic> (fact (dog (name delano) (fur long)))
```

## Hierarchical Facts

Relations can contain relations in addition to symbols.

```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
logic> (fact (dog (name clinton) (fur long)))
logic> (fact (dog (name delano) (fur long)))
logic> (fact (dog (name eisenhower) (fur short)))
```

## Hierarchical Facts

Relations can contain relations in addition to symbols.

```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
logic> (fact (dog (name clinton) (fur long)))
logic> (fact (dog (name delano) (fur long)))
logic> (fact (dog (name eisenhower) (fur short)))
logic> (fact (dog (name fillmore) (fur curly)))
```

## Hierarchical Facts

Relations can contain relations in addition to symbols.

```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
logic> (fact (dog (name clinton) (fur long)))
logic> (fact (dog (name delano) (fur long)))
logic> (fact (dog (name eisenhower) (fur short)))
logic> (fact (dog (name fillmore) (fur curly)))
logic> (fact (dog (name grover) (fur short)))
```

## Hierarchical Facts

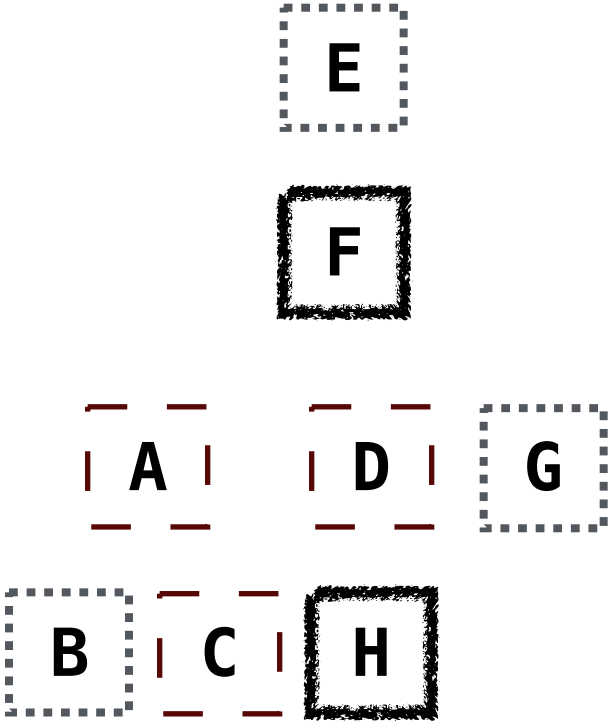
Relations can contain relations in addition to symbols.

```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
logic> (fact (dog (name clinton) (fur long)))
logic> (fact (dog (name delano) (fur long)))
logic> (fact (dog (name eisenhower) (fur short)))
logic> (fact (dog (name fillmore) (fur curly)))
logic> (fact (dog (name grover) (fur short)))
logic> (fact (dog (name herbert) (fur curly)))
```

## Hierarchical Facts

Relations can contain relations in addition to symbols.

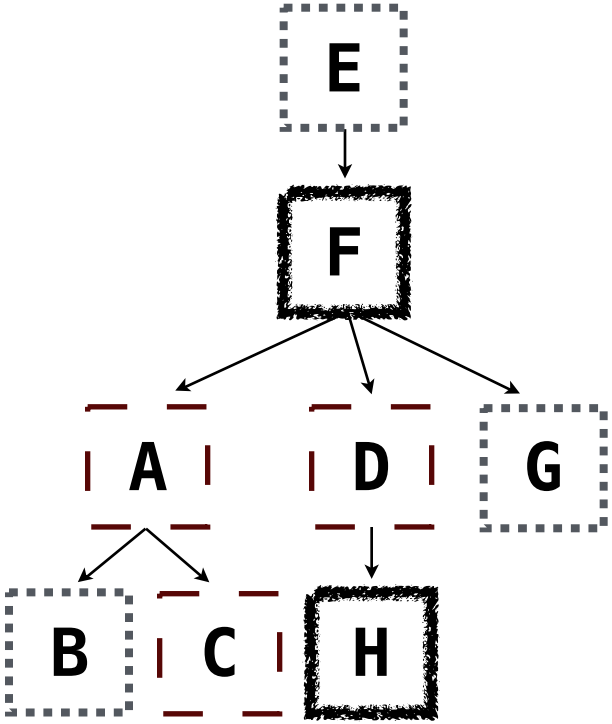
```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
logic> (fact (dog (name clinton) (fur long)))
logic> (fact (dog (name delano) (fur long)))
logic> (fact (dog (name eisenhower) (fur short)))
logic> (fact (dog (name fillmore) (fur curly)))
logic> (fact (dog (name grover) (fur short)))
logic> (fact (dog (name herbert) (fur curly)))
```



## Hierarchical Facts

Relations can contain relations in addition to symbols.

```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
logic> (fact (dog (name clinton) (fur long)))
logic> (fact (dog (name delano) (fur long)))
logic> (fact (dog (name eisenhower) (fur short)))
logic> (fact (dog (name fillmore) (fur curly)))
logic> (fact (dog (name grover) (fur short)))
logic> (fact (dog (name herbert) (fur curly)))
```

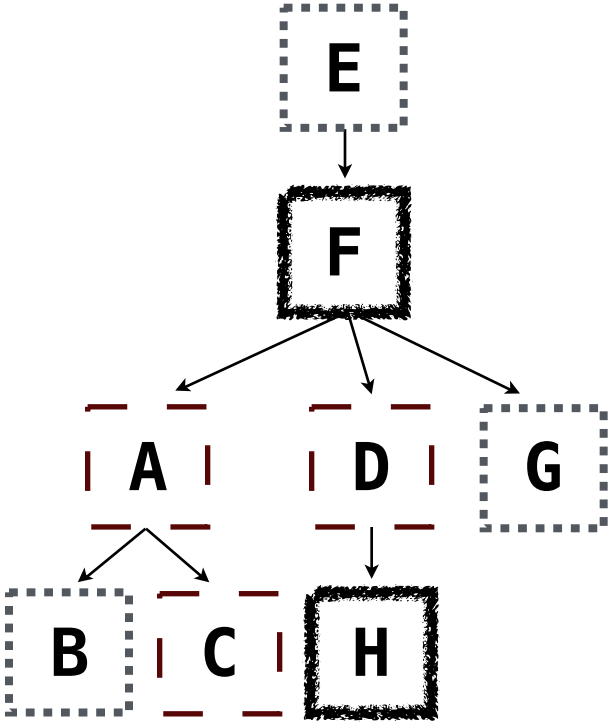


## Hierarchical Facts

Relations can contain relations in addition to symbols.

```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
logic> (fact (dog (name clinton) (fur long)))
logic> (fact (dog (name delano) (fur long)))
logic> (fact (dog (name eisenhower) (fur short)))
logic> (fact (dog (name fillmore) (fur curly)))
logic> (fact (dog (name grover) (fur short)))
logic> (fact (dog (name herbert) (fur curly)))
```

Variables can refer to symbols or whole relations.



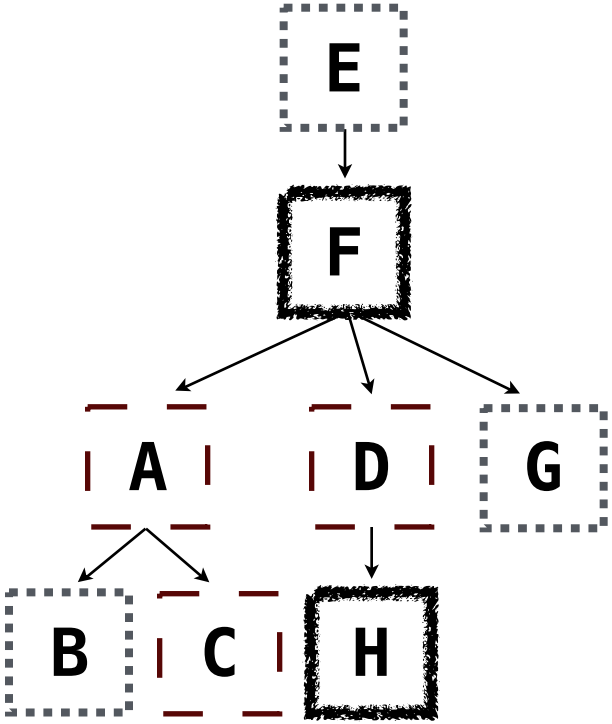
## Hierarchical Facts

Relations can contain relations in addition to symbols.

```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
logic> (fact (dog (name clinton) (fur long)))
logic> (fact (dog (name delano) (fur long)))
logic> (fact (dog (name eisenhower) (fur short)))
logic> (fact (dog (name fillmore) (fur curly)))
logic> (fact (dog (name grover) (fur short)))
logic> (fact (dog (name herbert) (fur curly)))
```

Variables can refer to symbols or whole relations.

```
logic> (query (dog (name clinton) (fur ?type)))
```



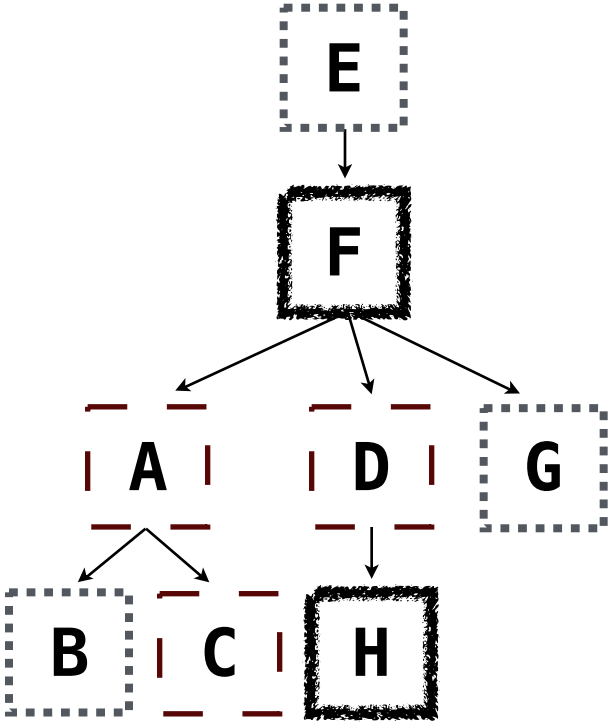
## Hierarchical Facts

Relations can contain relations in addition to symbols.

```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
logic> (fact (dog (name clinton) (fur long)))
logic> (fact (dog (name delano) (fur long)))
logic> (fact (dog (name eisenhower) (fur short)))
logic> (fact (dog (name fillmore) (fur curly)))
logic> (fact (dog (name grover) (fur short)))
logic> (fact (dog (name herbert) (fur curly)))
```

Variables can refer to symbols or whole relations.

```
logic> (query (dog (name clinton) (fur ?type)))
Success!
```



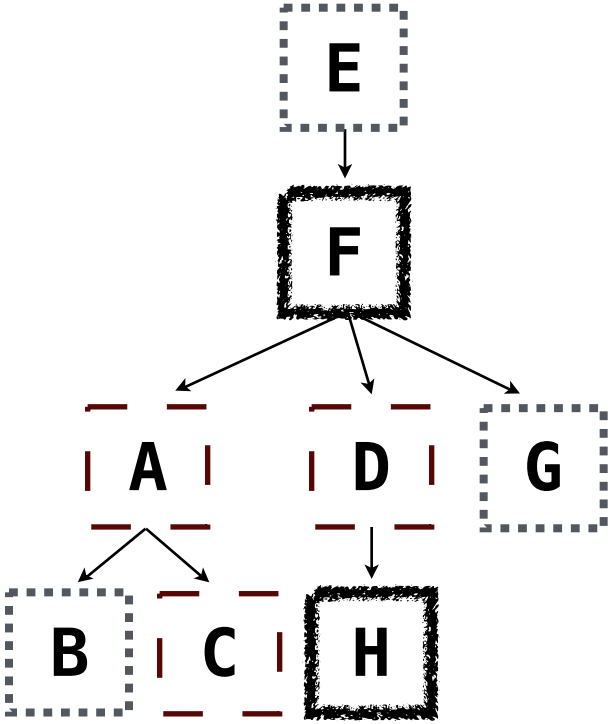
## Hierarchical Facts

Relations can contain relations in addition to symbols.

```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
logic> (fact (dog (name clinton) (fur long)))
logic> (fact (dog (name delano) (fur long)))
logic> (fact (dog (name eisenhower) (fur short)))
logic> (fact (dog (name fillmore) (fur curly)))
logic> (fact (dog (name grover) (fur short)))
logic> (fact (dog (name herbert) (fur curly)))
```

Variables can refer to symbols or whole relations.

```
logic> (query (dog (name clinton) (fur ?type)))
Success!
type: long
```



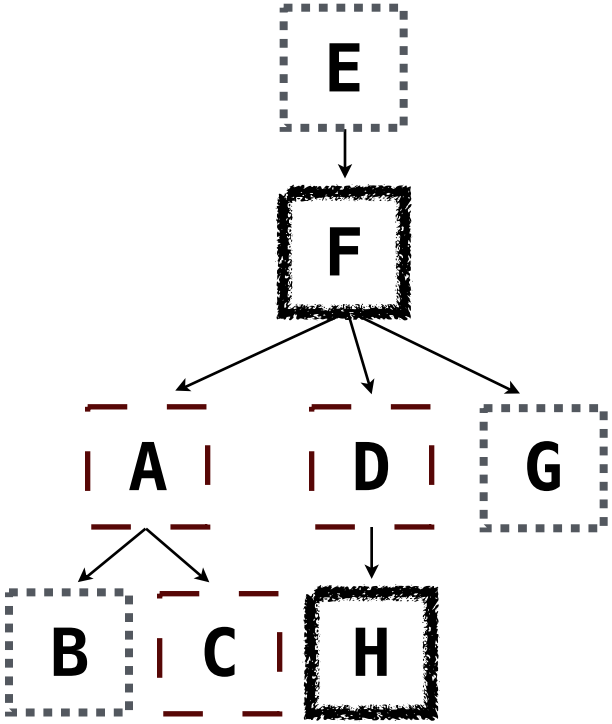
# Hierarchical Facts

Relations can contain relations in addition to symbols.

```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
logic> (fact (dog (name clinton) (fur long)))
logic> (fact (dog (name delano) (fur long)))
logic> (fact (dog (name eisenhower) (fur short)))
logic> (fact (dog (name fillmore) (fur curly)))
logic> (fact (dog (name grover) (fur short)))
logic> (fact (dog (name herbert) (fur curly)))
```

Variables can refer to symbols or whole relations.

```
logic> (query (dog (name clinton) (fur ?type)))
Success!
type: long

logic> (query (dog (name clinton) ?stats))
```



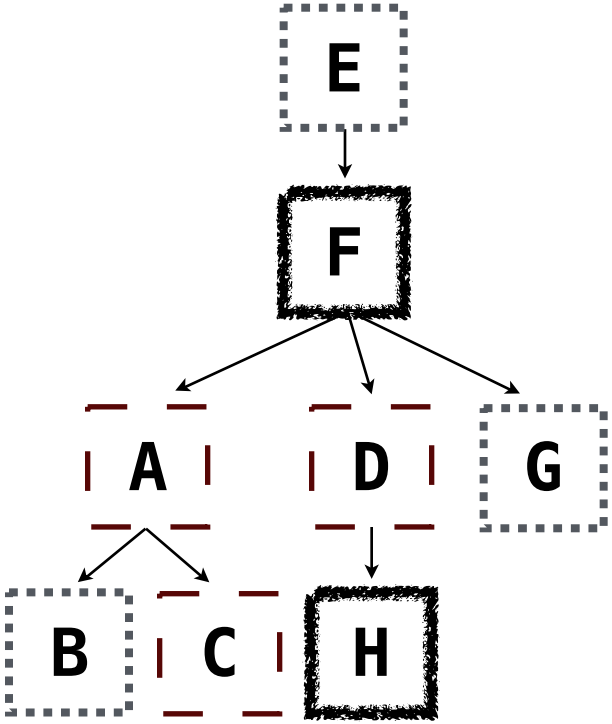
# Hierarchical Facts

Relations can contain relations in addition to symbols.

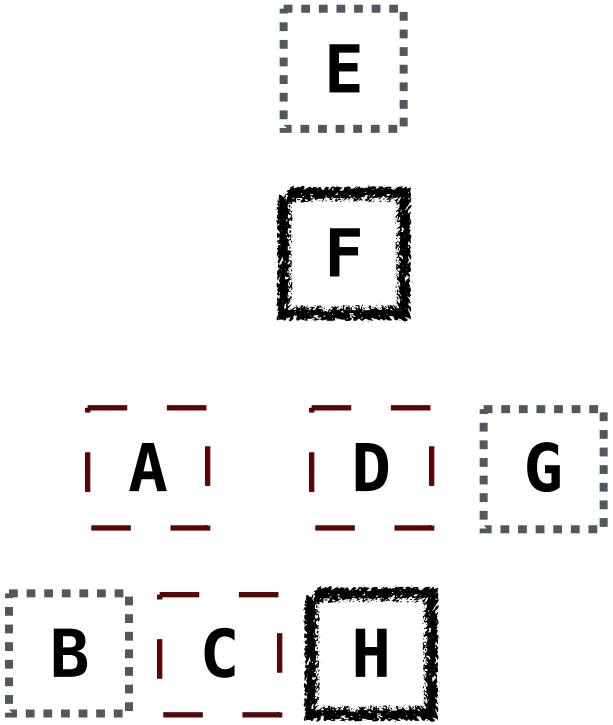
```
logic> (fact (dog (name abraham) (fur long)))
logic> (fact (dog (name barack) (fur short)))
logic> (fact (dog (name clinton) (fur long)))
logic> (fact (dog (name delano) (fur long)))
logic> (fact (dog (name eisenhower) (fur short)))
logic> (fact (dog (name fillmore) (fur curly)))
logic> (fact (dog (name grover) (fur short)))
logic> (fact (dog (name herbert) (fur curly)))
```

Variables can refer to symbols or whole relations.

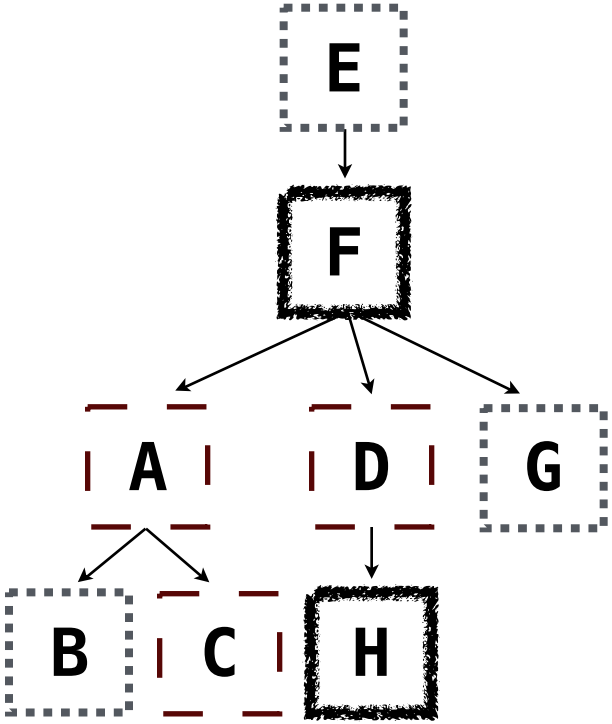
```
logic> (query (dog (name clinton) (fur ?type)))
Success!
type: long

logic> (query (dog (name clinton) ?stats))
Success!
stats: (fur long)
```



# Combining Multiple Data Sources

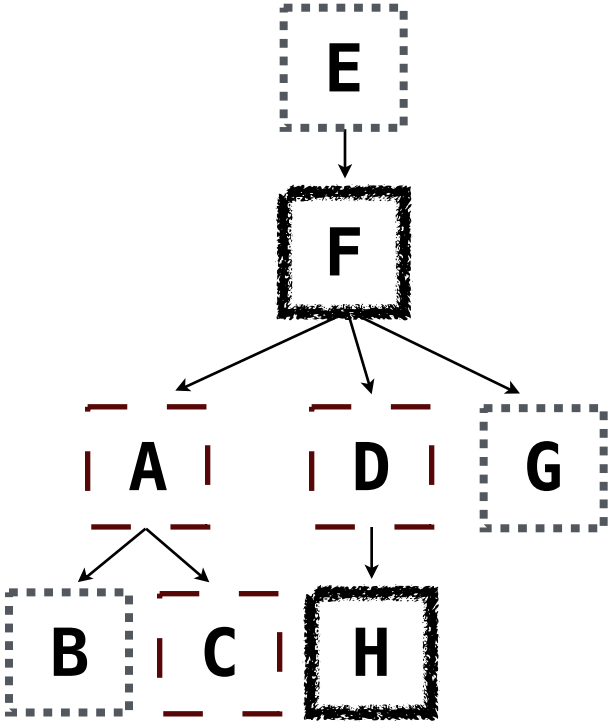


# Combining Multiple Data Sources



## Combining Multiple Data Sources

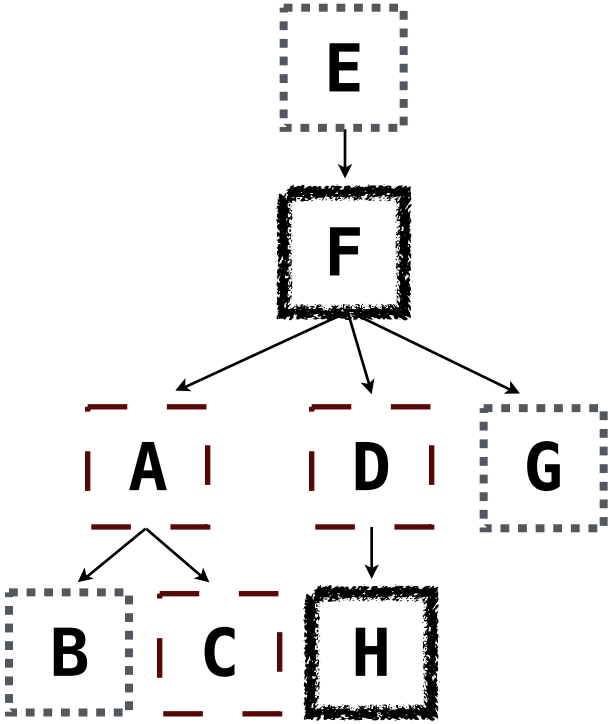
```
Which dogs have an ancestor of the same fur?
```



## Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

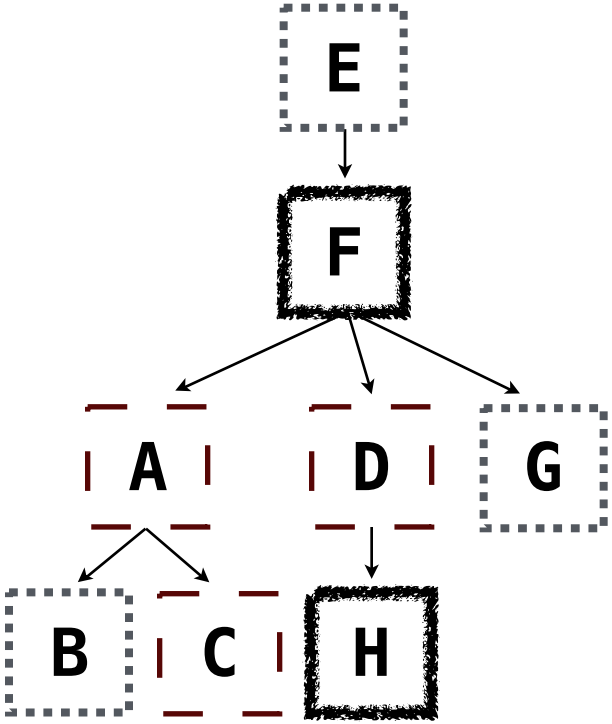
```
logic> (query (dog (name ?x) (fur ?fur))
```



## Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

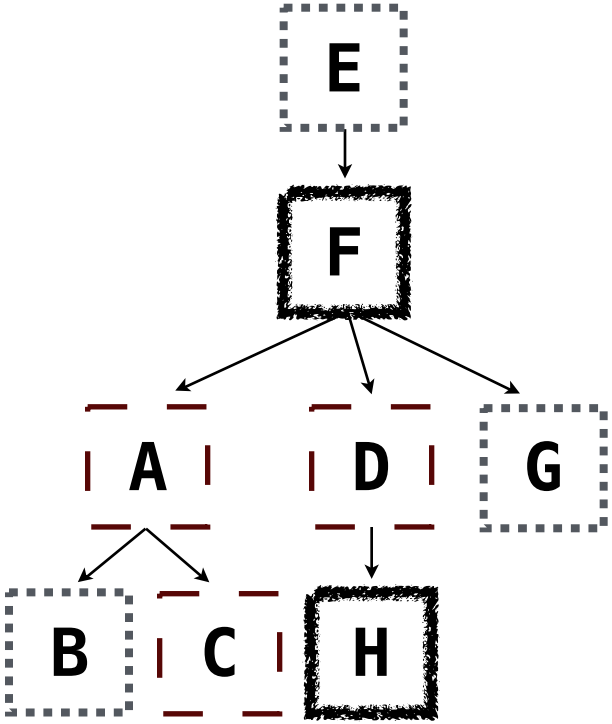
```
logic> (query (dog (name ?x) (fur ?fur))
              (ancestor ?y ?x)
```



## Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

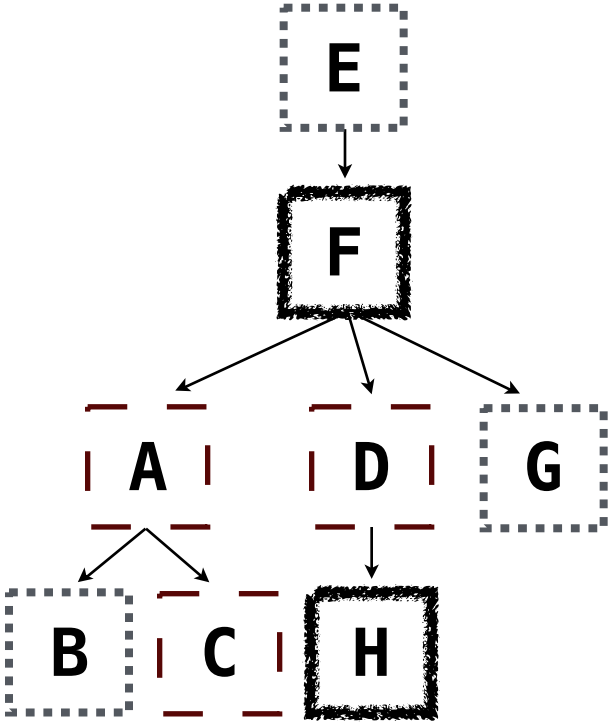
```
logic> (query (dog (name ?x) (fur ?fur))
              (ancestor ?y ?x)
              (dog (name ?y) (fur ?fur)))
```



## Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

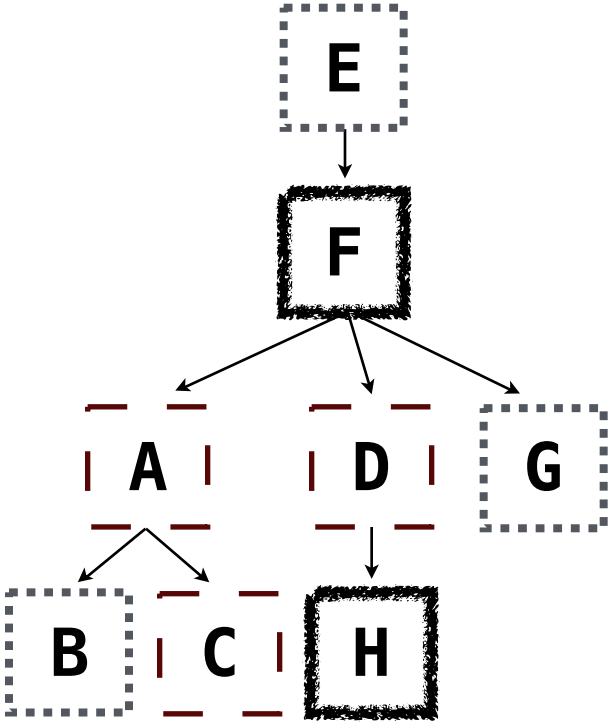
```
logic> (query (dog (name ?x) (fur ?fur))
              (ancestor ?y ?x)
              (dog (name ?y) (fur ?fur)))
Success!
```



## Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

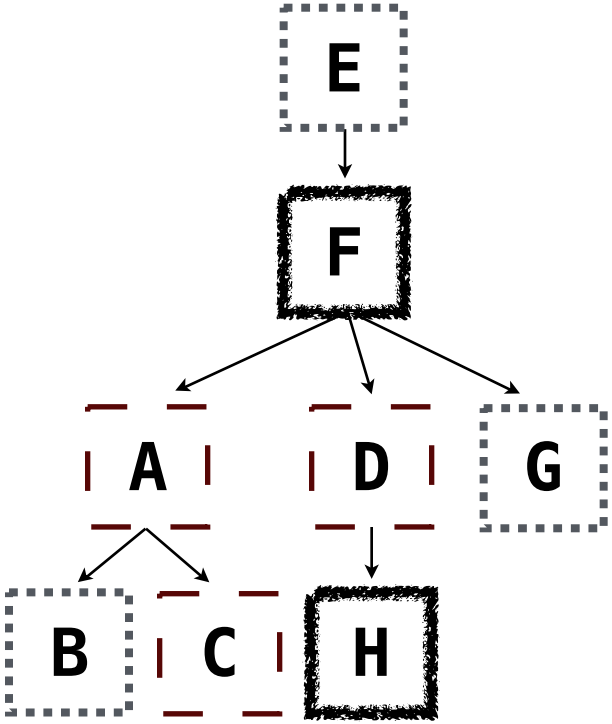
```
logic> (query (dog (name ?x) (fur ?fur))
              (ancestor ?y ?x)
              (dog (name ?y) (fur ?fur)))
Success!
x: barack    fur: short   y: eisenhower
```



## Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

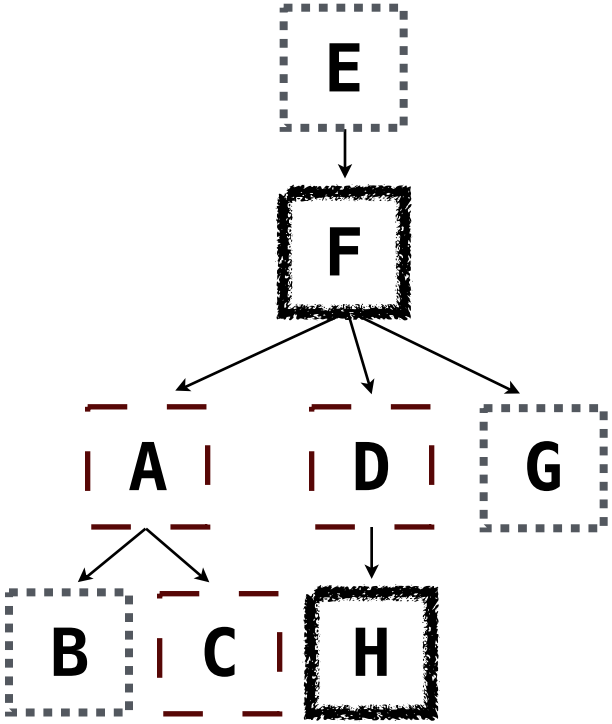
```
logic> (query (dog (name ?x) (fur ?fur))
              (ancestor ?y ?x)
              (dog (name ?y) (fur ?fur)))
Success!
x: barack    fur: short   y: eisenhower
x: clinton   fur: long    y: abraham
```



## Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

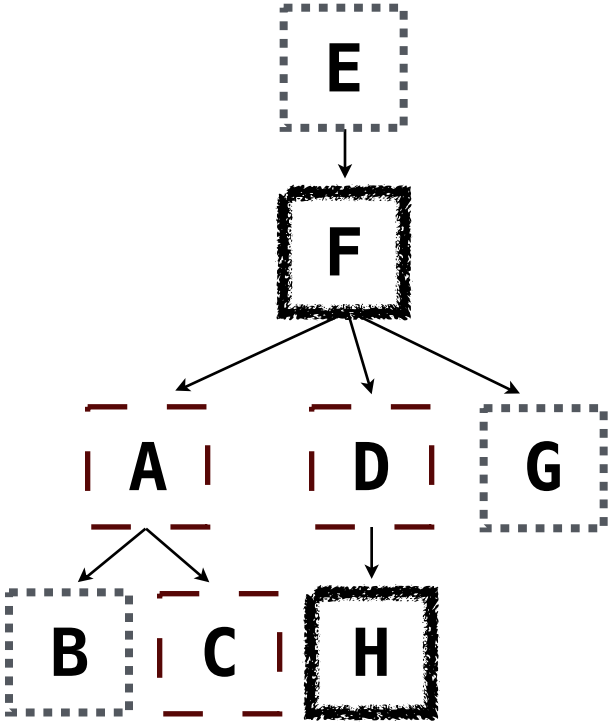
```
logic> (query (dog (name ?x) (fur ?fur))
              (ancestor ?y ?x)
              (dog (name ?y) (fur ?fur)))
Success!
x: barack     fur: short    y: eisenhower
x: clinton    fur: long     y: abraham
x: grover     fur: short    y: eisenhower
```



## Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

```
logic> (query (dog (name ?x) (fur ?fur))
              (ancestor ?y ?x)
              (dog (name ?y) (fur ?fur)))
Success!
x: barack    fur: short   y: eisenhower
x: clinton   fur: long    y: abraham
x: grover    fur: short   y: eisenhower
x: herbert   fur: curly   y: fillmore
```



# Appending Lists

(Demo)

# Lists in Logic

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

## Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

## Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

## Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

## Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

**() (c d) => (c d)**

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)

?x

```
() (c d) => (c d)
```

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)

?x ?x

() (c d) => (c d)

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)

?x ?x

**() (c d) => (c d)**

**(b) (c d) => (b c d)**

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)

?x ?x

**() (c d) => (c d)**

**(b) (c d) => (b c d)**

**(e b) (c d) => (e b c d)**

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)

?x ?x

**() (c d) => (c d)**

**(b) (c d) => (b c d)**

**(e b) (c d) => (e b c d)**

**(e . (b)) (c d) => (e . (b c d))**

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)

?x ?x

**() (c d) => (c d)**

**(b) (c d) => (b c d)**

**(e b) (c d) => (e b c d)**

**(e . (b)) (c d) => (e . (b c d))**

?a

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)

?x ?x

() (c d) => (c d)

(b) (c d) => (b c d)

(e b) (c d) => (e b c d)

(e . (b)) (c d) => (e . (b c d))

?a ?r

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)

?x ?x

**() (c d) => (c d)**

**(b) (c d) => (b c d)**

**(e b) (c d) => (e b c d)**

**(e . (b)) (c d) => (e . (b c d))**

?a ?r

(?a . ?r)

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)

```
                    ?x        ?x
() (c d) => (c d)

(b) (c d) => (b c d)

(e b) (c d) => (e b c d)

(e . (b)) (c d) => (e . (b c d))
 ?a   ?r    ?y
(?a . ?r)
```

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)

```
                    ?x        ?x
() (c d) => (c d)

(b) (c d) => (b c d)

(e b) (c d) => (e b c d)

(e . (b)) (c d) => (e . (b c d))
 ?a   ?r    ?y      ?a
(?a . ?r)
```

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)

```
                    ?x        ?x
() (c d) => (c d)

(b) (c d) => (b c d)

(e b) (c d) => (e b c d)

(e . (b)) (c d) => (e . (b c d))
 ?a   ?r   ?y      ?a    ?z
(?a . ?r)
```

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)

```
                    ?x        ?x
() (c d) => (c d)

(b) (c d) => (b c d)

(e b) (c d) => (e b c d)

(e . (b)) (c d) => (e . (b c d))
 ?a   ?r    ?y     ?a     ?z
(?a . ?r)          (?a . ?z)
```

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

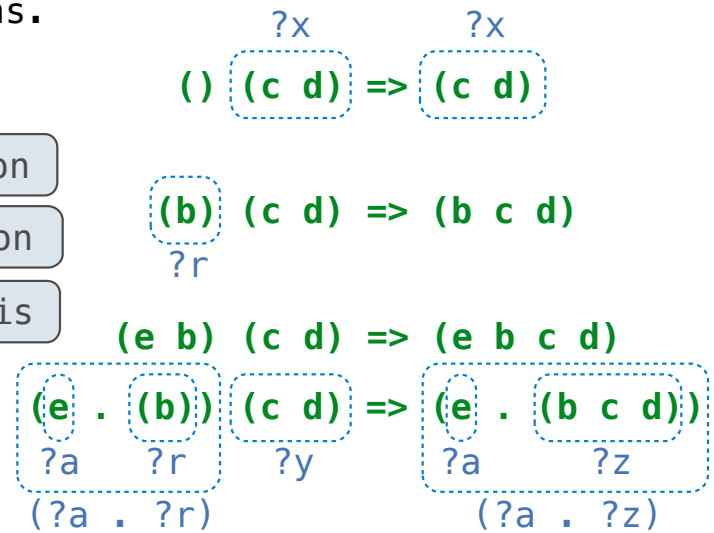
```
(fact (append-to-form (?a . ?r) ?y (?a . ?z))
      (append-to-form       ?r  ?y       ?z ))
```

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)




The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

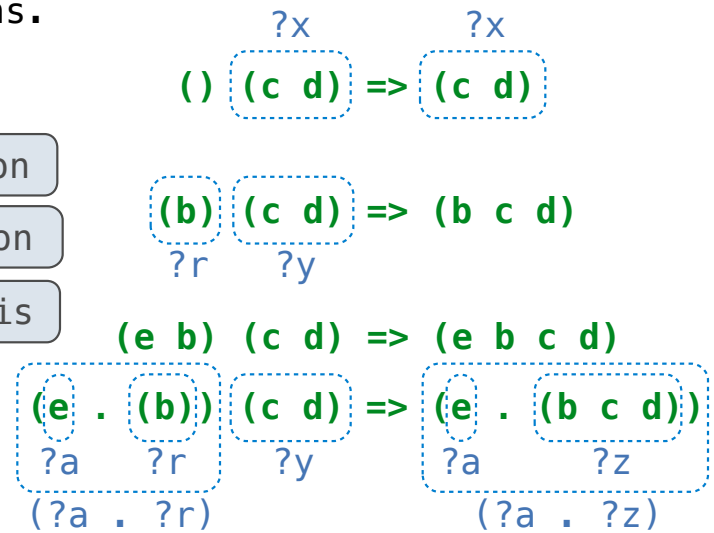
```
(fact (append-to-form (?a . ?r) ?y (?a . ?z))
      (append-to-form       ?r  ?y       ?z ))
```

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)




The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

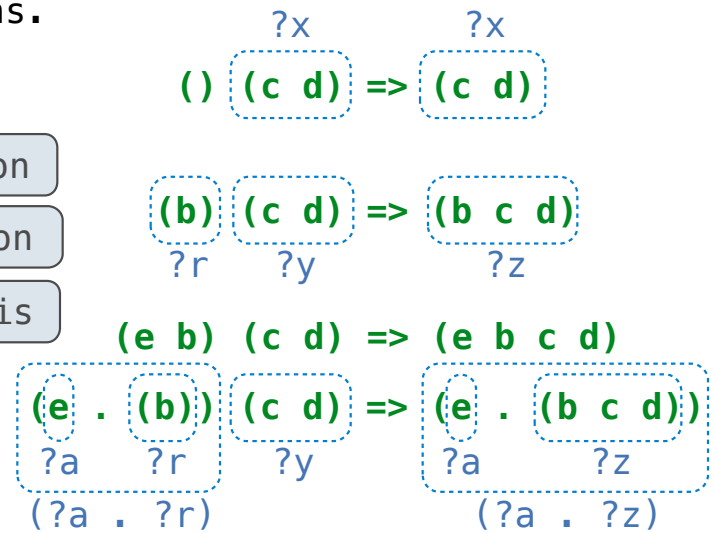
```
(fact (append-to-form (?a . ?r) ?y (?a . ?z))
      (append-to-form       ?r  ?y       ?z ))
```

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)




The interpreter lists all bindings of variables to values that it can find to satisfy the query.

# Lists in Logic

Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

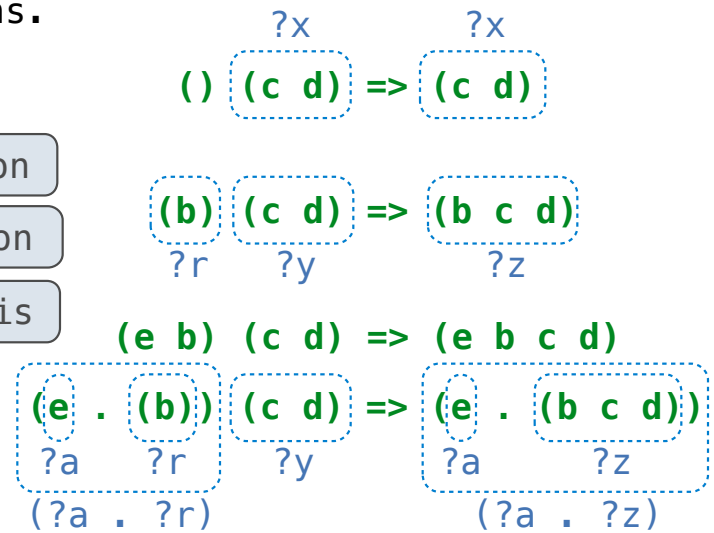
```
(fact (append-to-form (?a . ?r) ?y (?a . ?z))
      (append-to-form       ?r  ?y       ?z ))
```

Conclusion

Hypothesis

```
(query (append-to-form ?left (c d) (e b c d)))
Success!
left: (e b)
```

What ?left can append with (c d) to create (e b c d)




The interpreter lists all bindings of variables to values that it can find to satisfy the query.

(Demo)

# Which Hypotheses Complete append-3?

## Which Hypotheses Complete append-3?

```
I can append (a b) and (1 2) and (x y) to form (a b 1 2 x y).

(fact (append-3 ?x ?y ?z ?xyz)

  1:  (append-to-form ?x ?y ?xy)
      (append-to-form ?y ?z ?yz) )

  2:  (= (append-to-form ?x ?y) ?xy)
      (= (append-to-form ?y ?z) ?yz) )

  3:  (append-to-form ?x ?y ?xy)
      (append-to-form ?xy ?z ?xyz) )

  4:  (= (append-to-form ?x ?y) ?xy)
      (= (append-to-form ?xy ?z) ?xyz) )

  5:  None of the above
```

# Define Base Fact of append-to-form So That No Lists Can Be Empty

## Define Base Fact of append-to-form So That No Lists Can Be Empty

```
; (append-to-form () (1 2 3) (1 2 3))
; (append-to-form (1) (2 3) (1 2 3))
; (append-to-form (1 2) (3) (1 2 3))
; (append-to-form (1 2 3) () (1 2 3))
```

# Define Base Fact of append-to-form So That No Lists Can Be Empty

```
; (append-to-form () (1 2 3) (1 2 3))
; (append-to-form (1) (2 3) (1 2 3))
; (append-to-form (1 2) (3) (1 2 3))
; (append-to-form (1 2 3) () (1 2 3))
```

# Define Base Fact of append-to-form So That No Lists Can Be Empty

```
; (append-to-form () (1 2 3) (1 2 3))
; (append-to-form (1) (2 3) (1 2 3))
; (append-to-form (1 2) (3) (1 2 3))
; (append-to-form (1 2 3) () (1 2 3))
```

# Define Base Fact of append-to-form So That No Lists Can Be Empty

~~; (append-to-form () (1 2 3) (1 2 3))~~

```
; (append-to-form (1) (2 3) (1 2 3))
; (append-to-form (1 2) (3) (1 2 3))
```

~~; (append-to-form (1 2 3) () (1 2 3))~~

*Recursive fact*:

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

# Define Base Fact of append-to-form So That No Lists Can Be Empty

~~; (append-to-form () (1 2 3) (1 2 3))~~

; (append-to-form (1) (2 3) (1 2 3))

; (append-to-form (1 2) (3) (1 2 3))

~~; (append-to-form (1 2 3) () (1 2 3))~~

**1:** (fact (append-to-form () ?x ?x))

**2:** (fact (append-to-form ?a ?x (?a . ?x)))

**3:** (fact (append-to-form ?a (?b . ?x) (?a ?b . ?x)))

**4:** (fact (append-to-form (?a) ?x (?a . ?x)))

**5:** (fact (append-to-form (?a) (?b . ?x) (?a ?b . ?x)))

*Recursive fact*:

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

# Permuting Lists

# Anagrams in Logic

# Anagrams in Logic

```
A permutation (i.e., anagram) of a list is:
```

## Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

```
(fact (insert ?a ?r (?a . ?r)))
```

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element

```
(fact (insert ?a ?r (?a . ?r)))
```

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element List

```
(fact (insert ?a ?r (?a . ?r)))
```

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element | List | List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element | List | List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))

(fact (insert ?a (?b . ?r) (?b . ?s))
```

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))

(fact (insert ?a (?b . ?r) (?b . ?s))
```

List with ?a somewhere

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

```
(fact (anagram () ()))
```

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

```
(fact (anagram () ()))
```

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

```
(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
```

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

```
(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
      (insert   ?a   ?s   ?b))
```

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

a r t

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s))
```

List with ?a somewhere

```
(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
      (insert   ?a   ?s   ?b)
```

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

a | r t

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

```
(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
      (insert   ?a   ?s   ?b))
```

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

a | r t

r t

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

```
(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
      (insert   ?a   ?s  ?b))
```

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

a | r t

r t

ar t

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

```
(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
      (insert   ?a   ?s   ?b))
```

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

```
(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
      (insert   ?a   ?s  ?b))
```

a | r t

r t

ar t

rat

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

```
(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
      (insert   ?a   ?s  ?b))
```

a | r t

r t

ar t

rat

r ta

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

```
(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
      (insert   ?a   ?s  ?b))
```

a | r t

r t

ar t

rat

r ta

t r

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

```
(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
      (insert   ?a   ?s  ?b))
```

a | r t

r t

ar t

rat

r ta

t r

at r

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

```
(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
      (insert   ?a   ?s  ?b))
```

a | r t

r t

ar t

rat

r ta

t r

at r

tar

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element

List

List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

```
(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
      (insert   ?a   ?s  ?b))
```

a | r t

r t
ar t
rat
r ta

t r
at r
tar
t ra

# Anagrams in Logic

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.

Element | List | List with ?a in front

```
(fact (insert ?a ?r (?a . ?r)))
```

Bigger list with ?a somewhere

```
(fact (insert ?a (?b . ?r) (?b . ?s)))
```

List with ?a somewhere

```
(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
      (insert   ?a   ?s  ?b))
```

a | r t

r t
ar t
rat
r ta

t r
at r
tar
t ra

(Demo)

# Unification

# Pattern Matching

## Pattern Matching

The basic operation of the Logic interpreter is to attempt to *unify* two relations.

## Pattern Matching

The basic operation of the Logic interpreter is to attempt to *unify* two relations.

Unification is finding an assignment to variables that makes two relations the same.

## Pattern Matching

The basic operation of the Logic interpreter is to attempt to *unify* two relations.

Unification is finding an assignment to variables that makes two relations the same.

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

## Pattern Matching

The basic operation of the Logic interpreter is to attempt to *unify* two relations.

Unification is finding an assignment to variables that makes two relations the same.

```
( (a  b) c  (a  b) )
(   ?x   c    ?x   )
```

## Pattern Matching

The basic operation of the Logic interpreter is to attempt to *unify* two relations.

Unification is finding an assignment to variables that makes two relations the same.

```
( (a  b) c  (a  b) )
(   ?x   c    ?x   )
```

```
True, {x: (a b)}
```

## Pattern Matching

The basic operation of the Logic interpreter is to attempt to *unify* two relations.

Unification is finding an assignment to variables that makes two relations the same.

```
( (a  b) c  (a  b) )
(   ?x   c    ?x   )
```

```
True, {x: (a b)}
```

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

## Pattern Matching

The basic operation of the Logic interpreter is to attempt to *unify* two relations.

Unification is finding an assignment to variables that makes two relations the same.

```
( (a  b) c  (a  b) )
(   ?x   c    ?x   )
```

```
True, {x: (a b)}
```

```
( (a  b) c  (a  b) )
( (a ?y) ?z (a  b) )
```

## Pattern Matching

The basic operation of the Logic interpreter is to attempt to *unify* two relations.

Unification is finding an assignment to variables that makes two relations the same.

```
( (a  b) c  (a  b) )
(   ?x   c    ?x   )
```

```
True, {x: (a b)}
```

```
( (a  b) c  (a  b) )
( (a ?y) ?z (a  b) )
```

```
True, {y: b, z: c}
```

## Pattern Matching

The basic operation of the Logic interpreter is to attempt to *unify* two relations.

Unification is finding an assignment to variables that makes two relations the same.

```
( (a  b) c  (a  b) )
(   ?x   c    ?x   )
```

```
True, {x: (a b)}
```

```
( (a  b) c  (a  b) )
( (a ?y) ?z (a  b) )
```

```
True, {y: b, z: c}
```

```
( (a  b) c  (a  b) )
(   ?x  ?x    ?x   )
```

## Pattern Matching

The basic operation of the Logic interpreter is to attempt to *unify* two relations.

Unification is finding an assignment to variables that makes two relations the same.

```
( (a  b) c  (a  b) )
(   ?x   c    ?x   )
```

```
True, {x: (a b)}
```

```
( (a  b) c  (a  b) )
( (a ?y) ?z (a  b) )
```

```
True, {y: b, z: c}
```

```
( (a  b) c  (a  b) )
(   ?x  ?x    ?x   )
```

```
False
```

# Unification

# Unification

```
Unification recursively unifies each pair of corresponding elements in two relations,
accumulating an assignment.
```

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

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

(   ?x   c   ?x   )
```

{ }

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

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

(   ?x   c   ?x   )
```

```
{                }
```

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

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

(   ?x   c   ?x   )
```

```
{  x: (a b)  }
```

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

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

(   ?x   c   ?x   )
```

```
{  x: (a b)  }
```

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

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

(   ?x   c   ?x   )
```

```
{  x: (a b)  }
```

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

```
( (a  b) c (a  b) )
( ?x     c ?x     )
```

Lookup

```
(a  b)
(a  b)
```

```
{ x: (a b) }
```

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

```
( (a  b) c (a  b) )
( ?x     c ?x     )
```

Lookup

```
(a  b)
(a  b)
```

```
{ x: (a b) }
```

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

```
( (a  b) c (a  b) )
( ?x     c ?x     )
```

Lookup

```
(a  b)
(a  b)
```

```
{  x: (a b)  }
```

*Success!*

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

```
( (a  b) c (a  b) )
( ?x     c ?x     )
```

Lookup

```
(a  b)
(a  b)
```

```
{ x: (a b) }
```

*Success!*

```
( (a  b) c  (a  b) )
(   ?x   ?x   ?x   )
```

```
{ }
```

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

```
( (a  b) c (a  b) )
( ?x     c ?x     )
```

Lookup

```
(a  b)
(a  b)
```

```
{ x: (a b) }
```

*Success!*

```
( (a  b) c  (a  b) )
( ?x     ?x ?x     )
```

```
{ }
```

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

```
( (a  b) c (a  b) )
( ?x     c ?x     )
```

Lookup

```
(a  b)
(a  b)
```

```
{ x: (a b) }
```

*Success!*

```
( (a  b) c  (a  b) )
( ?x     ?x ?x     )
```

```
{ x: (a b) }
```

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

```
( (a  b) c (a  b) )
( ?x     c ?x     )
```

Lookup

```
(a  b)
(a  b)
```

```
{ x: (a b) }
```

*Success!*

```
( (a  b) c  (a  b) )
( ?x     ?x ?x     )
```

```
{ x: (a b) }
```

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

```
( (a  b) c (a  b) )
( ?x     c ?x     )
```

Lookup

```
(a  b)
(a  b)
```

```
{ x: (a b) }
```

*Success!*

```
( (a  b) c   (a  b) )
( ?x     ?x  ?x     )
```

Lookup

```
c
(a b)
```

```
{ x: (a b) }
```

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

```
( (a  b) c (a  b) )
( ?x     c ?x     )
```

Lookup

```
(a  b)
(a  b)
```

```
{ x: (a b) }
```

*Success!*

Symbols/relations without variables only unify if they are the same

```
( (a  b) c   (a  b) )
( ?x     ?x  ?x     )
```

Lookup

```
c
(a b)
```

```
{ x: (a b) }
```

# Unification

Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment.
2. Establish new bindings to unify elements.

```
( (a  b) c (a  b) )
( ?x     c ?x     )
```

Lookup

```
(a  b)
(a  b)
```

```
{ x: (a b) }
```

*Success!*

Symbols/relations without variables only unify if they are the same

```
( (a  b) c   (a  b) )
( ?x     ?x  ?x     )
```

Lookup

```
c
(a b)
```

```
{ x: (a b) }
```

*Failure.*

# Unifying Variables

## Unifying Variables

```
Two relations that contain variables can be unified as well.
```

## Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x        ?x    )

((a ?y c) (a b ?z))
```

## Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x        ?x    )

((a ?y c) (a b ?z))
```

```
True, {
```

## Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x        ?x    )
((a ?y c) (a b ?z))
```

```
True, {
```

## Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x        ?x    )
((a ?y c) (a b ?z))
```

```
True, {x: (a ?y c),
```

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(   ?x       ?x    )
((a ?y c) (a b ?z))
```

```
True, {x: (a ?y c),
```

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(   ?x        ?x    )
((a ?y c)  (a b ?z))
```

Lookup

```
(a ?y  c)

(a  b ?z)
```

```
True, {x: (a ?y c),
```

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x        )   (    ?x        )
((a ?y c))   ((a b ?z))
```

Lookup

```
(a ?y  c)
(a  b ?z)
```

```
True, {x: (a ?y c),
```

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x      )   (    ?x      )
((a ?y c))       (a b ?z))
```

Lookup

```
(a ?y  c)
(a  b ?z)
```

```
True, {x: (a ?y c),
```

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x        )   (    ?x        )
((a ?y c))     (a b ?z))
```

Lookup

```
(a ?y  c)
(a  b ?z)
```

```
True, {x: (a ?y c),
       y: b,
```

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x      )   (    ?x      )
((a ?y c))       (a b ?z))
```

```
True, {x: (a ?y c),
       y: b,
```


```
(a  ?y  c)
(a   b ?z)
```

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x      )   (    ?x      )
((a ?y c))       (a b ?z))
```

```
True, {x: (a ?y c),
       y: b,
       z: c}
```

Lookup

```
(a  ?y  c)
(a   b ?z)
```

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x      )   (    ?x      )
((a ?y c))       ((a b ?z))
```

Lookup

```
(a  ?y  c)
(a   b ?z)
```

```
True, {x: (a ?y c),
       y: b,
       z: c}
```

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x      )   (    ?x      )
((a ?y c))       ((a b ?z))
```

Lookup

```
(a  ?y  c)
(a   b ?z)
```

```
True, {x: (a ?y c),
       y: b,
       z: c}
```

Substituting values for variables may require multiple steps.

This process is called *grounding*. Two unified expressions have the same grounded form.

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x        )   (    ?x        )
((a ?y c)) ((a b ?z))
```

Lookup

```
(a  ?y  c)
(a   b ?z)
```

```
True, {x: (a ?y c),
       y: b,
       z: c}
```

Substituting values for variables may require multiple steps.

This process is called *grounding*. Two unified expressions have the same grounded form.

**`lookup('?x')`**

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x        )   (    ?x        )
((a ?y c))   ((a b ?z))
```

Lookup

```
(a  ?y  c)
(a   b ?z)
```

```
True, {x: (a ?y c),
       y: b,
       z: c}
```

Substituting values for variables may require multiple steps.

This process is called *grounding*. Two unified expressions have the same grounded form.

**lookup('?x')** ⇨ **(a ?y c)**

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x       )   (    ?x       )
((a ?y c))       ((a b ?z))
```

Lookup

```
(a  ?y  c)
(a   b ?z)
```

```
True, {x: (a ?y c),
       y: b,
       z: c}
```

Substituting values for variables may require multiple steps.

This process is called *grounding*. Two unified expressions have the same grounded form.

**lookup('?x')** ⇨ **(a ?y c)**  **lookup('?y')**

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x      )   (    ?x      )
((a ?y c))       (a b ?z))
```

Lookup

```
(a  ?y  c)
(a   b ?z)
```

```
True, {x: (a ?y c),
       y: b,
       z: c}
```

Substituting values for variables may require multiple steps.

This process is called *grounding*. Two unified expressions have the same grounded form.

**lookup('?x')** ⇨ **(a ?y c)**  **lookup('?y')** ⇨ **b**

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x        )   (    ?x        )
((a ?y c))   ((a b ?z))
```

Lookup

```
(a  ?y  c)
(a   b ?z)
```

```
True, {x: (a ?y c),
       y: b,
       z: c}
```

Substituting values for variables may require multiple steps.

This process is called *grounding*. Two unified expressions have the same grounded form.

**lookup('?x')** ⇨ **(a ?y c)**  **lookup('?y')** ⇨ **b**  **ground('?x')**

# Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x      )   (    ?x      )
((a ?y c))       (a b ?z))
```

```
True, {x: (a ?y c),
       y: b,
       z: c}
```

Lookup

```
(a  ?y  c)
(a   b ?z)
```

Substituting values for variables may require multiple steps.

This process is called *grounding*. Two unified expressions have the same grounded form.

**lookup('?x')** ⇨ **(a ?y c)**  **lookup('?y')** ⇨ **b**  **ground('?x')** ⇨ **(a b c)**

## Implementing Unification

```
def unify(e, f, env):
  e = lookup(e, env)
  f = lookup(f, env)
  if e == f:
      return True
  elif isvar(e):
      env.define(e, f)
      return True
  elif isvar(f):
      env.define(f, e)
      return True
  elif scheme_atomp(e) or scheme_atomp(f):
      return False
  else:
      return unify(e.first,  f.first,  env) and unify(e.second, f.second, env)
```

## Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first,  f.first,  env) and unify(e.second, f.second, env)
```

1. Look up variables in the current environment

# Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first,  f.first,  env) and unify(e.second, f.second, env)
```

1. Look up variables in the current environment

2. Establish new bindings to unify elements.

# Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first,  f.first,  env) and unify(e.second, f.second, env)
```

1. Look up variables in the current environment

Symbols/relations without variables only unify if they are the same

2. Establish new bindings to unify elements.

# Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first,  f.first,  env) and unify(e.second, f.second, env)
```

1. Look up variables in the current environment

Symbols/relations without variables only unify if they are the same

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

# Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first, f.first, env) and unify(e.second, f.second, env)
```

1. Look up variables in the current environment

Symbols/relations without variables only unify if they are the same

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

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

(   ?x   c   ?x   )
```

# Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first, f.first, env) and unify(e.second, f.second, env)
```

1. Look up variables in the current environment

Symbols/relations without variables only unify if they are the same

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

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

(    ?x    c    ?x    )

env: {                }
```

# Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first,  f.first,  env) and unify(e.second, f.second, env)
```

1. Look up variables in the current environment

Symbols/relations without variables only unify if they are the same

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

```
( (a  b) c (a  b) )
(   ?x   c   ?x   )
```

env: { }

# Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first,  f.first,  env) and unify(e.second, f.second, env)
```

1. Look up variables in the current environment

Symbols/relations without variables only unify if they are the same

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

```
( (a  b) c (a  b) )
(   ?x   c   ?x   )
```

env: { }

# Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first, f.first, env) and unify(e.second, f.second, env)
```

1. Look up variables in the current environment

Symbols/relations without variables only unify if they are the same

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

```
( (a  b) c (a  b) )
( ?x     c   ?x   )
```

env: { x: (a b) }

# Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first, f.first, env) and unify(e.second, f.second, env)
```

1. Look up variables in the current environment

Symbols/relations without variables only unify if they are the same

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

```
( (a  b) c (a  b) )
(   ?x   c   ?x   )
```

env: { x: (a b) }

# Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first, f.first, env) and unify(e.second, f.second, env)
```

1. Look up variables in the current environment

Symbols/relations without variables only unify if they are the same

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

```
( (a  b) c (a  b) )
(   ?x   c   ?x   )
```

env: { x: (a b) }

# Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first, f.first, env) and unify(e.second, f.second, env)
```

1. Look up variables in the current environment

Symbols/relations without variables only unify if they are the same

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

```
( (a  b) c (a  b) )
(   ?x   c   ?x   )
```

env: { x: (a b) }

# Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first,  f.first,  env) and unify(e.second, f.second, env)
```

1. Look up variables in the current environment

Symbols/relations without variables only unify if they are the same

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

```
( (a  b) c (a  b) )
(   ?x   c   ?x   )
```

Lookup

```
(a  b)
(a  b)
```

env: { x: (a b) }

# Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first, f.first, env) and unify(e.second, f.second, env)
```

1. Look up variables in the current environment

Symbols/relations without variables only unify if they are the same

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

```
( (a  b) c (a  b) )
(   ?x   c   ?x   )
```

Lookup

```
(a  b)
(a  b)
```

env: { x: (a b) }

# Search

# Searching for Proofs

# Searching for Proofs

```
The Logic interpreter searches
the space of facts to find
unifying facts and an env that
prove the query to be true.
```

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
```

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
```

```
(app (?a . ?r) ?y (?a . ?z))
```

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
```

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
```

```
(app (e . ?r) (c d) (e b c d))
```

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
```

```
(app (e . ?r) (c d) (e b c d))
```

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
```

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
```

Variables are local to facts & queries

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
```

Variables are local to facts & queries

```
(app (e . ?r) (c d) (e b c d))
```

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
```

Variables are local to facts & queries

```
(app (e . ?r) (c d) (e b c d))
```

```
(app (b . ?r2) (c d) (b c d))
```

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
(app ?r2 (c d) (c d))
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app (b . ?r2) (c d) (b c d))
```

Variables are local to facts & queries

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
(app ?r2 (c d) (c d))

(app () ?x ?x)
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app (b . ?r2) (c d) (b c d))
```

Variables are local to facts & queries

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app (b . ?r2) (c d) (b c d))
```

Variables are local to facts & queries

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app (b . ?r2) (c d) (b c d))
```

Variables are local to facts & queries

```
(app () (c d) (c d))
```

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app (b . ?r2) (c d) (b c d))
```

Variables are local to facts & queries

```
(app () (c d) (c d))
```

*?left*:

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app (b . ?r2) (c d) (b c d))
```

Variables are local to facts & queries

```
(app () (c d) (c d))
```

*?left*:

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))

(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))

(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app (b . ?r2) (c d) (b c d))
```

Variables are local to facts & queries

```
(app () (c d) (c d))
```

*?left*:

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))
(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))
(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app (b . ?r2) (c d) (b c d))
```

Variables are local to facts & queries

```
?left: (e .
```

```
(app () (c d) (c d))
```

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))
(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))
(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

(app (e . ?r) (c d) (e b c d))

(app (b . ?r2) (c d) (b c d))

Variables are local to facts & queries

(app () (c d) (c d))

*?left*: (e .

*?r*:

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))
(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))
(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

(app (e . ?r) (c d) (e b c d))

(app (b . ?r2) (c d) (b c d))

Variables are local to facts & queries

(app () (c d) (c d))

*?left*: **(e .**

*?r*:

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))
(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))
(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

(app (e . ?r) (c d) (e b c d))

(app (b . ?r2) (c d) (b c d))

Variables are local to facts & queries

(app () (c d) (c d))

*?left*: **(e .**

*?r*:

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))
(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))
(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
```

```
(app (b . ?r2) (c d) (b c d))
```

Variables are local to facts & queries

```
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

```
(app () (c d) (c d))
```

*?left*: **(e .**

*?r*: **(b .**

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))
(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))
(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
```

```
(app (b . ?r2) (c d) (b c d))
```

Variables are local to facts & queries

```
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

```
(app () (c d) (c d))
```

*?left*: **(e .**

*?r*: **(b .**

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))
(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))
(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
```

```
(app (b . ?r2) (c d) (b c d))
```

Variables are local to facts & queries

```
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

```
(app () (c d) (c d))
```

*?left*: **(e .**

*?r*: **(b . ())**

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))
(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))
(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
```

```
(app (b . ?r2) (c d) (b c d))
```

Variables are local to facts & queries

```
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

```
(app () (c d) (c d))
```

*?left*: **(e .**

*?r*: **(b . ())** ⇨ **(b)**

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))
(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))
(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
```

```
(app (b . ?r2) (c d) (b c d))
```

Variables are local to facts & queries

```
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

```
(app () (c d) (c d))
```

*?left*: **(e . (b))**

*?r*: **(b . ())** ⇨ **(b)**

# Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact  (app () ?x ?x))
(fact  (app (?a . ?r) ?y (?a . ?z))
       (app       ?r  ?y       ?z ))
(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
    conclusion <- hypothesis
(app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
    conclusion <- hypothesis
(app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
(app () ?x ?x)
```

```
(app (e . ?r) (c d) (e b c d))
```

```
(app (b . ?r2) (c d) (b c d))
```

Variables are local to facts & queries

```
(app () (c d) (c d))
```

*?left*: **(e . (b)) ⇨ (e b)**

*?r*: **(b . ()) ⇨ (b)**

# Depth-First Search

## Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

## Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

## Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

```
def search(clauses, env):
```

## Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

```
def search(clauses, env):
  for fact in facts:
```

# Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

```
def search(clauses, env):
  for fact in facts:
    env_head = an environment extending env
```

# Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

```
def search(clauses, env):
  for fact in facts:
    env_head = an environment extending env
    if unify(conclusion of fact, first clause, env_head):
```

# Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

```
def search(clauses, env):
  for fact in facts:
    env_head = an environment extending env
    if unify(conclusion of fact, first clause, env_head):
```

Environment now contains new unifying bindings

## Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

```
def search(clauses, env):
  for fact in facts:
    env_head = an environment extending env
    if unify(conclusion of fact, first clause, env_head):
      for env_rule in search(hypotheses of fact, env_head):
```

Environment now contains new unifying bindings

# Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

```
def search(clauses, env):
  for fact in facts:
    env_head = an environment extending env
    if unify(conclusion of fact, first clause, env_head):
      for env_rule in search(hypotheses of fact, env_head):
        for result in search(rest of clauses, env_rule):
```

Environment now contains new unifying bindings

# Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

```
def search(clauses, env):
  for fact in facts:
    env_head = an environment extending env
    if unify(conclusion of fact, first clause, env_head):
      for env_rule in search(hypotheses of fact, env_head):
        for result in search(rest of clauses, env_rule):
          yield each successful result
```

Environment now contains new unifying bindings

# Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

```
def search(clauses, env):
  for fact in facts:
    env_head = an environment extending env
    if unify(conclusion of fact, first clause, env_head):
      for env_rule in search(hypotheses of fact, env_head):
        for result in search(rest of clauses, env_rule):
          yield each successful result
```

Environment now contains new unifying bindings

- Limiting depth of the search avoids infinite loops.

# Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

```
def search(clauses, env):
  for fact in facts:
    env_head = an environment extending env
    if unify(conclusion of fact, first clause, env_head):
      for env_rule in search(hypotheses of fact, env_head):
        for result in search(rest of clauses, env_rule):
          yield each successful result
```

Environment now contains new unifying bindings

- Limiting depth of the search avoids infinite loops.
- Each time a fact is used, its variables are renamed.

# Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

```
def search(clauses, env):
  for fact in facts:
    env_head = an environment extending env
    if unify(conclusion of fact, first clause, env_head):
      for env_rule in search(hypotheses of fact, env_head):
        for result in search(rest of clauses, env_rule):
          yield each successful result
```

Environment now contains new unifying bindings

- Limiting depth of the search avoids infinite loops.
- Each time a fact is used, its variables are renamed.
- Bindings are stored in separate frames to allow backtracking.

# Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

```
def search(clauses, env):
  for fact in facts:
    env_head = an environment extending env
    if unify(conclusion of fact, first clause, env_head):
      for env_rule in search(hypotheses of fact, env_head):
        for result in search(rest of clauses, env_rule):
          yield each successful result
```

Environment now contains new unifying bindings

- Limiting depth of the search avoids infinite loops.
- Each time a fact is used, its variables are renamed.
- Bindings are stored in separate frames to allow backtracking.

(Demo)

# Addition

(Demo)