Logic Programming
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

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

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

```
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 (\texttt{abs} \ -3) calls \texttt{abs} on \texttt{-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))
```
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
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
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. 

\[(\text{fact } \langle \text{conclusion}\rangle \ \langle \text{hypothesis}_0\rangle \ \langle \text{hypothesis}_1\rangle \ \ldots \ \langle \text{hypothesis}_N\rangle)\]

Means \(\langle \text{conclusion}\rangle\) is true if all the \(\langle \text{hypothesis}_k\rangle\) are true.
Compound Facts

A fact can include multiple relations and variables as well.

\[
\text{fact } \langle \text{conclusion} \rangle \ \langle \text{hypothesis}_0 \rangle \ \langle \text{hypothesis}_1 \rangle \ \ldots \ \langle \text{hypothesis}_N \rangle
\]

Means \langle \text{conclusion} \rangle \ is \ true \ if \ all \ the \ \langle \text{hypothesis}_k \rangle \ are \ true.

\[
\text{logic} \succ (\text{fact} \ (\text{child} \ ?c \ ?p) \ (\text{parent} \ ?p \ ?c))
\]
**Compound Facts**

A fact can include multiple relations and variables as well.

\[(\text{fact } \langle \text{conclusion} \rangle \langle \text{hypothesis}_0 \rangle \langle \text{hypothesis}_1 \rangle ... \langle \text{hypothesis}_N \rangle)\]

Means \(\langle \text{conclusion} \rangle\) is true if all the \(\langle \text{hypothesis}_K \rangle\) are true.

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

\[
\text{logic} > (\text{query} (\text{child} \text{ herbert delano}))
\]
Compound Facts

A fact can include multiple relations and variables as well.

\[ \text{fact} \ <\text{conclusion}> \ <\text{hypothesis}_0> \ <\text{hypothesis}_1> \ldots \ <\text{hypothesis}_N> \]

Means \(<\text{conclusion}>\) is true if all the \(<\text{hypothesis}_k>\) are true.

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

\text{logic}\> (\text{query} \ (\text{child} \ \text{herbert delano}))

\text{Success}!
Compound Facts

A fact can include multiple relations and variables as well.

\[(\text{fact <conclusion>} \ <\text{hypothesis}_0> \ <\text{hypothesis}_1> \ ... \ <\text{hypothesis}_N>)\]

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


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

logic> (query (child eisenhower clinton))
Compound Facts

A fact can include multiple relations and variables as well.

\[
\text{(fact} \ <\text{conclusion}> \ <\text{hypothesis}_0> \ <\text{hypothesis}_1> \ ... \ <\text{hypothesis}_N>)
\]

Means \(<\text{conclusion}>\) is true if all the \(<\text{hypothesis}_K>\) are true.


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

logic> (query (child eisenhower clinton))
Failure.
**Compound Facts**

A fact can include multiple relations and variables as well.

\[(\text{fact} \ <\text{conclusion}> \ <\text{hypothesis}_0> \ <\text{hypothesis}_1> \ ... \ <\text{hypothesis}_N>)\]

Means \(<\text{conclusion}>\) is true if all the \(<\text{hypothesis}_k>\) are true.

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

\[
\text{logic} > (\text{query} \ (\text{child} \ \text{herbert delano}))
\text{Success!}
\]

\[
\text{logic} > (\text{query} \ (\text{child} \ \text{eisenhower clinton}))
\text{Failure.}
\]

\[
\text{logic} > (\text{query} \ (\text{child} \ ?kid \ \text{fillmore}))
\]
Compound Facts

A fact can include multiple relations and variables as well.

\[
\text{fact } \langle \text{conclusion} \rangle \langle \text{hypothesis}_0 \rangle \langle \text{hypothesis}_1 \rangle \cdots \langle \text{hypothesis}_N \rangle
\]

Means \( \langle \text{conclusion} \rangle \) is true if all the \( \langle \text{hypothesis}_K \rangle \) are true.

\[
\text{logic} > (\text{fact} \ (\text{child} \ ?c \ ?p) \ (\text{parent} \ ?p \ ?c))
\]
\[
\text{logic} > (\text{query} \ (\text{child} \ \text{herbert} \ \text{delano}))
\text{Success!}
\]
\[
\text{logic} > (\text{query} \ (\text{child} \ \text{eisenhower} \ \text{clinton}))
\text{Failure.}
\]
\[
\text{logic} > (\text{query} \ (\text{child} \ ?\text{kid} \ \text{fillmore}))
\text{Success!}
\]
Compound Facts

A fact can include multiple relations and variables as well.

\[
(\text{fact} \ <\text{conclusion}> \ <\text{hypothesis}_0> \ <\text{hypothesis}_1> \ ... \ <\text{hypothesis}_N>)
\]

Means \(<\text{conclusion}()>\) is true if all the \(<\text{hypothesis}_k>\) are true.

\[
\text{logic}> (\text{fact} \ (\text{child} \ ?c \ ?p) \ (\text{parent} \ ?p \ ?c))
\]
\[
\text{logic}> (\text{query} \ (\text{child} \ \text{herbert} \ \text{delano}))
\text{Success!}
\]
\[
\text{logic}> (\text{query} \ (\text{child} \ \text{eisenhower} \ \text{clinton}))
\text{Failure.}
\]
\[
\text{logic}> (\text{query} \ (\text{child} \ ?kid \ \text{fillmore}))
\text{Success!}
\text{kid: abraham}
\]
Compound Facts

A fact can include multiple relations and variables as well.

\[
\text{fact} \langle \text{conclusion} \rangle \langle \text{hypothesis}_0 \rangle \langle \text{hypothesis}_1 \rangle \ldots \langle \text{hypothesis}_N \rangle
\]

Means \text{conclusion} is true if all the \text{hypothesis}_K are true.


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

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

logic> (query (child ?kid fillmore))
Success!
kid: abraham
kid: delano
Compound Facts

A fact can include multiple relations and variables as well.

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

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


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

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

logic> (query (child ?kid fillmore))
Success!
kid: abraham
kid: delano
kid: grover
Compound Queries
Compound Queries

An assignment must satisfy all relations in a query.
Compound Queries

An assignment must satisfy all relations in a query.

\[(query <relation_0> <relation_1> \ldots <relation_N>)\]
Compound Queries

An assignment must satisfy all relations in a query.

\[(\text{query} \ <\text{relation}_0> \ <\text{relation}_1> \ \ldots \ <\text{relation}_N>)\]

is satisfied if all the \(<\text{relation}_k>\) are true.
Compound Queries

An assignment must satisfy all relations in a query.

\[(\text{query } \langle \text{relation}_0 \rangle \langle \text{relation}_1 \rangle \ldots \langle \text{relation}_N \rangle)\]

is satisfied if all the \(\langle \text{relation}_K \rangle\) are true.

Compound Queries

An assignment must satisfy all relations in a query.

\[(\text{query} \; \langle \text{relation}_0 \rangle \; \langle \text{relation}_1 \rangle \; \ldots \; \langle \text{relation}_N \rangle)\]

is satisfied if all the \( \langle \text{relation}_K \rangle \) are true.

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

\[
\text{logic} > (\text{query} \; (\text{parent} \; ?\text{grampa} \; ?\text{kid})
\]

![Family Tree Diagram]

- Delano
- Herbert
- Fillmore
- Eisenhower
Compound Queries

An assignment must satisfy all relations in a query.

\[(\text{query} \ <\text{relation}_0> \ <\text{relation}_1> \ ... \ <\text{relation}_N>)\]

is satisfied if all the \(<\text{relation}_k>\) are true.

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

\[
\text{logic} > (\text{query} \ (\text{parent} \ ?grampa \ ?kid) \ (\text{child} \ \text{clinton} \ ?kid))
\]
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> (query (parent ?grampa ?kid)
              (child clinton ?kid))

Success!
Compound Queries

An assignment must satisfy all relations in a query.

\[
(\text{query } \langle \text{relation}_0 \rangle \langle \text{relation}_1 \rangle \ldots \langle \text{relation}_N \rangle)
\]

is satisfied if all the \( \langle \text{relation}_k \rangle \) are true.

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

An assignment must satisfy all relations in a query.

\[(\text{query } \text{relation}_0 \text{ relation}_1 \ldots \text{relation}_N)\]

is satisfied if all the \(\text{relation}_k\) are true.

```
```

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

**Success!**

grampa: fillmore kid: abraham

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

```
Eisenhower

Fillmore

Abraham

 Barack

Clinton

 Herbert

 Delano

Grover

```
Compound Queries

An assignment must satisfy all relations in a query.

\[(\text{query } \langle \text{relation}_0 \rangle \langle \text{relation}_1 \rangle \ldots \langle \text{relation}_N \rangle)\]

is satisfied if all the \(\langle \text{relation}_K \rangle\) are true.

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

\[
\text{logic} > (\text{query} (\text{parent} ?grampa ?kid) (\text{child clinton} ?kid))
\]

Success!

gorampa: fillmore       kid: abraham

\[
\text{logic} > (\text{query} (\text{child} ?y ?x) (\text{child} ?x eisenhower))
\]
Compound Queries

An assignment must satisfy all relations in a query.

\[(\text{query } \text{relation}_0 \text{ relation}_1 \ldots \text{ relation}_N)\]

is satisfied if all the \(\text{relation}_k\) are true.

```
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.

\[(\text{query} \ <\text{relation}_0> \ <\text{relation}_1> \ ... \ <\text{relation}_N>)\]

is satisfied if all the \(<\text{relation}_K>\) are true.

```
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
```
**Compound Queries**

An assignment must satisfy all relations in a query.

\[
\text{(query } \langle \text{relation}_0 \rangle \ \langle \text{relation}_1 \rangle \ \ldots \ \langle \text{relation}_N \rangle \text{)}
\]

is satisfied if all the \( \langle \text{relation}_K \rangle \) are true.

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

\[\text{logic} \ (\text{query (parent ?grampa ?kid) (child clinton ?kid)})\]

**Success!**
grampa: fillmore    kid: abraham

\[\text{logic} \ (\text{query (child ?y ?x) (child ?x eisenhower)})\]

**Success!**
y: abraham    x: fillmore
y: delano    x: fillmore
Compound Queries

An assignment must satisfy all relations in a query.

\[(\text{query } \text{<relation}_0 > \text{<relation}_1 > \ldots \text{<relation}_N >)\]

is satisfied if all the \text{<relation}_k are true.

```
```

```
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
y: delano      x: fillmore
y: grover      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.

\[
\text{logic} \triangleright (\text{fact} (\text{ancestor} \ ?a \ ?y) (\text{parent} \ ?a \ ?y))
\]
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))
```
Recursive Facts

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

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


logic> (query (ancestor ?a herbert))
Recursive Facts

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

```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (query (ancestor ?a herbert))
Success!
```
Recursive Facts

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

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


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

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

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

a: delano
a: fillmore
Recursive Facts

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

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

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

\[
\text{logic} > (\text{query} (\text{ancestor} \ ?a \ \text{herbert}))
\]

Success!

a: delano
a: fillmore
a: eisenhower

![Diagram](image-url)
Recursive Facts

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

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

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

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

\[
\text{logic} > (\text{query} (\text{ancestor} ?a \text{ barack})
\]

![Diagram of family tree with names representing recursive facts]
Recursive Facts

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

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

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

\[
\text{logic}> (\text{query} \ (\text{ancestor} \ ?a \ \text{herbert}))
\]

\text{Success!}

a: delano
a: fillmore
a: eisenhower

\[
\text{logic}> (\text{query} \ (\text{ancestor} \ ?a \ \text{barack})
\quad (\text{ancestor} \ ?a \ \text{herbert}))
\]

![Family Tree Diagram]
Recursive Facts

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

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

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

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

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

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

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

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

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

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

\texttt{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))
```
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))

(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))

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

(parent delano herbert) ; (1), a simple fact
(ancestor delano herbert) ; (2), from (1) and the 1st ancestor fact
(parent fillmore delano) ; (3), a simple fact
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))

(ancestor delano herbert) ; (2), from (1) and the 1st ancestor 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
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
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?

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

![Diagram of a tree structure with nodes labeled A, B, C, D, E, F, G, and H, indicating relationships between the dogs and their ancestors.]
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 (app () ?x ?x))
Lists in Logic

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

Expressions and their relations are Scheme lists.

\[(\text{fact} \ (\text{app} \ () \ ?x \ ?x))\]  
*Simple fact: Conclusion*
Lists in Logic

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

Expressions and their relations are Scheme lists.

\[
\text{fact (app () ?x ?x)) \quad \text{(Simple fact: Conclusion)}
\]

\[
\text{fact (app (?a . ?r) ?y (?a . ?z))}
\]
\[
\text{(app ?r ?y ?z )}
\]

\[
\text{query (app ?left (c d) (e b c d)))}
\]

*Success!*

*left*: (e b)
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```scheme
(fact (app () ?x ?x))  < Simple fact: Conclusion
(fact (app (?a . ?r) ?y (?a . ?z))  < Conclusion
  (app       ?r  ?y       ?z ))

(query (app ?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 (app () ?x ?x))  \hspace{1cm} \text{Simple fact: Conclusion}

(fact (app (?a . ?r) ?y (?a . ?z))
  (app ?r ?y ?z )) \hspace{1cm} \text{Conclusion}

\hspace{1cm} \text{Hypothesis}

(query (app ?left (c d) (e b c d)))

\textbf{Success!}

\textbf{left: (e b)}
Lists in Logic

Expressions begin with query or fact followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (app () ?x ?x))  Simple fact: Conclusion
(fact (app (?a . ?r) ?y (?a . ?z))  Conclusion
  (app ?r ?y ?z ))  Hypothesis

(query (app ?left (c d) (e b c d)))
Success!
left: (e b)  What ?left can append with (c d) to create (e b c d)
```
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

\[
\text{(fact \ (app ~_~ ?x ~_~ ?x))}
\]

\[
\text{(fact \ (app ~_~ ?a . ~_~ ?r ~_~ ?y ~_~ (?a . ~_~ ?z))}
\]

\[
\text{(app ~_~ ?r ~_~ ?y ~_~ ?z ))}
\]

\[
\text{(query \ (app ~_~ ?left ~_~ (c d) ~_~ (e b c d)))}
\]

\[
\text{Success!}
\]

\[
\text{left: \ (e b)}
\]

\[
\text{What ?left can append with (c d) to create (e b c d)}
\]

\[
\text{(c d) => (c d)}
\]
Lists in Logic

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

Expressions and their relations are Scheme lists.

\[
\text{(fact (app () ?x ?x))}
\]

\[
\text{(fact (app (?a . ?r) ?y (?a . ?z))}
\]

\[
\text{(app ?r ?y ?z )}
\]

\[
\text{(query (app ?left (c d) (e b c d)))}
\]

Success!
left: (e b)

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

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

Expressions and their relations are Scheme lists.

- `(fact (app () ?x ?x))`
- `(query (app ?left (c d) (e b c d)))`

Success!

**left:** `(e b)`

What `?left` can append with `(c d)` to create `(e b c d)`
Lists in Logic

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

Expressions and their relations are Scheme lists.

\[
\text{(fact (app () ?x ?x))}
\]

\[
\text{(fact (app (?a . ?r) ?y (?a . ?z))}
\]

\[
\text{(app ?r ?y ?z))}
\]

\[
\text{(query (app ?left (c d) (e b c d)))}
\]

Success!

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

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

Expressions and their relations are Scheme lists.

- \((\text{fact} \ (\text{app} \ () \ ?x \ ?x))\)
- \((\text{fact} \ (\text{app} \ (?a \ . \ ?r) \ ?y \ (?a \ . \ ?z)) \ (\text{app} \ ?r \ ?y \ ?z))\)
- \((\text{query} \ (\text{app} \ ?\text{left} \ (c \ d) \ (e \ b \ c \ d)))\)

**Success!**

- left: \((e \ b)\)

What ?left can append with \((c \ d)\) to create \((e \ b \ c \ d)\)
Lists in Logic

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

Expressions and their relations are Scheme lists.

\[
\text{(fact (app () ?x ?x))}
\]

\[
\text{(fact (app (?a . ?r) ?y (?a . ?z))}
\]

\[
\text{(app ?r ?y ?z))}
\]

\[
\text{(query (app ?left (c d) (e b c d)))}
\]

Success!

left: (e b)

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

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

Expressions and their relations are Scheme lists.

- (fact (app () ?x ?x))  \(\text{Simple fact: Conclusion}\)
- (fact (app (?a . ?r) ?y (?a . ?z))  \(\text{Conclusion}\)
  (app ?r ?y ?z ))  \(\text{Hypothesis}\)

(query (app ?left (c d) (e b c d)))

Success!

left: (e b)  \(\text{What ?left can append with (c d) to create (e b c d)}\)
Lists in Logic

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

Expressions and their relations are Scheme lists.

- (fact (app () ?x ?x))  
  Simple fact: Conclusion

- (fact (app (?a . ?r) ?y (?a . ?z))  
  Conclusion

- (app       ?r  ?y       ?z ))  
  Hypothesis

- (query (app ?left (c d) (e b c d)))
  Success!

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

- () (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
Lists in Logic

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

Expressions and their relations are Scheme lists.

(\(\text{fact (app () ?x ?x)}\))  \(\text{Simple fact: Conclusion}\)

(\(\text{fact (app (?a . ?r) ?y (?a . ?z))}\)) \(\text{Conclusion}\)

(\(\text{query (app ?left (c d) (e b c d))}\))  \(\text{Success! left: (e b)}\)

What ?left can append with (c d) to create (e b c d)
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

- (fact (app () ?x ?x))  
  - Simple fact: Conclusion
- (fact (app (?a . ?r) ?y (?a . ?z))
  - (app ?r ?y ?z))
  - Conclusion
- (query (app ?left (c d) (e b c d)))
  - Success!
  - left: (e b)
  - What ?left can append with (c d) to create (e b c d)
Lists in Logic

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

Expressions and their relations are Scheme lists.

\[(\text{fact } (\text{app } () \ ?x \ ?x))\]  
**Simple fact: Conclusion**

\[(\text{fact } (\text{app } (?a . \ ?r) \ ?y \ (?a . \ ?z)) \ (\text{app } \ ?r \ ?y \ ?z ))\]  
**Conclusion**  
**Hypothesis**

\[(\text{query } (\text{app } \ ?\text{left } (c \ d) \ (e \ b \ c \ d))))\]  
**Success!**  
**left**: \((e \ b)\)  
What \(?\text{left}\) can append with \((c \ d)\) to create \((e \ b \ c \ d)\)

\[() \ (c \ d) \Rightarrow (c \ d)\]  
\[(b) \ (c \ d) \Rightarrow (b \ c \ d)\]  
\[(e \ b) \ (c \ d) \Rightarrow (e \ b \ c \ d)\]  
\[(e . \ (b)) \ (c \ d) \Rightarrow (e . \ (b \ c \ d))\]
Lists in Logic

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

Expressions and their relations are Scheme lists.

(fact (app () ?x ?x))  \(<\text{Simple fact: Conclusion}>\)

(fact (app (?a . ?r) ?y (?a . ?z))
  (app ?r ?y ?z ))  \(<\text{Conclusion}>\)

(query (app ?left (c d) (e b c d)))

\textbf{Success!}

\textbf{left:} (e b)

\textbf{What} ?left can append with (c d) to create (e b c d)
Lists in Logic

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

Expressions and their relations are Scheme lists.

(fact (app () ?x ?x))  \(\rightarrow\) Simple fact: Conclusion

(fact (app (?a . ?r) ?y (?a . ?z))
    (app ?r ?y ?z ))  \(\rightarrow\) Conclusion

(query (app ?left (c d) (e b c d)))
Success!
left: (e b)  \(\rightarrow\) What ?left can append with (c d) to create (e b c d)
Lists in Logic

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

Expressions and their relations are Scheme lists.

- **Simple fact: Conclusion**
  - \((\text{fact } (\text{app } () ?x ?x))\)
  - \((\text{fact } (\text{app } (?a . ?r) ?y (?a . ?z))\)\)

- **Hypothesis**
  - \((\text{app } ?r ?y ?z ))\)

- **Conclusion**
  - \((\text{query } (\text{app } ?\text{left } (c d) (e b c d)))\)

Success!

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

\(\) \((c d)\) \(\Rightarrow\) \((c d)\)

\((b)\) \((c d)\) \(\Rightarrow\) \((b c d)\)

\((e b)\) \((c d)\) \(\Rightarrow\) \((e b c d)\)

\((e . (b)):\) \((c d)\) \(\Rightarrow\) \((e . (b c d)):\)
Lists in Logic

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

Expressions and their relations are Scheme lists.

- (fact (app () ?x ?x))  
  Simple fact: Conclusion

- (fact (app (?a . ?r) ?y (?a . ?z))  
  Conclusion
  (app ?r ?y ?z ))  
  Hypothesis

- (query (app ?left (c d) (e b c d)))
  Success!
  left: (e b)  
  What ?left can append with (c d) to create (e b c d)
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

\[
\text{(fact (app () ?x ?x))} \quad \text{Simple fact: Conclusion}
\]

\[
\text{(fact (app (?a . ?r) ?y (?a . ?z))}
\text{(app ?r ?y ?z ))} \quad \text{Conclusion}
\]

\[
\text{(query (app ?left (c d) (e b c d))}}
\text{Success!}
\text{left: (e b)} \quad \text{What ?left can append with (c d) to create (e b c d)}
\]
Lists in Logic

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

Expressions and their relations are Scheme lists.

- \((\text{fact } (\text{app } () \ ?x \ ?x))\)  
  - Simple fact: Conclusion

- \((\text{fact } (\text{app } (?a \ . \ ?r) \ ?y \ (?a \ . \ ?z))\)  
  - Conclusion

- \((\text{app } \ ?r \ ?y \ ?z ))\)  
  - Hypothesis

- \((\text{query } (\text{app } \ ?\text{left } (c \ d) \ (e \ b \ c \ d))))\)  
  - Success!

  - left: \((e \ b)\)  
    - What \(?\text{left}\) can append with \((c \ d)\) to create \((e \ b \ c \ d)\)

The interpreter lists all bindings that it can find to satisfy the query.
List in Logic

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

Expressions and their relations are Scheme lists.

- `(fact (app () ?x ?x))`  
  *Simple fact: Conclusion*

  *Conclusion*

- `(query (app ?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 that it can find to satisfy the query.

(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 \textit{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.

\[
\begin{align*}
( (a \ b) \ c \ & \ (a \ b) ) \\
( & \ ?x \ c \ ?x \ )
\end{align*}
\]
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.

\[
\begin{align*}
( (a \ b) & \ c & (a \ b) ) \\
( \ ?x \ c & ?x ) & \quad \text{True, \ \{x: (a \ b)\}}
\end{align*}
\]
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.

\[
\begin{align*}
( (a \ b) & \ c \ (a \ b) ) \\
( \quad ?x & \ c \quad ?x \quad ) & \quad \text{True, } \{x: (a \ b)\} \\
( (a \ b) & \ c \ (a \ b) )
\end{align*}
\]
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.

\[
\begin{align*}
( (a \ b) & c \ (a \ b) ) \\
( ?x & c & ?x & ) & \text{ True, } \{x: (a \ b)\} \\
( (a \ b) & c & (a \ b) ) \\
( (a \ ?y) & ?z & (a \ b) )
\end{align*}
\]
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.

\[
( (a \ b) \ c \ (a \ b) ) \quad \text{True, \{x: (a b)\}}
\]

\[
( \ ?x \ c \ ?x \ ) \quad \text{True, \{x: (a b)\}}
\]

\[
( (a \ b) \ c \ (a \ b) ) \quad \text{True, \{y: b, z: c\}}
\]

\[
( (a \ ?y) \ ?z \ (a \ b) )
\]
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) )  True, {x: (a b)}
(   ?x   c    ?x   )

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

( (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.

- \(( (a \ b) \ c \ (a \ b) )\)  \(( ?x \ c \ ?x \ )\)  \(\text{True, } \{x: (a \ b)\}\)
- \(( (a \ b) \ c \ (a \ b) )\)  \(( (a \ ?y) \ ?z \ (a \ b) )\)  \(\text{True, } \{y: b, \ z: c\}\)
- \(( (a \ b) \ c \ (a \ b) )\)  \(( ?x \ ?x \ ?x \ )\)  \(\text{False}\)
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.

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

```plaintext
{
}
```
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.

\[
\begin{align*}
&\text{( (a b) c (a b) )} \\
&\text{( ?x c ?x )}
\end{align*}
\]
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.

\[
(\begin{array}{c}
(a \ b) \\
?x \\
\end{array} \quad c \quad (a \ b)
\end{array})
\]

\[
(\begin{array}{c}
?x \\
c \\
?x
\end{array})
\]

\{
\text{x: (a b)}
\}
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.

\[
\begin{align*}
\text{Unification} & \quad \text{Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.} \\
1. \text{Look up variables in the current environment.} & \\
2. \text{Establish new bindings to unify elements.} & \\
\end{align*}
\]

\[
\begin{align*}
\text{Unification} & \quad \text{Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.} \\
1. \text{Look up variables in the current environment.} & \\
2. \text{Establish new bindings to unify elements.} & \\
\end{align*}
\]
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) (a b)
(c ?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.

\[
\begin{align*}
&\begin{array}{c}
(a \ b) \\
?x \\
\end{array}
&\begin{array}{c}
\text{c} \\
\text{c} \\
\end{array}
&\begin{array}{c}
(a \ b) \\
?x \\
\end{array}
\end{align*}
\]

Lookup

\[
\begin{align*}
&(a \ b) \\
&(a \ b) \\
&\{x: (a \ b)\}
\end{align*}
\]
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)  
(  ?x   c   ?x   )
Lookup

(a b)
(a b)

{x: (a b)}
```

Success!
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.

\[
\begin{align*}
&(a \ b) \ c \ (a \ b) \\
&(?x \ c \ ?x \\
& \text{Lookup}
\end{align*}
\]

\[
\begin{align*}
&(a \ b) \\
&(a \ b)
\end{align*}
\]

\[
\begin{align*}
&\{ \ x: (a \ b) \ \} \\
&\text{Success!}
\end{align*}
\]
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.

\[
\begin{align*}
&\text{Unification:} \\
&\text{Success!}
\end{align*}
\]
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.

\[
\begin{align*}
&\left( \begin{array}{c} \text{(a b)} \\ \text{c} \end{array} \right) \quad \left( \begin{array}{c} \text{(a b)} \\ \text{c} \end{array} \right) \\
&\left( \begin{array}{c} \text{?x} \\ \text{c} \end{array} \right) \quad \left( \begin{array}{c} \text{?x} \\ \text{c} \end{array} \right)
\end{align*}
\]

\[
\begin{align*}
&\{ \text{x: (a b)} \} \\
&\text{Success!}
\end{align*}
\]
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.

\[
\begin{align*}
&((a, b), c, (a, b)) \\
&((?, x), c, (?, x))
\end{align*}
\]

\[
\begin{align*}
&((a, b), c, (a, b)) \\
&((?, x), ?, x, ?x)
\end{align*}
\]

\[
\{ \text{x: (a b)} \}
\]

\[
\{ \text{x: (a b)} \}
\]

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.

\[
\begin{align*}
\text{Unification} & : (\ (a\ b) \ ) \ c \ (a\ b) \\
\text{Unification} & : (\ ?x \ c \ ?x \ )
\end{align*}
\]

\{
\begin{align*}
x & : \ (a\ b) \ \\
\text{Success!}
\end{align*}
\}
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.

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

Success!

Failure.

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

\[( \ ?x \ ?x \ ) \quad \rightarrow \quad \text{True, } \{\]

\[((a \ ?y \ c) \ (a \ b \ ?z))\]
Unifying Variables

Two relations that contain variables can be unified as well.

\[
\begin{align*}
\{ \quad \text{True,} \quad \{ \\
\}
\end{align*}
\]
Unifying Variables

Two relations that contain variables can be unified as well.

\[
\begin{align*}
\text{True, } \{x: (a \ ?y \ c), \}
\end{align*}
\]
Unifying Variables

Two relations that contain variables can be unified as well.

\[
\begin{align*}
\text{True, } \{x: (a \ ?y \ c), \}
\end{align*}
\]
Unifying Variables

Two relations that contain variables can be unified as well.

True, \{x: (a ?y c), ...

Lookup

(a ?y c)

(a b ?z)
Unifying Variables

Two relations that contain variables can be unified as well.

\[
\begin{align*}
&\{ x: (a ~?y~ c), (a~ b~ ?z) \} \\
&\text{True,} \{ x: (a ~?y~ c), (a~ b~ ?z) \}
\end{align*}
\]
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.

\[
\begin{align*}
\text{True, } & \{ x: (a \ ?y \ c), \\
& y: b, \\
& \text{Lookup} \}
\end{align*}
\]
Unifying Variables

Two relations that contain variables can be unified as well.

\[
\begin{align*}
&\text{(x)} & &\text{(x)} \\
&(a \ ?y \ c) & &(a \ b \ ?z) \\
\end{align*}
\]

True, \{x: (a ?y c), y: b, \}
Two relations that contain variables can be unified as well.

True, \{x: (a ?y c), y: b, z: c\}
Unifying Variables

Two relations that contain variables can be unified as well.

\[
\begin{array}{c}
?x \\
(a \ ?y \ c) \\
(a \ b \ ?z)
\end{array}
\quad \quad \quad \quad \quad \quad
?x \\
((a \ ?y \ c) \\
(a \ b \ ?z))
\]

True, \ \{x: (a \ ?y \ c), \\
y: b, \\
z: c\}\}
Two relations that contain variables can be unified as well.

\[
\begin{align*}
\langle x &\rangle \\
\langle a \ ?y \ c \rangle
\end{align*}
\quad \quad 
\begin{align*}
\langle x &\rangle \\
\langle a \ b \ ?z \rangle
\end{align*}
\quad \quad 
\text{True, } \{x: (a \ ?y \ c),} \\
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{y: b,}} \\
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{z: c}\}
\]

Substituting values for variables may require multiple steps.

This process is called \textit{grounding}. Two unified expressions have the same grounded form.
Unifying Variables

Two relations that contain variables can be unified as well.

\[
\begin{align*}
&\text{(} ?x \text{)} \\
&\text{(} (a \ ?y \ c) \text{)} \\
&\text{(} (a \ b \ ?z) \text{)}
\end{align*}
\]

\[\text{True, } \{x: (a \ ?y \ c), \ y: b, \ z: c\}\]

Substituting values for variables may require multiple steps.

This process is called \textit{grounding}. Two unified expressions have the same grounded form.

\texttt{lookup('?x')}
Unifying Variables

Two relations that contain variables can be unified as well.

\[
\begin{align*}
\text{(a ?y c)} & \quad \text{(?x)} \\
\text{(a b ?z)} & \quad \text{(a ?y c)}
\end{align*}
\]

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

Substituting values for variables may require multiple steps.

This process is called \textit{grounding}. Two unified expressions have the same grounded form.

\texttt{lookup('?x')} \rightarrow (a ?y c)
Unifying Variables

Two relations that contain variables can be unified as well.

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.

\texttt{lookup('?x')} \rightarrow (a ?y c) \quad \texttt{lookup('?y')}
Unifying Variables

Two relations that contain variables can be unified as well.

\[(\text{a} \ ?y \ c) \quad (\text{a} \ \text{b} \ \text{c})\]

True, \{x: (a \ ?y \ c),
\quad y: \text{b},
\quad z: \text{c}\}

Substituting values for variables may require multiple steps.

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

\[\text{lookup('?x')} \Rightarrow (\text{a} \ \text{y} \ c) \quad \text{lookup('?y')} \Rightarrow \text{b}\]
Unifying Variables

Two relations that contain variables can be unified as well.

\[
\begin{align*}
\text{(a } ?y \ c) & \quad \text{(a } b \ ?z) \\
?x & \quad ?x
\end{align*}
\]

True, \{x: \text{(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.

\[
\text{lookup('?x')} \Rightarrow (a \ ?y \ c) \quad \text{lookup('?y')} \Rightarrow b \quad \text{ground('?x')}
\]
Unifying Variables

Two relations that contain variables can be unified as well.

\[
\begin{align*}
\text{lookup}(\text{'?x'}) & \Rightarrow (a \ ?y \ c) & \text{lookup}(\text{'?y'}) & \Rightarrow b & \text{ground}(\text{'?x'}) & \Rightarrow (a \ b \ c)
\end{align*}
\]

Substituting values for variables may require multiple steps.

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

```python
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.
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

```python
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.

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

Recursively unify the first and rest of any lists.
Implementing Unification

```python
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

```python
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)
```

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

1. Look up variables in the current environment.

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

```
env: {
      }
```

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

( ?x c ?x )
```
Implementing Unification

```python
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.

```
(env: {  
(a b) c (a b)  
?x c ?x  
})
```
Implementing Unification

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

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.

env: { }
Implementing Unification

```python
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.

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

Recursively unify the first and rest of any lists.

( (a b) c (a b) )

( ?x c ?x )

env: { x: (a b) }
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)

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

1. Look up variables in the current environment

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

env: { x: (a b) }
Implementing Unification

```python
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)
```

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

1. Look up variables in the current environment.

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

```python
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)
```

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

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

Recursively unify the first and rest of any lists.

```
(env: { x: (a b) })
((a b) c (a b))
((?x c ?x))
```
Implementing Unification

```python
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.

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

Recursively unify the first and rest of any lists.

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

```
(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)
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)))
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 (?a2 . ?r2) ?y2 (?a2 . ?z2))
(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 (?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
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
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))

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.

\[
\text{(fact (app ( ) ?x ?x))}
\]

\[
\text{(fact (app (?a . ?r) ?y (?a . ?z))}
\]

\[
\text{(app ?r ?y ?z))}
\]

\[
\text{(query (app ?left (c d) (e b c d)))}
\]

\[
\text{(app ?left (c d) (e b c d))}
\]

\[
\{a: e, y: (c d), z: (b c d), left: (?a . ?r)\}
\]

\[
\text{(app (?a . ?r) ?y (?a . ?z))}
\]

\[
\text{conclusion <- hypothesis}
\]

\[
\text{(app ?r (c d) (b c d))}
\]

\[
\{a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)\}
\]

\[
\text{(app (?a2 . ?r2) ?y2 (?a2 . ?z2))}
\]

\[
\text{conclusion <- hypothesis}
\]

\[
\text{(app ?r2 (c d) (c d))}
\]

\[
\text{Variables are local to facts & queries}
\]

\[
\text{(app ( ) ?x ?x)}
\]
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.

\[
\text{(query } (\text{app left (c d) (e b c d)})\text{)}
\]

\[
\text{(fact } (\text{app () ?x ?x})\text{)}
\]

\[
\text{(fact } (\text{app (?a . ?r) ?y (?a . ?z)})
\text{(app ?r ?y ?z}))\text{)}
\]

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)

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

?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))

(app (b . ?r2) (c d) (b c d))

Variables are local to facts & queries

?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 () (c d) (c d))

(app () ?x ?x)

Variables are local to facts & queries

?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)

Variables are local to facts & queries

?left: (e .)

(app (e . ?r) (c d) (e b c d))

(app (b . ?r2) (c d) (b c d))

(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)

?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)

Variables are local to facts & queries

(app (e . ?r) (c d) (e b c d))

(app (b . ?r2) (c d) (b 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.

\[
\text{app} \ ?\text{left} \ (c \ d) \ (e \ b \ c \ d))
\]

\[
\{a: e, \ y: (c \ d), \ z: (b \ c \ d), \ \text{left: } (?a . \ ?r)\}
\]

\[
\text{app} \ ?\text{r} \ (c \ d) \ (b \ c \ d))
\]

\[
\{a2: b, \ y2: (c \ d), \ z2: (c \ d), \ r: (?a2 . \ ?r2)\}
\]

\[
\text{app} \ ?\text{r2} \ (c \ d) \ (c \ d))
\]

\[
\{r2: (), \ x: (c \ d)\}
\]

\[
\text{app} \ () \ ?\text{x} \ ?\text{x})
\]

Variables are local to facts & queries

\[
\text{conclusion <- hypothesis}
\]

\[
\text{app} \ ?\text{r} \ (c \ d) \ (e \ b \ c \ d))
\]

\[
\{a: e, \ y: (c \ d), \ z: (b \ c \ d), \ \text{left: } (?a . \ ?r)\}
\]

?\text{left}: (e .

?\text{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 (e . ?r) (c d) (e b c d))
```

```
(app (?a . ?r) ?y (?a . ?z))
```

```
(app (e . ?r) (c d) (e b c d))
```

```
conclusion <- hypothesis
(app ?r (c d) (b c d))
  {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
```

```
(app (b . ?r2) (c d) (b c d))
```

```
(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
```

```
(app (e . ?r) (c d) (e b c d))
```

```
conclusion <- hypothesis
(app ?r2 (c d) (c d))
  {r2: (), x: (c d)}
```

```
(app () (c d) (c d))
```

```
(app () ?x ?x)
```

```
Variables are local to facts & queries
```

```
?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)))

variables are local to facts & queries

?left: (e .
?r: (b .

Variables are local to facts & queries

?left: (e .
?r: (b .

Variables are local to facts & queries

?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.

(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 () (c d) (c d))

(app () ?x ?x)

(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 (e . ?r) (c d) (e b c d))

(app (b . ?r2) (c d) (b c d))

(app (e . ?r) (c d) (e b c d))

(app (b . ?r2) (c d) (b c d))

?left: (e .)

?r: (b . ())

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)
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.

\begin{align*}
(fact \ (app \ () \ ?x \ ?x)) \\
(fact \ (app \ (?a \ . \ ?r) \ ?y \ (?a \ . \ ?z)) \\
\quad (app \ ?r \ ?y \ ?z)) \\
(query \ (app \ ?left \ (c \ d) \ (e \ b \ c \ d)))
\end{align*}

\begin{align*}
(app \ ?left \ (c \ d) \ (e \ b \ c \ d)) \\
\{a: e, \ y: (c \ d), \ z: (b \ c \ d), \ left: (?a \ . \ ?r)\}
\end{align*}

\begin{align*}
(app \ (?a \ . \ ?r) \ ?y \ (?a \ . \ ?z)) \\
\text{conclusion} \leftarrow \text{hypothesis}
\end{align*}

\begin{align*}
(app \ ?r \ (c \ d) \ (b \ c \ d)) \\
\{a2: b, \ y2: (c \ d), \ z2: (c \ d), \ r: (?a2 \ . \ ?r2)\}
\end{align*}

\begin{align*}
(app \ (?a2 \ . \ ?r2) \ ?y2 \ (?a2 \ . \ ?z2)) \\
\text{conclusion} \leftarrow \text{hypothesis}
\end{align*}

\begin{align*}
(app \ ?r2 \ (c \ d) \ (c \ d)) \\
\{r2: (), \ x: (c \ d)\}
\end{align*}

\begin{align*}
(app \ () \ (c \ d) \ (c \ d)) \\
(app \ () \ ?x \ ?x)
\end{align*}

\begin{align*}
(app \ (e \ . \ ?r) \ (c \ d) \ (e \ b \ c \ d)) \\
\{a: e, \ y: (c \ d), \ z: (b \ c \ d), \ left: (?a \ . \ ?r)\}
\end{align*}

\begin{align*}
(app \ (b \ . \ ?r2) \ (c \ d) \ (b \ c \ d)) \\
\{r2: (), \ x: (c \ d)\}
\end{align*}

\begin{align*}
(app \ (e \ . \ ?r) \ (c \ d) \ (e \ b \ c \ d)) \\
\{a: e, \ y: (c \ d), \ z: (b \ c \ d), \ left: (?a \ . \ ?r)\}
\end{align*}

Variables are local to facts & queries

?left: (e \ . \ (b))

?r: (b \ . \ ()) \rightarrow (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

Variables are local to facts & queries

(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))

?left: (e . (b)) \rightarrow (e b)

?r: (b . ()) \rightarrow (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.

```python
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.

```python
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.

```python
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.

```python
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.

```python
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):
```
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.

```python
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.

```python
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.

```python
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
```

- 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
```

• 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.

```python
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
```

- 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.

```python
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
```

- 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)