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

logic> (fact (parent delano herbert))
A simple fact expression in the Logic language declares a relation to be true

Let's say I want to track 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))
```

![Diagram of genealogy]

```
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))
```

```
Delano

Abraham

Clinton

Barack

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))
logic> (fact (parent abraham clinton))
logic> (fact (parent fillmore abraham))
```
A simple fact expression in the Logic language declares a relation to be true.

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

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

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

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

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

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

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

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

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

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

```
logic> (fact (parent delano herbert))
logic> (fact (parent abraham barack))
logic> (fact (parent abraham clinton))
logic> (fact (parent fillmore abraham))
logic> (fact (parent fillmore delano))
logic> (fact (parent fillmore grover))
logic> (fact (parent eisenhower fillmore))
```
Relations are Not Procedure Calls
Relations are Not Procedure Calls

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

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

- *Scheme*: the expression `(abs -3)` calls `abs` on `-3`. It returns 3.
Relations are Not Procedure Calls

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

- **Scheme**: the expression `(abs -3)` calls *abs* on `-3`. It returns 3.
- **Logic**: `(abs -3 3)` asserts that *abs* of `-3` is 3.
Relations are Not Procedure Calls

In Logic, a relation is not a call expression.

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

To assert that $1 + 2 = 3$, we use a relation: `(add 1 2 3)`
Relations are Not Procedure Calls

In Logic, a relation is not a call expression.
• Scheme: the expression \( (\text{abs} \ -3) \) calls \text{abs} on \(-3\). It returns 3.
• Logic: \( (\text{abs} \ -3 \ 3) \) asserts that \text{abs} of \(-3\) is 3.

To assert that \(1 + 2 = 3\), we use a relation: \( (\text{add} \ 1 \ 2 \ 3) \)

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 \((\text{abs} \ -3)\) calls \text{abs} on \(-3\). It returns 3.
- **Logic**: \((\text{abs} \ -3 \ 3)\) asserts that \text{abs} of \(-3\) is 3.

To assert that \(1 + 2 = 3\), we use a relation: \((\text{add} \ 1 \ 2 \ 3)\)

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

\((\text{add} \ ? \ 2 \ 3)\)
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.

\[
\begin{align*}
  (add \ ? \ 2 \ 3) & \quad 1 \\
  (add \ 1 \ ? \ 3) & \quad 2
\end{align*}
\]
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 \((\text{abs} \ -3)\) calls \textit{abs} on \(-3\). It returns 3.
- **Logic**: \((\text{abs} \ -3\ 3)\) asserts that \textit{abs} of \(-3\) is 3.

To assert that \(1 + 2 = 3\), we use a relation: \((\text{add} \ 1 \ 2 \ 3)\)

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

\[
\begin{align*}
(\text{add} \ ? \ 2 \ 3) &= 1 \\
(\text{add} \ 1 \ ? \ 3) &= 2 \\
(\text{add} \ 1 \ 2 \ ?) &= 3
\end{align*}
\]
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 \((\text{abs} \ -3)\) calls \text{abs} on \(-3\). It returns 3.
- **Logic:** \((\text{abs} \ -3 \ 3)\) asserts that \text{abs} of \(-3\) is 3.

To assert that \(1 + 2 = 3\), we use a relation: \((\text{add} \ 1 \ 2 \ 3)\)

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

\[
\begin{align*}
(\text{add} \ ? \ 2 \ 3) & \quad 1 \\
(\text{add} \ 1 \ ? \ 3) & \quad 2 \\
(\text{add} \ 1 \ 2 \ ?) & \quad 3 \\
(\ ? \ 1 \ 2 \ 3) & \quad \text{add}
\end{align*}
\]
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
(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 `?`

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

Diagram:

```
  Eisenhower
    /   \
 Fillmore   \
    /     \
 Abraham  Delano  Grover
      /   /   \
 Barack Clinton Herbert
```
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
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
```

Each line is an assignment of variables to values.

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
```

Each line is an assignment of variables to values.

A variable can have any name.
Compound Facts and Queries
Compound Facts

Eisenhower
  ↓
  Fillmore
  ↓
  Abraham
  ↓
  Barack

  Delano
  ↓
  Clinton
  ↓
  Herbert

  Grover
Compound Facts

A fact can include multiple relations and variables as well.
Compound Facts

A fact can include multiple relations and variables as well.

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

A fact can include multiple relations and variables as well.

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

Means \text{<conclusion>} is true if all the \text{<hypothesis}_K\text{>} are true.
A fact can include multiple relations and variables as well.

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

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


Diagram:

- Eisenhower
- Fillmore
- Abraham
- Delano
- Grover
- Barack
- Clinton
- Herbert
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.

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

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

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

A fact can include multiple relations and variables as well.

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

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

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

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

\text{Success!}
**Compound Facts**

A fact can include multiple relations and variables as well.

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

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

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

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

Success!

![Diagram of family tree](image-url)
**Compound Facts**

A fact can include multiple relations and variables as well.

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

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

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

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

\text{Success!}

\[
\text{logic> (query (child eisenhower clinton))}
\]
**Compound Facts**

A fact can include multiple relations and variables as well.

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

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.

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

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.
```

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

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

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

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

\[
\text{logic}\geq (\text{query (child ?kid fillmore)})
\text{Success!}
\]
**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
Compound Queries

- Eisenhower
- Fillmore
  - Abraham
  - Delano
  - Grover
  - Barack
  - Clinton
  - Herbert
Compound Queries

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

An assignment must satisfy all relations in a query.

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

An assignment must satisfy all relations in a query.

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

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

An assignment must satisfy all relations in a query.

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

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

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

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

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

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

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

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

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

An assignment must satisfy all relations in a query.

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

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

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

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

**Success!**

grampa: fillmore  kid: abraham

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

```
Eisenhower
  └── Fillmore
      ├── Abraham
      └── Delano
          └── Grover
              └── Herbert
                  ├── Barack
                  └── Clinton
```
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
```

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

Compound Queries

An assignment must satisfy all relations in a query.

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

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
```
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.

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

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

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

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

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

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

Diagram: A graph showing a family tree with recursive facts illustrated.
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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

logic> (query (ancestor ?a herbert))
Searching to Satisfy Queries

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

```
logic> (query (ancestor ?a herbert))
Success!
```
Searching to Satisfy Queries

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

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

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

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

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

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

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

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

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

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

logic> (fact (parent delano herbert))
Searching to Satisfy Queries

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

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

```logic
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.

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

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

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

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

(p parent delano herbert) ; (1), a simple fact
(ancestor delano herbert) ; (2), from (1) and the 1st ancestor fact
(p parent fillmore delano) ; (3), a simple fact
(ancestor fillmore herbert) ; (4), from (2), (3), & the 2nd ancestor fact
```
Hierarchical Facts
Hierarchical Facts
Hierarchical Facts

Relations can contain relations in addition to symbols.
Hierarchical Facts

Relations can contain relations in addition to symbols.

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

Relations can contain relations in addition to symbols.

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

Relations can contain relations in addition to symbols.

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

Relations can contain relations in addition to symbols.

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

Relations can contain relations in addition to symbols.

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

Relations can contain relations in addition to symbols.

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

Relations can contain relations in addition to symbols.

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

Relations can contain relations in addition to symbols.

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

Relations can contain relations in addition to symbols.

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

Relations can contain relations in addition to symbols.

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

Relations can contain relations in addition to symbols.

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

Variables can refer to symbols or whole relations.

Diagram:

```
   E
   |
  F
 /|
A B C
 /|
 D G
```

Diagram:

```
   E
   |
  F
 /|
A B C
 /|
 D G
```
Hierarchical Facts

Relations can contain relations in addition to symbols.

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

Variables can refer to symbols or whole relations.

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

Relations can contain relations in addition to symbols.

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

Variables can refer to symbols or whole relations.

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

Relations can contain relations in addition to symbols.

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

Variables can refer to symbols or whole relations.

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

Relations can contain relations in addition to symbols.

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

Variables can refer to symbols or whole relations.

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

Relations can contain relations in addition to symbols.

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

Variables can refer to symbols or whole relations.

```
logic> (query (dog (name clinton) (fur ?type)))
Success!
  type: long
logic> (query (dog (name clinton) ?stats))
Success!
  stats: (fur long)
```
Combining Multiple Data Sources
Combining Multiple Data Sources
Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?
Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

logic> (query (dog (name ?x) (fur ?fur))
Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

```
logic> (query (dog (name ?x) (fur ?fur))
(ancestor ?y ?x))
```
Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

logic> (query (dog (name ?x) (fur ?fur))
  (ancestor ?y ?x)
  (dog (name ?y) (fur ?fur)))
Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

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

Success!
Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

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

Success!

x: barack fur: short y: eisenhower
Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

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

```
  x: barack    fur: short      y: eisenhower
  x: clinton   fur: long       y: abraham
```

Diagram:

```
    E
   /|
  F  A
  |  |
B  C  D  G
  |  |
    H
```
Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

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

Success!

x: barack  fur: short  y: eisenhower
x: clinton  fur: long   y: abraham
x: grover  fur: short  y: eisenhower
Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

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

Success!

x: barack    fur: short    y: eisenhower
x: clinton    fur: long     y: abraham
x: grover     fur: short    y: eisenhower
x: herbert    fur: curly    y: fillmore
Appending Lists

(Demo)
Lists in Logic
Lists in Logic

Expressions begin with *query* or *fact* followed by relations.
Lists in Logic

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

Expressions and their relations are Scheme lists.
Lists in Logic

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

Expressions and their relations are Scheme lists.

(fact (append-to-form () ?x ?x))
Lists in Logic

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

Expressions and their relations are Scheme lists.

\[(\text{fact (append-to-form () ?x ?x))}\]

*Simple fact: Conclusion*
Lists in Logic

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

Expressions and their relations are Scheme lists.

(fact (append-to-form () ?x ?x)) Simple fact: Conclusion

(fact (append-to-form (?a . ?r) ?y (?a . ?z))
  (append-to-form       ?r  ?y       ?z ))
Lists in Logic

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

Expressions and their relations are Scheme lists.

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

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

(\(\text{append-to-form ?r \?y \?z })\))
Lists in Logic

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

Expressions and their relations are Scheme lists.

(\text{fact} \ (\text{append-to-form} \ () \ ?x \ ?x)) < Simple fact: Conclusion
(\text{fact} \ (\text{append-to-form} \ (?a \ . \ ?r) \ ?y \ (?a \ . \ ?z)) \ (\text{append-to-form} \ ?r \ ?y \ ?z)) < Conclusion

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

Expressions and their relations are Scheme lists.

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

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

(query (append-to-form ?left (c d) (e b c d)))

Success!
left: (e b)
Lists in Logic

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

Expressions and their relations are Scheme lists.

\(\text{(fact (append-to-form () ?x ?x))}\)  \(<\text{Simple fact: Conclusion}\>

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

\(\text{(append-to-form ?r ?y ?z )}\)  \(<\text{Conclusion}\>

\(\text{(query (append-to-form ?left (c d) (e b c d)))}\)

**Success!**

**left: (e b)**
Lists in Logic

Expressions begin with \textit{query} or \textit{fact} followed by relations.

Expressions and their relations are Scheme lists.

\begin{align*}
\text{(fact (append-to-form () ?x ?x))} & \quad \text{Simple fact: Conclusion} \\
\text{(fact (append-to-form (?a . ?r) ?y (?a . ?z))} & \quad \text{Conclusion} \\
\quad \text{(append-to-form ?r ?y ?z)} & \quad \text{Hypothesis}
\end{align*}

\begin{align*}
\text{(query (append-to-form ?left (c d) (e b c d)))} \\
\text{Success!} \\
\text{left: (e b)}
\end{align*}

The interpreter lists all bindings of variables to values that it can find to satisfy the query.
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))  \text{Simple fact: Conclusion}
(fact (append-to-form (?a . ?r) ?y (?a . ?z))  \text{Conclusion}
    (append-to-form    ?r   ?y     ?z ))  \text{Hypothesis}
```

```
(query (append-to-form ?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)}
```

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

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

Expressions and their relations are Scheme lists.

\[
\text{(fact (append-to-form () ?x ?x))}
\]

\[
\]

\[
\text{(query (append-to-form ?left (c d) (e b c d)))}
\]

**Success!**

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

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

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

Expressions and their relations are Scheme lists.

\[
\text{(fact (append-to-form () ?x ?x))}
\]

\[
\]

\[
\text{(query (append-to-form ?left (c d) (e b c d)))}
\]

Success!

left: (e b)

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

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

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

Expressions and their relations are Scheme lists.

(\text{fact} \ (\text{append-to-form} \ () \ ?x \ ?x))< \text{Simple fact: Conclusion}

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

\text{Hypothesis}

(\text{query} \ (\text{append-to-form} \ ?\text{left} \ (c \ d) \ (e \ b \ c \ d)))

\text{Success!}

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

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

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

Expressions and their relations are Scheme lists.

\[(\text{fact \ (append-to-form (\() \ ?x \ ?x\))})\quad \text{Simple fact: Conclusion} \]

\[(\text{fact \ (append-to-form \ (?a \ . \ ?r) \ ?y \ (?a \ . \ ?z))})\quad \text{Conclusion} \]
\[(\text{append-to-form \ ?r \ ?y \ ?z})\quad \text{Hypothesis} \]

\[(\text{query \ (append-to-form \ ?left \ (c \ d) \ (e \ b \ c \ d))})\]

Success!

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

The interpreter lists all bindings of variables to values that it can find to satisfy the query.
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

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

(query (append-to-form ?left (c d) (e b c d)))

Success!

left: (e b)

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

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

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

Expressions and their relations are Scheme lists.

\[
\text{Simple fact: Conclusion}
\]

\[
\text{Conclusion}
\]

\[
\text{Hypothesis}
\]

**Success!**

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

The interpreter lists all bindings of variables to values that it can find to satisfy the query.
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

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

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

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

Expressions and their relations are Scheme lists.

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

Success!

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

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

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

Expressions and their relations are Scheme lists.

\[
\text{(fact (append-to-form () ?x ?x))}
\]

Simple fact: Conclusion

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

Conclusion

\[
\text{(append-to-form ?r ?y ?z )}
\]

Hypothesis

\[
\text{(query (append-to-form ?left (c d) (e b c d)))}
\]

Success!

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

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

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

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

Expressions and their relations are Scheme lists.

Expressions:
- `(fact (append-to-form () ?x ?x))`  
  - *Simple fact: Conclusion*
  - *Conclusion*
  - *Hypothesis*

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

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

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

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

Expressions and their relations are Scheme lists.

- (fact (append-to-form () ?x ?x))
- (fact (append-to-form (?a . ?r) ?y (?a . ?z))
  (append-to-form ?r ?y ?z))
- (query (append-to-form ?left (c d) (e b c d)))

Success!

left: (e b)

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

Expressions begin with \textit{query} or \textit{fact} followed by relations.

Expressions and their relations are Scheme lists.

\begin{itemize}
  \item \begin{Verbatim}
(fact (append-to-form () \(\textit{?x}\) \(\textit{?x}\))
\end{Verbatim}
\textbf{Simple fact: Conclusion}

\item \begin{Verbatim}
(fact (append-to-form (?a . ?r) \(\textit{?y}\) (?a . ?z))
(append-to-form \(\textit{?r}\) \(\textit{?y}\) \(\textit{?z}\))
\end{Verbatim}
\textbf{Conclusion}

\item \begin{Verbatim}
(fact (append-to-form (?a . ?r) \(\textit{?y}\) (?a . ?z))
(append-to-form \(\textit{?r}\) \(\textit{?y}\) \(\textit{?z}\))
\end{Verbatim}
\textbf{Hypothesis}
\end{itemize}

\begin{itemize}
  \item \begin{Verbatim}
(query (append-to-form ?left (c d) (e b c d)))
\end{Verbatim}
\textbf{Success!}

\item \begin{Verbatim}
left: (e b)
\end{Verbatim}
\textbf{What ?left can append with (c d) to create (e b c d)}
\end{itemize}

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

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

Expressions and their relations are Scheme lists.

\[
\text{fact (append-to-form () ?x ?x))} \quad \text{Simple fact: Conclusion} \\
\text{fact (append-to-form (?a . ?r) ?y (?a . ?z))} \quad \text{Conclusion} \\
\text{(append-to-form ?r ?y ?z)} \quad \text{Hypothesis}
\]

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

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

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

Expressions and their relations are Scheme lists.

(fact (append-to-form () ?x ?x))<Simple fact: Conclusion

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

(append-to-form ?r ?y ?z )<Hypothesis

(query (append-to-form ?left (c d) (e b c d)))

Success!

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

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

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

Expressions and their relations are Scheme lists.

(fact (append-to-form () ?x ?x))<Simple fact: Conclusion

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

(append-to-form ?r ?y ?z )<Hypothesis

(query (append-to-form ?left (c d) (e b c d)))

Success!

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

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

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

Expressions and their relations are Scheme lists.

```
(fact (append-to-form () ?x ?x))
(fact (append-to-form (?a . ?r) ?y (?a . ?z))
  (append-to-form ?r ?y ?z))
(query (append-to-form ?left (c d) (e b c d)))
```

Success!

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

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

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

Expressions and their relations are Scheme lists.

- \( (\text{fact} \ (\text{append-to-form} \ () \ ?x \ ?x)) \)
  - Simple fact: Conclusion
  - \( () \ (c \ d) \Rightarrow (c \ d) \)

- \( (\text{fact} \ (\text{append-to-form} \ (?a \ . \ ?r) \ ?y \ (?a \ . \ ?z)) \)
  - Conclusion
  - \( (b) \ (c \ d) \Rightarrow (b \ c \ d) \)
  - Hypothesis

- \( (\text{query} \ (\text{append-to-form} \ ?\text{left} \ (c \ d) \ (e \ b \ c \ d))) \)
  - Success!
  - \( \text{left}: (e \ b) \)

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

- \( (e \ . \ (b)) \ (c \ d) \Rightarrow (e \ . \ (b \ c \ d)) \)
  - \( ?a \ ?r \ ?y \ ?z \)
  - \( (?a \ . \ ?r) \ (c \ d) \Rightarrow (e \ . \ (b \ c \ d)) \)
  - \( (?a \ . \ ?r) \ (c \ d) \Rightarrow (e \ . \ (b \ c \ d)) \)

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

(Demo)
Which Hypotheses Complete append-3?
Which Hypotheses Complete append-3?

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

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

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

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

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

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

5:  None of the above
Define Base Fact of append-to-form So That No Lists Can Be Empty
Define Base Fact of append-to-form So That No Lists Can Be Empty

; (append-to-form () (1 2 3) (1 2 3))
; (append-to-form (1) (2 3) (1 2 3))
; (append-to-form (1 2) (3) (1 2 3))
; (append-to-form (1 2 3) () (1 2 3))
Define Base Fact of append-to-form So That No Lists Can Be Empty

; (append-to-form () (1 2 3) (1 2 3))
; (append-to-form (1) (2 3) (1 2 3))
; (append-to-form (1 2) (3) (1 2 3))
; (append-to-form (1 2 3) () (1 2 3))
Define Base Fact of `append-to-form` So That No Lists Can Be Empty

; (append-to-form () (1 2 3) (1 2 3))
; (append-to-form (1) (2 3) (1 2 3))
; (append-to-form (1 2) (3) (1 2 3))
; (append-to-form (1 2 3) () (1 2 3))
Define Base Fact of append-to-form So That No Lists Can Be Empty

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

Recursive fact:  (fact (append-to-form (?a . ?r) ?y (?a . ?z))
                (append-to-form ?r ?y ?z))
Define Base Fact of `append-to-form` So That No Lists Can Be Empty

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

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

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

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

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

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

Recursive fact: (fact (append-to-form (?a . ?r) ?y (?a . ?z))
                       (append-to-form ?r ?y ?z))
Permuting Lists
Anagrams in Logic

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

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

- The empty list for an empty list.
Anagrams in Logic

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

• The empty list for an empty list.
• The first element of the list inserted into an anagram of the rest of the list.
Anagrams in Logic

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

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

(fact (insert ?a ?r (?a . ?r)))
Anagrams in Logic

A permutation (i.e., anagram) of a list is:
• The empty list for an empty list.
• The first element of the list inserted into an anagram of the rest of the list.

\[(\text{fact (insert } \text{?a } \text{?r (}\text{?a . } \text{?r)}))\]
Anagrams in Logic

A permutation (i.e., anagram) of a list is:
• The empty list for an empty list.
• The first element of the list inserted into an anagram of the rest of the list.

(fact (insert ?a ?r (?a . ?r)))
Anagrams in Logic

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

• The empty list for an empty list.

• The first element of the list inserted into an anagram of the rest of the list.

\[(\text{fact } (\text{insert } ?a \ ?r \ (?a \ . \ ?r)))\]
Anagrams in Logic

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

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

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

(fact (insert ?a (?b . ?r) (?b . ?s)))
```
Anagrams in Logic

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

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

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

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

Element → List → List with ?a in front

List with ?a somewhere
Anagrams in Logic

A permutation (i.e., anagram) of a list is:
• The empty list for an empty list.
• The first element of the list inserted into an anagram of the rest of the list.

(fact (insert ?a ?r (?a . ?r)))  \( \text{List with \ ?a\ in\ front} \)

(fact (insert ?a (?b . ?r) (?b . ?s)))  \( \text{Bigger list with \ ?a\ somewhere} \)

(fact (insert ?a (?b . ?r) (?b . ?s)))  \( \text{List with \ ?a\ somewhere} \)
Anagrams in Logic

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

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

\[(\text{fact } (\text{insert } ?a \ ?r \ (\text{?a . ?r})))\]  
\[(\text{fact } (\text{insert } ?a \ (?b \ ?r) \ (?b \ ?s)))\]
\[(\text{fact } (\text{anagram } () ()())\)]
Anagrams in Logic

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

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

(fact (insert ?a ?r (?a . ?r)))  Bigger list with ?a somewhere

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

(fact (anagram () ()()))  List with ?a somewhere
Anagrams in Logic

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

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

\[
\text{(fact (insert ?a ?r (?a . ?r)))}
\]

\[
\text{(fact (insert ?a (?b . ?r) (?b . ?s))}
\]

\[
\text{(fact (anagram () ()))}
\]

\[
\text{(fact (anagram (?a . ?r) ?b)}
\]
Anagrams in Logic

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

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

\[
\begin{align*}
\text{(fact (insert ?a ?r (?a . ?r)))} & \quad \text{Bigger list with ?a somewhere} \\
\text{(fact (insert ?a (?b . ?r) (?b . ?s)))} & \\
\text{(fact (anagram () ()))} & \quad \text{List with ?a somewhere} \\
\text{(fact (anagram (?a . ?r) ?b)} & (\text{insert ?a ?s ?b})
\end{align*}
\]
Anagrams in Logic

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

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

\[
\text{fact (insert ?a ?r (?a . ?r))}
\]

\[
\text{fact (insert ?a (?b . ?r) (?b . ?s))}
\]

\[
\text{fact (anagram () ())}
\]

\[
\text{fact (anagram (?a . ?r) ?b)}
\quad \text{(insert ?a ?s ?b)}
\]
Anagrams in Logic

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

• The empty list for an empty list.

• The first element of the list inserted into an anagram of the rest of the list.

\[
\text{(fact (insert \ ?a \ ?r \ (?a . \ ?r)))} \\
\text{(fact (insert \ ?a \ (?b . \ ?r) \ (?b . \ ?s)))}
\]

\[
\text{(fact (anagram \ () \ ()))} \\
\text{(fact (anagram \ (?a . \ ?r) \ ?b)} \\
\text{(insert \ ?a \ ?s \ ?b)}
\]
Anagrams in Logic

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

• The empty list for an empty list.

• The first element of the list inserted into an anagram of the rest of the list.

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

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

(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
  (insert ?a ?s ?b))
Anagrams in Logic

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

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

\[
\text{fact (insert ?a ?r (?a . ?r))}
\]

\[
\text{fact (insert ?a (?b . ?r) (?b . ?s))}
\]

\[
\text{(fact (anagram () ()))}
\]

\[
\text{(fact (anagram (?a . ?r) ?b) (insert ?a ?s ?b))}
\]
Anagrams in Logic

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

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

\[
\text{(fact (insert } ?a \ ?r \ (?a . \ ?r)) \quad \text{Bigger list with } ?a \text{ somewhere}
\]

\[
\text{(fact (insert } ?a \ (?b . \ ?r) \ (?b . \ ?s))}
\]

\[
\text{(fact (anagram } () \ ()))
\]

\[
\text{(fact (anagram } (?a . \ ?r) \ ?b)
\text{ (insert } \ ?a \ ?s \ ?b)
\]
Anagrams in Logic

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

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

\[
\text{(fact (insert } ?a \ ?r \ (?a \ . \ ?r) ) (\text{List with } ?a \ \text{in front})}
\]

\[
\text{(fact (insert } ?a \ (?b \ . \ ?r) \ (?b \ . \ ?s)) (\text{Bigger list with } ?a \ \text{somewhere})}
\]

\[
\text{(fact (anagram } () ()()) (\text{List with } ?a \ \text{somewhere})}
\]

\[
\text{(fact (anagram } (?a \ . \ ?r) ?b) (\text{insert } ?a \ ?s \ ?b)}
\]
Anagrams in Logic

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

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

\[
\begin{align*}
\text{(fact (insert ?a ?r (?a . ?r)))} & \quad \text{Bigger list with ?a somewhere} \\
\text{(fact (insert ?a (?b . ?r) (?b . ?s))} & \quad \text{List with ?a somewhere} \\
\end{align*}
\]

\[
\begin{align*}
\text{(fact (anagram () ()}) & \quad \text{List with ?a somewhere} \\
\text{(fact (anagram (?a . ?r) ?b) (insert ?a ?s ?b)\}
\end{align*}
\]
Anagrams in Logic

A permutation (i.e., anagram) of a list is:
• The empty list for an empty list.
• The first element of the list inserted into an anagram of the rest of the list.

\[
\text{fact}\ (\text{insert\ } ?a\ ?r\ (?, a, r))
\]

\[
\text{fact}\ (\text{insert\ } ?a\ (?, b, r)\ (?, b, s))
\]

\[
\text{fact}\ (\text{anagram\ } ()\ ())
\]

\[
\text{fact}\ (\text{anagram\ } (?, a, r)\ ?b)\ \\
\quad \text{(insert\ } ?a\ ?s\ ?b)\]

\[
\text{List} \quad \text{List with } ?a\ \text{in front} \quad \text{Bigger list with } ?a\ \text{somewhere}
\]

\[
\text{List with } ?a\ \text{somewhere}
\]

\[
\text{Element} \quad \text{List} \quad \text{List with } ?a\ \text{in front}
\]

\[
\text{List with } ?a\ \text{somewhere}
\]
Anagrams in Logic

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

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

\[(\text{fact } (\text{insert } ?a \ ?r \ ((?a \ . \ ?r))))\]  
\[(\text{fact } (\text{insert } ?a \ (?b \ . \ ?r) \ ((?b \ . \ ?s))))\]

\[(\text{fact } (\text{anagram } () \ ()))\]  
\[(\text{fact } (\text{anagram } (?a \ . \ ?r) \ ?b))\]

\[(\text{insert } ?a \ ?s \ ?b)\]
Anagrams in Logic

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

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

\[(\text{fact (insert } ?a \ ?r \ (?a \ . \ ?r)))\]
\[(\text{fact (insert } ?a \ (?b \ . \ ?r) \ (?b \ . \ ?s)))\]
\[(\text{fact (anagram } () \ ()())\]
\[(\text{fact (anagram } (?a \ . \ ?r) ?b) \ (\text{insert } ?a \ ?s ?b))\]
Anagrams in Logic

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

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

\[
\begin{align*}
\text{(fact (insert ?a ?r (?a . ?r)))} & \quad \text{List with ?a in front} \\
\text{(fact (insert ?a (?b . ?r) (?b . ?s))} & \quad \text{Bigger list with ?a somewhere} \\
\text{(fact (anagram () ()}) & \quad \text{List with ?a somewhere} \\
\text{(fact (anagram (?a . ?r) ?b) (insert ?a ?s ?b))} & \quad \text{(Demo)}
\end{align*}
\]
Unification
Pattern Matching
Pattern Matching

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

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

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

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

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

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

The basic operation of the Logic interpreter is to attempt to *unify* two relations.
Unification is finding an assignment to variables that makes two relations the same.

\[
\begin{align*}
\text{pattern 1: } & ( (a \ b) \ c \ (a \ b) ) \\
\text{pattern 2: } & ( \ ?x \ c \ \ ?x \ )
\end{align*}
\]
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.

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

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

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

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

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

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

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

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

\{ \}
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.

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

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

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

{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.

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

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

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

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

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

x: (a b)

{ x: (a b) }

Success!
Unification

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

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

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

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

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.

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

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

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.

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

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

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

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

\[
\begin{align*}
\text{Success!}
\end{align*}
\]

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

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

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.

Symbols/relations without variables only unify if they are the same.
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))
Two relations that contain variables can be unified as well.

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

Two relations that contain variables can be unified as well.

\[
\begin{align*}
& (\text{(a ?y c)} \quad \text{(a b ?z)}) \\
\rightarrow & \quad \text{True, } \{ x \rightarrow ?x \}\n\end{align*}
\]
Unifying Variables

Two relations that contain variables can be unified as well.

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

Two relations that contain variables can be unified as well.

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

Unifying Variables

Two relations that contain variables can be unified as well.

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

Two relations that contain variables can be unified as well.

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

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

Two relations that contain variables can be unified as well.

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

Two relations that contain variables can be unified as well.

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

Two relations that contain variables can be unified as well.

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

Two relations that contain variables can be unified as well.

```
(a ?y c)

(a b ?z)
```

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

Two relations that contain variables can be unified as well.

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

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

Two relations that contain variables can be unified as well.

\[
(\begin{array}{c}
?x \\
(a \ ?y \ c)
\end{array}) \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ quad
Unifying Variables

Two relations that contain variables can be unified as well.

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

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.

\text{lookup('?x')}
### Unifying Variables

Two relations that contain variables can be unified as well.

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

Substituting values for variables may require multiple steps.

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

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

Two relations that contain variables can be unified as well.

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

Substituting values for variables may require multiple steps.

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

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

Two relations that contain variables can be unified as well.

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

Lookup

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

Substituting values for variables may require multiple steps.

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

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

Two relations that contain variables can be unified as well.

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

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.

\[\text{lookup('?x')} \Rightarrow (a \ ?y \ c) \quad \text{lookup('?y')} \Rightarrow b \quad \text{ground('?x')}\]
Two relations that contain variables can be unified as well.

\[
\begin{align*}
& \text{True, } \{ x: (a \ ?y \ c), \\
& \hspace{1cm} y: b, \\
& \hspace{1cm} z: c \} \\
\end{align*}
\]

Substituting values for variables may require multiple steps.

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

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

1. Look up variables in the current environment
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.
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.
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.
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

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

Recursively unify the first and rest of any lists.

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

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)
```

1. Look up variables in the current environment
   ( (a b) c (a b) )
   ( ?x c ?x )

2. Establish new bindings to unify elements.
   `env: { x: (a b) }`

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.

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

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

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

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

1. Look up variables in the current environment

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

Recursively unify the first and rest of any lists.

env: { x: (a b) }

Lookup

((a b) c (a b))

((?x c ?x))

((a b) (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)
```

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: { x: (a b) })

((a b) c (a b))

((?x) c (?x))
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.
The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

(fact (app () ?x ?x))
(fact (app (?a . ?r) ?y (?a . ?z))
  (app       ?r  ?y       ?z ))
(query (app ?left (c d) (e b c d)))
Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

(fact (app () ?x ?x))
(fact (app (?a . ?r) ?y (?a . ?z))
  (app ?r ?y ?z ))
(query (app left (c d) (e b c d)))

(app left (c d) (e b c d))
Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

(fact (app () ?x ?x))
(fact (app (?a . ?r) ?y (?a . ?z))
  (app ?r ?y ?z))
(query (app ?left (c d) (e b c d)))

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

(app (?a . ?r) ?y (?a . ?z))
Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

(fact (app () ?x ?x))
(fact (app (?a . ?r) ?y (?a . ?z))
   (app ?r ?y ?z ))
(query (app ?left (c d) (e b c d)))

(app ?left (c d) (e b c d))
   {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
(app (?a . ?r) ?y (?a . ?z))
Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

(fact (app () ?x ?x))
(fact (app (?a . ?r) ?y (?a . ?z))
  (app ?r ?y ?z))
(query (app ?left (c d) (e b c d)))

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

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

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

(app (e . ?r) (c d) (e b c d))
The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

<table>
<thead>
<tr>
<th>(fact (app () ?x ?x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(fact (app (?a . ?r) ?y (?a . ?z))</td>
</tr>
<tr>
<td>(app ?r ?y ?z ))</td>
</tr>
<tr>
<td>(query (app ?left (c d) (e b c d)))</td>
</tr>
</tbody>
</table>

```
(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))
```
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))
```
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
The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

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

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

\[
\text{conclusion <- hypothesis}
\]
\[
\text{(app ?r (c d) (b c d))}
\]
\[
\begin{aligned}
\{a2: &\ b, y2: (c d), z2: (c d), r: (?a2 . ?r2)\} \\
\text{(app (?a2 . ?r2) ?y2 (?a2 . ?z2))}
\end{aligned}
\]

Variables are local to facts & queries
Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

(fact (app () ?x ?x))

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

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

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

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

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

conclusion <- hypothesis

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

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

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

Variables are local to facts & queries

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

(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.

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

\[
\begin{align*}
\text{(app ?left (c d) (e b c d))} & \quad \{a: e, y: (c d), z: (b c d), left: (?a . ?r)\} \\
\text{(app (?a . ?r) ?y (?a . ?z))} & \quad \text{conclusion <- hypothesis} \\
\text{(app ?r (c d) (b c d))} & \quad \{a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)\} \\
\text{(app (?a2 . ?r2) ?y2 (?a2 . ?z2))} & \quad \text{conclusion <- hypothesis} \\
\text{(app ?r2 (c d) (c d))} & \quad \text{Variables are local to facts & queries}
\end{align*}
\]
Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

(fact (app () ?x ?x))
(fact (app (?a . ?r) ?y (?a . ?z))
  (app ?r ?y ?z))
(query (app ?left (c d) (e b c d)))

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

  conclusion <- hypothesis

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

  conclusion <- hypothesis

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

Variables are local to facts & queries

(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.

(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))
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)))

\[
\begin{align*}
\text{(app ?left (c d) (e b c d))} & \quad \{a: e, y: (c d), z: (b c d), \text{left: (}???\}) \\
\text{(app (?a . ?r) ?y (?a . ?z))} & \quad \text{\textbf{conclusion} <- \textbf{hypothesis}} \\
\text{(app ?r (c d) (b c d))} & \quad \{a2: b, y2: (c d), z2: (c d), r: (}???\} \\
\text{(app (?a2 . ?r2) ?y2 (?a2 . ?z2))} & \quad \text{\textbf{conclusion} <- \textbf{hypothesis}} \\
\text{(app ?r2 (c d) (c d))} & \quad \{r2: (), x: (c d)} \\
\text{(app () ?x ?x)} & \quad \text{(app () (c d) (c d))} \\
\end{align*}
\]
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.

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

?left:

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

?left: (e .
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 .)
?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

?left: (e .
?r:
Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

(fact (app () ?x ?x))
(fact (app (?a . ?r) ?y (?a . ?z))
  (app   ?r ?y   ?z ))
(query (app ?left (c d) (e b c d)))

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

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

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

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

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

Variables are local to facts & queries

?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

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

(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)
```

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

- Variables are local to facts & queries
- ?left: (e .)
- ?r: (b .)
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))
```
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 () ?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 (c d) (c d))

?left: (e .

?r: (b . ()) → (b)

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.

\[
\begin{align*}
\text{query} & \leq \text{hypothesis} \\
\text{conclusion} & \leftarrow \text{hypothesis}
\end{align*}
\]
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)

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

?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.

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.

```python
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):
```

Environment now contains new unifying bindings
Depth-First Search

The space of facts is searched exhaustively, starting from the query and following a depth-first exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

def search(clauses, env):
    for fact in facts:
        env_head = an environment extending env
        if unify(conclusion of fact, first clause, env_head):
            for env_rule in search(hypotheses of fact, env_head):
                for result in search(rest of clauses, env_rule):

Environment now contains new unifying bindings
**Depth-First Search**

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order.

Depth-first search: Each proof approach is explored exhaustively before the next.

```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.

```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.
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)