Announcements

☐ HW12 due tonight

☐ HW13 out

☐ Scheme project, contest due Monday
Expressions begin with *query* or *fact* followed by relations.
Expressions begin with *query* or *fact* followed by relations

Expressions and their relations are Scheme lists
Logic Language Review

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

Expressions and their relations are Scheme lists

```scheme
logic> (fact (parent eisenhower fillmore))
logic> (fact (parent fillmore abraham))
logic> (fact (parent abraham clinton))
```
Expressions begin with *query* or *fact* followed by relations

Expressions and their relations are Scheme lists

```
logic> (fact (parent eisenhower fillmore))
logic> (fact (parent fillmore abraham))
logic> (fact (parent abraham clinton))
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
```
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists:

```
logic> (fact (parent eisenhower fillmore))
logic> (fact (parent fillmore abraham))
logic> (fact (parent abraham clinton))
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (query (ancestor ?who abraham))
```
Expressions begin with *query* or *fact* followed by relations

Expressions and their relations are Scheme lists

```scheme
logic> (fact (parent eisenhower fillmore))
logic> (fact (parent fillmore abraham))
logic> (fact (parent abraham clinton))
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (query (ancestor ?who abraham))
Success!
who: fillmore
who: eisenhower
```
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists:

```
logic> (fact (parent eisenhower fillmore))
logic> (fact (parent fillmore abraham))
logic> (fact (parent abraham clinton))
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (query (ancestor ?who abraham))
Success!
who: fillmore
who: eisenhower
```

If a fact has more than one relation, the first is the *conclusion*, and it is satisfied if the remaining relations, the *hypotheses*, are satisfied.
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists:

```scheme
(logic> (fact (parent eisenhower fillmore)))
(logic> (fact (parent fillmore abraham)))
(logic> (fact (parent abraham clinton)))
(logic> (fact (ancestor ?a ?y) (parent ?a ?y)))
(logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y)))
(logic> (query (ancestor ?who abraham)))
```

**Success!**

who: fillmore
who: eisenhower

If a fact has more than one relation, the first is the *conclusion*, and it is satisfied if the remaining relations, the *hypotheses*, are satisfied.

If a query has more than one relation, all must be satisfied.
Logic Language Review

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

Expressions and their relations are Scheme lists

```scheme
logic> (fact (parent eisenhower fillmore))
logic> (fact (parent fillmore abraham))
logic> (fact (parent abraham clinton))
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (query (ancestor ?who abraham))
Success!
who: fillmore
who: eisenhower
```

If a fact has more than one relation, the first is the *conclusion*, and it is satisfied if the remaining relations, the *hypotheses*, are satisfied

If a query has more than one relation, all must be satisfied

The interpreter lists all bindings that it can find to satisfy the query
Hierarchical Facts
Hierarchical Facts

Relations can contain relations in addition to atoms
Hierarchical Facts

Relations can contain relations in addition to atoms

\[
\text{logic} \ (\text{fact} \ (\text{dog} \ (\text{name} \ abraham) \ (\text{color} \ white)))
\]
Hierarchical Facts

Relations can contain relations in addition to atoms

logic> (fact (dog (name abraham) (color white)))
logic> (fact (dog (name barack) (color tan)))
logic> (fact (dog (name clinton) (color white)))
logic> (fact (dog (name delano) (color white)))
logic> (fact (dog (name eisenhower) (color tan)))
logic> (fact (dog (name fillmore) (color brown)))
logic> (fact (dog (name grover) (color tan)))
logic> (fact (dog (name herbert) (color brown)))
Hierarchical Facts

Relations can contain relations in addition to atoms

logic> (fact (dog (name abraham) (color white)))
logic> (fact (dog (name barack) (color tan)))
logic> (fact (dog (name clinton) (color white)))
logic> (fact (dog (name delano) (color white)))
logic> (fact (dog (name eisenhower) (color tan)))
logic> (fact (dog (name fillmore) (color brown)))
logic> (fact (dog (name grover) (color tan)))
logic> (fact (dog (name herbert) (color brown)))
Hierarchical Facts

Relations can contain relations in addition to atoms

logic> (fact (dog (name abraham) (color white)))
logic> (fact (dog (name barack) (color tan)))
logic> (fact (dog (name clinton) (color white)))
logic> (fact (dog (name delano) (color white)))
logic> (fact (dog (name eisenhower) (color tan)))
logic> (fact (dog (name fillmore) (color brown)))
logic> (fact (dog (name grover) (color tan)))
logic> (fact (dog (name herbert) (color brown)))

Variables can refer to atoms or relations
Hierarchical Facts

Relations can contain relations in addition to atoms

```
logic> (fact (dog (name abraham) (color white)))
logic> (fact (dog (name barack) (color tan)))
logic> (fact (dog (name clinton) (color white)))
logic> (fact (dog (name delano) (color white)))
logic> (fact (dog (name eisenhower) (color tan)))
logic> (fact (dog (name fillmore) (color brown)))
logic> (fact (dog (name grover) (color tan)))
logic> (fact (dog (name herbert) (color brown)))
```

Variables can refer to atoms or relations

```
logic> (query (dog (name clinton) (color ?color)))
Success!
color: white
```
Hierarchical Facts

Relations can contain relations in addition to atoms

logic> (fact (dog (name abraham) (color white)))
logic> (fact (dog (name barack) (color tan)))
logic> (fact (dog (name clinton) (color white)))
logic> (fact (dog (name delano) (color white)))
logic> (fact (dog (name eisenhower) (color tan)))
logic> (fact (dog (name fillmore) (color brown)))
logic> (fact (dog (name grover) (color tan)))
logic> (fact (dog (name herbert) (color brown)))

Variables can refer to atoms or relations

logic> (query (dog (name clinton) (color ?color)))
Success!
color: white

logic> (query (dog (name clinton) ?info))
Success!
info: (color white)
Hierarchical Facts

Relations can contain relations in addition to atoms

```
logic> (fact (dog (name abraham) (color white)))
logic> (fact (dog (name barack) (color tan)))
logic> (fact (dog (name clinton) (color white)))
logic> (fact (dog (name delano) (color white)))
logic> (fact (dog (name eisenhower) (color tan)))
logic> (fact (dog (name fillmore) (color brown)))
logic> (fact (dog (name grover) (color tan)))
logic> (fact (dog (name herbert) (color brown)))
```

Variables can refer to atoms or relations

```
logic> (query (dog (name clinton) (color ?color)))
Success!
color: white

logic> (query (dog (name clinton) ?info))
Success!
info: (color white)
```
Example: Combining Multiple Data Sources

Which dogs have an ancestor of the same color?
Example: Combining Multiple Data Sources

Which dogs have an ancestor of the same color?

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

Which dogs have an ancestor of the same color?

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

```
  E
  /\  \\
 /    \
F     \
|     |
|     |
A -- D -- G
|     |
B -- C -- H
```
Example: Combining Multiple Data Sources

Which dogs have an ancestor of the same color?

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

Which dogs have an ancestor of the same color?

```
logic> (query (dog (name ?name) (color ?color))
  (ancestor ?ancestor ?name)
  (dog (name ?ancestor) (color ?color)))
Success!
name: barack    color: tan    ancestor: eisenhower
name: clinton   color: white  ancestor: abraham
name: grover    color: tan    ancestor: eisenhower
name: herbert   color: brown  ancestor: fillmore
```
Example: Appending Lists
Example: Appending Lists

Two lists append to form a third list if:
Example: Appending Lists

Two lists append to form a third list if:

• The first list is empty and the second and third are the same
Example: Appending Lists

Two lists append to form a third list if:

• The first list is empty and the second and third are the same

  () (a b c) (a b c)
Example: Appending Lists

Two lists append to form a third list if:

• The first list is empty and the second and third are the same

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

logic> (fact (append-to-form () ?x ?x))
Example: Appending Lists

Two lists append to form a third list if:

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

- Both of the following hold:

```
logic> (fact (append-to-form () ?x ?x))
```
Example: Appending Lists

Two lists append to form a third list if:

• The first list is empty and the second and third are the same

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

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

logic> (fact (append-to-form () ?x ?x))
Example: Appending Lists

Two lists append to form a third list if:

• The first list is empty and the second and third are the same

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

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

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

logic> (fact (append-to-form () ?x ?x))
Example: Appending Lists

Two lists append to form a third list if:

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

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

logic> (fact (append-to-form () ?x ?x))
Example: Appending Lists

Two lists append to form a third list if:

• The first list is empty and the second and third are the same

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

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

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

logic> (fact (append-to-form () ?x ?x))
Example: Appending Lists

Two lists append to form a third list if:

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

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

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

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

(fact (append-to-form (?a . ?r) ?y (?a . ?z)))
```
Example: Appending Lists

Two lists append to form a third list if:

• The first list is empty and the second and third are the same
  \[(\text{true}) (a \ b \ c) (a \ b \ c)\]

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

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

\[
\text{logic}\> (\text{fact} \ (\text{append-to-form} \ (?a . \ ?r) \ ?y \ (?a . \ ?z))
\]
Example:Appending Lists

Two lists append to form a third list if:

• The first list is empty and the second and third are the same

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

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

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

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

Two lists append to form a third list if:

• The first list is empty and the second and third are the same

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

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

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

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

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

Two lists append to form a third list if:

• The first list is empty and the second and third are the same

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

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

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

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

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

Two lists append to form a third list if:

• The first list is empty and the second and third are the same

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

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

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

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

logic> (fact (append-to-form (?a . ?r) ?y (?a . ?z))
  (append-to-form ?r ?y ?z))
Logic Example: Anagrams
Logic Example: Anagrams

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

A permutation (i.e., anagram) of a list is:
• The empty list for an empty list
Logic Example: Anagrams

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
Logic Example: Anagrams

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

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

a r t
Logic Example: Anagrams

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
Logic Example: Anagrams

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
Logic Example: Anagrams

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

Arthur
Logic Example: Anagrams

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

\[
\begin{array}{c}
\text{a} \\
\text{r} \\
\text{t}
\end{array}
\]

\[
\begin{array}{c}
\text{r} \\
\text{t}
\end{array}
\]

\[
\begin{array}{c}
\text{ar} \\
\text{t}
\end{array}
\]

\[
\begin{array}{c}
\text{rat}
\end{array}
\]
Logic Example: Anagrams

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
Logic Example: Anagrams

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

\[
\begin{align*}
\text{ar} & \rightarrow \text{art}\\
\text{art} & \rightarrow \text{rat}\\
\text{rat} & \rightarrow \text{rat}\\
\text{rat} & \rightarrow \text{art}\\
\text{art} & \rightarrow \text{ar}
\end{align*}
\]
Logic Example: Anagrams

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
Logic Example: Anagrams

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
Logic Example: Anagrams

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
Logic Example: Anagrams

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))))
Logic Example: Anagrams

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

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

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))))
Logic Example: Anagrams

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))))
Logic Example: Anagrams

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

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

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

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

\[
(fact \ (insert \ ?a \ ?r \ ((?a \ . \ ?r))))
\]

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

\[
(fact \ (anagram \ () \ ()))
\]
Logic Example: Anagrams

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))
  (insert ?a ?r ?s))
(fact (anagram () ()))
(fact (anagram (?a . ?r) ?b)}
Logic Example: Anagrams

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

(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
  (insert ?a ?s ?b)
  (anagram ?r ?s))
Pattern Matching
Pattern Matching

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

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

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

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

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

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

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

\[
( (a \ b) \ c \ (a \ b) ) \quad ( \ ?x \ c \ ?x )
\]
Pattern Matching

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

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

\[(a \ b) c \quad (a \ b)\]
\[(\ ?x \quad c \quad ?x)\]

True, \(\{x: (a \ b)\}\)
Pattern Matching

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

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

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

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

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

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

True, \( \{x: (a \ b)\} \)
Pattern Matching

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

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

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

\[
\begin{align*}
& ( (a \ b) \ c \ (a \ b) ) \\
& ( (a \ ?y) \ ?z \ (a \ b) ) \\
& \text{True, } \{x: (a \ b)\} \\
& \text{True, } \{y: b, z: c\}
\end{align*}
\]
The basic operation of the Logic interpreter is to attempt to unify two relations

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

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

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

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

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

1. Look up variables in the current environment
Unification

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

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

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

\{ \}
Unification unifies each pair of corresponding elements in two relations, accumulating an assignment

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

$$
\begin{array}{c}
(a \ b) \ c \ (a \ b) \\
(?x \ c) \ (?x)
\end{array}
$$

$$
\{\ }
$$
Unification unifies each pair of corresponding elements in two relations, accumulating an assignment

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

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

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

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

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

Unification 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

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

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

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

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

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

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

\[
\{ \ x : (a \ b) \ \}
\]
Unification unifies each pair of corresponding elements in two relations, accumulating an assignment.

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

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

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

*Success!*
Unification 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

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

Success!
Unification

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

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

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

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

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

Success!
Unification

Unification 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

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

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

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

Success!
Unification 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) \equiv (a \ b) \\
&(\ ?x \ c \ ?x) \\
\end{align*}
```

```
\begin{align*}
&(a \ b \ c) \equiv (a \ b) \\
&(\ ?x \ ?x \ ?x) \\
\end{align*}
```

\textbf{Success!}

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

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

Unification 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

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

Unification 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

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

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

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

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

Success!
Unification

Unification 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!  
Failure.
Unification with Two Variables
Unification with Two Variables

Two relations that contain variables can be unified as well.
Unification with Two Variables

Two relations that contain variables can be unified as well

( ?x ?x )

( (a ?y c) (a b ?z) )
Unification with Two Variables

Two relations that contain variables can be unified as well

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

\[( (a \quad ?y \quad c) \quad (a \quad b \quad ?z) ) \]

\[\]
Unification with Two Variables

Two relations that contain variables can be unified as well

\[
\begin{align*}
&\quad \text{(}\ ?x \quad ?x \quad \text{)} \\
&\equiv \quad \text{(}\ (a \ ?y \ c) \quad (a \ b \ ?z) \quad ) \\
&\quad \text{True, } \{ \\
\end{align*}
\]
Unification with Two Variables

Two relations that contain variables can be unified as well

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

True, \( \{ x: (a \text{ ?y } c) \} \)
Unification with Two Variables

Two relations that contain variables can be unified as well

\[(\mathbf{?x}) \quad (\mathbf{a \ ?y \ c})\]
\[(\mathbf{a \ b \ ?z})\]

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

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

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

\[
\begin{align*}
(a \ ?y \ c) \\
(a \ b \ ?z)
\end{align*}
\]
Two relations that contain variables can be unified as well:

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

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

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

\[
\begin{align*}
    & (a ?y c) \\
    & (a b ?z)
\end{align*}
\]
Two relations that contain variables can be unified as well.

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

Lookup

\[
\begin{align*}
(\ a \ ?y \ c) \\
(\ a \ b \ ?z) \\
(\ )
\end{align*}
\]
Unification with Two Variables

Two relations that contain variables can be unified as well

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

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

True, \( \{ x: (a \ ?y \ c), y: b, \} \)
Unification with Two Variables

Two relations that contain variables can be unified as well:

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

True, \{x: (a \ ?y \ c), \ y: b,\}
Unification with Two Variables

Two relations that contain variables can be unified as well:

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

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

Two relations that contain variables can be unified as well:

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

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

Two relations that contain variables can be unified as well

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

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

Substituting values for variables may require multiple steps
Unification with Two Variables

Two relations that contain variables can be unified as well:

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

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

Substituting values for variables may require multiple steps:

\text{lookup('?x')}
Unification with Two Variables

Two relations that contain variables can be unified as well:

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

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

Substituting values for variables may require multiple steps:

\[\text{lookup('?x')} \Rightarrow (a \ ?y \ c)\]
Unification with Two Variables

Two relations that contain variables can be unified as well

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

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

Substituting values for variables may require multiple steps

\[
\text{lookup('?x')} \quad \rightarrow \quad (a \ ?y \ c) \quad \text{lookup('?y')}
\]
Unification with Two Variables

Two relations that contain variables can be unified as well

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

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

Substituting values for variables may require multiple steps

\[
\text{lookup('?x')} \Rightarrow (a ?y c) \quad \text{lookup('?y')} \Rightarrow b
\]
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)
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first, f.first, env) and \
        unify(e.second, f.second, env)
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first, f.first, env) and \
            unify(e.second, f.second, env)
Implementing Unification

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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)

    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first, f.first, env) and \
            unify(e.second, f.second, env)
Searching for Proofs

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

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

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

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

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

(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.

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

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

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

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

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

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

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

(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

(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)))

(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

(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

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

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

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

\text{conclusion} \leftarrow \text{hypothesis}

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

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

Variables are local to facts and 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

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

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

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

\text{conclusion } \leftarrow \text{ hypothesis}

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

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

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

\text{Variables are local to facts and 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.

(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 and 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

(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))

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

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

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

Variables are local to facts and 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

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

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

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

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

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

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

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

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

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

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

Variables are local to facts and 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

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

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

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

conclusion <- hypothesis

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

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

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

conclusion <- hypothesis

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

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

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

Variables are local to facts and 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.

(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 and queries.

(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.

(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)}

(left: (fact (app () ?x ?x))

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

(app ?r ?y ?z))

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

Variables are local to facts and 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

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

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

\[
\text{conclusion } \leftarrow \text{ hypothesis}
\]

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

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

\[
\text{conclusion } \leftarrow \text{ hypothesis}
\]

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

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

Variables are local to facts and 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

\[
(app \ ?\text{left} \ (c \ d) \ (e \ b \ c \ d))\\
\text{\{a: e, y: (c d), z: (b c d), left: (?a . ?r)\}}\\
(app \ (?a . ?r) \ ?y \ (?a . ?z))\\
\text{conclusion \leftarrow hypothesis}\\
(app \ ?r \ (c \ d) \ (b \ c \ d)))\\
\text{\{a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)\}}\\
\text{\{a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)\}}\\
\text{conclusion \leftarrow hypothesis}\\
\text{\{r2: (), x: (c d)\}}\\
\text{\{r2: (), x: (c d)\}}\\
\text{\{r2: (), x: (c d)\}}\\
\text{left:}\\
\text{\{r2: (), x: (c d)\}}\\
\text{Variables are local to facts and 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

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

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

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

Conclusion <- hypothesis

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

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

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

Conclusion <- hypothesis

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

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

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

Left: (e .)

Variables are local to facts and 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

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

Variables are local to facts and 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.

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

```
(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 (?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)}
```

```
(left: (app () ?x ?x)
```

Variables are local to facts and 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.

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

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

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

**Variables are local to facts and 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

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

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

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

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

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

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

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

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

Variables are local to facts and 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

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

Variables are local to facts and queries

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

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

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

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

\[
\text{conclusion} \leftarrow \text{hypothesis}
\]

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

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

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

\[
\text{conclusion} \leftarrow \text{hypothesis}
\]

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

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

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

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

Variables are local to facts and queries.
Underspecified Queries
Now that we know about Unification, let’s look at an underspecified query
Underspecified Queries

Now that we know about Unification, let’s look at an underspecified query.

What are the results of these queries?
Now that we know about Unification, let’s look at an underspecified query.

What are the results of these queries?

> (fact (append-to-form () ?x ?x))
Underspecified Queries

Now that we know about Unification, let’s look at an underspecified query.

What are the results of these queries?

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

> (fact (append-to-form (?a . ?r) ?x (?a . ?s))
   (append-to-form ?r ?x ?s))
Underspecified Queries

Now that we know about Unification, let’s look at an underspecified query

What are the results of these queries?

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

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

> (query (append-to-form (1 2) (3) ?what))
Now that we know about Unification, let’s look at an underspecified query

What are the results of these queries?

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

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

> (query (append-to-form (1 2) (3) ?what))
Success!
what: (1 2 3)
Now that we know about Unification, let’s look at an underspecified query.

What are the results of these queries?

> (fact (append-to-form () ?x ?x))
> (fact (append-to-form (?a . ?r) ?x (?a . ?s))
  (append-to-form ?r ?x ?s))
> (query (append-to-form (1 2) (3) ?what))
Success!
what: (1 2 3)
> (query (append-to-form (1 2 . ?r) (3) ?what)
Underspecified Queries

Now that we know about Unification, let’s look at an underspecified query

What are the results of these queries?

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

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

> (query (append-to-form (1 2) (3) ?what))
Success!
what: (1 2 3)

> (query (append-to-form (1 2 . ?r) (3) ?what))
Success!
Underspecified Queries

Now that we know about Unification, let’s look at an underspecified query

What are the results of these queries?

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

\[
> \text{(fact (append-to-form (?a . ?r) ?x (?a . ?s))}
\]
\[
> \text{(append-to-form ?r ?x ?s))}
\]

\[
> \text{(query (append-to-form (1 2) (3) ?what))}
\]

Success!
what: (1 2 3)

\[
> \text{(query (append-to-form (1 2 . ?r) (3) ?what)}
\]

Success!
r: () what: (1 2 3)
Underspecified Queries

Now that we know about Unification, let’s look at an underspecified query

What are the results of these queries?

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

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

> (query (append-to-form (1 2) (3) ?what))
Success!
what: (1 2 3)

> (query (append-to-form (1 2 . ?r) (3) ?what)
Success!
r: () what: (1 2 3)
r: (?s_6) what: (1 2 ?s_6 3)
Now that we know about Unification, let’s look at an underspecified query.

What are the results of these queries?

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

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

> (query (append-to-form (1 2) (3) ?what))
Success!
what: (1 2 3)

> (query (append-to-form (1 2 . ?r) (3) ?what))
Success!
r: () what: (1 2 3)
r: (?s_6) what: (1 2 ?s_6 3)
r: (?s_6 ?s_8) what: (1 2 ?s_6 ?s_8 3)
Underspecified Queries

Now that we know about Unification, let’s look at an underspecified query

What are the results of these queries?

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

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

> (query (append-to-form (1 2) (3) ?what))
Success!
what: (1 2 3)

> (query (append-to-form (1 2 . ?r) (3) ?what)
Success!
r: () what: (1 2 3)
r: (?s_6) what: (1 2 ?s_6 3)
r: (?s_6 ?s_8) what: (1 2 ?s_6 ?s_8 3)
r: (?s_6 ?s_8 ?s_10) what: (1 2 ?s_6 ?s_8 ?s_10 3)
Now that we know about Unification, let’s look at an underspecified query.

What are the results of these queries?

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

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

> (query (append-to-form (1 2) (3) ?what))
Success!
what: (1 2 3)

> (query (append-to-form (1 2 . ?r) (3) ?what))
Success!
r: ()  what: (1 2 3)
r: (?s_6)  what: (1 2 ?s_6 3)
r: (?s_6 ?s_8) what: (1 2 ?s_6 ?s_8 3)
r: (?s_6 ?s_8 ?s_10)  what: (1 2 ?s_6 ?s_8 ?s_10 3)
r: (?s_6 ?s_8 ?s_10 ?s_12)  what: (1 2 ?s_6 ?s_8 ?s_10 ?s_12 3)
Underspecified Queries

Now that we know about Unification, let’s look at an underspecified query.

What are the results of these queries?

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

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

> (query (append-to-form (1 2) (3) ?what))
Success!
what: (1 2 3)

> (query (append-to-form (1 2 . ?r) (3) ?what)
Success!
  r: ()  what: (1 2 3)
  r: (?s_6)  what: (1 2 ?s_6 3)
  r: (?s_6 ?s_8) what: (1 2 ?s_6 ?s_8 3)
  r: (?s_6 ?s_8 ?s_10) what: (1 2 ?s_6 ?s_8 ?s_10 3)
  r: (?s_6 ?s_8 ?s_10 ?s_12) what: (1 2 ?s_6 ?s_8 ?s_10 ?s_12 3)
...
Search for possible unification
Search for possible unification

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

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

A possible proof is explored exhaustively before another one is considered.
Search for possible unification

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

A possible proof is explored exhaustively before another one is considered.

def search(clauses, env):

Search for possible unification

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

A possible proof is explored exhaustively before another one is considered.

```python
def search(clauses, env):
    for fact in facts:
```
Search for possible unification

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

A possible proof is explored exhaustively before another one is considered.

\begin{verbatim}
def search(clauses, env):
    for fact in facts:
        env_head <- unify(conclusion of fact, first clause, env)
\end{verbatim}
Search for possible unification

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

A possible proof is explored exhaustively before another one is considered.

```python
def search(clauses, env):
    for fact in facts:
        env_head <- unify(conclusion of fact, first clause, env)
        if unification succeeds:
```
Search for possible unification

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

A possible proof is explored exhaustively before another one is considered.

```python
def search(clauses, env):
    for fact in facts:
        env_head <- unify(conclusion of fact, first clause, env)
        if unification succeeds:
            env_rule <- search(hypotheses of fact, env_head)
```
Search for possible unification

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

A possible proof is explored exhaustively before another one is considered.

```python
def search(clauses, env):
    for fact in facts:
        env_head <- unify(conclusion of fact, first clause, env)
        if unification succeeds:
            env_rule <- search(hypotheses of fact, env_head)
            result <- search(rest of clauses, env_rule)
```
Search for possible unification

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

A possible proof is explored exhaustively before another one is considered.

```python
def search(clauses, env):
    for fact in facts:
        env_head <- unify(conclusion of fact, first clause, env)
        if unification succeeds:
            env_rule <- search(hypotheses of fact, env_head)
            result <- search(rest of clauses, env_rule)
            yield each result
```
Search for possible unification

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

A possible proof is explored exhaustively before another one is considered.

```python
def search(clauses, env):
    for fact in facts:
        env_head <- unify(conclusion of fact, first clause, env)
        if unification succeeds:
            env_rule <- search(hypotheses of fact, env_head)
            result <- search(rest of clauses, env_rule)
            yield each result
```

Some good ideas:
Search for possible unification

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

A possible proof is explored exhaustively before another one is considered.

```python
def search(clauses, env):
    for fact in facts:
        env_head <- unify(conclusion of fact, first clause, env)
        if unification succeeds:
            env_rule <- search(hypotheses of fact, env_head)
            result <- search(rest of clauses, env_rule)
            yield each result
```

Some good ideas:

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

A possible proof is explored exhaustively before another one is considered.

```python
def search(clauses, env):
    for fact in facts:
        env_head <- unify(conclusion of fact, first clause, env)
        if unification succeeds:
            env_rule <- search(hypotheses of fact, env_head)
            result <- search(rest of clauses, env_rule)
            yield each result
```

Some good ideas:
- Limiting depth of the search avoids infinite loops
- Each time a fact is used, its variables are renamed
Search for possible unification

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

A possible proof is explored exhaustively before another one is considered.

```python
def search(clauses, env):
    for fact in facts:
        env_head <- unify(conclusion of fact, first clause, env)
        if unification succeeds:
            env_rule <- search(hypotheses of fact, env_head)
            result <- search(rest of clauses, env_rule)
            yield each result
```

Some good ideas:

- 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.
def search(clauses, env, depth):

    if clauses is nil:
        yield env

    elif DEPTH_LIMIT is None or depth <= DEPTH_LIMIT:
        for fact in facts:
            fact = rename_variables(fact, get_unique_id())
            env_head = Frame(env)
            if unify(fact.first, clauses.first, env_head):
                for env_rule in search(fact.second, env_head, depth+1):
                    for result in search(clauses.second, env_rule, depth+1):
                        yield result
def search(clauses, env, depth):
    if clauses is nil:
        yield env
    elif DEPTH_LIMIT is None or depth <= DEPTH_LIMIT:
        for fact in facts:
            fact = rename_variables(fact, get_unique_id())
            env_head = Frame(env)
            if unify(fact.first, clauses.first, env_head):
                for env_rule in search(fact.second, env_head, depth+1):
                    for result in search(clauses.second, env_rule, depth+1):
                        yield result
Implementing Search

def search(clauses, env, depth):
    if clauses is nil:
        yield env
    elif DEPTH_LIMIT is None or depth <= DEPTH_LIMIT:
        for fact in facts:
            fact = rename_variables(fact, get_unique_id())
            env_head = Frame(env)
            if unify(fact.first, clauses.first, env_head):
                for env_rule in search(fact.second, env_head, depth+1):
                    for result in search(clauses.second, env_rule, depth+1):
                        yield result
def search(clauses, env, depth):

    if clauses is nil:
        yield env

    elif DEPTH_LIMIT is None or depth <= DEPTH_LIMIT:

        for fact in facts:

            fact = rename_variables(fact, get_unique_id())

            env_head = Frame(env)

            if unify(fact.first, clauses.first, env_head):

                for env_rule in search(fact.second, env_head, depth+1):

                    for result in search(clauses.second, env_rule, depth+1):

                        yield result
def search(clauses, env, depth):
    if clauses is nil:
        yield env
    elif DEPTH_LIMIT is None or depth <= DEPTH_LIMIT:
        for fact in facts:
            fact = rename_variables(fact, get_unique_id())
            env_head = Frame(env)
            if unify(fact.first, clauses.first, env_head):
                for env_rule in search(fact.second, env_head, depth+1):
                    for result in search(clauses.second, env_rule, depth+1):
                        yield result
    Whatever calls search can access all yielded results
An Evaluator in Logic
An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:
An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:

```
logic> (fact (ints 1 2))
```
An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:

```
logic> (fact (ints 1 2))
logic> (fact (ints 2 3))
```
An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:

```
logic> (fact (ints 1 2))
logic> (fact (ints 2 3))
logic> (fact (ints 3 4))
```
An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:

```
logic> (fact (ints 1 2))
logic> (fact (ints 2 3))
logic> (fact (ints 3 4))
logic> (fact (ints 4 5))
```
We can define an evaluator in Logic; first, we define numbers:

```
logic> (fact (ints 1 2))
logic> (fact (ints 2 3))
logic> (fact (ints 3 4))
logic> (fact (ints 4 5))
```

Then we define addition:
An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:

\[
\begin{align*}
\text{logic}> & \ (\text{fact } (\text{ints } 1 \ 2)) \\
\text{logic}> & \ (\text{fact } (\text{ints } 2 \ 3)) \\
\text{logic}> & \ (\text{fact } (\text{ints } 3 \ 4)) \\
\text{logic}> & \ (\text{fact } (\text{ints } 4 \ 5)) \\
\end{align*}
\]

Then we define addition:

\[
\begin{align*}
\text{logic}> & \ (\text{fact } (\text{add } 1 \ ?x \ ?y) \ (\text{ints } ?x \ ?y)) \\
\end{align*}
\]
An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:

```
logic> (fact (ints 1 2))
logic> (fact (ints 2 3))
logic> (fact (ints 3 4))
logic> (fact (ints 4 5))
```

Then we define addition:

```
logic> (fact (add 1 ?x ?y) (ints ?x ?y))
logic> (fact (add ?x ?y ?z)
```
An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:

```logic
logic> (fact (ints 1 2))
logic> (fact (ints 2 3))
logic> (fact (ints 3 4))
logic> (fact (ints 4 5))
```

Then we define addition:

```logic
logic> (fact (add 1 ?x ?y) (ints ?x ?y))
logic> (fact (add ?x ?y ?z)
```

Finally, we define the evaluator:
We can define an evaluator in Logic; first, we define numbers:

logic> (fact (ints 1 2))
logic> (fact (ints 2 3))
logic> (fact (ints 3 4))
logic> (fact (ints 4 5))

Then we define addition:

logic> (fact (add 1 ?x ?y) (ints ?x ?y))
logic> (fact (add ?x ?y ?z)

Finally, we define the evaluator:

logic> (fact (eval ?x ?x) (ints ?x ?something))
An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:

```
logic> (fact (ints 1 2))
logic> (fact (ints 2 3))
logic> (fact (ints 3 4))
logic> (fact (ints 4 5))
```

Then we define addition:

```
logic> (fact (add 1 ?x ?y) (ints ?x ?y))
logic> (fact (add ?x ?y ?z)
```

Finally, we define the evaluator:

```
logic> (fact (eval ?x ?x) (ints ?x ?something))
logic> (fact (eval (+ ?op0 ?op1) ?val)
    (add ?a0 ?a1 ?val) (eval ?op0 ?a0) (eval ?op1 ?a1))
```
An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:

```prolog
logic> (fact (ints 1 2))
logic> (fact (ints 2 3))
logic> (fact (ints 3 4))
logic> (fact (ints 4 5))
```

Then we define addition:

```prolog
logic> (fact (add 1 ?x ?y) (ints ?x ?y))
logic> (fact (add ?x ?y ?z)
```

Finally, we define the evaluator:

```prolog
logic> (fact (eval ?x ?x) (ints ?x ?something))
logic> (fact (eval (+ ?op0 ?op1) ?val)
          (add ?a0 ?a1 ?val) (eval ?op0 ?a0) (eval ?op1 ?a1))
logic> (query (eval (+ 1 (+ ?what 2)) 5))
```
An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:

```
(FACT (INTS 1 2))
(FACT (INTS 2 3))
(FACT (INTS 3 4))
(FACT (INTS 4 5))
```

Then we define addition:

```
(FACT (ADD 1 ?X ?Y) (INTS ?X ?Y))
(FACT (ADD ?X ?Y ?Z)
```

Finally, we define the evaluator:

```
(FACT (EVAL ?X ?X) (INTS ?X ?SOMETHING))
(FACT (EVAL (+ ?OP0 ?OP1) ?VAL)
(QUERY (EVAL (+ 1 (+ ?WHAT 2)) 5))
Success!
```
An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:

```
logic> (fact (ints 1 2))
logic> (fact (ints 2 3))
logic> (fact (ints 3 4))
logic> (fact (ints 4 5))
```

Then we define addition:

```
logic> (fact (add 1 ?x ?y) (ints ?x ?y))
logic> (fact (add ?x ?y ?z)
```

Finally, we define the evaluator:

```
logic> (fact (eval ?x ?x) (ints ?x ?something))
logic> (fact (eval (+ ?op0 ?op1) ?val)
  (add ?a0 ?a1 ?val) (eval ?op0 ?a0) (eval ?op1 ?a1))
logic> (query (eval (+ 1 (+ ?what 2)) 5))
Success!
what: 2
```
An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:

```
logic> (fact (ints 1 2))
logic> (fact (ints 2 3))
logic> (fact (ints 3 4))
logic> (fact (ints 4 5))
```

Then we define addition:

```
logic> (fact (add 1 ?x ?y) (ints ?x ?y))
logic> (fact (add ?x ?y ?z)
```

Finally, we define the evaluator:

```
logic> (fact (eval ?x ?x) (ints ?x ?something))
logic> (fact (eval (+ ?op0 ?op1) ?val)
    (add ?a0 ?a1 ?val) (eval ?op0 ?a0) (eval ?op1 ?a1))
```

```
logic> (query (eval (+ 1 (+ ?what 2)) 5))
Success!
what: 2
what: (+ 1 1)
```