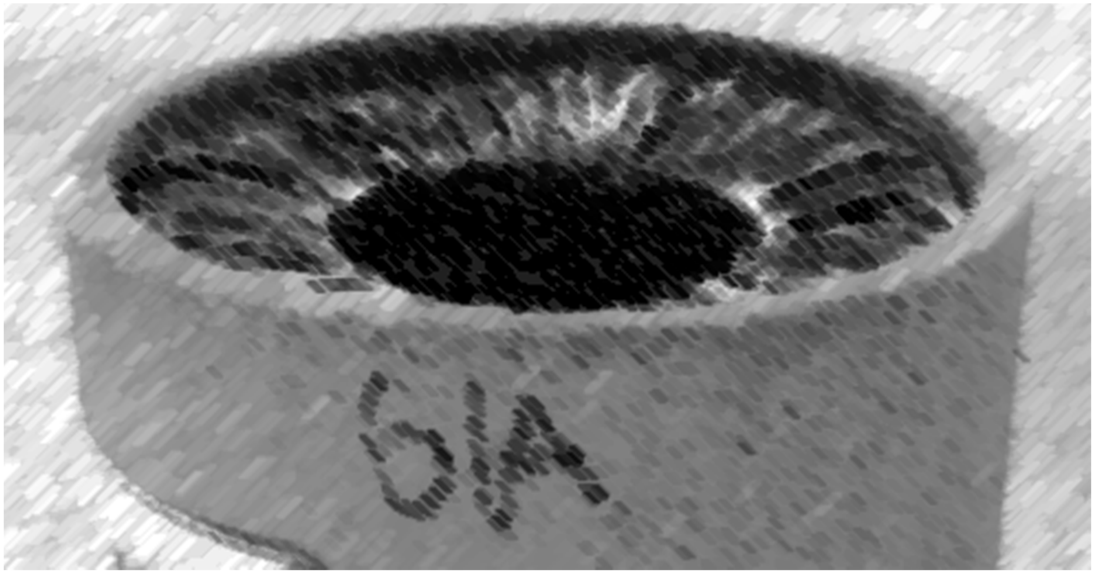


CS61A Lecture 40

Amir Kamil and Stephen Martinis

UC Berkeley

April 24, 2013

Announcements

- HW12 due tonight
- HW13 out
- Scheme project, contest due Monday

Logic Language Review

Logic Language Review

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

Logic Language Review

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

```
logic> (fact (parent eisenhower fillmore))
logic> (fact (parent fillmore abraham))
logic> (fact (parent abraham clinton))
```

Logic Language Review

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

Logic Language Review

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

Logic Language Review

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

Logic Language Review

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> (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

Logic Language Review

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> (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

```
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

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

```
logic> (fact (dog (name abraham) (color white)))
```

Hierarchical Facts

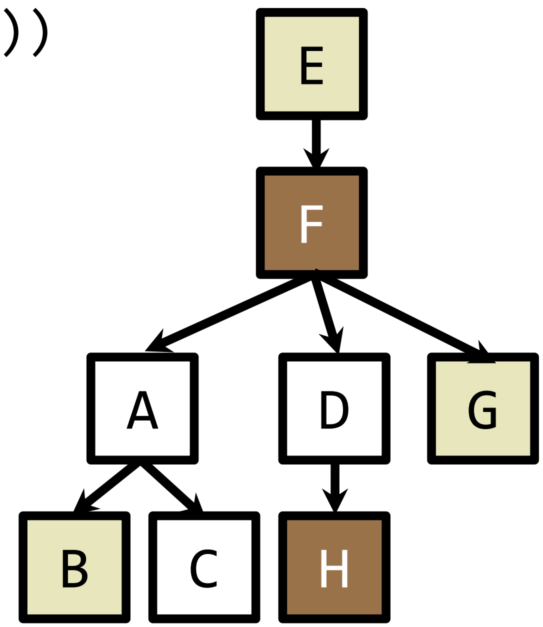
Relations can contain relations in addition to atoms

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

Hierarchical Facts

Relations can contain relations in addition to atoms

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

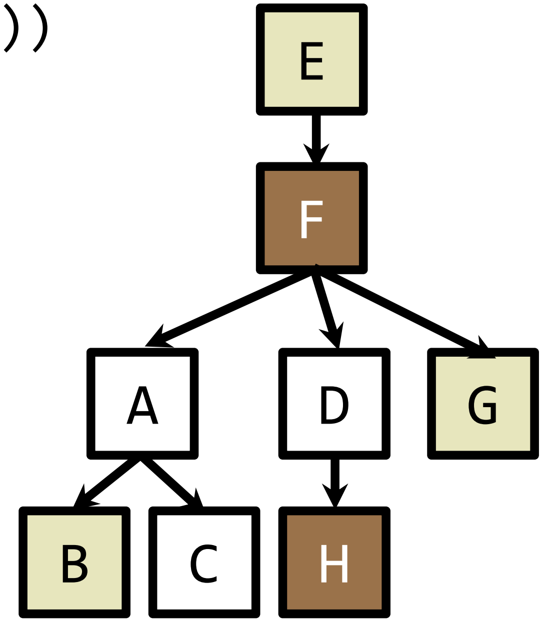


Hierarchical Facts

Relations can contain relations in addition to atoms

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

Variables can refer to atoms or relations



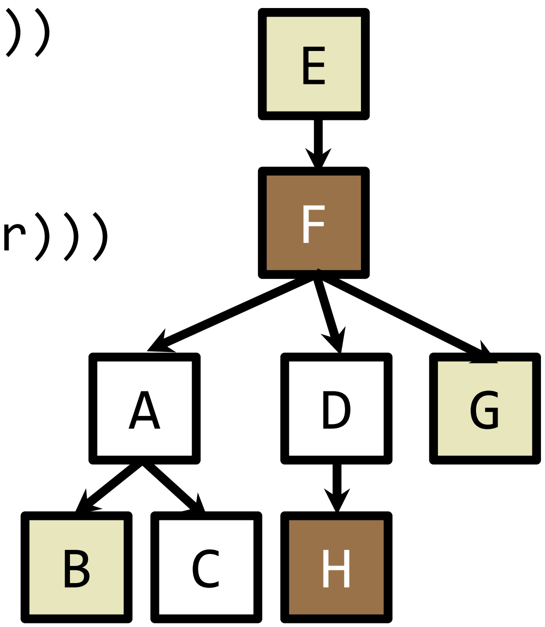
Hierarchical Facts

Relations can contain relations in addition to atoms

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

Variables can refer to atoms or relations

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



Hierarchical Facts

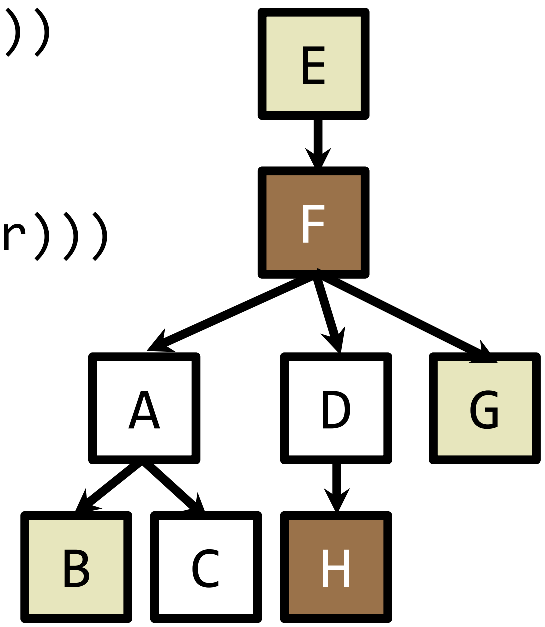
Relations can contain relations in addition to atoms

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

Variables can refer to atoms or relations

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

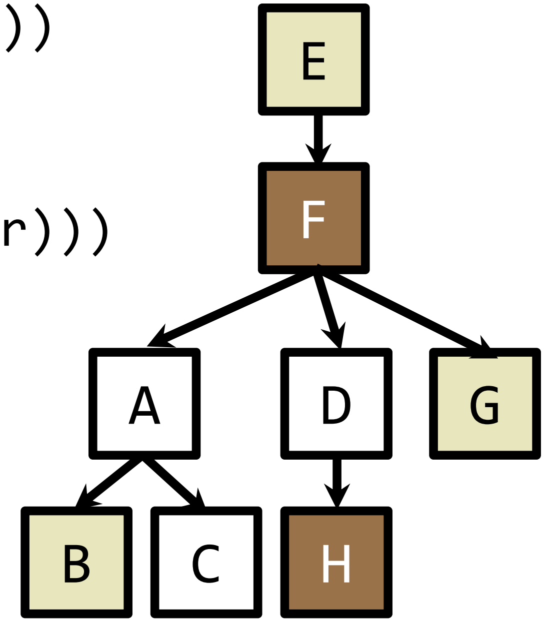
```
logic> (query (dog (name clinton) ?info))
Success!
info: (color white)
```



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)))
```



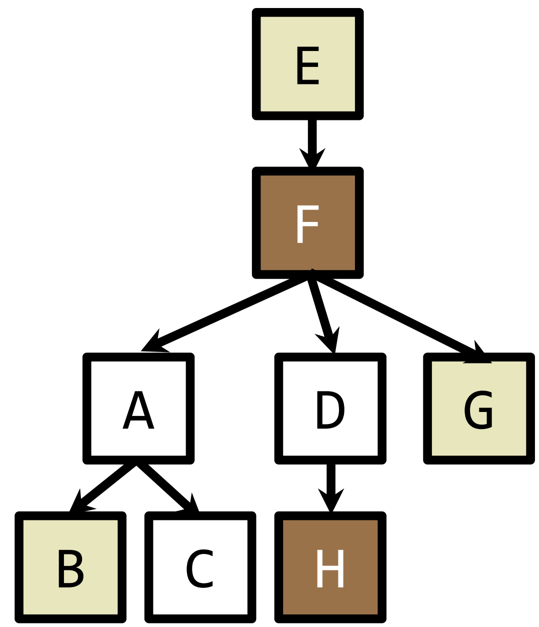
Variables can refer to atoms or relations

```
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

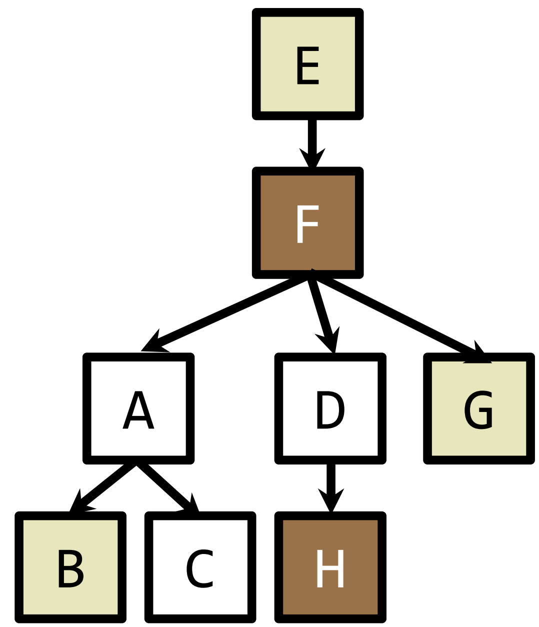
Which dogs have an ancestor of the same color?



Example: Combining Multiple Data Sources

Which dogs have an ancestor of the same color?

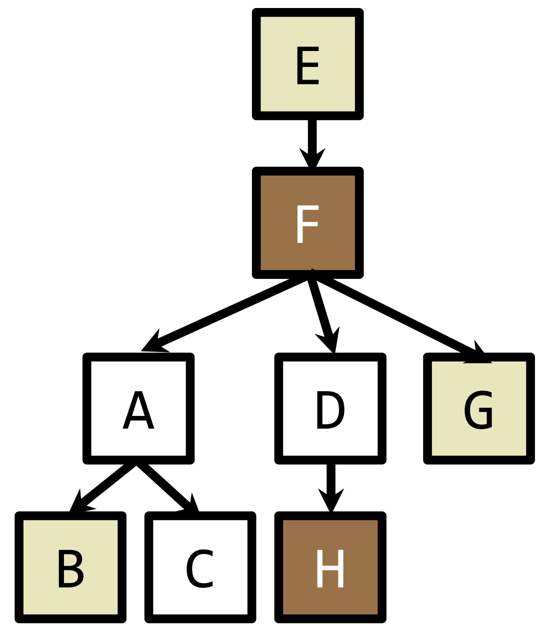
```
logic> (query (dog (name ?name) (color ?color))
```



Example: Combining Multiple Data Sources

Which dogs have an ancestor of the same color?

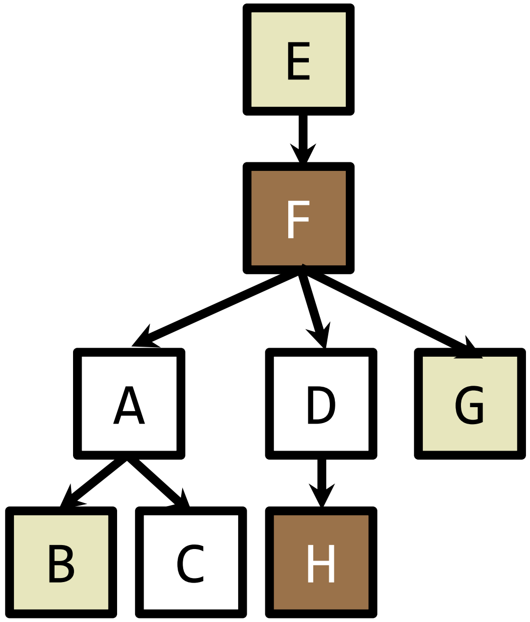
```
logic> (query (dog (name ?name) (color ?color))
              (ancestor ?ancestor ?name)
```



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)))
```

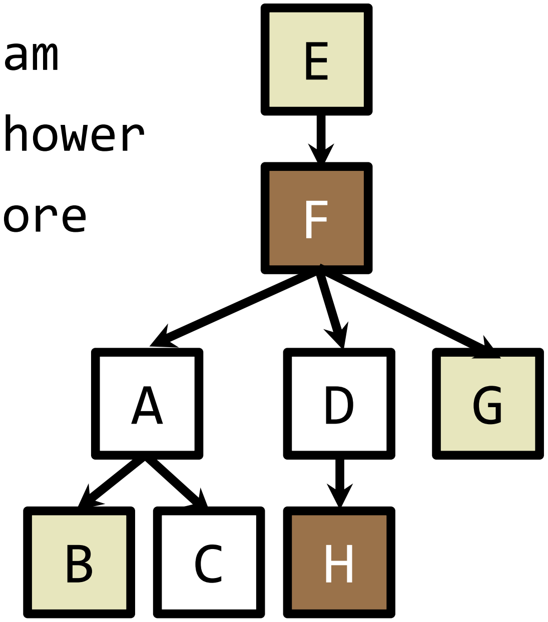


Example: Combining Multiple Data Sources

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))
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))
```

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

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

a | r t

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

```
a | r t

  r t
 ar 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

```
a | r t

  r t
 ar t
 rat
```

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

 r t
ar t
 rat
 r ta
```

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

  r t
 ar t
  rat
  r ta

  t 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

```
a|r t

 r t
ar t
 rat
 r ta

 t r
at 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

```
a | r t

  r t
 ar t
  rat
  r ta

  t r
 at r
  tar
```

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

  r t
 ar t
  rat
  r ta

  t r
 at r
  tar
  t ra
```

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)))
```

```
a | r t

  r t
 ar t
  rat
  r ta

  t r
 at r
  tar
  t ra
```

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

Element

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

```
a | r t

  r t
 ar t
  rat
  r ta

  t r
 at r
  tar
  t ra
```

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

Element List

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

```
a | r t

  r t
 ar t
  rat
  r ta

  t r
 at r
  tar
  t ra
```

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

Element | List | List with element

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

a | r t

r t
ar t
rat
r ta

t r
at r
tar
t ra

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

Element | List | List with element

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

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

```
a | r t

  r t
ar t
 rat
 r ta

  t r
at r
 tar
 t ra
```

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

Element | List | List with element

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

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

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

```
a | r t

  r t
 ar t
  rat
  r ta

  t r
 at r
  tar
  t ra
```

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

Element | List | List with element

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

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

(fact (anagram () ()))

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

```
a|r t

  r t
 ar t
  rat
  r ta

  t r
 at r
  tar
  t ra
```

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

Element | List | List with element

```
(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 | r t

  r t
 ar t
  rat
  r ta

  t r
 at r
  tar
  t ra
```

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

Element | List | List with element

```
(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))
```

```
a | r t

  r t
ar t
  rat
  r ta

  t r
at r
  tar
  t ra
```

Pattern Matching

Pattern Matching

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

Pattern Matching

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

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

Pattern Matching

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

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

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

Pattern Matching

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

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

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

Pattern Matching

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

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

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

▷ `True, {x: (a b)}`

Pattern Matching

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

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

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

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

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

Pattern Matching

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

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

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

→ `True, {x: (a b)}`

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

Pattern Matching

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

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

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

▷ `True, {x: (a b)}`

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

▷ `True, {y: b, z: c}`

Pattern Matching

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

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

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

→ `True, {x: (a b)}`

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

→ `True, {y: b, z: c}`

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

Pattern Matching

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

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

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

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

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

Unification

Unification

Unification 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

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

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

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    )
```

```
{   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

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

(   ?x   c   ?x   )
```

```
{  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

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

(   ?x   c   ?x   )
```

```
{  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

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

(    ?x   c    ?x   )
```

Lookup

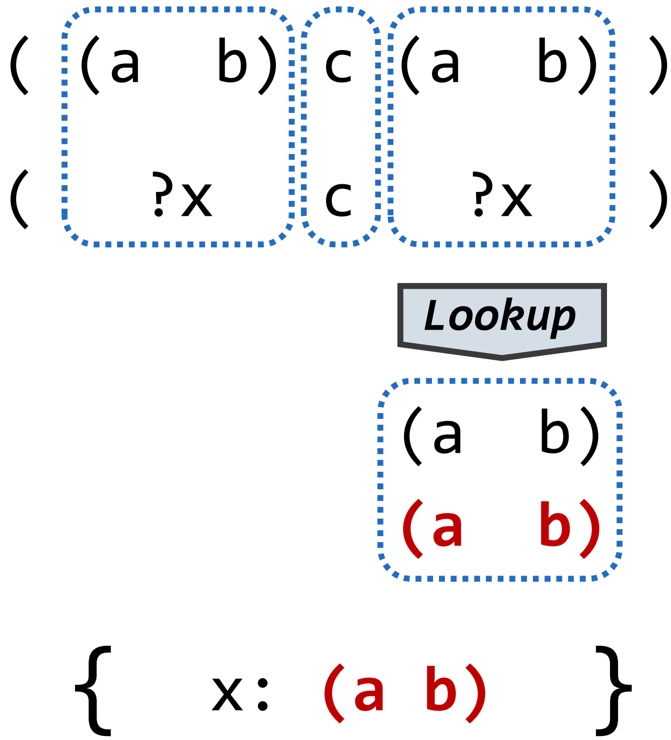
```
(a   b)
(a   b)
```

```
{   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



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

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

(    ?x    c     ?x   )
```

Lookup

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

{ 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

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

Lookup

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

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

Success!

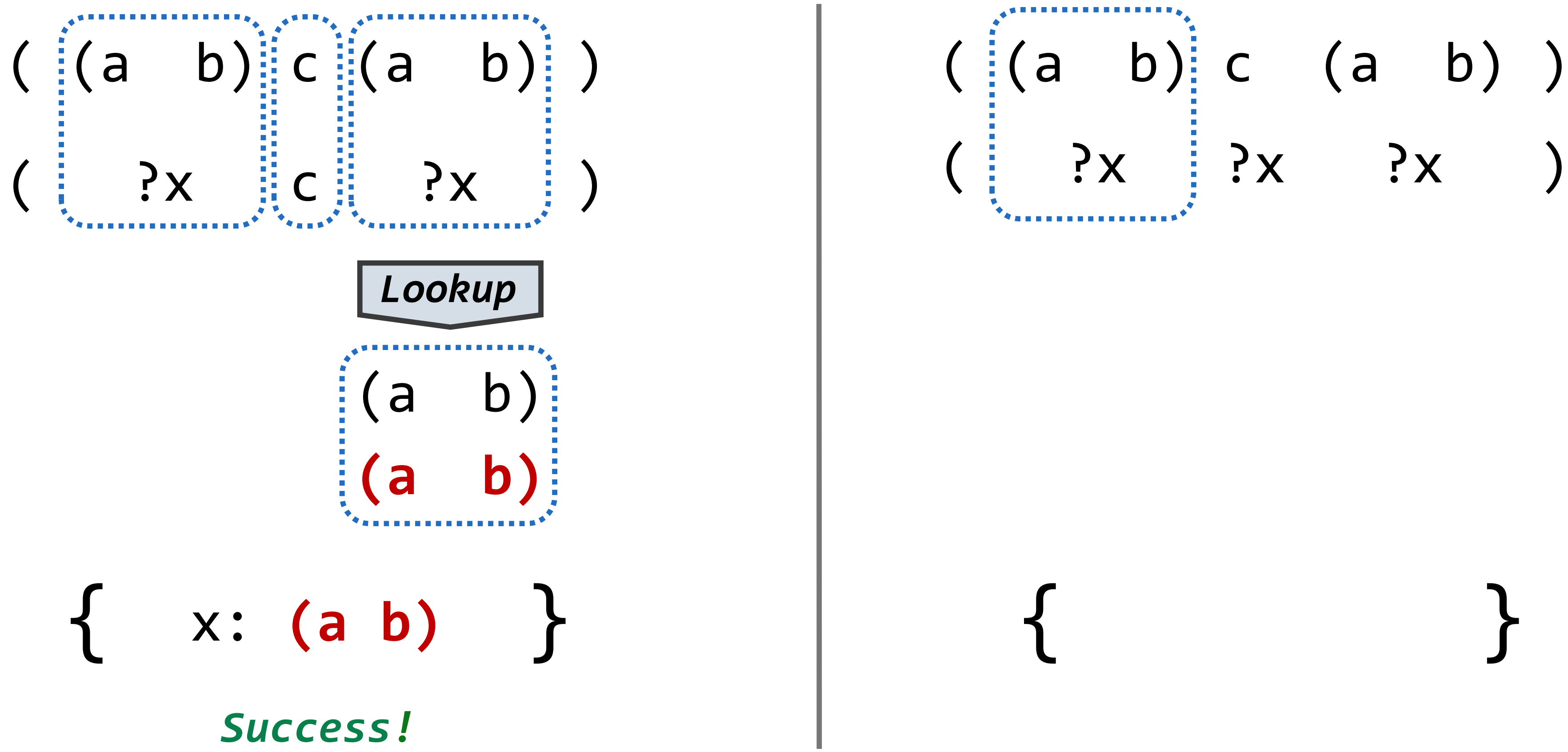
```
( (a   b) c   (a   b) )
(    ?x     ?x     ?x   )
```

```
{            }
```

Unification

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

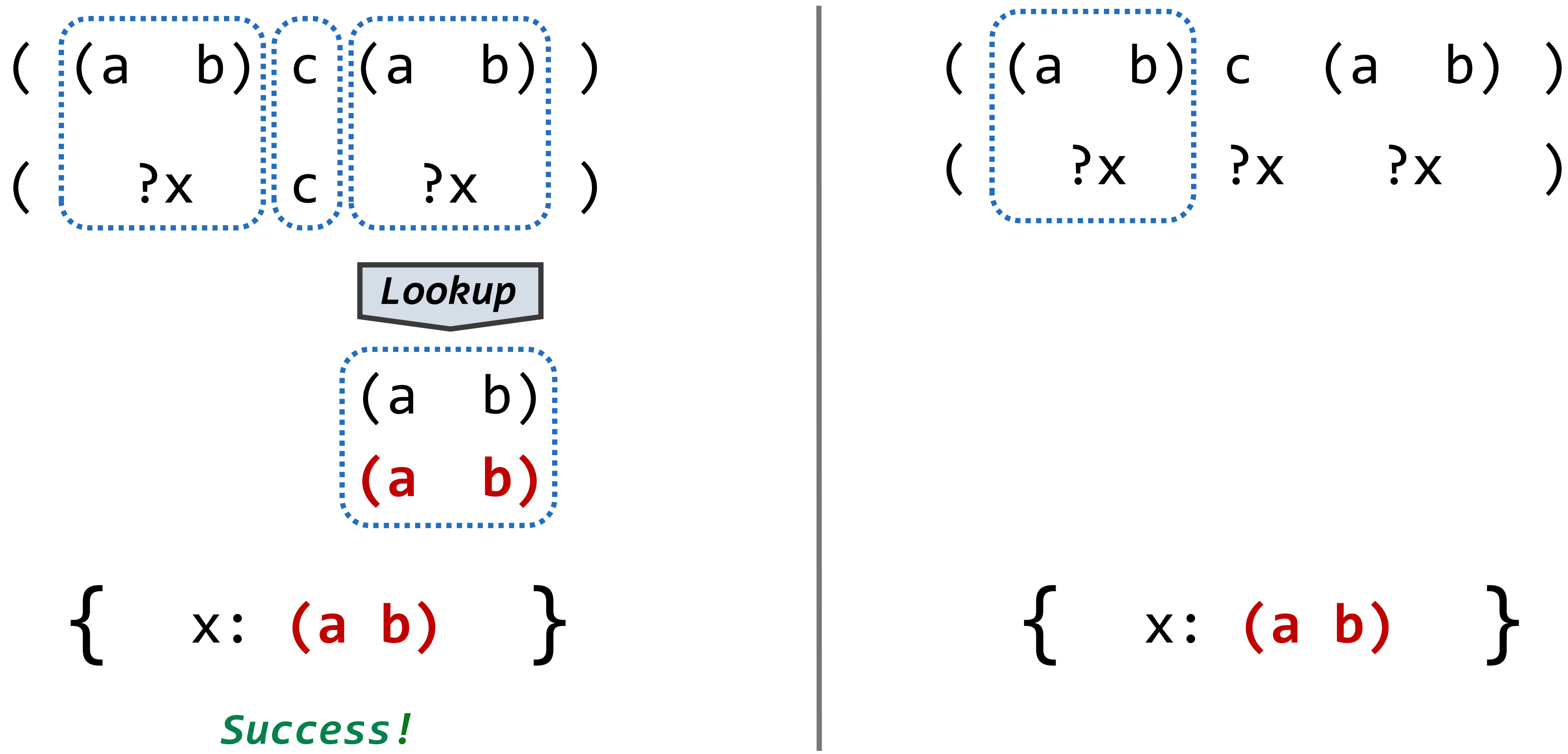
1. Look up variables in the current environment
2. Establish new bindings to unify elements



Unification

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

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

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

Lookup

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

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

Success!

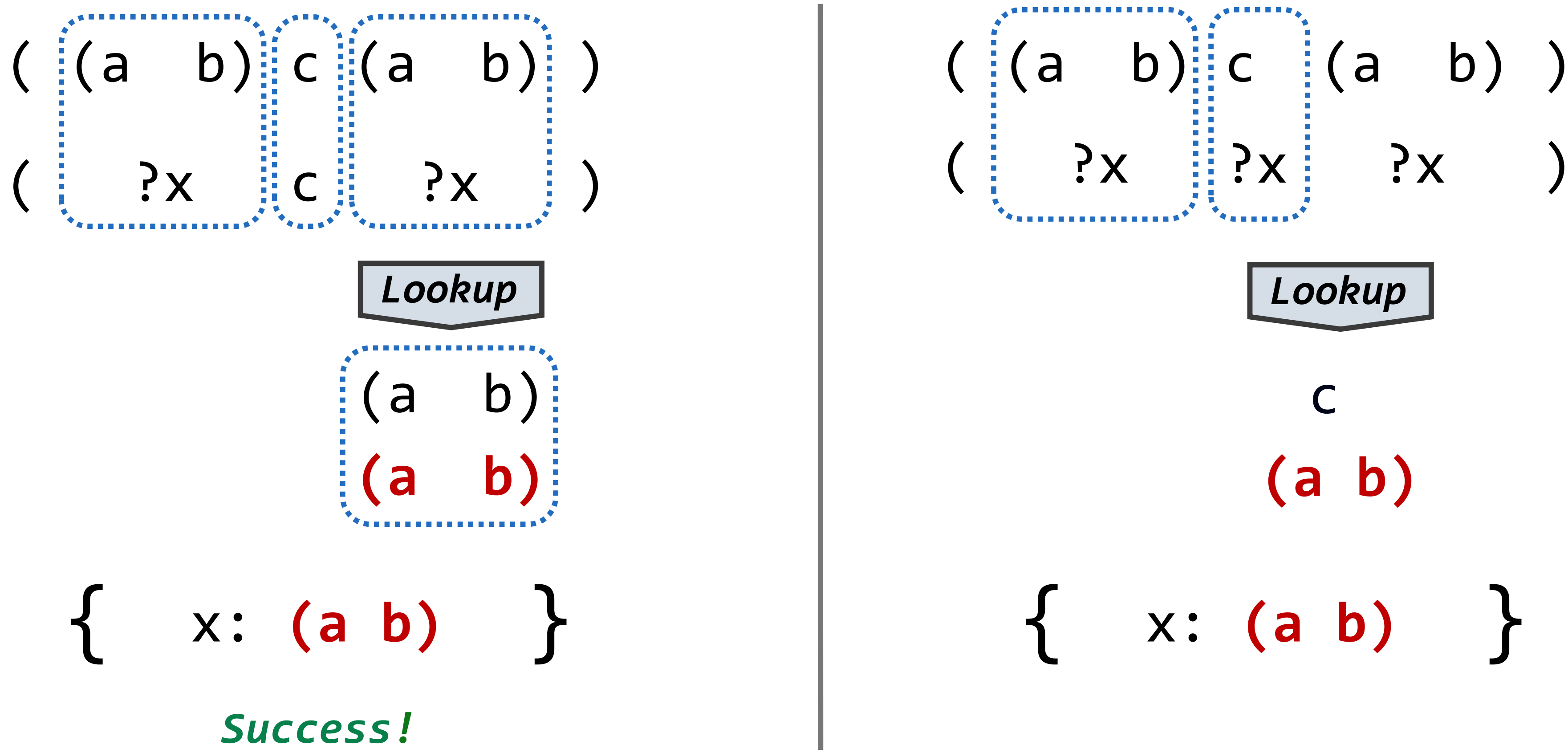
```
( (a   b) c (a   b) )
(    ?x   ?x   ?x   )
```

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

Unification

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

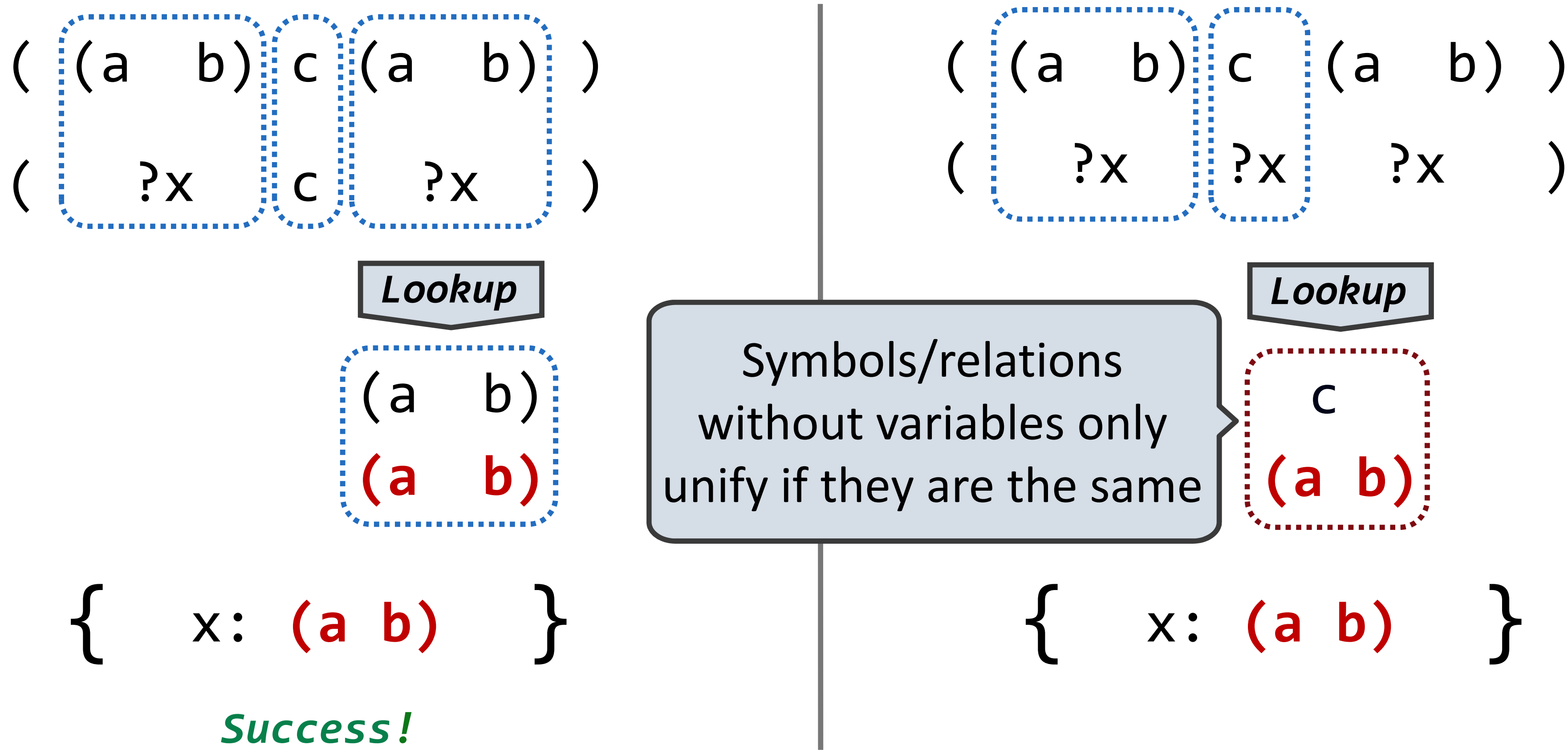
1. Look up variables in the current environment
2. Establish new bindings to unify elements



Unification

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

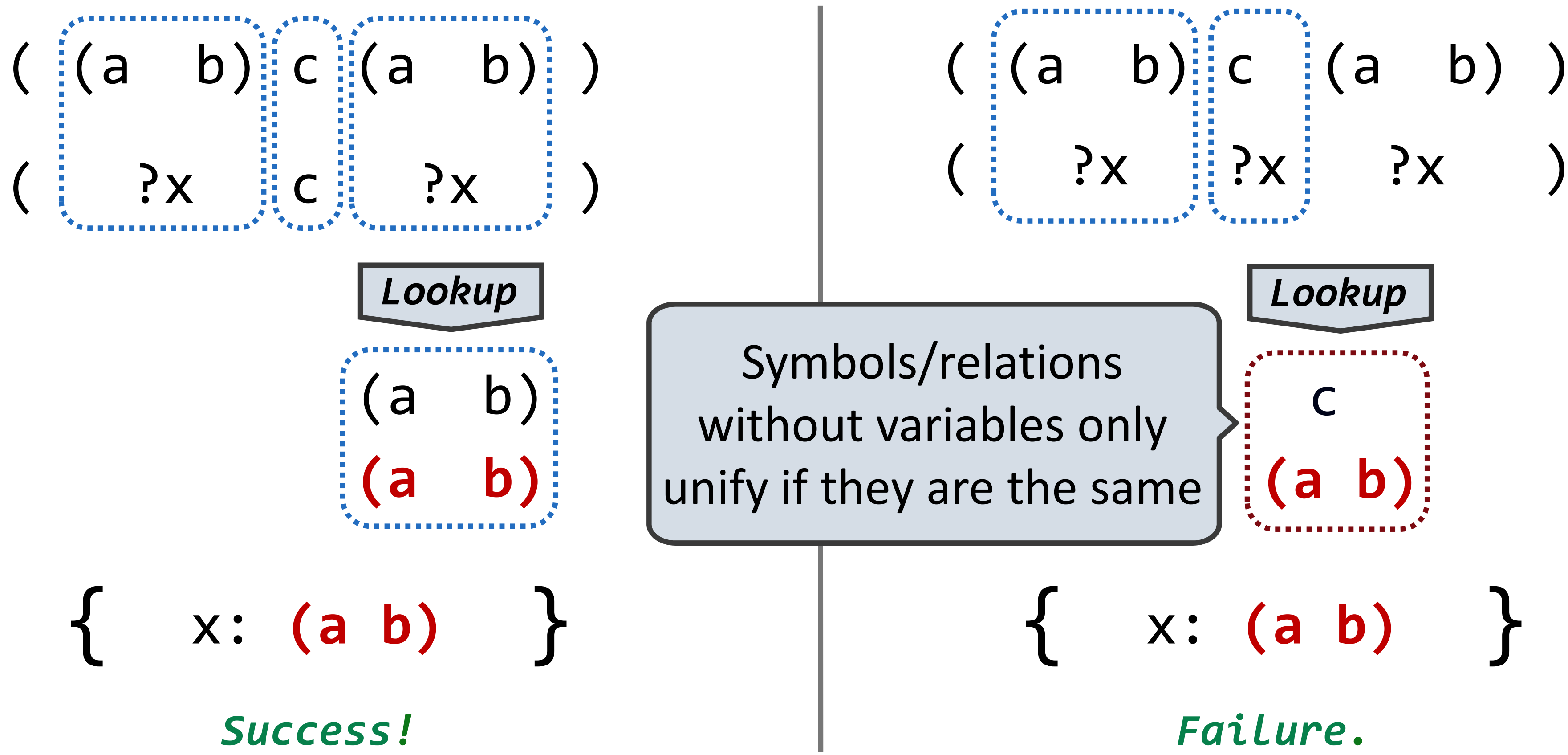
1. Look up variables in the current environment
2. Establish new bindings to unify elements



Unification

Unification 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 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          ?x     )

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

▷

```
True, {
```

Unification with Two Variables

Two relations that contain variables can be unified as well

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

Unification with Two Variables

Two relations that contain variables can be unified as well

```
(      ?x            ?x       )

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

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

Unification with Two Variables

Two relations that contain variables can be unified as well

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

▷ `True, {x:` **`(a ?y c)`**`,`

Unification with Two Variables

Two relations that contain variables can be unified as well

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

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

Lookup

(a ?y c)

(a b ?z)

Unification with Two Variables

Two relations that contain variables can be unified as well

```
(    ?x          ?x      )

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

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

Lookup

```
(a ?y  c)

(a  b ?z)
```

Unification with Two Variables

Two relations that contain variables can be unified as well

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

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

Lookup

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

Unification with Two Variables

Two relations that contain variables can be unified as well

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

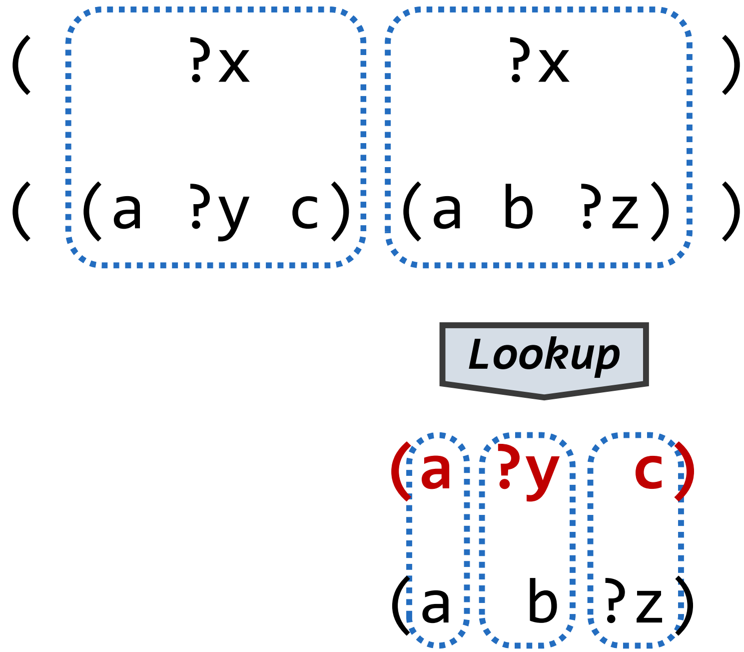
True, {x: **(a ?y c)**,
y: **b**,

Lookup

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

Unification with Two Variables

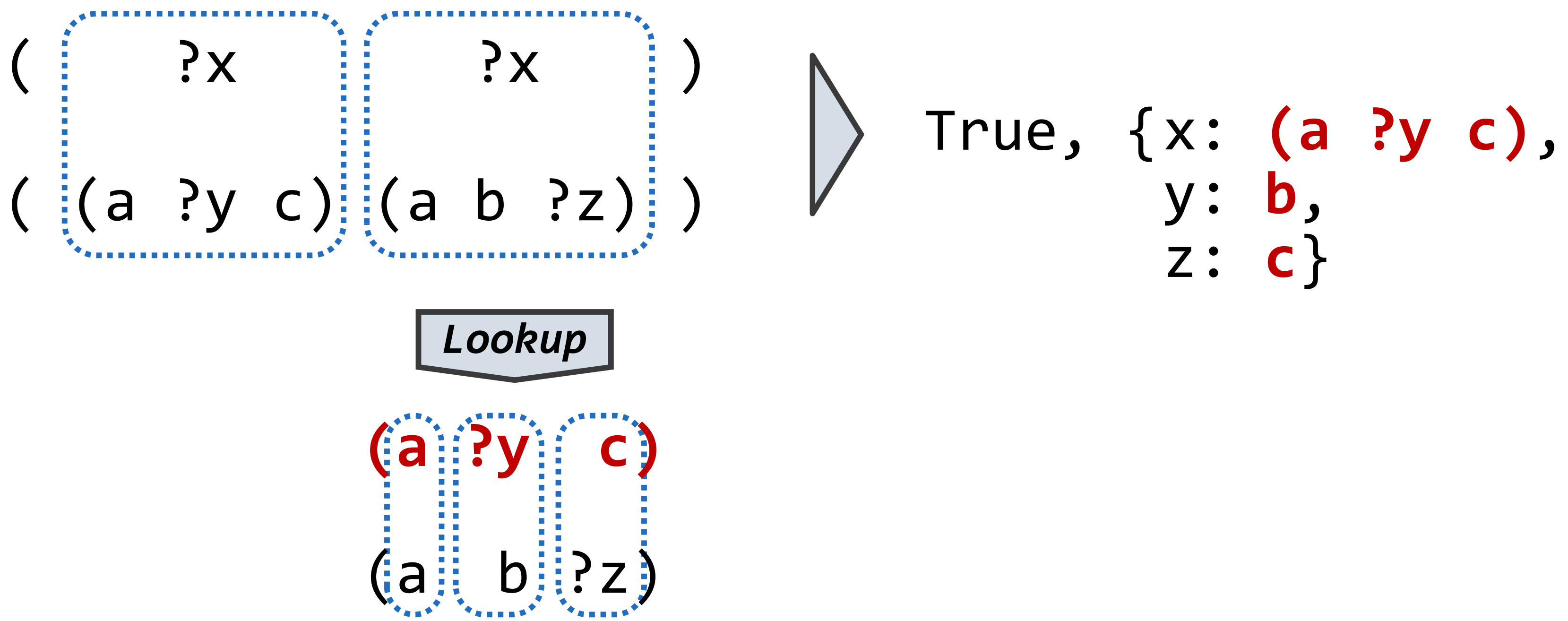
Two relations that contain variables can be unified as well




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

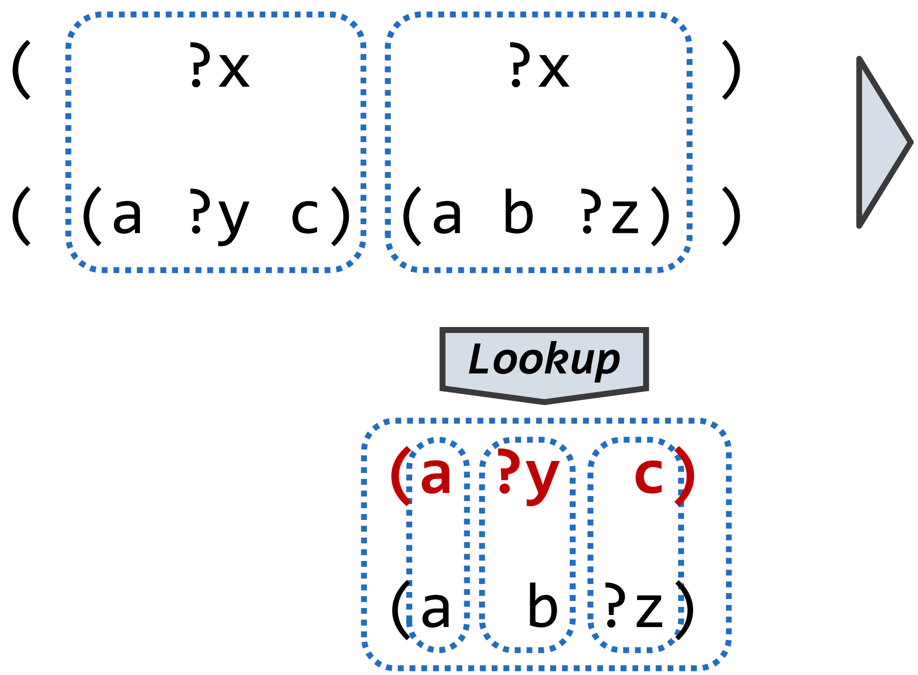
Unification with Two Variables

Two relations that contain variables can be unified as well



Unification with Two Variables

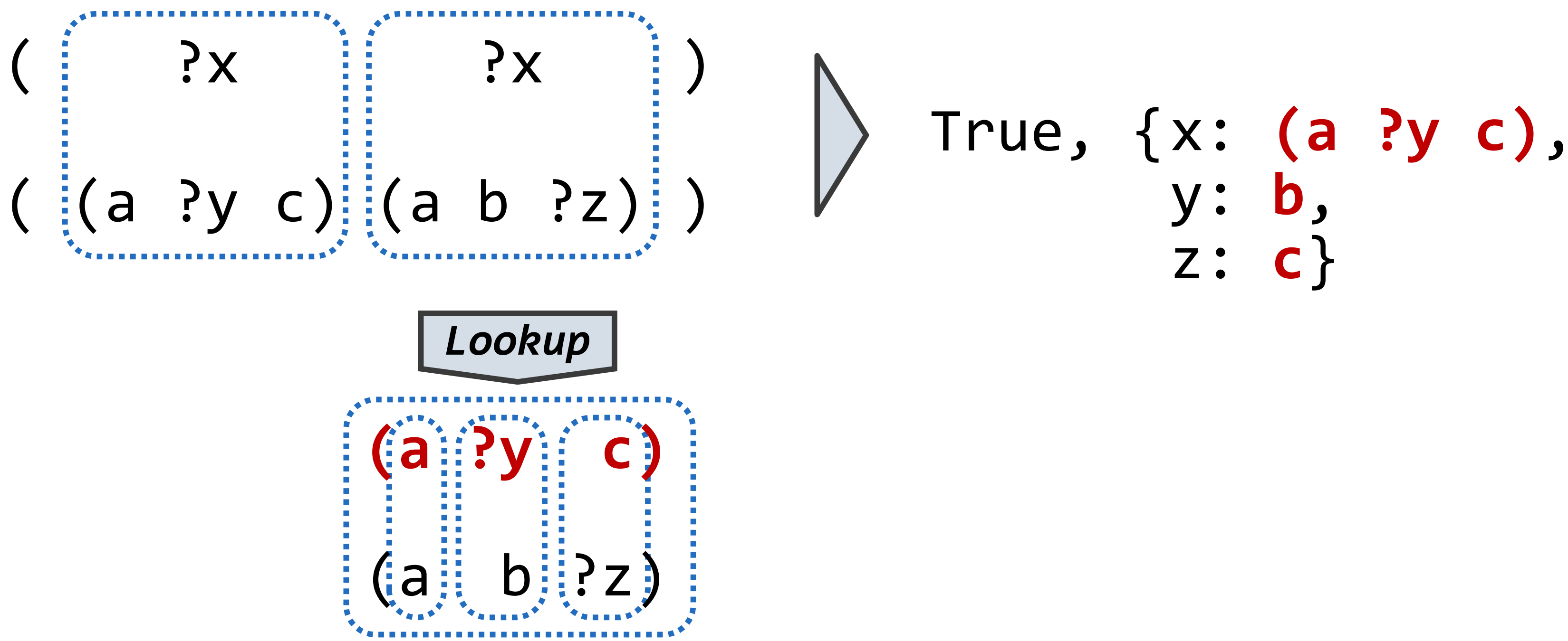
Two relations that contain variables can be unified as well




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

Unification with Two Variables

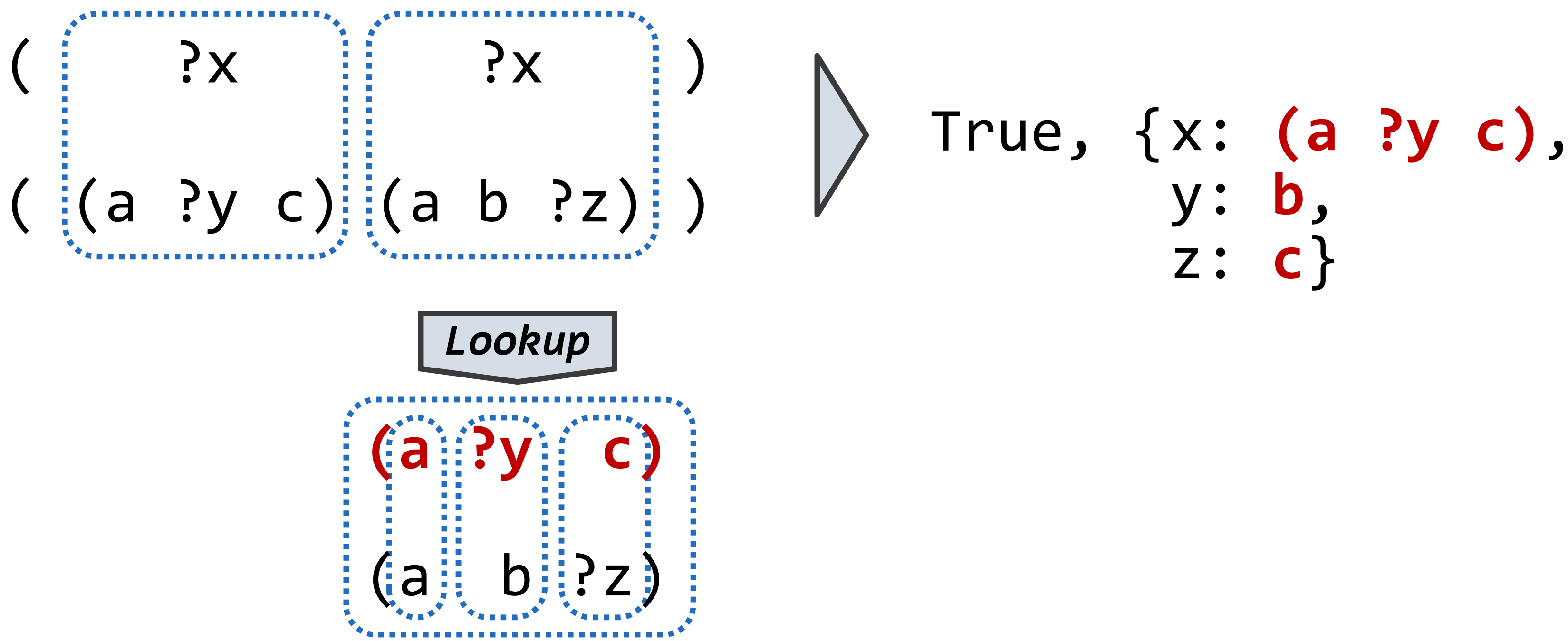
Two relations that contain variables can be unified as well



Substituting values for variables may require multiple steps

Unification with Two Variables

Two relations that contain variables can be unified as well

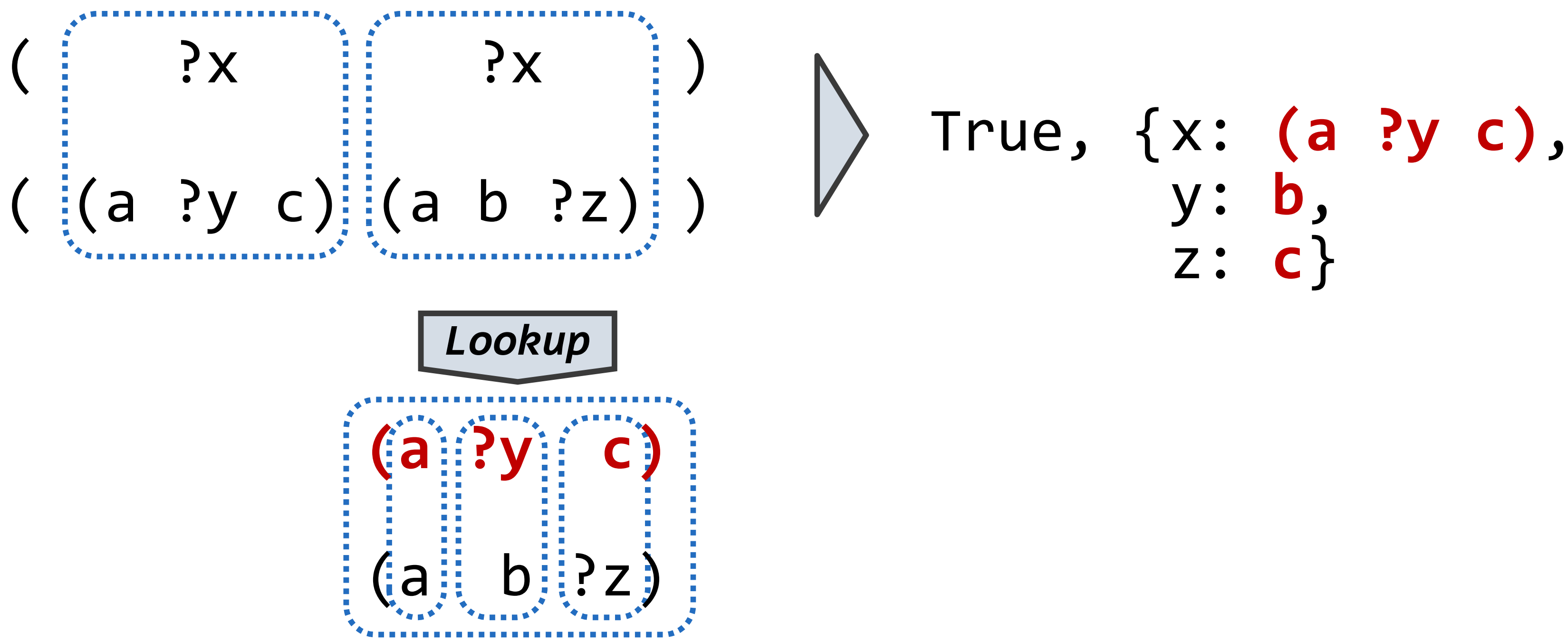


Substituting values for variables may require multiple steps

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

Unification with Two Variables

Two relations that contain variables can be unified as well

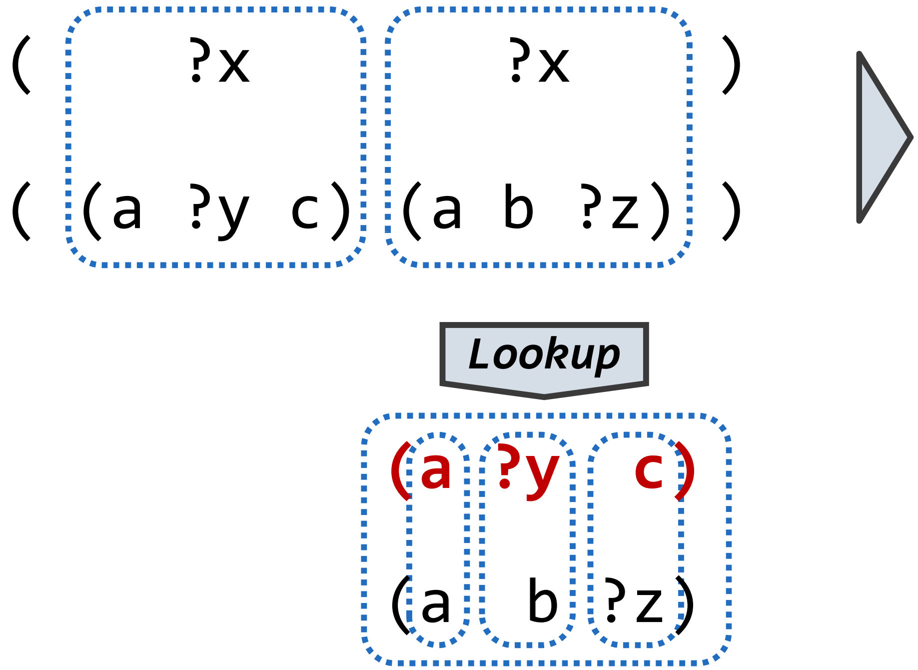


Substituting values for variables may require multiple steps

```
lookup('?x') ⇨ (a ?y c)
```

Unification with Two Variables

Two relations that contain variables can be unified as well



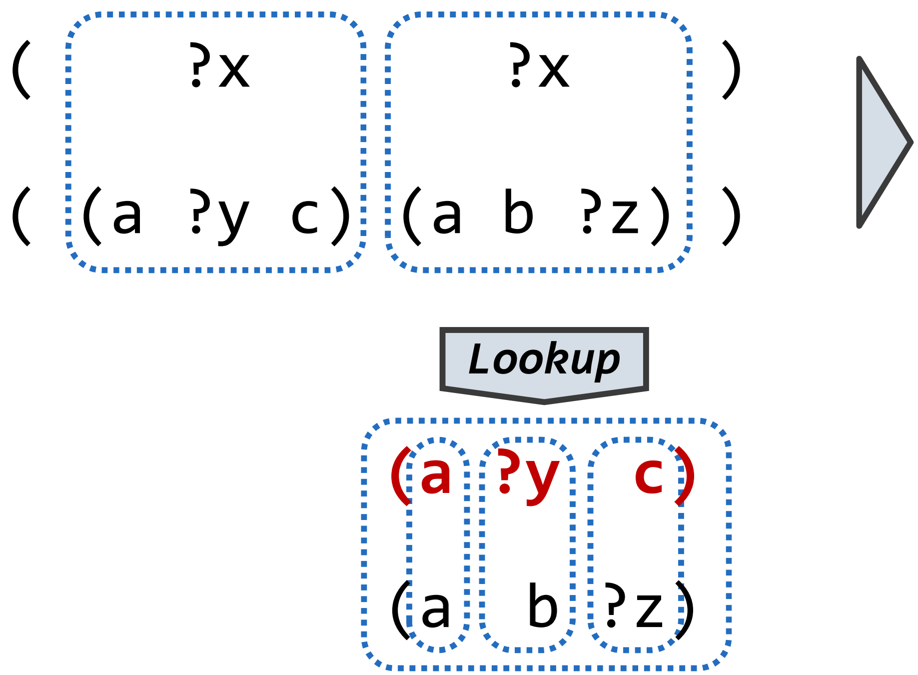

```
True, {x: (a ?y c),
       y: b,
       z: c}
```

Substituting values for variables may require multiple steps

```
lookup('?x')  ⇒  (a ?y c)        lookup('?y')
```

Unification with Two Variables

Two relations that contain variables can be unified as well




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

Substituting values for variables may require multiple steps

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

Implementing Unification

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

Implementing Unification

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

1. Look up variables in the current environment

Implementing Unification

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

1. Look up variables in the current environment

2. Establish new bindings to unify elements.

Implementing Unification

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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.

Implementing Unification

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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.

Unification recursively unifies each pair of elements

Searching for Proofs

Searching for Proofs

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

Searching for Proofs

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

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

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

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

Searching for Proofs

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

```
(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

```
(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))

        conclusion <- hypothesis

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

Searching for Proofs

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

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

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

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

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

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

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

        conclusion <- hypothesis

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


        (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 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

```
(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 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

```
(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 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

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

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

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

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

Variables are local to facts 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

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

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

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

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

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

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

```
(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 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

```
(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 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

```
(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 and queries

left:

Searching for Proofs

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

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

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

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

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

Variables are local to facts and queries

left:

Searching for Proofs

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

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

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

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

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

Variables are local to facts and queries

left:

Searching for Proofs

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

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

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

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

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

Variables are local to facts and 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 and 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 and 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 and queries

left: (e . (b .

Searching for Proofs

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

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

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

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

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

Variables are local to facts and queries

left: (e . (b .

Searching for Proofs

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

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

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

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

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

Variables are local to facts and queries

left: (e . (b . ()))

Searching for Proofs

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

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

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

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

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

Variables are local to facts and queries

left: (e . (b . ())) ⇨ (e b)

Underspecified 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?

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))
```

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))
```

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)
```

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)
```

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?

```
> (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)
```

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)
```

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)
```

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)
```

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)
```

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

```
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 *depth-first* exploration order

A possible proof is explored exhaustively before another one is considered

```
def search(clauses, env):
  for fact in facts:
    env_head <- unify(conclusion of fact, first clause, 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

```
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

```
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

```
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

```
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

```
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

```
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

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):
  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

```
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

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
```

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
```

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
```

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
```

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
```

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))
```

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:

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))
```

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)
             (ints ?x-1 ?x) (ints ?z-1 ?z) (add ?x-1 ?y ?z-1))
```

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)
             (ints ?x-1 ?x) (ints ?z-1 ?z) (add ?x-1 ?y ?z-1))
```

Finally, we define the evaluator:

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)
             (ints ?x-1 ?x) (ints ?z-1 ?z) (add ?x-1 ?y ?z-1))
```

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)
             (ints ?x-1 ?x) (ints ?z-1 ?z) (add ?x-1 ?y ?z-1))
```

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:

```
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)
             (ints ?x-1 ?x) (ints ?z-1 ?z) (add ?x-1 ?y ?z-1))
```

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))
```

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)
             (ints ?x-1 ?x) (ints ?z-1 ?z) (add ?x-1 ?y ?z-1))
```

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!
```

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)
             (ints ?x-1 ?x) (ints ?z-1 ?z) (add ?x-1 ?y ?z-1))
```

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)
             (ints ?x-1 ?x) (ints ?z-1 ?z) (add ?x-1 ?y ?z-1))
```

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)
```