61A Lecture 32

Friday, November 22

Announcements

Announcements

- Homework 10 due Tuesday 11/26 @ 11:59pm

Announcements

- Homework 10 due Tuesday 11/26 @ 11:59pm
- No lecture on Wednesday 11/27 or Friday 11/29

Announcements

- Homework 10 due Tuesday 11/26 @ 11:59pm
- No lecture on Wednesday 11/27 or Friday 11/29
- No discussion section Wednesday 11/27 through Friday 11/29

Announcements

- Homework 10 due Tuesday 11/26 @ 11:59pm
- No lecture on Wednesday 11/27 or Friday 11/29
- No discussion section Wednesday 11/27 through Friday 11/29
    - Lab will be held on Wednesday 11/27

Announcements

- Homework 10 due Tuesday 11/26 @ 11:59pm
- No lecture on Wednesday 11/27 or Friday 11/29
- No discussion section Wednesday 11/27 through Friday 11/29
    - Lab will be held on Wednesday 11/27
- Recursive art contest entries due Monday 12/2 @ 11:59pm

Appending Lists

(Demo)

Lists in Logic

Lists in Logic

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

Simple fact: Conclusion

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

Simple fact: Conclusion

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

Simple fact: Conclusion

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

Conclusion

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

Simple fact: Conclusion

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

Conclusion

Hypothesis

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

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

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

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

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

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

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

Simple fact: Conclusion

Conclusion

Hypothesis

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

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

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

Simple fact: Conclusion

Conclusion

Hypothesis

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

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

Simple fact: Conclusion

Conclusion

Hypothesis

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

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

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

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

Simple fact: Conclusion

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

Conclusion

Hypothesis

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

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

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

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

Expressions and their relations are Scheme lists.

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

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

Simple fact: Conclusion

Conclusion

Hypothesis

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

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

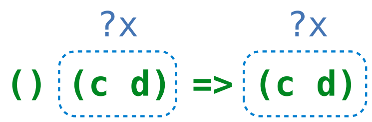
In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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



Expressions and their relations are Scheme lists.

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

Simple fact: Conclusion

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

Conclusion

Hypothesis

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

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

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

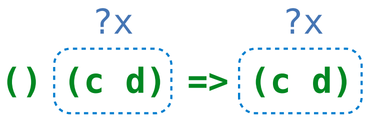
```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis



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

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

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

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

Simple fact: Conclusion

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

Conclusion

Hypothesis

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

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

?x ?x

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

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

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

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

Simple fact: Conclusion

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

Conclusion

Hypothesis

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

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

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

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

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

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

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

Simple fact: Conclusion

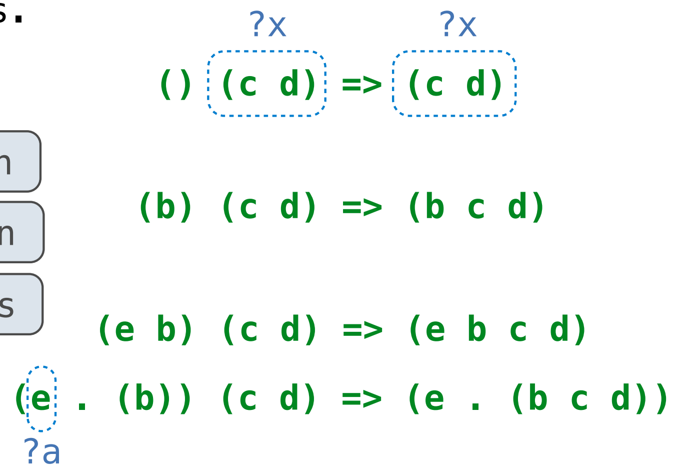
```
(fact (append-to-form (?a . ?r) ?y (?a . ?z))
      (append-to-form       ?r  ?y       ?z ))
```

Conclusion

Hypothesis

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

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



In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

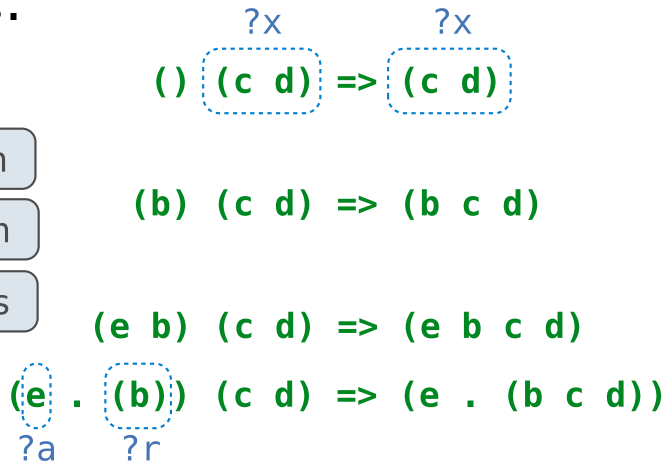
Simple fact: Conclusion

Conclusion

Hypothesis

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

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



?x ?x

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

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

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

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

?a ?r

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

Simple fact: Conclusion

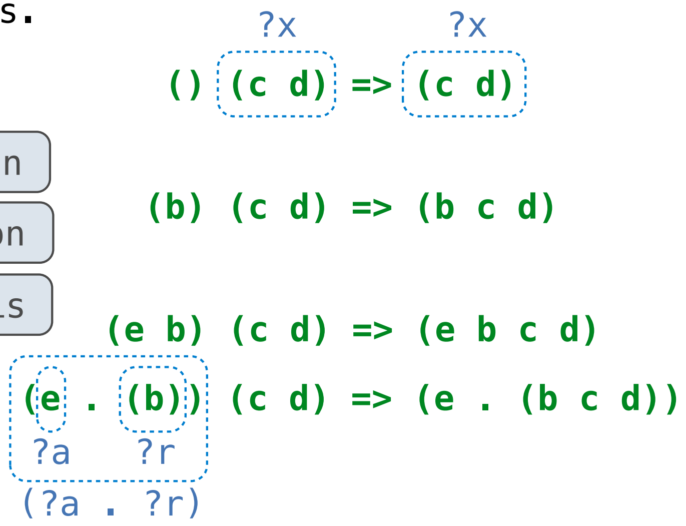
```
(fact (append-to-form (?a . ?r) ?y (?a . ?z))
      (append-to-form       ?r  ?y       ?z ))
```

Conclusion

Hypothesis

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

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



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

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

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

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

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

Simple fact: Conclusion

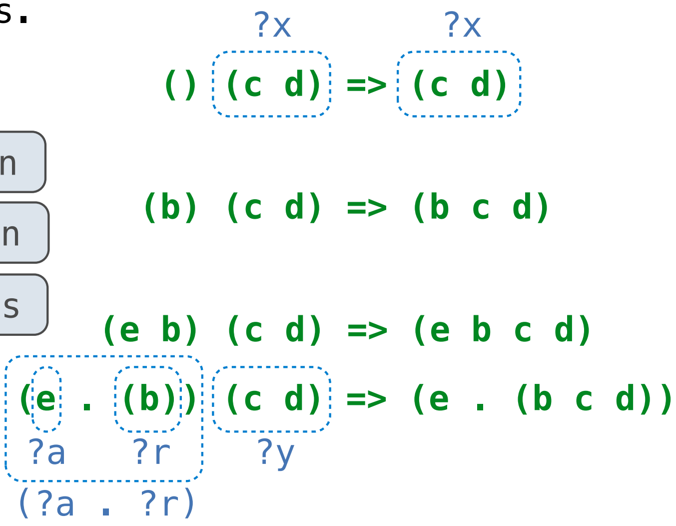
```
(fact (append-to-form (?a . ?r) ?y (?a . ?z))
      (append-to-form       ?r  ?y       ?z ))
```

Conclusion

Hypothesis

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

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



?x ?x

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

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

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

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

?a ?r ?y

(?a . ?r)

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

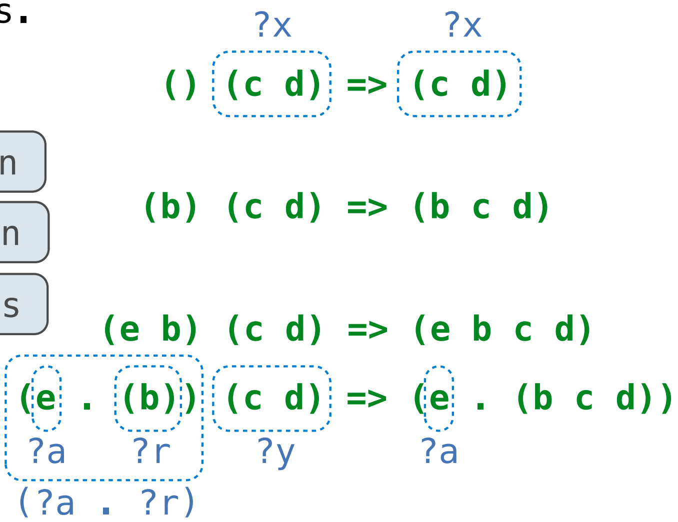
```
(fact (append-to-form () ?x ?x))
```

Simple fact: Conclusion

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

Conclusion

Hypothesis



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

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

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

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

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

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

Simple fact: Conclusion

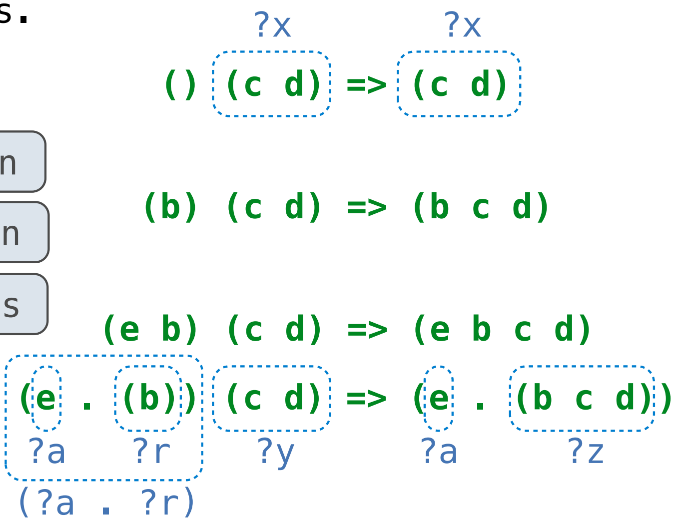
```
(fact (append-to-form (?a . ?r) ?y (?a . ?z))
      (append-to-form       ?r  ?y       ?z ))
```

Conclusion

Hypothesis

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

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



?x ?x

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

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

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

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

?a ?r ?y ?a ?z

(?a . ?r)

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

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

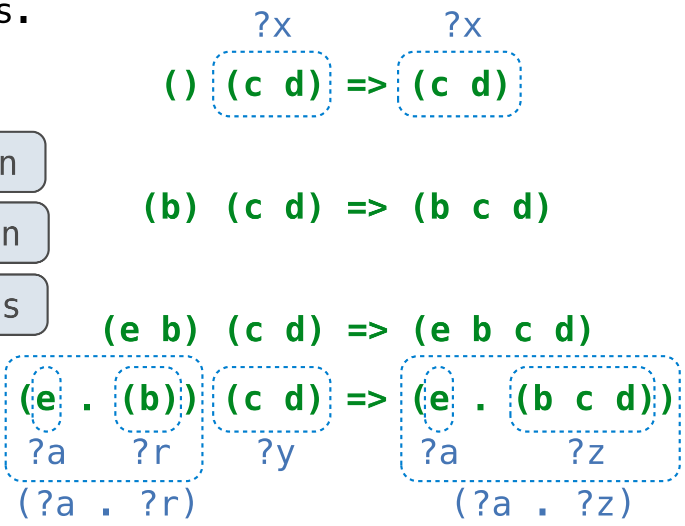
Simple fact: Conclusion

Conclusion

Hypothesis

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

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



In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

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

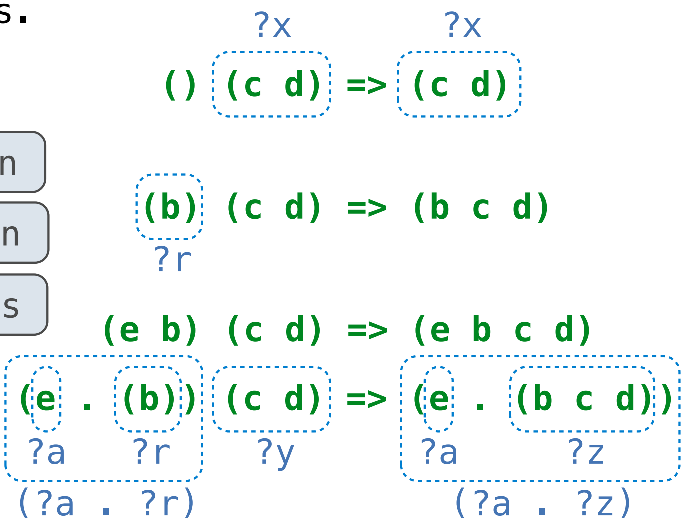
Simple fact: Conclusion

Conclusion

Hypothesis

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

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



?x ?x

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

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

?r

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

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

?a ?r ?y ?a ?z

(?a . ?r) (?a . ?z)

In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

Simple fact: Conclusion

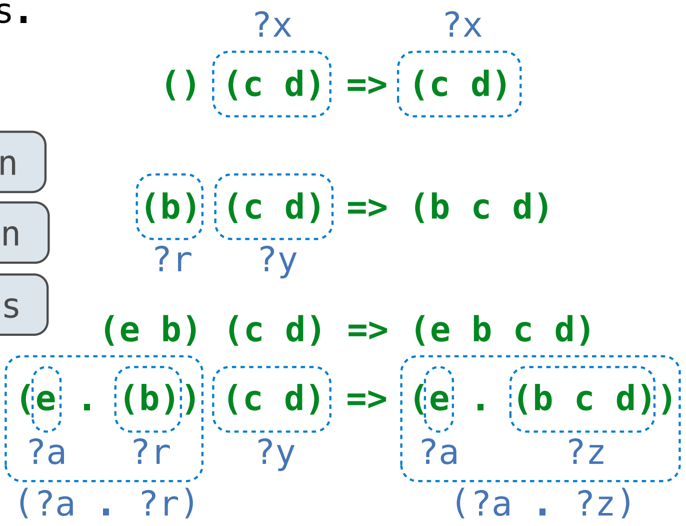
```
(fact (append-to-form (?a . ?r) ?y (?a . ?z))
      (append-to-form       ?r  ?y       ?z ))
```

Conclusion

Hypothesis

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

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



In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

Simple fact: Conclusion

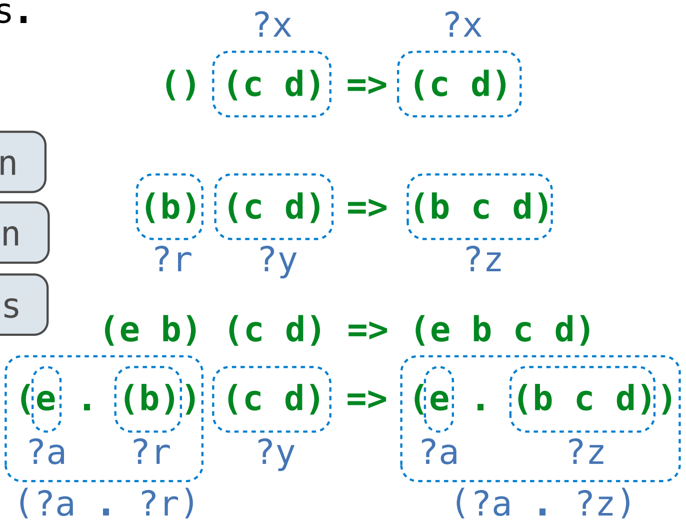
```
(fact (append-to-form (?a . ?r) ?y (?a . ?z))
      (append-to-form       ?r  ?y       ?z ))
```

Conclusion

Hypothesis

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

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



In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

Lists in Logic

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

Expressions and their relations are Scheme lists.

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

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

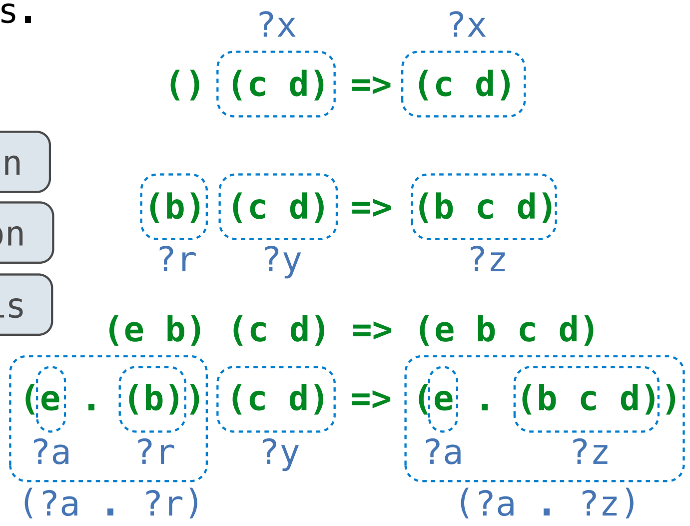
Simple fact: Conclusion

Conclusion

Hypothesis

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

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



In a **fact**, the first relation is the conclusion and the rest are hypotheses.

In a **query**, all relations must be satisfied.

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

(Demo)

Permuting Lists

Anagrams in Logic

Anagrams in Logic

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

Anagrams in Logic

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

- The empty list for an empty list.

Anagrams in Logic

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

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

Anagrams in Logic

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

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

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

Anagrams in Logic

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

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

Element

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

Anagrams in Logic

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

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

Element

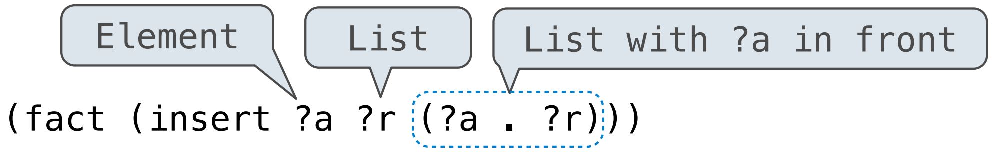
List

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

Anagrams in Logic

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

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



Element

List

List with ?a in front

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

Anagrams in Logic

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

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

Element

List

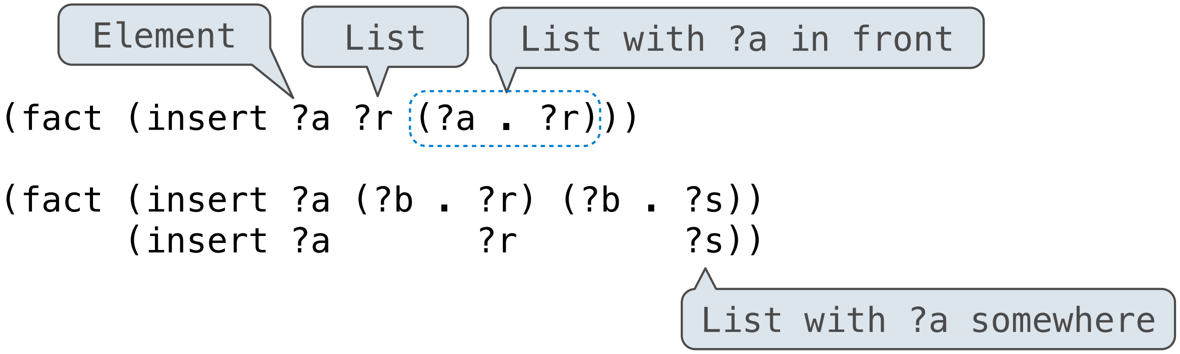
List with ?a in front

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

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

Anagrams in Logic

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

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

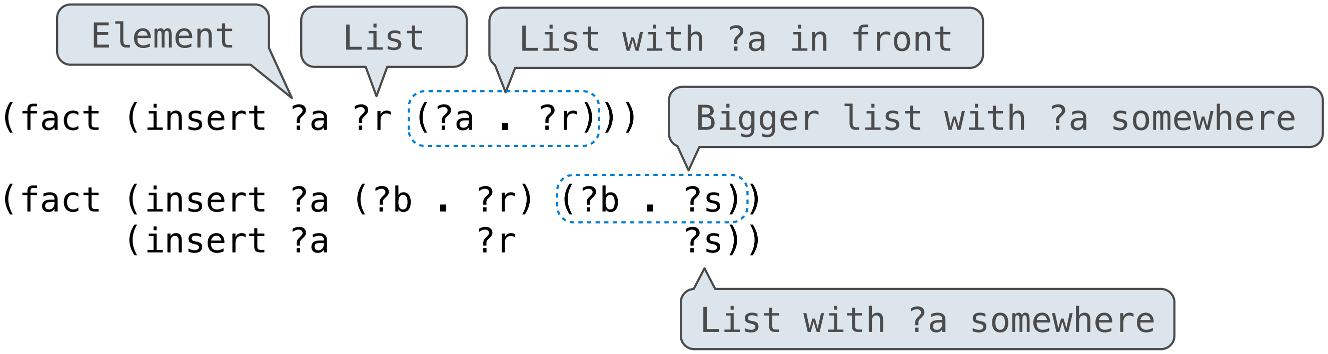


```
(fact (insert ?a ?r (?a . ?r)))

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

Anagrams in Logic

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

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



Element

List

List with ?a in front

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

Bigger list with ?a somewhere

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

List with ?a somewhere

Anagrams in Logic

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

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

Element

List

List with ?a in front

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

Bigger list with ?a somewhere

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

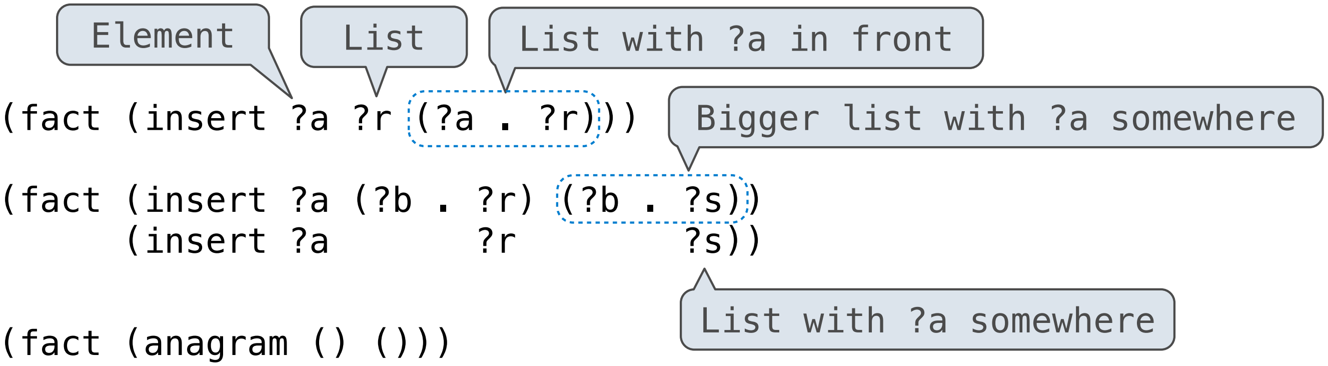
List with ?a somewhere

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

Anagrams in Logic

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

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

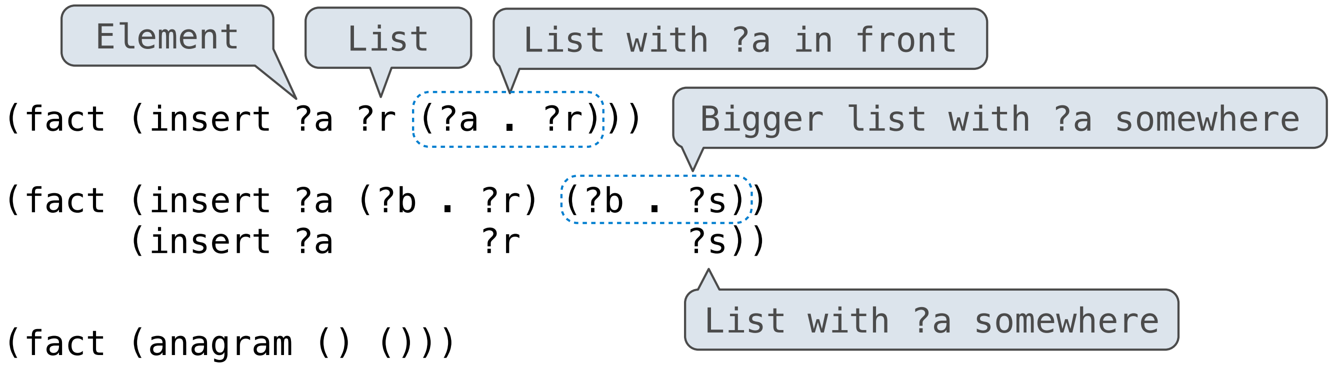


```
(fact (insert ?a ?r (?a . ?r)))

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


(fact (anagram () ()))

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

Anagrams in Logic

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

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



```
(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))
```

Anagrams in Logic

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

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

Element

List

List with ?a in front

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

Bigger list with ?a somewhere

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

List with ?a somewhere

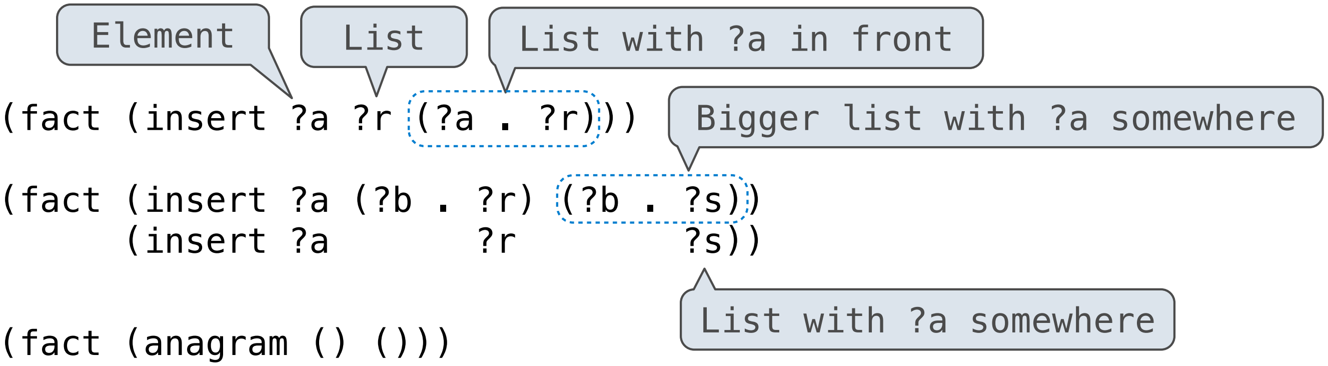
```
(fact (anagram () ()))

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

Anagrams in Logic

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

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

a r t



Element

List

List with ?a in front

Bigger list with ?a somewhere

List with ?a somewhere

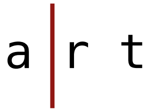
```
(fact (insert ?a ?r (?a . ?r)))

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

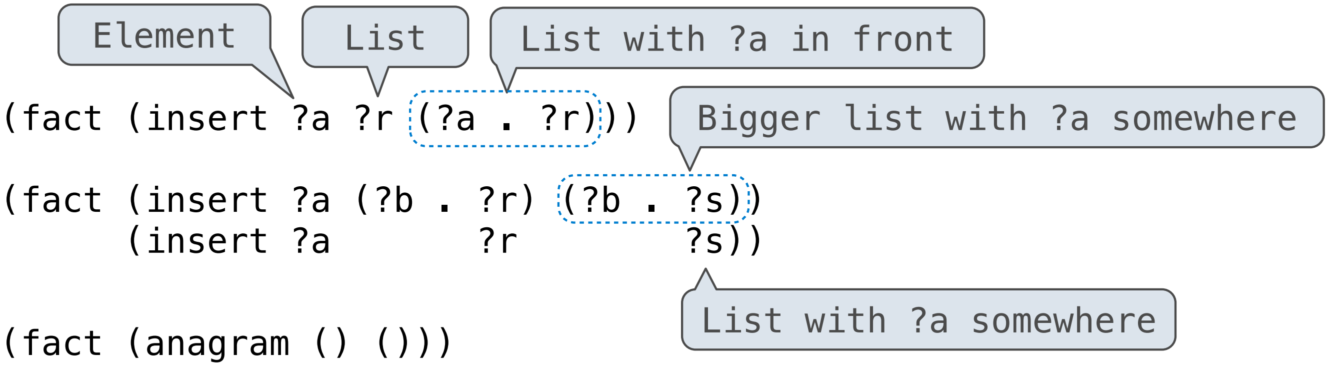
```
(fact (anagram () ()))

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

Anagrams in Logic

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

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





Element

List

List with ?a in front

Bigger list with ?a somewhere

List with ?a somewhere

```
(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))
```

Anagrams in Logic

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

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

Element

List

List with ?a in front

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

Bigger list with ?a somewhere

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

List with ?a somewhere

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

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

Anagrams in Logic

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

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

Element

List

List with ?a in front

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

Bigger list with ?a somewhere

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

List with ?a somewhere

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

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

a | r t

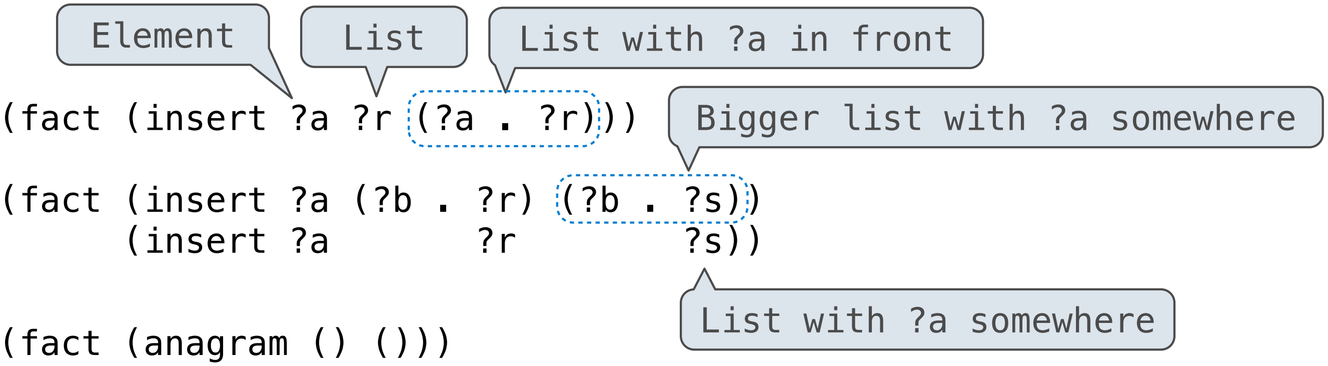
r t

ar t

Anagrams in Logic

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

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



Element

List

List with ?a in front

Bigger list with ?a somewhere

List with ?a somewhere

```
(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))
```

Anagrams in Logic

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

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

Element

List

List with ?a in front

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

Bigger list with ?a somewhere

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

List with ?a somewhere

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

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

a | r t

r t

ar t

r**a**t

r t**a**

Anagrams in Logic

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

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

Element

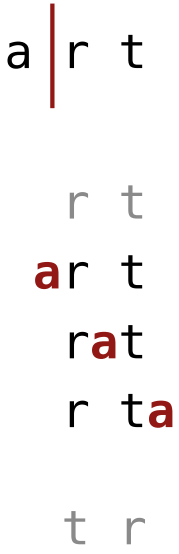
List

List with ?a in front

Bigger list with ?a somewhere

List with ?a somewhere

```
(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))
```



Anagrams in Logic

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

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

Element

List

List with ?a in front

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

Bigger list with ?a somewhere

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

List with ?a somewhere

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

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

a | r t

r t

ar t

r**a**t

r t**a**

t r

at r

Anagrams in Logic

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

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

Element

List

List with ?a in front

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

Bigger list with ?a somewhere

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

List with ?a somewhere

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

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

a | r t

r t

ar t

r**a**t

r t**a**

t r

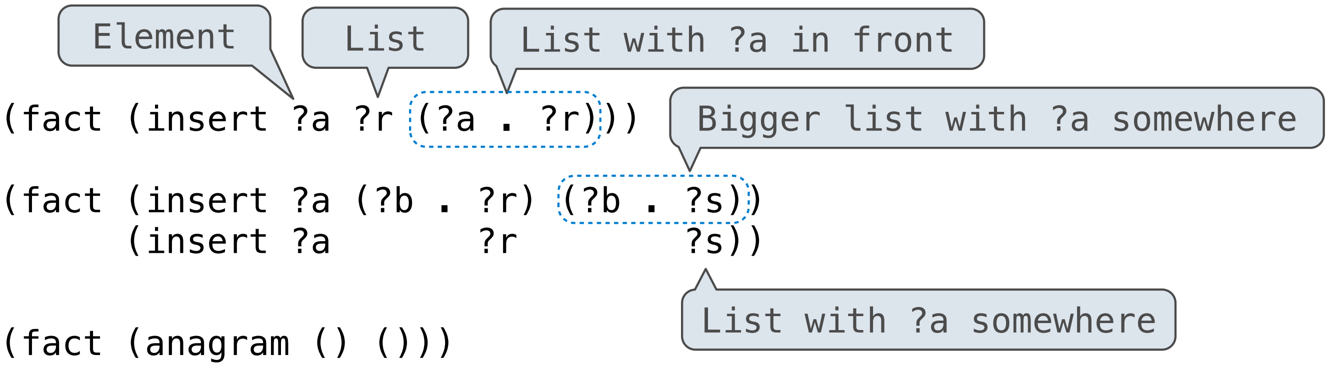
at r

t**a**r

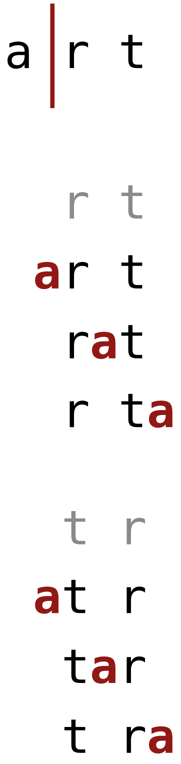
Anagrams in Logic

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

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



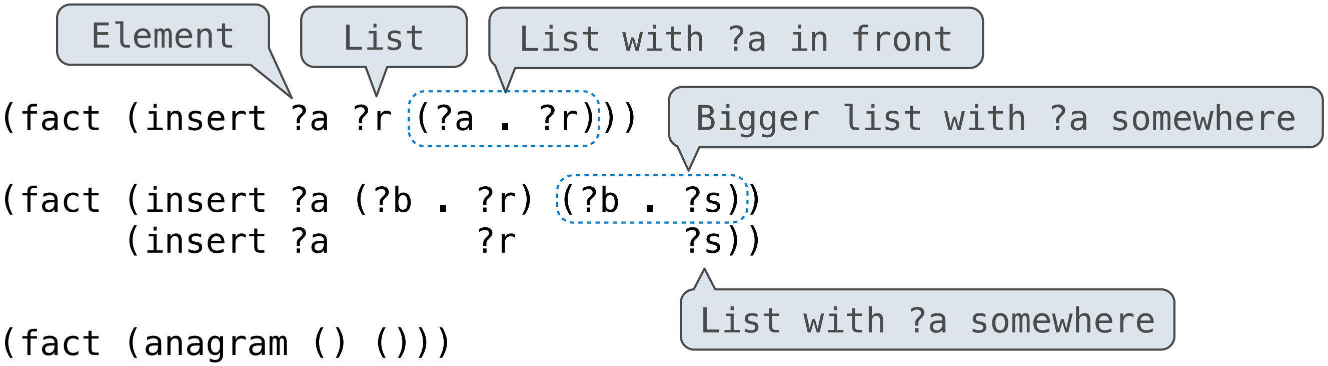
```
(fact (insert ?a ?r (?a . ?r)))

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


(fact (anagram () ()))

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



Anagrams in Logic

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

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



Element

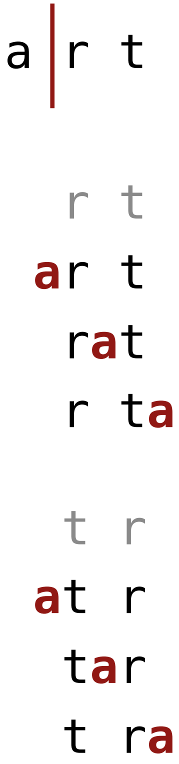
List

List with ?a in front

Bigger list with ?a somewhere

List with ?a somewhere

```
(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))
```



(Demo)

Unification

Pattern Matching

Pattern Matching

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

Pattern Matching

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

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

Pattern Matching

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

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

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

Pattern Matching

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

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

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

Pattern Matching

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

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

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


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

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


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


( (a  b) c  (a  b) )

(   ?x  ?x    ?x    )
```

Pattern Matching

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

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

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

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

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

Unification

Unification

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

Unification

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

1. Look up variables in the current environment.

Unification

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

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

Unification

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

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

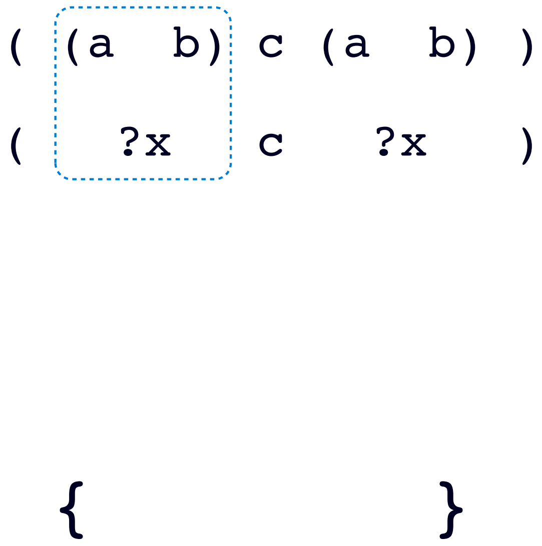
```
( (a  b) c (a  b) )

(   ?x   c   ?x   )
```

```
 {              }
```

Unification

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

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

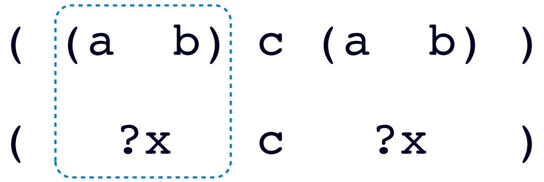


```
( (a  b) c (a  b) )

(   ?x   c   ?x    )


   {              }
```

Unification

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

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



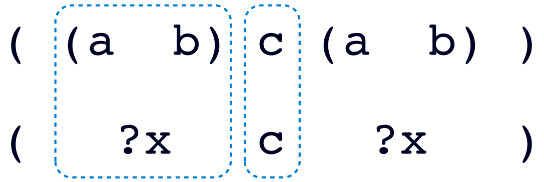
```
( (a   b) c (a   b) )

(   ?x    c    ?x    )
```

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

Unification

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

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



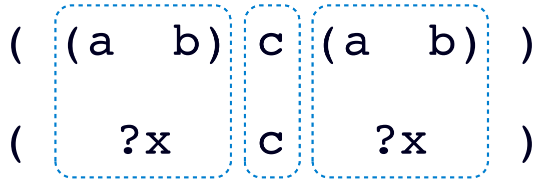
```
( (a   b) c (a   b) )

(   ?x    c   ?x    )
```

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

Unification

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

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



((a b) c (a b))

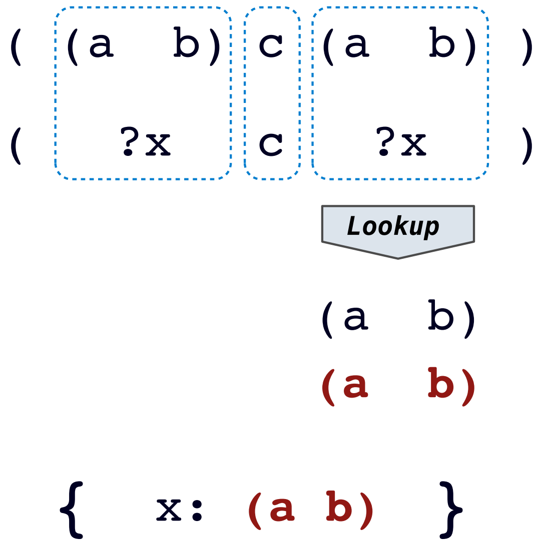
(?x c ?x)

{ x: (a b) }

Unification

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

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



((a b) c (a b))

(?x c ?x)

Lookup

(a b)

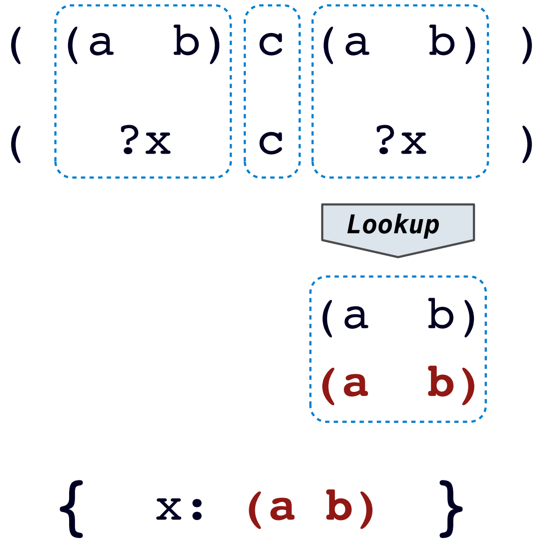
(a b)

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

Unification

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

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



((a b) c (a b))

(?x c ?x)

Lookup

(a b)

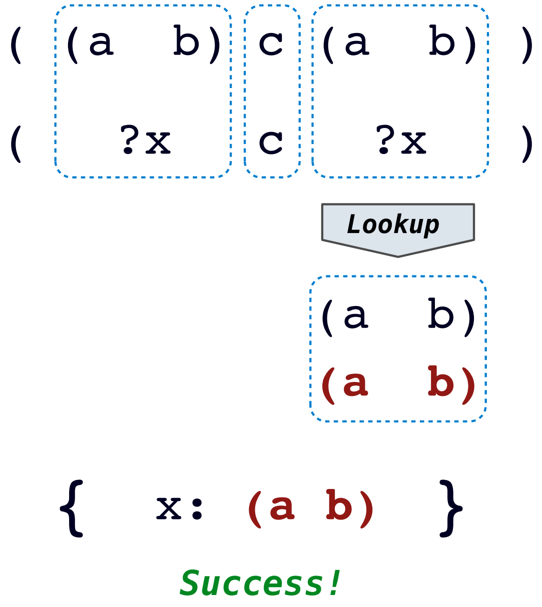
(a b)

{ x: (a b) }

Unification

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

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



((a b) c (a b))

(?x c ?x)

Lookup

(a b)

(a b)

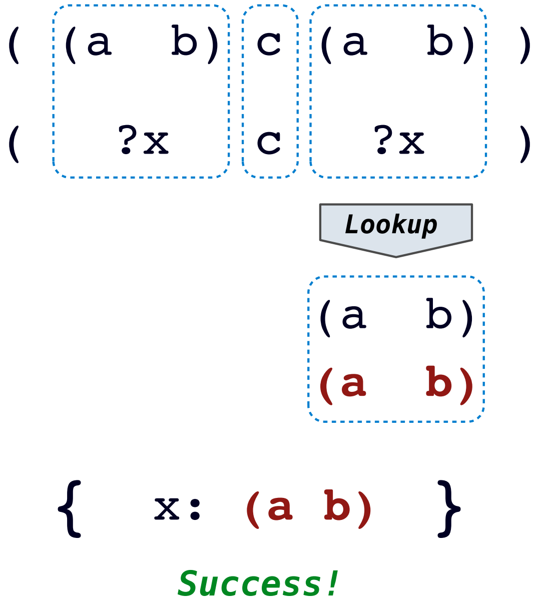
{ x: (a b) }

Success!

Unification

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

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



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

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

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

Success!

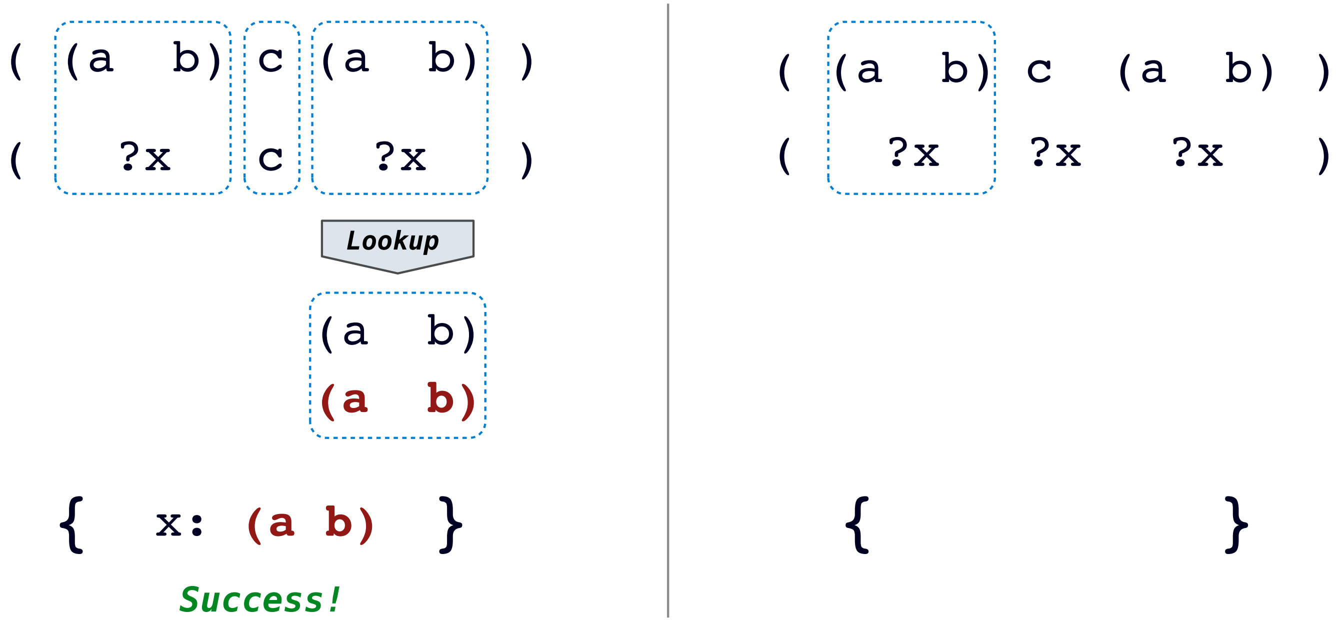
```
( (a  b) c (a  b) )
(   ?x   ?x   ?x  )
```

```
{             }
```

Unification

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

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



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

Lookup

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

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

Success!

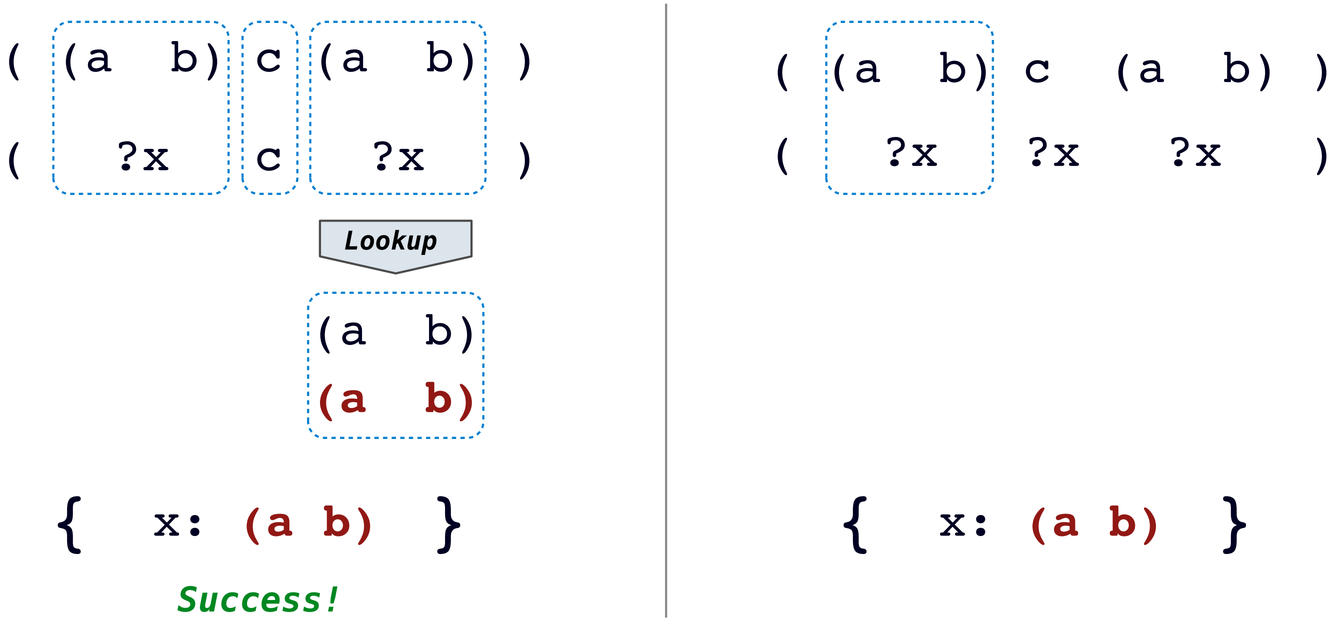
```
( (a   b) c   (a   b) )
(    ?x   ?x    ?x    )
```

```
{            }
```

Unification

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

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



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

Lookup

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

{ x: (a b) }

Success!

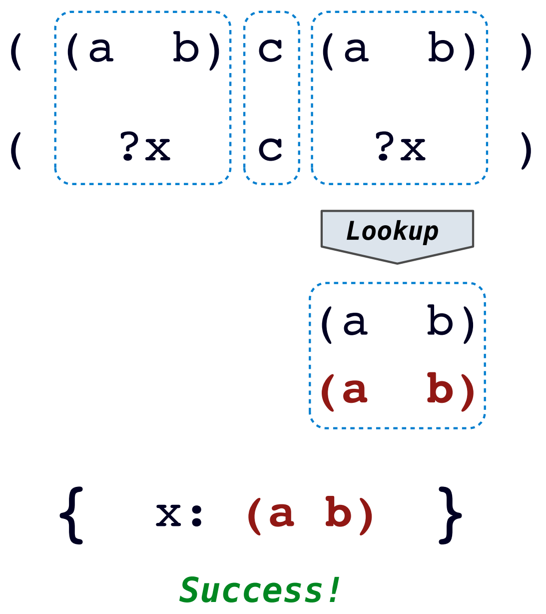
```
( (a  b) c  (a  b) )
(   ?x   ?x   ?x   )
```

{ x: (a b) }

Unification

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

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



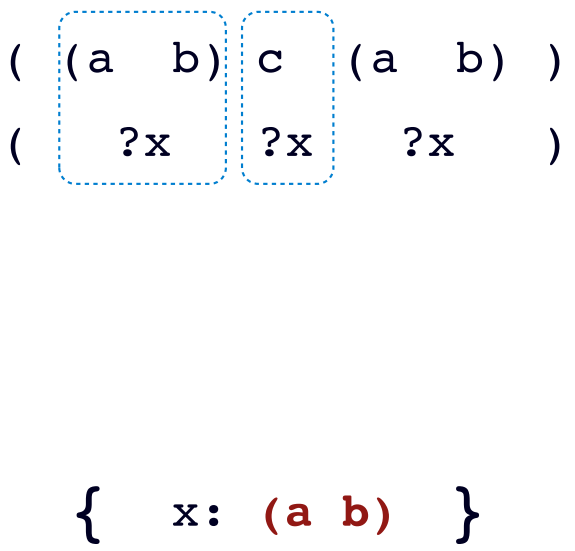
```
( (a   b)  c  (a   b)  )
(    ?x    c     ?x    )
```

Lookup

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

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

Success!



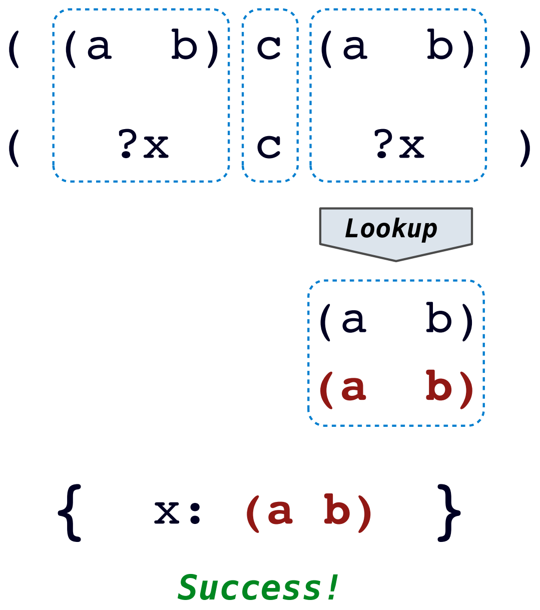
```
( (a   b)  c  (a   b)  )
(    ?x    ?x    ?x    )
```

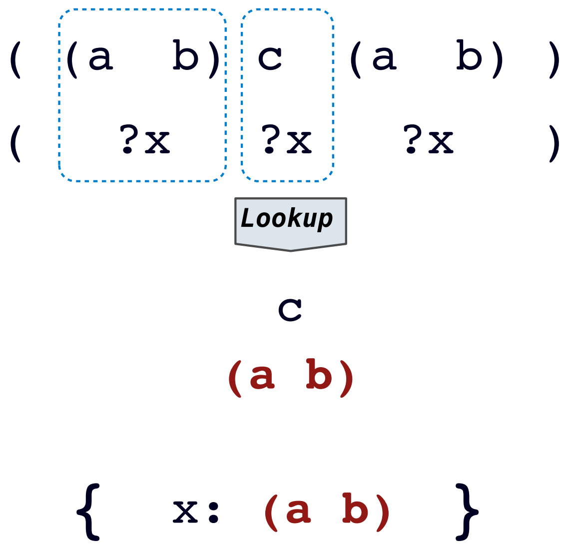
```
{   x:  (a b)   }
```

Unification

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

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

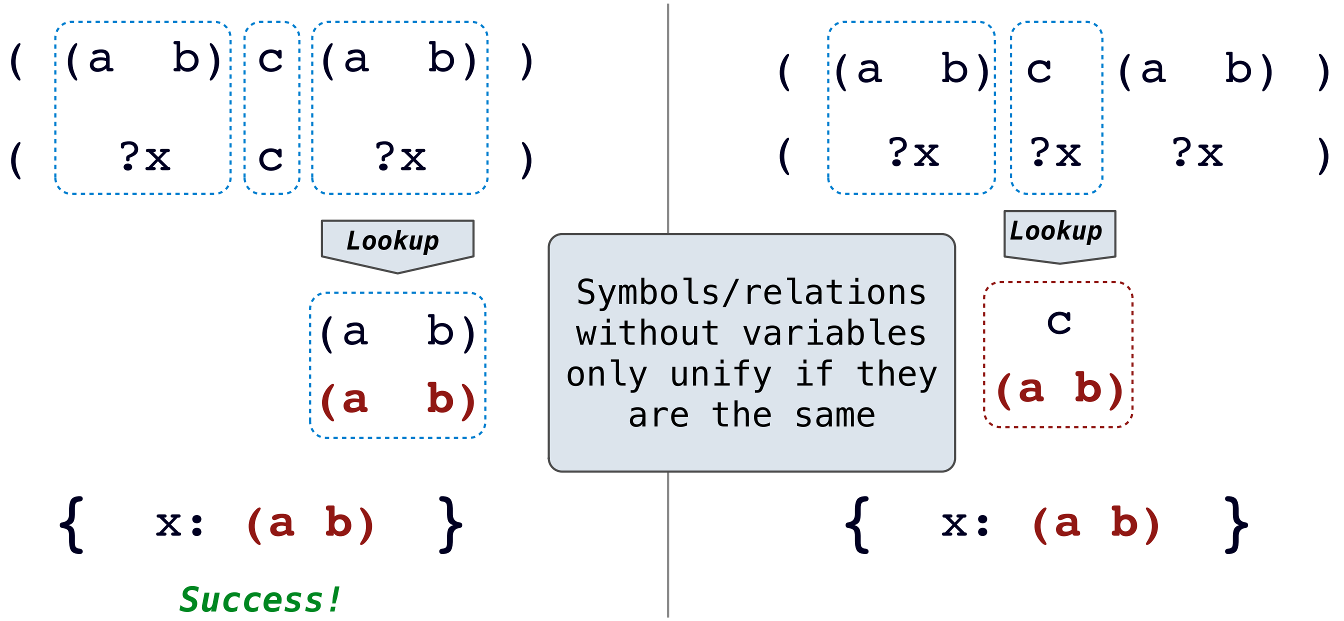




Unification

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

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



((a b) c (a b))

(?x c ?x)

Lookup

(a b)

(a b)

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

Success!

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

((a b) c (a b))

(?x ?x ?x)

Lookup

c

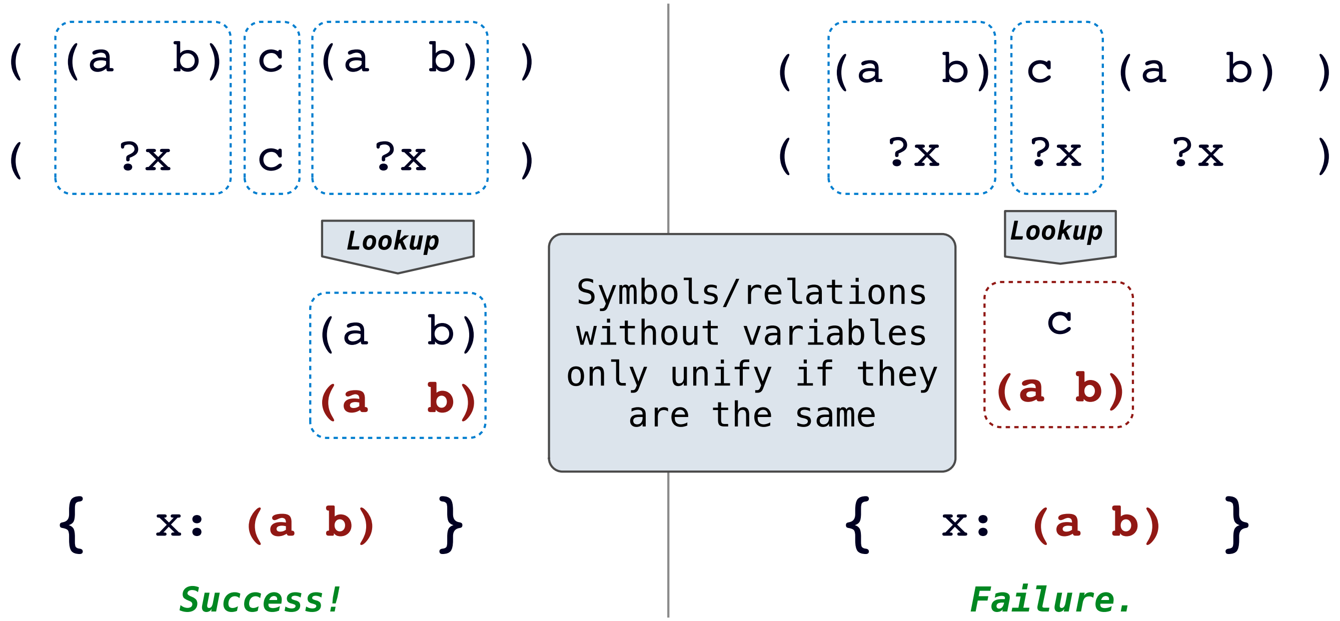
(a b)

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

Unification

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

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



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

Lookup

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

{ x: (a b) }

Success!

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

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

Lookup

```
  c
(a b)
```

{ x: (a b) }

Failure.

Unifying Variables

Unifying Variables

Two relations that contain variables can be unified as well.

Unifying Variables

Two relations that contain variables can be unified as well.

```
(    ?x        ?x    )

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

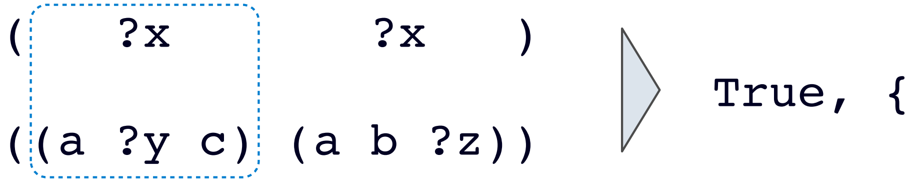
Unifying Variables

Two relations that contain variables can be unified as well.

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

Unifying Variables

Two relations that contain variables can be unified as well.

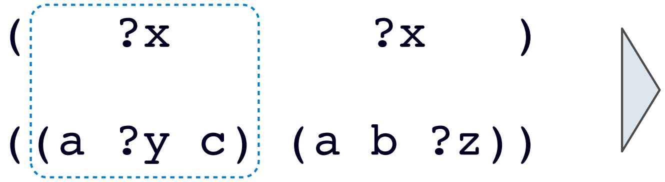


```
(    ?x           ?x     )

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

`True, {`

Unifying Variables

Two relations that contain variables can be unified as well.



```
(    ?x          ?x    )

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

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

Unifying Variables

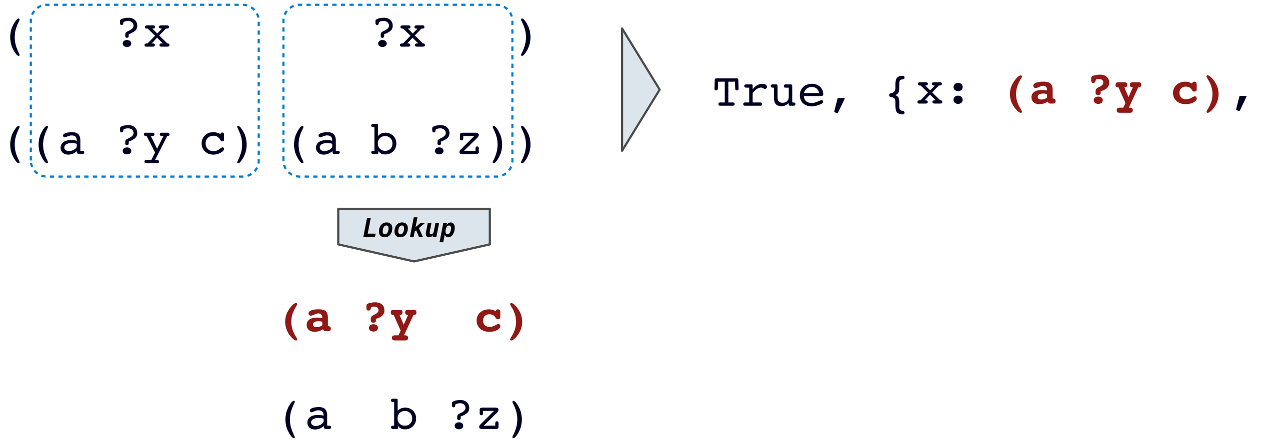
Two relations that contain variables can be unified as well.

```
(    ?x          ?x     )

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

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

Unifying Variables

Two relations that contain variables can be unified as well.



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

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

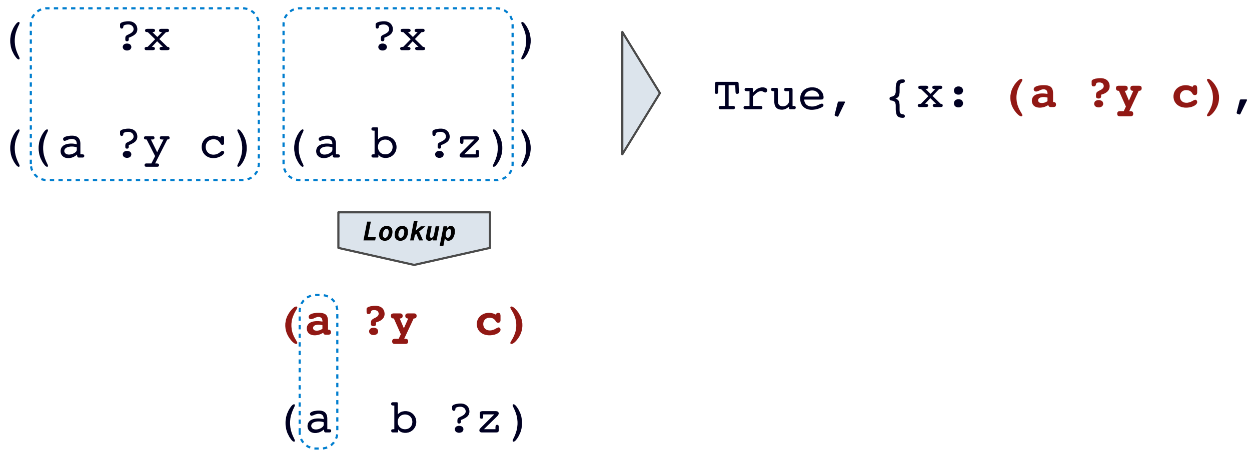
Lookup

(a ?y c)

(a b ?z)

Unifying Variables

Two relations that contain variables can be unified as well.



```
(    ?x          ?x     )

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

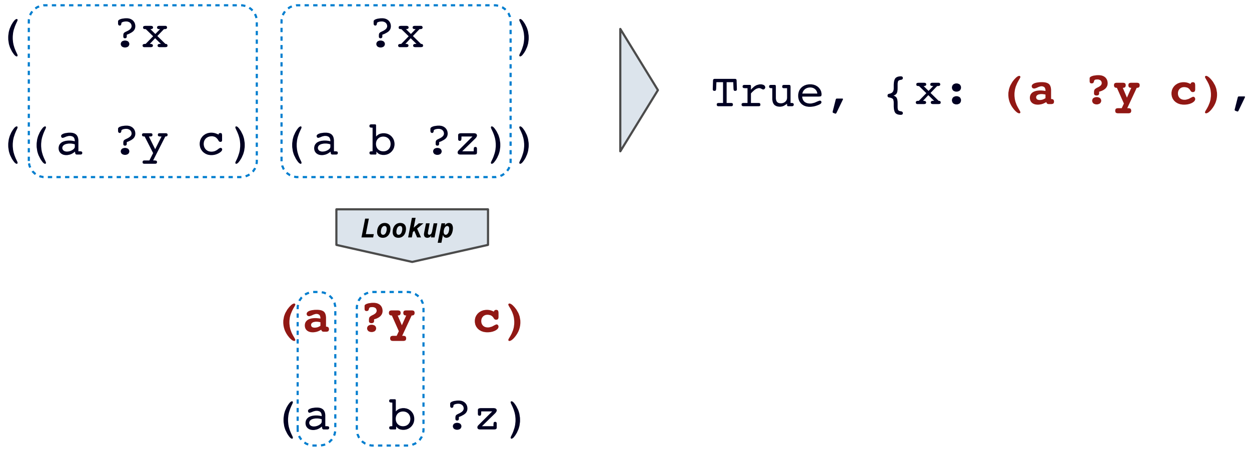
Lookup

```
(a ?y  c)

(a   b ?z)
```

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

Unifying Variables

Two relations that contain variables can be unified as well.



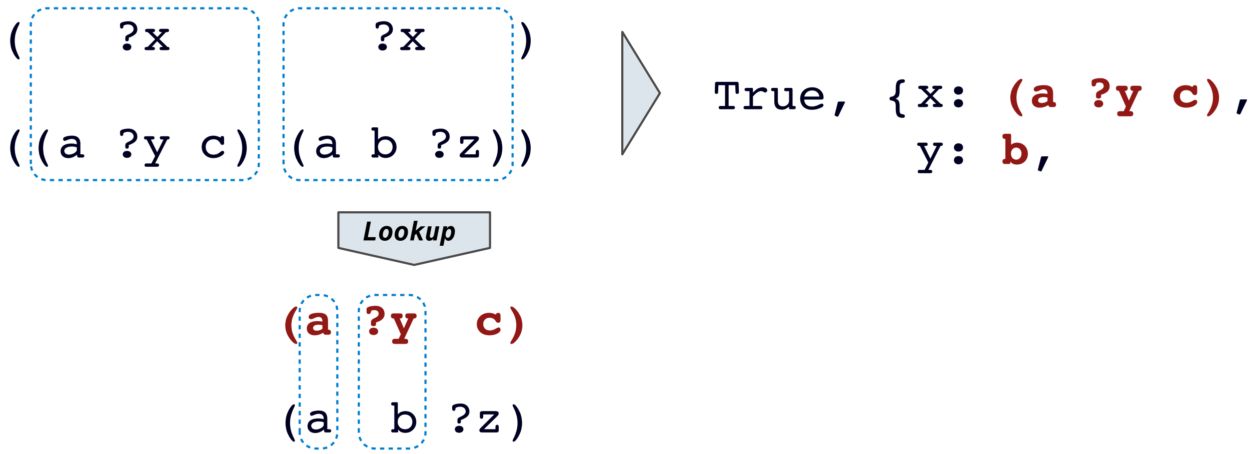
```
(      ?x           ?x     )
((a ?y c)    (a b ?z))
```

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

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

Unifying Variables

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



```
(    ?x          ?x      )

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

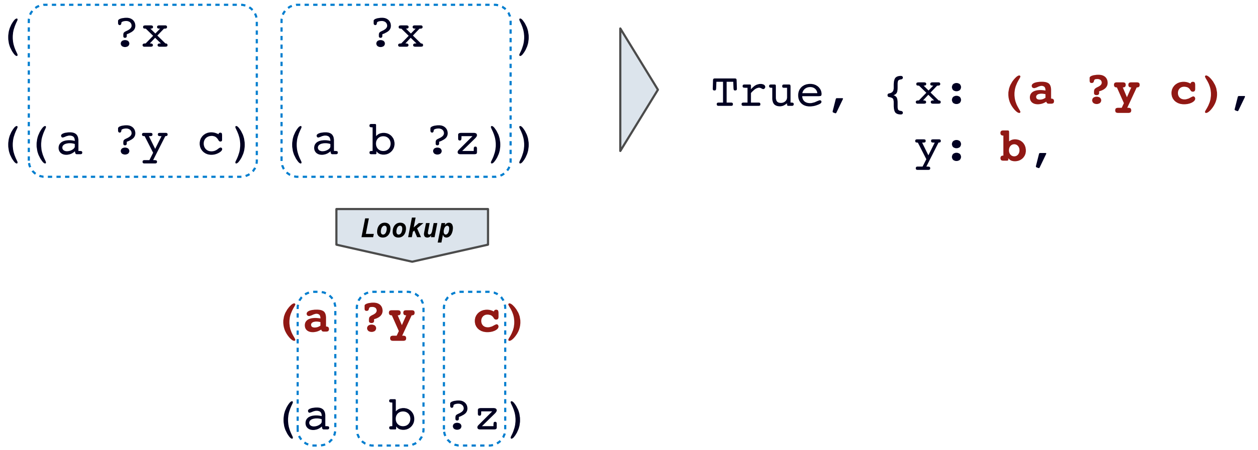
Lookup

```
(a ?y c)

(a  b ?z)
```

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

Unifying Variables

Two relations that contain variables can be unified as well.



```
(    ?x            ?x      )

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

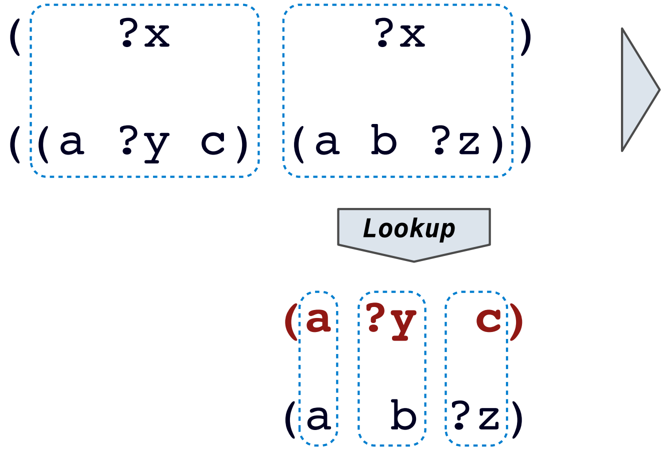
Lookup

```
(a  ?y   c)

(a   b  ?z)
```

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

Unifying Variables

Two relations that contain variables can be unified as well.



```
(   ?x          ?x      )

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

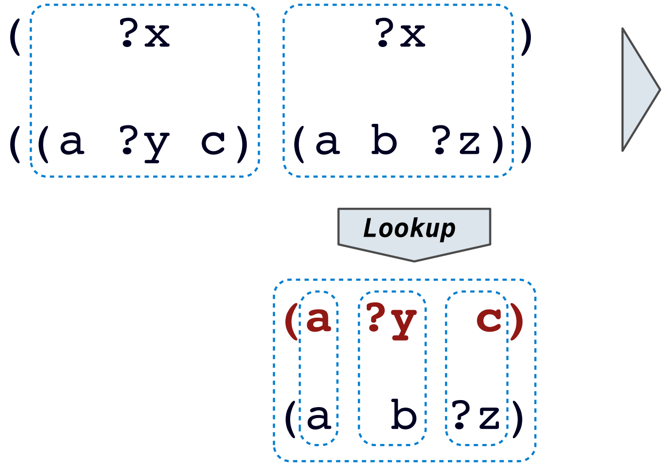
Lookup

```
(a ?y  c)

(a  b ?z)
```

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

Unifying Variables

Two relations that contain variables can be unified as well.



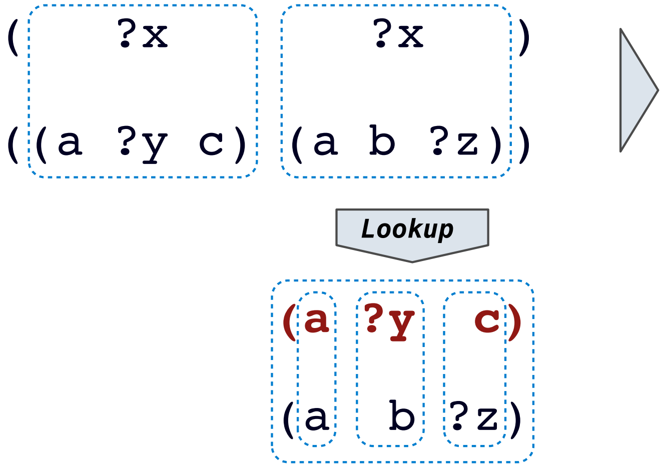
```
(    ?x         ?x    )
((a ?y c) (a b ?z))
```

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

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

Unifying Variables

Two relations that contain variables can be unified as well.



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

Lookup

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

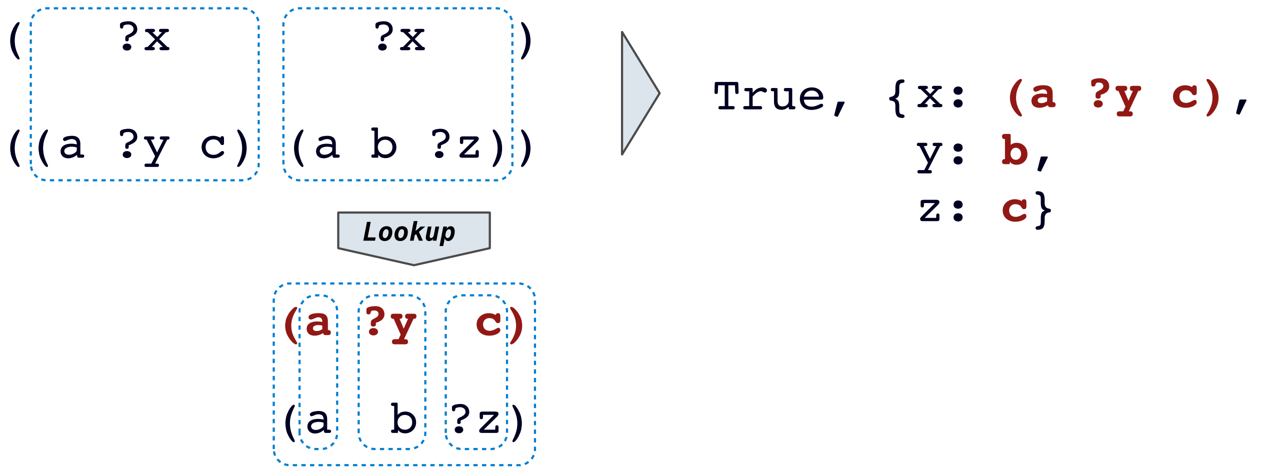
```
True, {x: (a ?y c),
       y: b,
       z: c}
```

Substituting values for variables may require multiple steps.

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

Unifying Variables

Two relations that contain variables can be unified as well.



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

Lookup

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

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

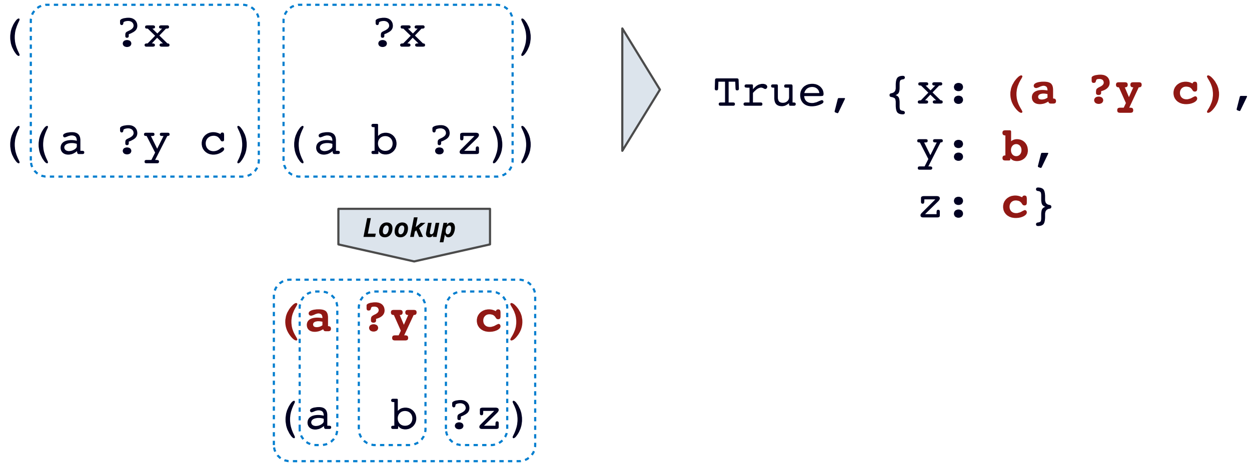
Substituting values for variables may require multiple steps.

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

`lookup('?x')`

Unifying Variables

Two relations that contain variables can be unified as well.



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

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

Lookup

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

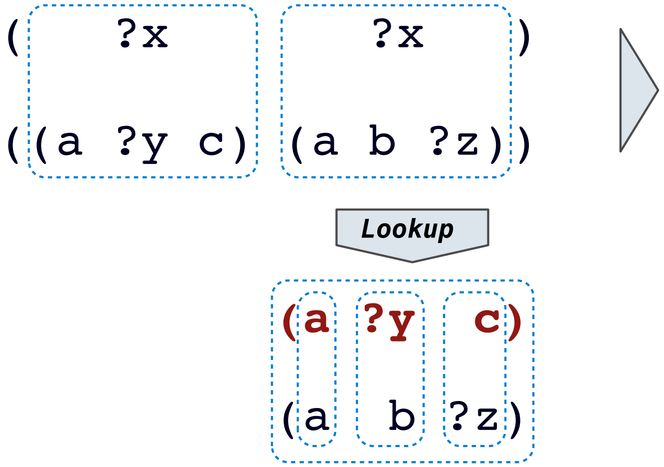
Substituting values for variables may require multiple steps.

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

lookup('?x') ⇨ **(a ?y c)**

Unifying Variables

Two relations that contain variables can be unified as well.



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

Lookup

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

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

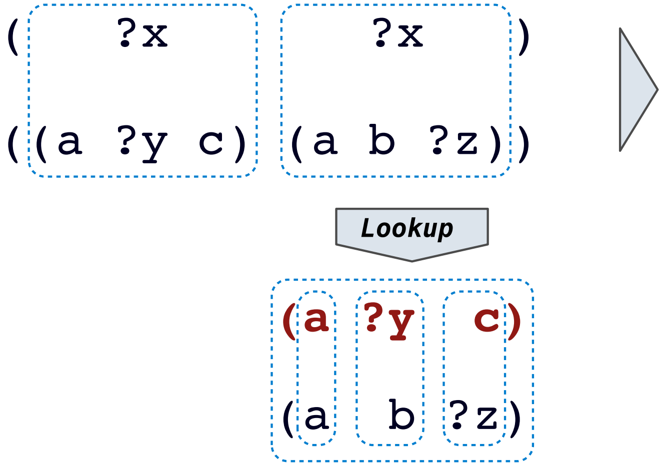
Substituting values for variables may require multiple steps.

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

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

Unifying Variables

Two relations that contain variables can be unified as well.



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

Lookup

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

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

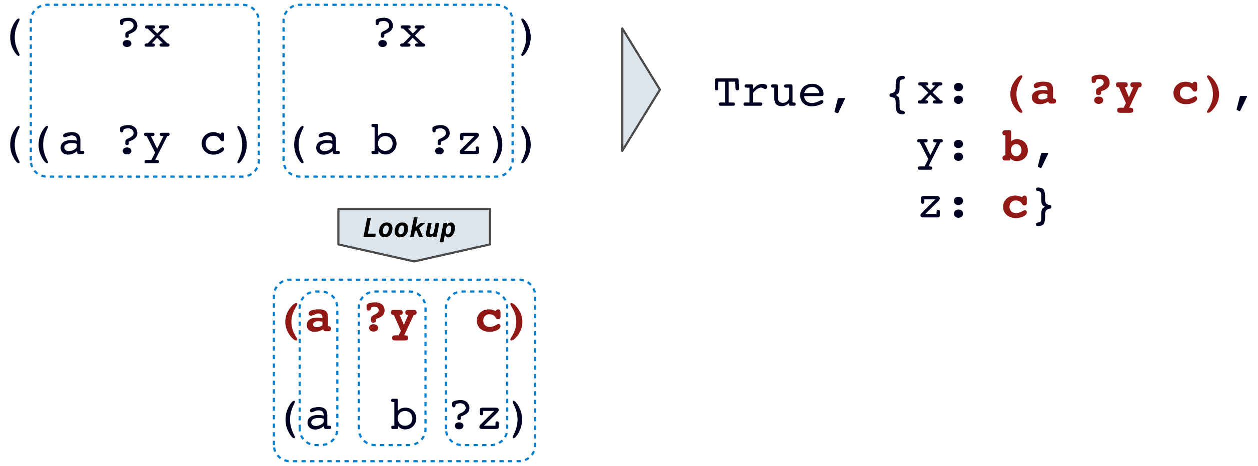
Substituting values for variables may require multiple steps.

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

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

Unifying Variables

Two relations that contain variables can be unified as well.



```
(   ?x        ?x     )

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

Lookup

```
(a ?y  c)

(a  b ?z)
```

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

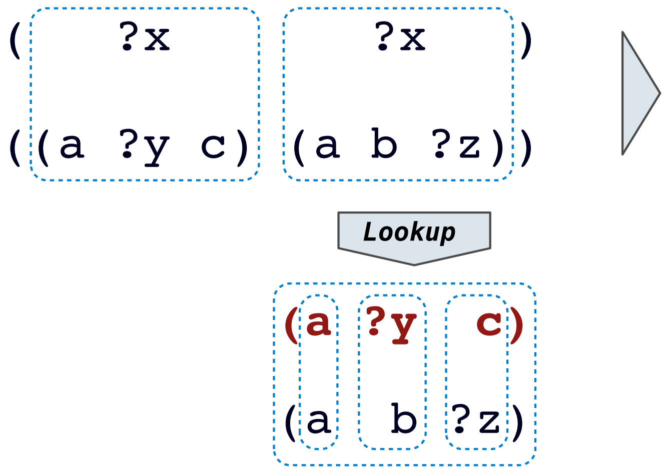
Substituting values for variables may require multiple steps.

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

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

Unifying Variables

Two relations that contain variables can be unified as well.



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

Substituting values for variables may require multiple steps.

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

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

Implementing Unification

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

Implementing Unification

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

1. Look up variables in the current environment

Implementing Unification

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

1. Look up variables in the current environment

2. Establish new bindings to unify elements.

Implementing Unification

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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.

Implementing Unification

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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

Implementing Unification

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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

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

(    ?x    c    ?x    )
```

Implementing Unification

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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

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

env: { }

Implementing Unification

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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

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

env: { }

Implementing Unification

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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

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

env: { }

Implementing Unification

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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

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

env: { x: (a b) }

Implementing Unification

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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.

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

env: { x: **(a b)** }

Implementing Unification

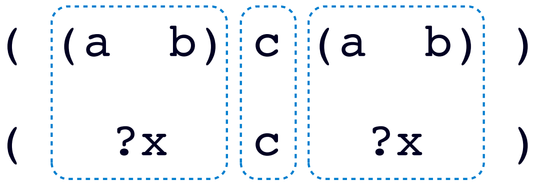
```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.



((a b) c (a b))

(?x c ?x)

env: { x: **(a b)** }

Implementing Unification

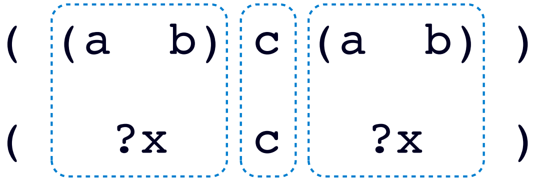
```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.



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

`( ?x c ?x )`

env: { x: **(a b)** }

Implementing Unification

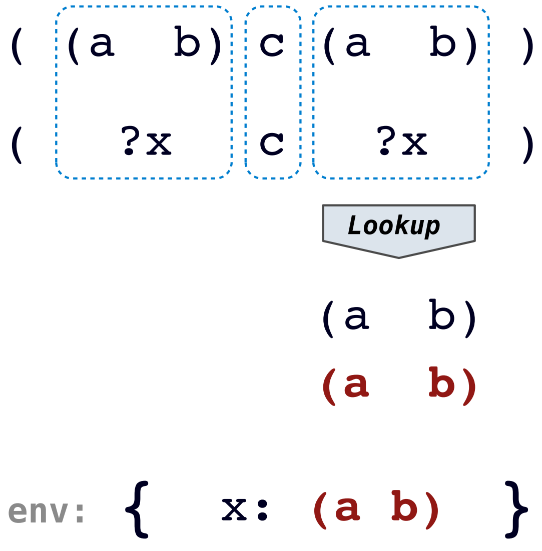
```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.



((a b) c (a b))

(?x c ?x)

Lookup

(a b)

(a b)

env: { x: (a b) }

Implementing Unification

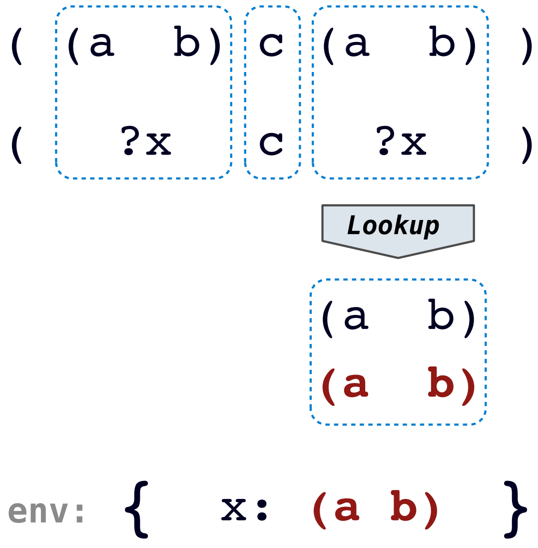
```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.

Recursively unify the first and rest of any lists.



Search

Searching for Proofs

Searching for Proofs

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

Searching for Proofs

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

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

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

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

Searching for Proofs

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

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

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

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

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

Searching for Proofs

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

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

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

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

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


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

Searching for Proofs

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

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

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

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

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

Searching for Proofs

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

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

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

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

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

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

Searching for Proofs

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

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

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

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

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

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

Searching for Proofs

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

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

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

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

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


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

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

Searching for Proofs

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

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

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

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

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

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

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

Variables are local to facts & queries

Searching for Proofs

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

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

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

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

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

Variables are local to facts & queries

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

Searching for Proofs

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

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

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

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

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

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

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

conclusion <- hypothesis

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

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

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

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

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

Variables are local to facts & queries

Searching for Proofs

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

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

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

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

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

Variables are local to facts & queries

Searching for Proofs

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

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

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

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

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

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

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

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

Variables are local to facts & queries

Searching for Proofs

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

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

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

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

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

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

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

Variables are local to facts & queries

Searching for Proofs

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

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

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

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

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

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

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

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

Variables are local to facts & queries

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

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

Searching for Proofs

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

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

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

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

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

Variables are local to facts & queries

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

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

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

?left:

Searching for Proofs

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

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

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

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

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

Variables are local to facts & queries

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

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

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

?left:

Searching for Proofs

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

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

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

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

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

Variables are local to facts & queries

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

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

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

?left:

Searching for Proofs

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

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

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

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

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

Variables are local to facts & queries

?left: **(e .**

Searching for Proofs

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

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

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

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

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

Variables are local to facts & queries

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

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

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

?left: **(e .**

?r:

Searching for Proofs

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

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

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

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

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

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

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

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

Variables are local to facts & queries

?left: **(e .**

?r:

Searching for Proofs

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

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

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

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

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

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

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

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

Variables are local to facts & queries

?left: **(e .**

?r:

Searching for Proofs

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

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

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

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

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

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

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

Variables are local to facts & queries

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

?left: **(e .**

?r: **(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 & queries

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

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

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

?left: **(e .**

?r: **(b .**

Searching for Proofs

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

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

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

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

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

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

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

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

Variables are local to facts & queries

?left: **(e .**

?r: **(b . ())**

Searching for Proofs

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

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

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

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

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

Variables are local to facts & queries

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

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

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

?left: **(e .**

?r: **(b . ())** ⇨ **(b)**

Searching for Proofs

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

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

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

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

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

Variables are local to facts & queries

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

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

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

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

?r: **(b . ())** ⇨ **(b)**

Searching for Proofs

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

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

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

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

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

Variables are local to facts & queries

?left: **(e . (b))** ⇨ **(e b)**

?r: **(b . ())** ⇨ **(b)**

Depth-First Search

Depth-First Search

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

Depth-First Search

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

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

Depth-First Search

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

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

```
def search(clauses, env):
```

Depth-First Search

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

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

```
def search(clauses, env):
  for fact in facts:
```

Depth-First Search

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

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

```
def search(clauses, env):
  for fact in facts:
    env_head = an environment extending env
```

Depth-First Search

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

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

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

Depth-First Search

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

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

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

Environment now contains new unifying bindings

Depth-First Search

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

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

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

Environment now contains new unifying bindings

Depth-First Search

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

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

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

Environment now contains new unifying bindings

Depth-First Search

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

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

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

Environment now contains new unifying bindings

Depth-First Search

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

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

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

Environment now contains new unifying bindings

- Limiting depth of the search avoids infinite loops.

Depth-First Search

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

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

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

Environment now contains new unifying bindings

- Limiting depth of the search avoids infinite loops.
- Each time a fact is used, its variables are renamed.

Depth-First Search

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

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

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

Environment now contains new unifying bindings

- Limiting depth of the search avoids infinite loops.
- Each time a fact is used, its variables are renamed.
- Bindings are stored in separate frames to allow backtracking.

Depth-First Search

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

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

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

Environment now contains new unifying bindings

- Limiting depth of the search avoids infinite loops.
- Each time a fact is used, its variables are renamed.
- Bindings are stored in separate frames to allow backtracking.

(Demo)

Addition

`(Demo)`