

# 61A Lecture 34

---

Monday, November 19

# Logic Language Review

---

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

Expressions and their relations are Scheme lists.

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

Simple fact

(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)

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

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

## 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

**a** r t

r **a** t

r t **a**

t r

**a** t r

t **a** r

t r **a**


Demo


## 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 )		

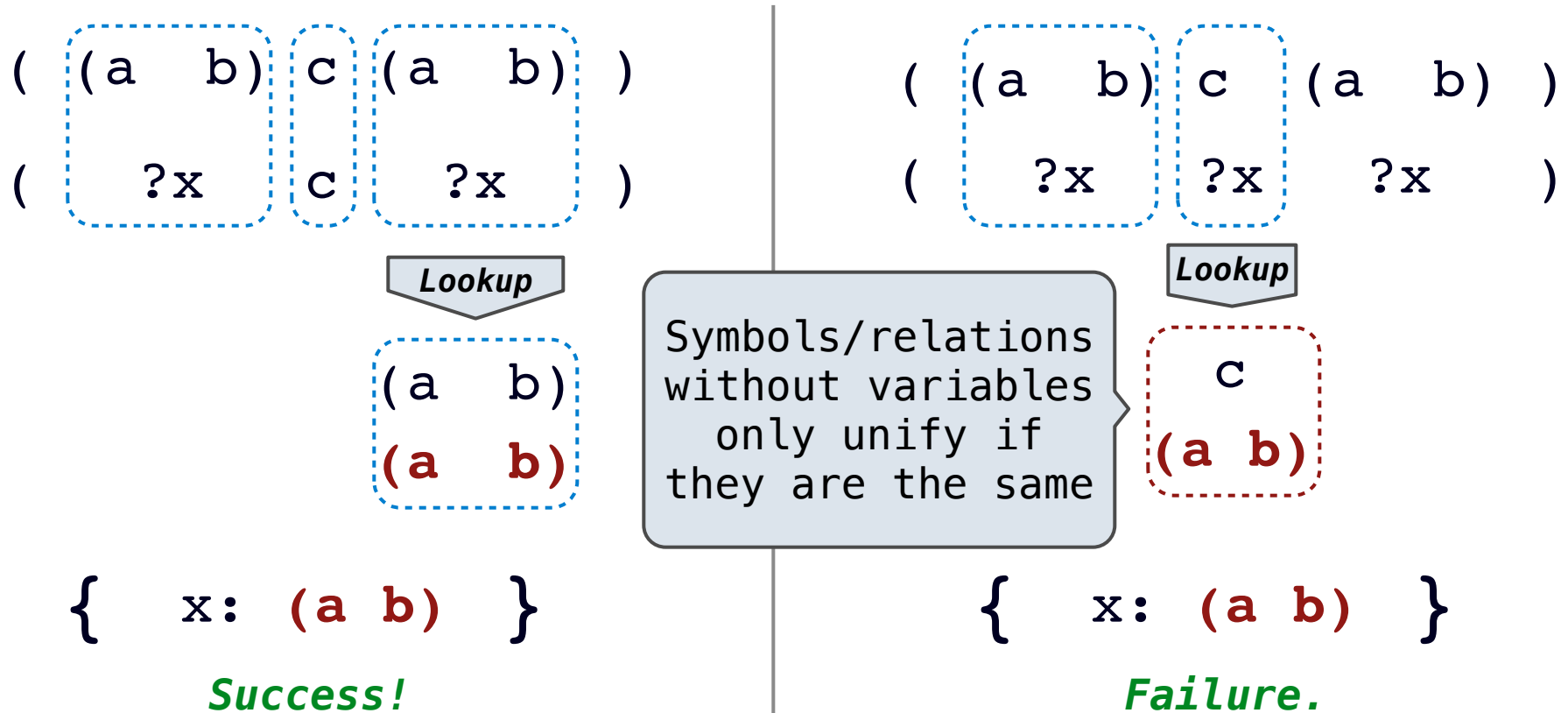
( (a b) c (a b) )		True, {y: b, z: c}
( (a ?y) ?z (a b) )		

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

# Unification

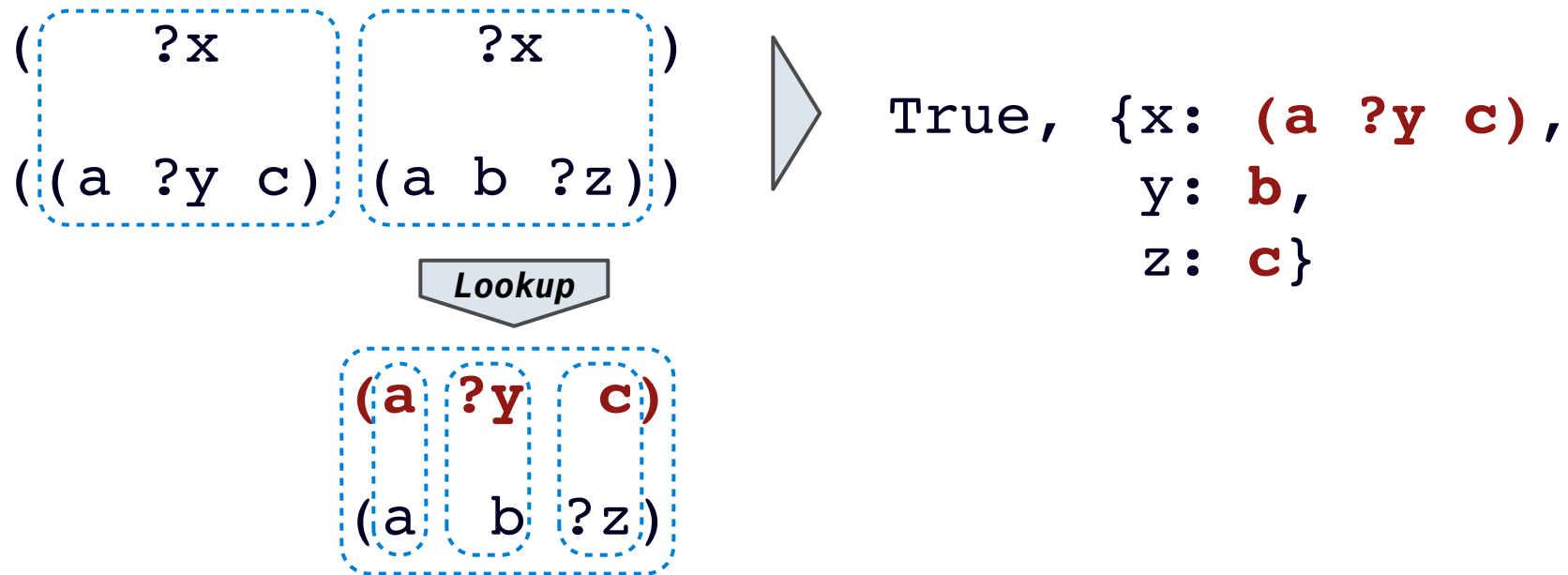
Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

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



## Unification with Two Variables

Two relations that contain variables can be unified as well.



Substituting values for variables may require multiple steps.

$\text{lookup}(' ?x ') \Rightarrow (a \ ?y \ c)$        $\text{lookup}(' ?y ') \Rightarrow b$

# 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.

Unification recursively unifies each pair of corresponding elements

# Searching for Proofs

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

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

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

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

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

```
conclusion <- hypothesis
```

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

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

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

```
conclusion <- hypothesis
```

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

```
{r2: (), x: (c d)}
```

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

Variables are local  
to facts & queries

left: **(e . (b . ()))**  $\Rightarrow$  **(e b)**



## Depth-First Search

---

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

Depth-first search: A possible proof approach is explored exhaustively before another one is considered.

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

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

# Implementing Depth-First 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