Logic Language Review

Expressions begin with *query* or *fact* followed by relations.
Logic Language Review

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

Expressions and their relations are Scheme lists.
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

(fact (append-to-form () ?x ?x))
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

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

*Simple fact*
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 ))
```
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))  Conclusion
  (append-to-form       ?r  ?y       ?z ))
```
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

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

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

\text{Hypothesis}
Logic Language Review

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

Expressions and their relations are Scheme lists.

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

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

\textit{(query} \ (\textit{append-to-form} \ ?left \ (c \ d) \ (e \ b \ c \ d))))
\textit{Success!}
\textit{left:} \ (e \ b)
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

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

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

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

If a query has more than one relation, all must be satisfied.
Expressions begin with *query* or *fact* followed by relations.

Expressions and their relations are Scheme lists.

(\texttt{fact (append-to-form (\() ?x \ ?x))\textbf{Simple fact}}

(\texttt{fact (append-to-form (\(?a \ . \ ?r) \ ?y (\(?a \ . \ ?z))\textbf{Conclusion}}
\texttt{(append-to-form \ ?r \ ?y \ ?z \))\textbf{Hypothesis}}

(\texttt{query (append-to-form \?left (c d) (e b c d)))\textbf{Success}}
\texttt{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
Logic Example: Anagrams

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

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

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

a r t
Logic Example: Anagrams

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

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

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

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

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

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

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

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

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

- The empty list for an empty list.
- The first element of the list inserted into an anagram of the rest of the list.
Logic Example: Anagrams

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

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

```
<table>
<thead>
<tr>
<th>a</th>
<th>r</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>t</td>
<td>r</td>
</tr>
<tr>
<td>r</td>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>t</td>
<td>r</td>
<td>a</td>
</tr>
</tbody>
</table>
```
Logic Example: Anagrams

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

(fact (insert ?a ?r (?a . ?r)))
Logic Example: Anagrams

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

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

(fact (insert ?a ?r (?a . ?r)))
Logic Example: Anagrams

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

(fact (insert ?a ?r (?a . ?r)))
Logic Example: Anagrams

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

(Number)

(fact (insert ?a ?r (?a . ?r)))
Logic Example: Anagrams

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

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

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

\[
\text{Element} \quad \text{List} \quad \text{List with element}
\]

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

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

\[
\text{(fact \ (anagram \ () \ ())))}
\]
Logic Example: Anagrams

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

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

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

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

(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))
Logic Example: Anagrams

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

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

\[
\text{Element} \quad \text{List} \quad \text{List with element}
\]

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

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

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

Demo
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.

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

\[
( \ ?x \ c \ \ ?x \ )
\]
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.

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

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

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

The basic operation of the Logic interpreter is to attempt to \textit{unify} two relations.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

\[
\begin{align*}
&\text{( (a b) c (a b) )} \\
&\text{(?x c ?x )} \\
&\{ \ x: (a b) \ \}
\end{align*}
\]
Unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment.

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

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

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

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

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

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

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

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

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

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

Lookup
Unification

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

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

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

\[
\begin{align*}
\text{Lookup} \\
\left( \begin{array}{c}
(a\ b) \\
(a\ b)
\end{array} \right)
\end{align*}
\]

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

Success!
Unification

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

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

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

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

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

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

\[ (\text{(a b)} \ c \ \text{(a b)}) \]
\[ (\ ?x \ c \ ?x \ ) \]

Success!
Unification

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

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

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

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

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

Success!
Unification

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

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

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

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

Success!
Unification

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

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

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

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

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

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

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

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

```
{ 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.

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

{ \( x : (a b) \) } \text{ Success!} \\
\{ \( x : (a b) \) \} \text{ Failure.}
Unification with Two Variables
Unification with Two Variables

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

Two relations that contain variables can be unified as well.

( ?x ?x )

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

Two relations that contain variables can be unified as well.

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

\[
\text{True, } \{ \}
\]
Unification with Two Variables

Two relations that contain variables can be unified as well.

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

Two relations that contain variables can be unified as well.

(\textcolor{red}{?x} \ ?x ) \quad \textcolor{red}{(a \ ?y c)} \quad \textcolor{red}{(a \ b \ ?z)}

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


Two relations that contain variables can be unified as well.

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

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

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

(a \ ?y \ c)

(a \ b \ ?z)\]
Two relations that contain variables can be unified as well.

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

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

Two relations that contain variables can be unified as well.

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

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

Two relations that contain variables can be unified as well.

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

Two relations that contain variables can be unified as well.

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

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

Two relations that contain variables can be unified as well.

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

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

Two relations that contain variables can be unified as well.

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

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

Substituting values for variables may require multiple steps.

lookup('?x')
Unification with Two Variables

Two relations that contain variables can be unified as well.

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

Substituting values for variables may require multiple steps.

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

Two relations that contain variables can be unified as well.

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

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

Substituting values for variables may require multiple steps.

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

Two relations that contain variables can be unified as well.

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

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

Substituting values for variables may require multiple steps.

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

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

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

1. Look up variables in the current environment
Implementing Unification

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

1. Look up variables in the current environment

2. Establish new bindings to unify elements.
Implementing Unification

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

1. Look up variables in the current environment

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

2. Establish new bindings to unify elements.
```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first, f.first, env) and \
               unify(e.second, f.second, env)
```
Searching for Proofs
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.

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

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

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

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

\[(\text{app } (?a \ . \ ?r) \ ?y \ (?a \ . \ ?z))\]
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))
The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

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

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

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

conclusion <- hypothesis

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

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

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

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

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

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

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

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

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

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

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

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

Variables are local to facts & queries
Searching for Proofs

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

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

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

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

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

\[
\text{(app } (?a . ?r) ?y (?a . ?z)) \\
\quad \text{conclusion <- hypothesis}
\]

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

\[
\text{(app } (?a2 . ?r2) ?y2 (?a2 . ?z2)) \\
\quad \text{conclusion <- hypothesis}
\]

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

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)

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.

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

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

\[(\text{app } (?a . ?r) ?y (?a . ?z))\]
\[\text{conclusion } <- \text{ hypothesis}\]

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

\[(\text{app } (?a2 . ?r2) ?y2 (?a2 . ?z2))\]
\[\text{conclusion } <- \text{ hypothesis}\]

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

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

Variables are local to facts & queries
Searching for Proofs

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

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

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

(app (?a . ?r) ?y (?a . ?z))
  conclusion <- hypothesis

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

(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
  conclusion <- hypothesis

(app ?r2 (c d) (c d))
  {r2: (), x: (c d)}

(app () ?x ?x)

Variables are local to facts & queries

left:
Searching for Proofs

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

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

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

(app (?a . ?r) ?y (?a . ?z))
  conclusion <- hypothesis

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

(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
  conclusion <- hypothesis

(app ?r2 (c d) (c d))
  {r2: (), x: (c d)}

(app () ?x ?x)

Variables are local to facts & queries

left:
Searching for Proofs

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

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

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

(left: Variables are local to facts & queries

Conclusion <- hypothesis

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

Conclusion <- hypothesis

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

(left: (app () ?x ?x))
Searching for Proofs

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

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

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

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

\[
(app \ ?r \ (c \ d) \ (b \ c \ d))
\]

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

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

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

\[
(app \ ?r2 \ (c \ d) \ (c \ d))
\]

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

\[
(app \ () \ ?x \ ?x)
\]

Variables are local to facts & queries

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

Variables are local to facts & queries

(app (?a2 . ?r2) ?y2 (?a2 . ?z2))
  conclusion <- hypothesis

(app ?r2 (c d) (c d))
  {r2: (), x: (c d)}

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.

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

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

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

conclusion <- hypothesis

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

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

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

conclusion <- hypothesis

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

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

(app () ?x ?x)

Variables are local to facts & queries

left: (e .)
Searching for Proofs

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

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

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

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

conclusion <- hypothesis

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

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

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

conclusion <- hypothesis

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

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

(left: (e . (b .

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.

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

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

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

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

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

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

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

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

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

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

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

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))))

Variables are local to facts & queries

(left: (e . (b . ())))
Searching for Proofs

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

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

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

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

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

Variables are local to facts & queries

```
left: (e . (b . ())) → (e 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: A possible proof approach is explored exhaustively before another one is considered.
Depth-First Search

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

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

\begin{verbatim}
def search(clauses, env):
\end{verbatim}
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.

```python
def search(clauses, env):
    for fact in facts:
```
Depth-First Search

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

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

```python
def search(clauses, env):
    for fact in facts:
        unify(conclusion of fact, first clause, env) -> 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: A possible proof approach is explored exhaustively before another one is considered.

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

```python
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
```
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.

```python
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
```
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.

```python
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
```
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.

```python
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.
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.

```python
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.
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.

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

    if clauses is nil:
        yield env

    elif DEPTH_LIMIT is None or depth <= DEPTH_LIMIT:

        for fact in facts:

            fact = rename_variables(fact, get_unique_id())

            env_head = Frame(env)

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

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

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

                        yield result
Implementing 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
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