Logic Programming — Prolog

Steven Zeil

Dec. 1, 2003

Contents

Logic Programming

1. Overview
2. Relations
3. Prolog Basics
4. Implementing LP
5. Impurities in Prolog

1 Overview

Logic programming is characterized by

- Statements
  - facts
  - inference rules
- Queries

- programmer describes world using logic
- programmer/user poses queries as to what is true in that world
- an inference engine tries to determine the truth/falsehood of the query from the logical description.

---

2 Relations

In logic programming, a programmer spends more effort on saying

- “what is true”

than on

- “how to compute it”

In effect, *functions* are replaced by *relations*.

Imperative/functional programmers see $+$ as the name for a function: you supply its operands and it returns a result.

But go back to grade school math, and you get a different view:

- What is 3 plus 2?
- What plus 2 equals 4?
- 3 plus what equals 6?

“plus” is not viewed here as an executable operator, but as a *relation among triplets of numbers*.

2.1 Definition

A **relation** of arity n is a set of n-tuples.

Given a relation R, we can ask questions like

- Is $(x_1, x_2, \ldots, x_n) \in R$?
- For what values of x_2 is $(x_1, x_2, \ldots, x_n) \in R$?

For example, consider the relation `prereq` defined as

$$\{(cs150, cs250), (cs150, cs281), (cs250, cs361), (cs281, cs361), (cs361, cs350), (cs250, cs355), (cs281, cs390), (mt163, cs281)\}$$

We can use this relation to answer questions like

- Is `cs281` an (immediate) prerequisite for `cs361`?
- What are the (immediate) prerequisites for `cs281`?

$$\{(cs150, cs250), (cs150, cs281), (cs250, cs361), (cs281, cs361), (cs361, cs350), (cs250, cs355), (cs281, cs390), (mt163, cs281)\}$$

Furthermore, we can, with a few appropriate rules, *infer* the answers to questions like

- Must `cs250` be taken before `cs390`?
- Can `cs281` and `cs361` be taken within the same semester?

Questions like

- Must `cs250` be taken before `cs390`?
- Can `cs261` and `cs361` be taken within the same semester?

actually introduce new relations.

The relation `mustTakeBefore` can be defined as

$$\{(cs150, cs250), (cs150, cs281), (cs150, cs350), (cs150, cs361), (cs150, cs390), (cs250, cs350), (cs250, cs361), (cs281, cs350), (cs281, cs361), (cs281, cs390), (mt163, cs281), (mt163, cs350), (mt163, cs361), (mt163, cs390), (cs361, cs350)\}$$

We can visualize this relation as:

	163	150	250	281	300	350	355	361	390
163				√		√		√	√
150			√	√		√	√	√	√
250						√	√	√	
281						√		√	√
300									
350									
355									
361						√			
390									

The relation `canTakeTogether` can be given as

	163	150	250	281	300	350	355	361	390
163	?	√	√		√				
150	√	?			√				
250	√		?	√	√				√
281			√	?	√		√		
300	√	√	√	√	?	√	√	√	√
350					√	?	√		√
355				√	√	√	?	√	√
361					√		√	?	√
390			√		√	√	√	√	?

	163	150	250	281	300	350	355	361	390
163	?	√	√		√				
150	√	?			√				
250	√		?	√	√				√
281			√	?	√		√		
300	√	√	√	√	?	√	√	√	√
350					√	?	√		√
355				√	√	√	?	√	√
361					√		√	?	√
390			√		√	√	√	√	?

This relation is symmetric. If $(x, y) \in$ `canTakeTogether`, then $(y, x) \in$ `canTakeTogether`.

Not all relations are binary. We have already mentioned `plus`:

$$\{(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 2), \ldots\}$$

which is not only not binary, it is not finite!

We can also have unary relations, such as `requiredCSCourse`: $\{cs150, cs250, cs281, cs300, cs350, cs355, cs361, cs390\}$

---

3 Prolog Basics

1. Data Types

2. Facts and Rules
3. Interacting with Prolog
4. Unification
5. Lists
6. Unification and Rules

3.1 Data Types

Data in Prolog is a mixture of ideas familiar from SML and Scheme:

1. Variables
2. Simple Terms
3. Compound Terms

$$\begin{array}{rcl} \langle \textit{term} \rangle & ::= & \langle \textit{simple-term} \rangle \\ & | & \langle \textit{compound-term} \rangle \\ & | & \texttt{variable} \end{array}$$

3.1.1 Variables

Variables in Prolog must begin with a capital letter:

`X`, `AVariable`, `Another_Variable`

- A special exception is "_", which is an anonymous variable (as in SML).
  - Each occurrence of "_" denotes a separate variable.
- Many Prolog systems generate internal variable names that begin with "_"
- Variables are not typed — they can be bound to any value.

3.1.2 Simple Terms

$$\langle \textit{simple-term} \rangle ::= \texttt{atom} | \texttt{number}$$

- Atoms must begin with lower-case letters or non-alphanumerics, or appear within single quotes:

  `atom`, `aTOM`, `+`, `++++`, `'Atom'`
- Numbers: `1997`, `3.14159`

3.1.3 Compound Terms

The general form of a compound term is

$$\begin{array}{rcl} \langle \textit{compound-term} \rangle & ::= & \texttt{atom(} \langle \textit{terms} \rangle \texttt{)} \\ \langle \textit{terms} \rangle & ::= & \langle \textit{term} \rangle \\ & | & \langle \textit{term} \rangle \texttt{,} \langle \textit{terms} \rangle \end{array}$$

Examples:

```
f(g(h), 2)
+(x,y)
node(10,
     leaf(1),
     node(20, empty, leaf(25)))
```

Certain compound forms have "convenient" alternate syntaxes:

- Arithmetic operators can be written in infix form: `1+2`

- Lists can be written as
  - `[a,b,c,d]` or as
  - `[x | L]` where `x` is the first element and `L` is the list of remaining elements.

Alternate syntax (cont.):

- Strings can be written in double quotes (`"abc"`).
  - A string is really a list of characters
    - and characters are really integers

3.2 Facts and Rules

1. Facts
2. Rules

3.2.1 Facts

$$\langle \textit{fact} \rangle ::= \langle \textit{term} \rangle \texttt{.}$$

Facts are simple statements of relationships:

```
prereq(cs150, cs250).
prereq(cs150, cs281).
prereq(cs250, cs361).
  ⋮
```

The collection of all `prereq` facts will define the `prereq` relation.

Some more facts:

```
requiredCScourse(cs150).
requiredCScourse(cs250).
requiredCScourse(cs281).
  ⋮
```

3.2.2 Rules

$$\langle rule \rangle ::= \langle term \rangle \texttt{ :- } \langle terms \rangle \texttt{ .}$$

Rules describe relations using other relations. A term is in the relation if it satisfies *any* rule for that relation:

```
mustTakeBefore(C,D) :- prereq(C,D).
mustTakeBefore(C,D) :-
    prereq(C,E), mustTakeBefore(E,D).
```

```
mustTakeBefore(C,D) :- prereq(C,D).
mustTakeBefore(C,D) :-
    prereq(C,E), mustTakeBefore(E,D).
```

Read this as

> C must be taken before D if C is a prerequisite of D, or if there is some course E that C is a prerequisite for, and E must be taken before D.

Describe a relation `cannotTakeTogether(C,D)`, for courses `C` and `D` that must be taken in separate semesters,

---

3.3 Interacting with Prolog

Most Prolog systems are in one of 2 modes at any given time.

- Query mode
- Definition mode

3.3.1 Query Mode

This is the interactive mode of prolog.

Queries may be entered as

$$\langle query \rangle ::= \langle terms \rangle \texttt{ .}$$

Sample queries:

```
X = abc.
f(g(h))=f(g(h)).
f(g(h))=f(h(g)).
X = "abc".
[X, b, c] = [a, b, Y].
```

- The Prolog system determines if the query is true/false by trying to find a satisfactory assignment for each variable in the query.
- If it cannot, it responds “No” or “Fail”, indicating that the query cannot be satisfied.

- If it can, it responds “Yes” or “Success”, and lists the variable bindings that it found to satisfy the query.

  The user can then

  – hit Return to terminate the query, or
  – type a semi-colon to request that the Prolog system find a different set of variable bindings that satisfy the query.

From query mode, one can ask the system to load a file (in definition mode) via

```
[filename-atom].
```

For example:

```
['prereq.pro'].
```

Alternatively, one can say:

```
consult('prereq.pro').
```

3.3.2 Definition Mode

Used when reading from a file.

The system is reading facts and rules and storing them in its internal data base.

Queries may be posted as

$$\langle query \rangle ::= \texttt{:- } \langle terms \rangle \texttt{ .}$$

A more typical Prolog session might look like:

```
['prereq.pro'].
prereq(X,cs361).
mustTakeBefore(X,cs361).
```

3.4 Unification

All variable binding in Prolog occurs via **unification**, a kind of pattern matching between terms.

- More general than SML's pattern matching
  - SML: each variable can occur only once in a pattern
  - Prolog: variables can occur multiple times

3.4.1 Unification: Definition

A term U is an **instance** of another term T if U is obtained by replacing variables $V_1, V_2, \ldots, V_k$ in T by terms $u_1, u_2, \ldots, u_k$.

Terms T_1 and T_2 **unify** if they have a common instance U.

3.4.2 Unification: Examples

Unification occurs each time a fact/rule is matched against a query:

query: `prereq(cs250, Z).`

rule 1: `prereq(cs150,cs250).`

The two terms do not unify.

- There's only one variable, `Z`
- and no substitution for it would make these terms identical.

---

query: `prereq(cs250, Z).`

rule 2: `prereq(cs250,cs350).`

This unifies with the query, under a substitution of `Z`→`cs350`.

An easy way to experiment with unification is by using `=`.

`=` in Prolog is neither an assignment nor an equality test. It means "unifies with".

```
X = 1.
1 = X.
X = y+1.
2+X=Y+3.
2+X=X+3.
Y+X=X+3.
f(X, g(h(z))) = f(x+y, Y).
f(X, g(h(X))) = f(x+y, Y).
f(X, g(h(Y))) = f(x+y, Y).
```

3.4.3 Most General Unifiers

In some cases, there is more than one unifier (substitution) that could be applied to unify two terms.

```
f(g(X)) = f(Y).
```

This could be unified by the substitution {`Y`→`g(X)`}, or by {`X`→`foo(z), Y`→`g(foo(z))`}, or by ...

Prolog always uses the **most general unifier**, the one that binds the fewest variables.

You can find the most general unifier by matching from the outermost operators:

Do `f(A,g(h(B)))` and `f(h(c), g(A))` unify?

- Outermost operators `f` match, so they unify if
  - `A` and `h(c)` unify, and
  - `g(h(B))` and `g(A)` unify

  with the *same substitution.*

- Do `A` and `h(c)` unify?
  - Yes, under substitution {`A`→`h(c)`}
- Do `g(h(B))` and `g(A)` unify under {`A`→`h(c)`}?
  - Apply substitution to get `g(h(B))` and `g(h(c))`
  - Outer operators (`g`) match, so they unify if
    - ∗ `h(B)` and `h(c)` unify.
    - ∗ Outer operators (`h`) match, so they unify if `B` and `c` unify.
      - · They do, under substitution {`B`→`c`}

Summary:

Do `f(A,g(h(B)))` and `f(h(c), g(A))` unify?

- Yes, under {`A`→`h(c), B`→`c`}.

3.5 Lists

Unification applies to lists as well:

```
[a, b, c] = L.
[a, b, c] = [a|L].
[a, b, c] = [x|L].
[a, b, c] = [X|L].
[a] = [X|L].
[] = [X|L].
[a, b, c] = [X, Y |L].
[a, b, c] = [X, Y, Z |L].
[a, b(X), c] = [X, Y |L].
```

3.5.1 Common List Utilities

We can use our new knowledge of unification to set up some list relations:

- `member(X,L)` is satisfied if `X` is an element of the list `L`
- `append(A,B,C)` is satisfied if `C` is a list formed by appending list `B` onto the end of list `A`.

---

member

```
member(X, [X|_]).
member(X, [_|L]) :- member(X,L).
```

Try this for queries:

```
member(2, [1,2,3]).
member(4, [1,2,3]).
member(100, "zeil").
member(A, [a,b,c]).
member(14, [a,B,c]).
member(14, [A,B,C]).
member(14, [A,A,A]).
member(100, L).
member("zeil", [X|L]).
```

Use the ';' response to try and force multiple solutions.

---

append

```
append([], B, B).
append([X|A], B, [X|C])
    :- append(A,B,C).
```

Try this for queries:

```
append([a,b], [d,e,f], C).
append([a,b], B, [a,b,c,d]).
append([a,b], B, [a,b]).
append(A, [d], [a,b,c,d]).
append(A, A, [a,b,c|L]).
```

3.6 Unification and Rules

When a rule `A :- B`$_1$`, ..., B`$_j$ is applied to a goal (query) `G`

- The rule's "head" `A` is unified with `G`.

  Let σ be the most general unifier of `A` and `G`.
- ...

- ...
- The substitution σ is applied to the rule body `B`$_1$`, ..., B`$_j$ to yield new subgoals `B`$_1\sigma$`, ..., B`$_j\sigma$.
- The Prolog system attempts to prove the subgoals.

Example: `append([a | R], S, [T | [d]])`

- Compare to the two `append` rules:

```
append([], B, B).
append([X|A], B, [X|C])
    :- append(A,B,C).
```

- The first rule's head does not unify, so we try the second rule.

- `append([a | R], S, [T | [d]])` unifies with `append([X|A], B, [X|C])` under unifier {`X`→`a`, `A`→`R`, `B`→`S`, `T`→`a`, `C`→`[d]`}.
- Apply that substitution to the right-hand-side of the rule. New goal is `append(R, S, [d])`.

Goal: `append(R, S, [d])`

- Compare to the two `append` rules:

```
append([], B, B).
append([X|A], B, [X|C])
    :- append(A,B,C).
```

- The first rule's head unifies with `append(R, S, [d])` under unifier {`R`→`[]`, `S`→`[d]`, `B`→`[d]`}.

- We're out of unproven goals, so we conclude that `append([a | R], S, [T | [d]])` is satisfiable, with unifier {`R`→`[]`, `S`→`[d]`, `T`→`a`}.

4 Implementing LP

A query is just a list of terms.
Each term is a *goal*.

- Goals are satisfied by unification with the heads of rules,
- but then the terms on the right-hand-side are added as additional goals.

The heart of a Prolog engine is the procedure

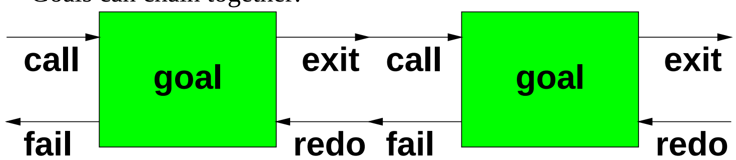
```
proc visit (goals: list<term>,
            s: unifier)
begin
  if goals == [] then
    succeed(s);
  else
    g = hd(goals);
    rest = tl(goals);
    solve(g, rest, s);
  end if;
end;
```

```
proc succeed(s: unifier)
begin
  print s;
  do
    get response;
  until (response == '\n')
    or (response == ';');
  if (response == '\n')
    abort; // and be happy
end;
```

```
proc solve(g: term, goals: list<term>,
           s: unifier)
begin
  for (i = 0; i < numRules; ++i)
    let rule[i] be A :- List;
    if (A unifies with g) then
      let σ be the m.g. unifier;
      newGoals = Listσ ∪ goalsσ;
      visit (newGoals, sσ);
    end if;
  end loop;
end;
```

A common visualization of the Prolog inference process is

Goals can chain together:

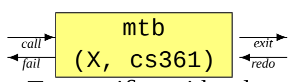


4.1 Example with Backtracking

Query: `mustTakeBefore(X,cs361)`
(Abbreviate `mustTakeBefore` as `mtb`)

```
visit([mtb(X, cs361)], {})
```

```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
```

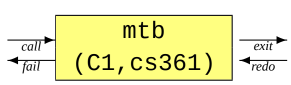


Term unifies with rule:

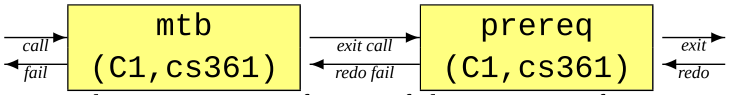
```
mustTakeBefore(C1,D1) :- prereq(C1,D1).
```

Unifier is {`X`→`C1`, `D1`→`cs361`}

```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C1,cs361)],
          {X→C1, D1→cs361})
```



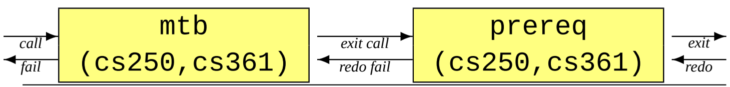

```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C1,cs361)],
          {X→C1, D1→cs361})
      solve(prereq(C1,cs361), [],
            {X→C1, D1→cs361})
```



Several `prereq` facts fail to unify, e.g. `prereq(cs150,cs250)`

Eventually, we get to `prereq(cs250,cs361)`

```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C1,cs361)],
          {X→C1, D1→cs361})
      solve(prereq(C1,cs361), [],
            {X→C1, D1→cs361})
        visit([],
              {X→cs250, D1→cs361,
               C1→cs250})
```

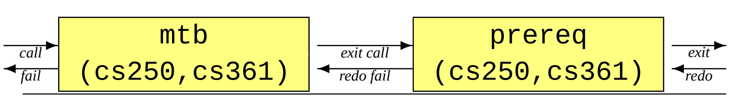



```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C1,cs361)],
          {X→C1, D1→cs361})
      solve(prereq(C1,cs361), [],
            {X→C1, D1→cs361})
        visit([],
              {X→cs250, D1→cs361,
               C1→cs250})
          succeed({X→cs250, D1→cs361,
                   C1→cs250})
```

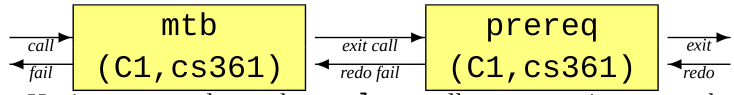
Prints `X=cs250`

Suppose we hit ; to force a new solution.

```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C1,cs361)],
          {X→C1, D1→cs361})
      solve(prereq(C1,cs361), [],
            {X→C1, D1→cs361})
        visit([],
              {X→cs250, D1→cs361,
               C1→cs250})
```

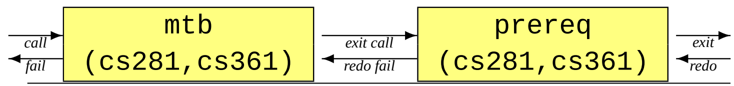



```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C1,cs361)],
          {X→C1, D1→cs361})
      solve(prereq(C1,cs361), [],
            {X→C1, D1→cs361})
```



Having returned to the `solve` call, we continue searching through `prereq` rules, and eventually unify with `prereq(cs281,cs361)`.

```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C1,cs361)],
          {X→C1, D1→cs361})
      solve(prereq(C1,cs361), [],
            {X→C1, D1→cs361})
        visit([], {X→cs281, D1→cs361,
                   C1→cs281})
```

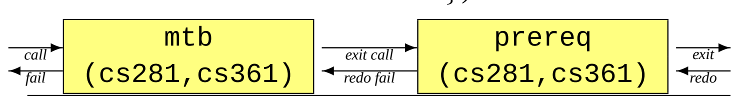



```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C1,cs361)],
          {X→C1, D1→cs361})
      solve(prereq(C1,cs361), [],
            {X→C1, D1→cs361})
        visit([], {X→cs281, D1→cs361,
                   C1→cs281})
          succeed({X→cs281, D1→cs361,
                   C1→cs281})
```

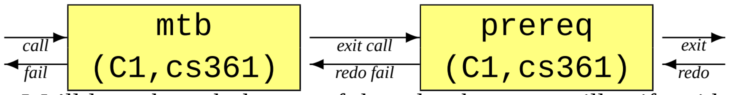
Prints `X=cs281`

Suppose we hit ; again to force a new solution.

```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C1,cs361)],
          {X→C1, D1→cs361})
      solve(prereq(C1,cs361), [],
            {X→C1, D1→cs361})
        visit([], {X→cs281, D1→cs361,
                   C1→cs281})
```

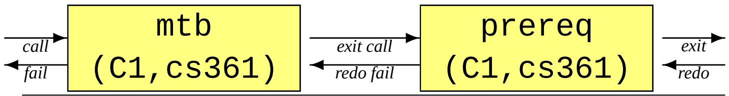



```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C1,cs361)],
          {X→C1, D1→cs361})
      solve(prereq(C1,cs361), [],
            {X→C1, D1→cs361})
```

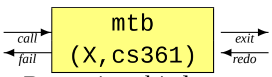


We'll loop through the rest of the rules, but none will unify with this term.

```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C1,cs361)],
          {X→C1, D1→cs361})
```



```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
```

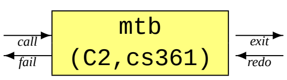


Resuming this loop, we unify with the head of

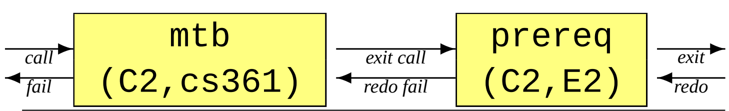
```
mustTakeBefore(C2,D2)
  :- prereq(C2,E2), mustTakeBefore(E2,D2).
```

Unifier is {`X→C2, D2→cs361`}

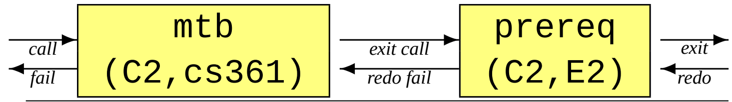
```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C2,E2), mtb(E2,cs361)],
          {X→C2, D2→cs361})
```



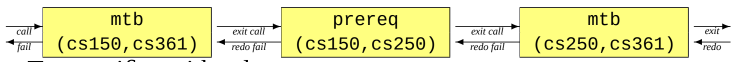
```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C2,E2), mtb(E2,cs361)],
          {X→C2, D2→cs361})
      solve(prereq(C2,E2),
            [mtb(E2,cs361)],
            {X→C2, D2→cs361})
```



```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C2,E2), mtb(E2,cs361)],
          {X→C2, D2→cs361})
      solve(prereq(C2,E2),
            [mtb(E2,cs361)],
            {X→C2, D2→cs361})
        visit([mtb(cs250,cs361)],
              {X→cs150, D2→cs361,
               C2→cs150, E2→cs250})
```



```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C2,E2), mtb(E2,cs361)],
          {X→C2, D2→cs361})
      solve(prereq(C2,E2),
            [mtb(E2,cs361)],
            {X→C2, D2→cs361})
        visit([mtb(cs250,cs361)],
              {X→cs150, D2→cs361,
               C2→cs150, E2→cs250})
          solve(mtb(cs250,cs361), [],
                {X→cs150, D2→cs361,
                 C2→cs150, E2→cs250})
```

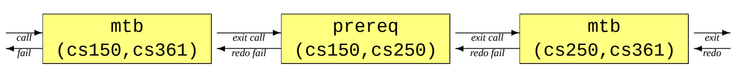


Term unifies with rule:

```
mustTakeBefore(C3,D3) :- prereq(C3,D3).
```

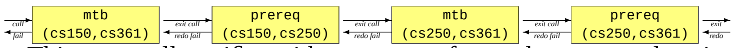
Unifier is {`C3→cs250, D3→cs361`}

```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C2,E2), mtb(E2,cs361)],
          {X→C2, D2→cs361})
      solve(prereq(C2,E2),
            [mtb(E2,cs361)],
            {X→C2, D2→cs361})
        visit([mtb(cs250,cs361)],
              {X→cs150, D2→cs361,
               C2→cs150, E2→cs250})
          solve(mtb(cs250,cs361), [],
                {X→cs150, D2→cs361,
                 C2→cs150, E2→cs250})
            visit([prereq(cs250,cs361)],
                  {X→cs150, D2→cs361,
                   C2→cs150, E2→cs250,
                   C3→cs250, D3→cs361})
```



```
visit([mtb(X, cs361)], {})
  solve(mtb(X,cs361), [], {})
    visit([prereq(C2,E2), mtb(E2,cs361)],
          {X→C2, D2→cs361})
      solve(prereq(C2,E2),
```

```
                [mtb(E2,cs361)],
                {X→C2, D2→cs361})
            visit([mtb(cs250,cs361)],
                  {X→cs150, D2→cs361,
                   C2→cs150, E2→cs250})
              solve(mtb(cs250,cs361), [],
                    {X→cs150, D2→cs361,
                       C2→cs150, E2→cs250})
                visit([prereq(cs250,cs361)],
                      {X→cs150, D2→cs361,
                       C2→cs150, E2→cs250,
                       C3→cs250, D3→cs361})
                   solve(prereq(cs250,cs361), [],
                      {X→cs150, D2→cs361,
                         C2→cs150, E2→cs250})
```



This eventually unifies with a `prereq` fact and we succeed again, printing “`X=cs150`”

---

5 Impurities in Prolog

. . .