# Logic Programming — Impurities in Prolog

Steven Zeil

Apr. 1, 2001

# Contents



# Logic Programming

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

# 1 Impurities in Prolog

1. Numerical Calculations
2. Order Dependencies
3. Occurs Checks
4. Failure as Negation
5. Cuts

## 1.1 Numerical Calculations

Evaluation of numerical expressions in Prolog is accomplished via `is`:

⟨*simple-term*⟩ `is` ⟨*term*⟩

(A simple term is an atom, variable, or number).

For example,

```
A is 1+2*3.
X=1, Y=2, Z is X/Y.
3 is 1+2.
X=1, Y=2, 3 is X/Y.
```

The last example fails, but is still legal as a test.

### 1.1.1 `is` versus `=`

- `is` evaluates an expression
- `=` unifies expressions

Compare the results of

```
A is 1+2*3.
A = 1+2*3.
X=1, Y=2, Z is X/Y.
X=1, Y=2, Z = X/Y.
3 is 1+2.
3 = 1+2.
```

### 1.1.2 `is` is not a pure relation.

The term on the right hand side of an `is` must be **grounded**, i.e., all its variables must be bound to atoms, numbers, or to other grounded expressions.

- This is valid:

  ```
  X=1, Y=2, 0.5 is X/Y.
  ```

- This is not:

  ```
  X=1, 0.5 is X/Y.
  ```

  It doesn’t merely fail, it is an error that halts the execution.

### 1.1.3 Numeric Tests

- Prolog provides the usual range of relational operators:
  - >, <, >=, =<
  - Numerical equality test: `=:=`
  - Numerical not-equals test: `=\=`
- These do an implicit evaluation of their operands.
  - Therefore, like `is`, they operate on grounded terms only.

---

```
A is 1+2, A =:= 3.   /* yes */
1+2 =\= 3.           /* no */
A is 1+2, A > 0.     /* yes */
1+2 =:= B.           /* error */
A is 1+2, A < A+1.   /* yes */
```

---

## 1.2 Order Dependencies

The order in which we write goals and rules can affect

- speed
- meaning
- validity

of a Prolog program.

---

1. Order and Grounding
2. Goal Ordering
3. Rule Ordering

---

### 1.2.1 Order and Grounding

An obvious example of order dependency arises with predicates that require grounded operands.

Compare, for example:

```
A=1+2, B=A.
B=A, A=1+2.
A=1+2, B=:=A.
B=:=A, A=1+2. /*error*/
```

---

### 1.2.2 Goal Ordering

Given

```
prefix(A,B) :- append(A, _, B).
suffix(A,B) :- append(_, A, B).
```

use ';' to examine the possible solutions of:

```
prefix(X, [a,b,c]).
suffix([a], X).
```

---

Now try the combined queries:

```
prefix(X, [a,b,c]), suffix([a], X).
```

and

```
suffix([a], X), prefix(X, [a,b,c]).
```

The second query goes into an "infinite" recursion after ';'.

---

```
prefix(X, [a,b,c]), suffix([a], X).
```

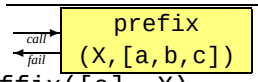
These are placed on a list of pending goals.

---

Remaining goals: `prefix(X, [a,b,c]), suffix([a], X)`

The first goal is then visited.

---

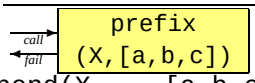



Remaining goals: `suffix([a], X)`

The current goal unifies with the rule

```
prefix(A,B) :- append(A, _, B).
```

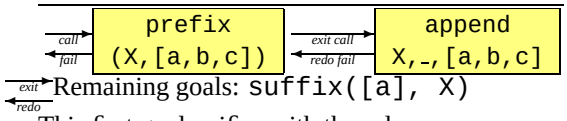
---




Remaining goals: `append(X, _, [a,b,c]), suffix([a], X)`

We then call (visit) the first of the remaining goals.

---



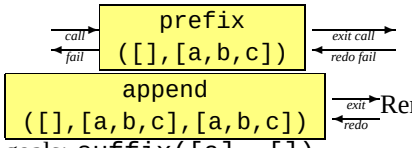

Remaining goals: `suffix([a], X)`

This first goal unifies with the rule

```
append([], B, B).
```

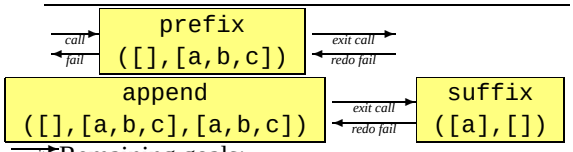
which unifies {`X`→`[]`}.

---




Remaining goals: `suffix([a], [])`

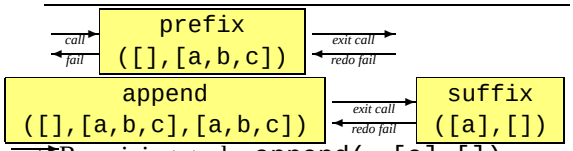
Again, we visit the first remaining goal.




Remaining goals:

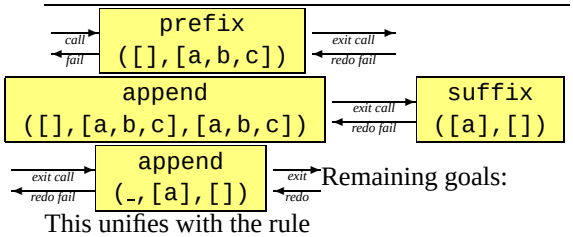
This unifies with the rule

```
suffix(A,B) :- append(_, A, B).
```




Remaining goals: `append(_,[a],[])`

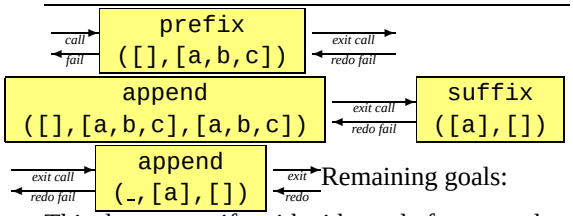
Visit first remaining goal.




Remaining goals:

This unifies with the rule

```
append([],B,B).
```

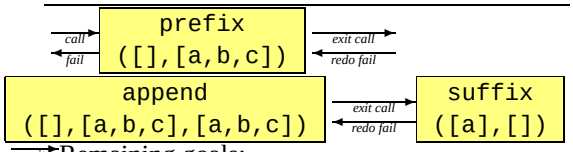



Remaining goals:

This does not unify with either rule for append:

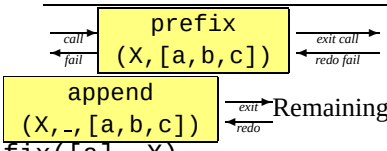
```
append([], B, B).
append([X|A], B, [X|C]) :- append(A,B
```

So this goal fails and we redo the prior goal.




Remaining goals:

But `suffix` only has one rule, and we've already tried that, so this goal fails as well.

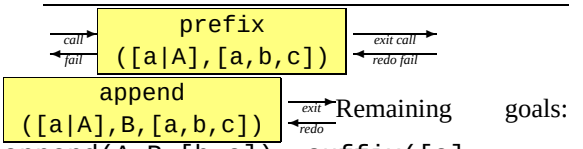



Remaining goals: `suffix([a], X)`

We move on to the 2nd rule for append, which unifies.

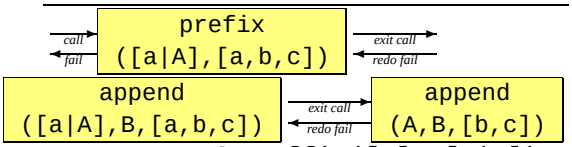
```
append([Y|A], B, [Y|C]) :- append(A,B,C).
```

with $\sigma$={`X`→`[a|A], C`→`[b,c]`}




Remaining goals: `append(A,B,[b,c]), suffix([a], [a|A])`

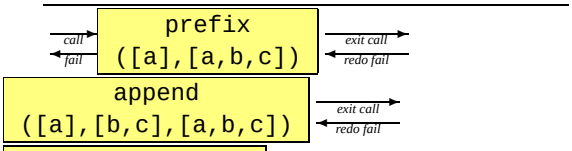
We now visit the first remaining goal.




Remaining goals: `suffix([a], [a|A])`

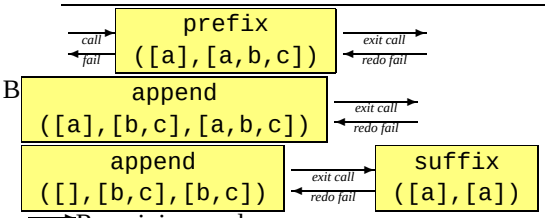
The current goal unifies with the rule:

```
append([], D, D).
```




Remaining goals: `suffix([a], [a])`

Visit the 1st remaining goal.




Remaining goals:

This unifies with

```
suffix(A,B) :- append(_, A, B).
```

**And now for something completely different**

Reverse the order of the goals:

```
suffix([a], X), prefix(X, [a,b,c]).
```

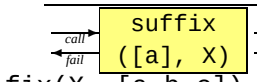
For the sake of brevity, we will omit the `append` subgoals.

---

Remaining goals: `suffix([a], X), prefix(X, [a,b,c])`

The first goal is then visited.

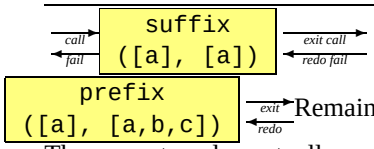
---



Remaining goals: `prefix(X, [a,b,c])`

The current goal eventually succeeds with $\sigma$=$\{$`X`$\rightarrow$`[a]`$\}$

---



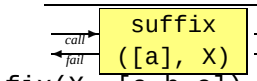
Remaining goals:

The current goal eventually succeeds and the system announces

```
X = [a]
```

Suppose we hit ';' to force a redo of the last goal. It then fails.

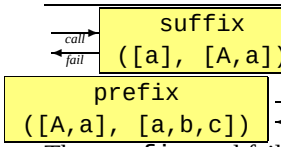
---



Remaining goals: `prefix(X, [a,b,c])`

On redo, the current goal eventually succeeds with $\sigma$=$\{$`X`$\rightarrow$`[A,a]`$\}$

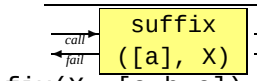
---



Remaining goals:

The `prefix` goal fails.

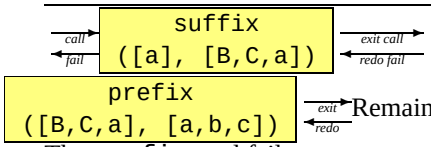
---



Remaining goals: `prefix(X, [a,b,c])`

On redo, the current goal eventually succeeds with $\sigma$=$\{$`X`$\rightarrow$`[B,C,a]`$\}$

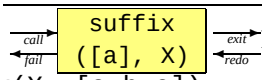
---



Remaining goals:

The `prefix` goal fails.

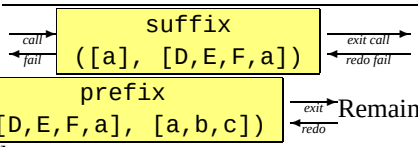
---



Remaining goals: `prefix(X, [a,b,c])`

On redo, the current goal eventually succeeds with $\sigma$=$\{$`X`$\rightarrow$`[D,E,F,a]`$\}$

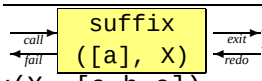
---



Remaining goals:

The `prefix` goal fails.

---



Remaining goals: `prefix(X, [a,b,c])`

On redo, the current goal eventually succeeds with $\sigma$=$\{$`X`$\rightarrow$`[G,H,I,J,a]`$\}$

This pattern continues forever, because

- there are an infinite number of lists that will satisfy `suffix([a], X)`,
- but none (except `[a]` will satisfy `prefix(X,[a,b,c])`.

---

### 1.2.3 Rule Ordering

- The Prolog engine attempts each unifying rule in the order they occur in the database.
- Changing that order alters the order of solutions attempted
- May affect meaning, speed, or termination of a query.

---

For example, we previously defined

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

The query `member(2,L)` would return successive solutions:

```
L=[2|A]
L=[2,A|B]
L=[2,A,B|C]
```

(Why isn't the first solution `L=[2]`? Did we miss a solution?)

---

---

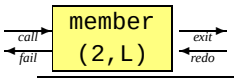
If we reverse the order of those two rules

```
member2(X, [_|L]) :- member2(X,L).
member2(X, [X|_]).
```

The query `member(2,L)` goes into an infinite recursion.

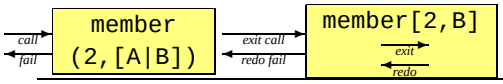
---

```
member2(X, [_|L]) :- member2(X,L).
member2(X, [X|_]).
```



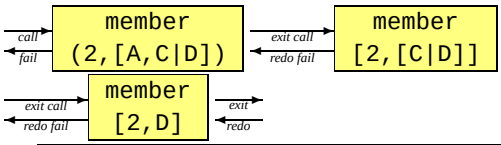

---

```
member2(X, [_|L]) :- member2(X,L).
member2(X, [X|_]).
```



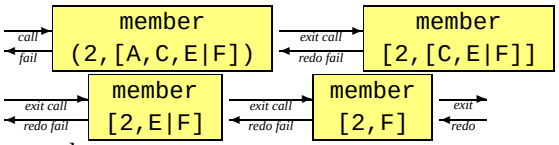

---

```
member2(X, [_|L]) :- member2(X,L).
member2(X, [X|_]).
```




---

```
member2(X, [_|L]) :- member2(X,L).
member2(X, [X|_]).
```




and so on...

---

- One way to combat rule order dependency is to always list "base cases" before recursive cases.
  - Not fool-proof, though.
- Trying to anticipate these dependencies is hard because we need to consider all possible combinations of grounded/ungrounded operands.

---

## 1.3 Occurs Checks

- Prolog is essentially tree-oriented.
- If a variable `X` is unified with a term that contains `X`, this tree structure is violated.
  - `X=f(X)`
  - `append(A, E, [a,b|E])`
- But for efficiency reasons, Prolog does not check for such cases.

---

## 1.4 Failure as Negation

Given the facts

```
student(john).
student(mary).
student(vladimir).
```

What is the response to the query: `student(brunhilda).`?

---

Why does it say "no"?

- Not because we *know* that Brunhilda is not a student,
- But because we cannot *prove* that she is

---

### 1.4.1 Closed World Assumption

Prolog operates under a **closed world assumption**:

- There are not facts or rules missing from out database's description of the "world".

---

### 1.4.2 "not" in a Closed World

- In a closed world, we should be able to prove anything that is true.
- So a natural definition of "not" would be:
  - `not(X)` succeeds if `X` fails
  - `not(X)` fails if `X` succeeds
- But this definition is very prone to order dependencies
- and can cause other surprises as well.

---

```
X=a, Y=b, X=Y.          /* no */
X=a, Y=b, not(X=Y).     /* yes */
X=a, not(X=Y), y=b.     /* yes */
X=a, not(X=Y).          /* no */
X=a, X=Y.               /* yes */
X=a, not(not(X=Y)).     /* yes */
```

Of course, if the closed world assumption does not hold, or other impurities prevent `X` from being proven, then `not(X)` may mean something entirely different.

## 1.5 Cuts

Given rules

$$\begin{array}{lll} A & :- & B_1, B_2, \ldots. \\ A & :- & C_1, C_2, \ldots. \\ & \vdots & \\ A & :- & D_1, D_2, \ldots, D_k, !, D_{k+1}, \ldots. \\ A & :- & E_1, E_2, \ldots. \\ & \vdots & \end{array}$$

The "`!`" is a **cut**.

When a cut is "solved" as a goal, we "commit" to the use of that rule.

- If a subsequent $D_i$ term in that rule fails, then
  - this rule fails
  - no other rules for `A` will be attempted
    - so the "parent" `A` goal will fail as well
- If the remaining terms in this rule succeed, but a later failure causes us to try to redo the parent `A` goal, then that parent goal fails.

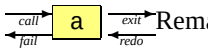
### 1.5.1 Example of Cut

Given the rules:

```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```

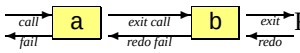
what happens to the query: `a`?

```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```



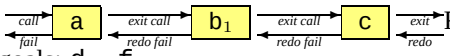
Remaining goals:

```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```



Remaining goals: `f`

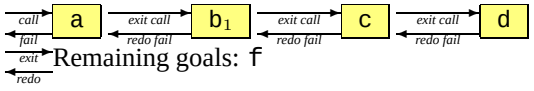
```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```



Remaining goals: `d, f`

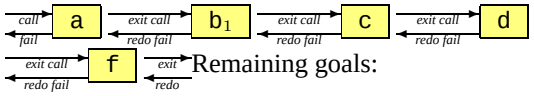
The notation $\texttt{b}_1$ indicates that we have tried (are trying) rule 1.

```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```



Remaining goals: `f`

```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```

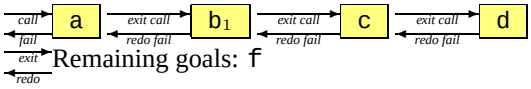


Remaining goals:

- `b` has "succeeded" using rule 1
- but `f` fails, forcing a redo.

---

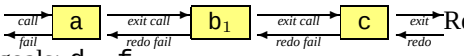
```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```



Remaining goals: `f`

---

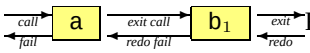
```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```



Remaining goals: `d, f`

---

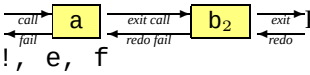
```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```



Remaining goals: `d, f`

---

```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```

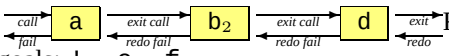


Remaining goals: `d, !, e, f`

---

```
a :- b, f.
b :- c, d.
b :- d, !, e.
```

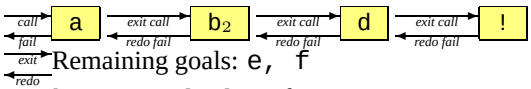
```
b :- c.
c.
d.
e.
```



Remaining goals: `!, e, f`

---

```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```

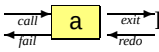


Remaining goals: `e, f`

The cut is "solved". It forces us to commit to the current rule.

---

```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```

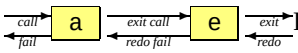


Remaining goals: `e, f`

- The "remaining goals" list still contains the remaining portion of the 2nd `b` rule.
- But the already successful portion of that rule disappears from the backtrack list
- as does the `b` marker that was the head of the current rule.
- These are removed so that a later failure will not induce a redo of those terms.

---

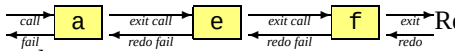
```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```



Remaining goals: `f`

---

```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```

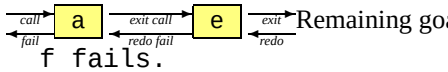


Remaining goals:

```
f fails.
```

---

```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```

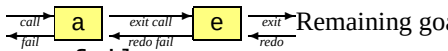


Remaining goals: `f`

```
f fails.
```

---

```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```

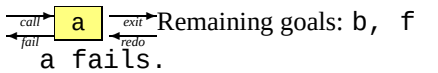


Remaining goals: `f`

```
e fails.
```

---

```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```



Remaining goals: `b, f`

```
a fails.
```

---

```
a :- b, f.
b :- c, d.
b :- d, !, e.
b :- c.
c.
d.
e.
```

The query fails.

---

### 1.5.2 Applications of Cut

Cut can be used to

- improve efficiency
- force termination
- prevent duplicate solutions

But cuts can also change the meaning of a term, and so must be used with care.

---

Consider the factorial relation:

```
fac(0, 1).
fac(N, F) :- N1 is N-1,
             fac(N1, F1),
             F is F1*N.
```

This works OK for `fac(4,A)`, but hangs up on

```
fac(4,A), A=10
```

- `A=10` forces redo of `fac(0,X)`
- on redo, tries to use 2nd rule to take $0*R$, where $R$ is found by `fac(-1,R)`.

---

Solution is to realize that any time we use the first rule, we don't want to redo with the second:

- The conditions for taking the two rules are mutually exclusive.

So add a cut:

```
fac(0, 1) :- !.
fac(N, F) :- N1 is N-1,
             fac(N1, F1),
             F is F1*N.
```

---

Consider a relation for checking to see if one list is a subset of another:

```
subset([], _).
subset([X|A], B) :- member(X, B),
                    subset(A,B).
```

What happens if this is invoked in a query:

```
L = [0,0,0,0,0, 1,1,1,1,1,
     2,2,2,2,2, 3,3,3,3,3,
     \ldots 9,9,9,9,9],
subset([1,2,3,4,5,6,7,8,9,a], L).
```

- We will get 9 successful `member` checks, then fail on `member(a,L)`
- On redo, the next most recent `member(9,L)` succeeds again.
- We fail again on `member(a,L)`
- On redo, the next most recent `member(9,L)` succeeds again.

⋮

-

---

One way to speed this up is to use a cut:

```
subset2([], _).
subset2([X|A], B) :- member(X, B), !,
                     subset2(A,B).
```

This works very well when both operands are grounded, but not with a query like:

```
subset2([23], L).
```

---

What does the following predicate do?

```
foo(X) :- X, fail.
foo(X).
```

---

---

---

Now let's add a single cut:
What does the following predicate do?

```
foo(X) :- X, !, fail.
foo(X).
```

---

---

The combination `!, fail` is a common Prolog idiom for aborting a set of rules.

---

# 2 Programming in Prolog

1. Algorithmic Styles
2. Dare to Fail!
3. Open Lists

---

## 2.1 Algorithmic Styles

1. Backtracking
2. Guess and Verify

---

### 2.1.1 Backtracking

Prolog lends itself well to backtracking algorithms:

Given a problem with $N$ variables $X_1, \ldots, X_N$, each of which can take on one of $k$ values: $v_1, v_2, \ldots, v_k$, do

```
backtrack(i)
begin
  if (i > N) then
    print X's
    abort;
  else
    for j := 1 to k loop
      X[i] := v[j];
      backtrack(i+1);
    end loop
  end if
end;

backtrack(1);
```

---

In Prolog, this looks like

```
val(v1).
val(v2).
  ⋮
backtrack([]).
backtrack([X|L]) :-
      val(X), backtrack(L).
```

with the query:

```
backtrack([X1, X2, X3, \ldots]).
```

---

Problem is, backtracking is $O(k^N)$.

This makes Prolog one of the worlds easiest languages for writing exponential-time algorithms!

---

### 2.1.2 Guess and Verify

A better strategy in Prolog is to use what information you have to make a reasonable guess about the value of some variable, then check to see if that guess is correct.

A good example is

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

- The `prereq` in the second rule is a "guess", based upon the available list of facts.
- The remaining `mustTakeBefore` determines whether that guess is valid.

---

Note that this is much better than

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

A characteristic of a good "guess" is that it should have fine (preferably small) possible solutions.

---

## 2.2 Dare to Fail!

We've already seen failure used in some creative ways to induce iteration.

Consider this as a way of generating guesses for a generate-and-test algorithm:

---

```
generate(0).
generate(X) :- generate(Y), X is Y+1.


squares :- generate(I),
           J is I*I,
           write(J), nl,
           I>99.
```

This prints the first 101 integer squares. Why?

---

---

## 2.3 Open Lists

Sometimes it's useful to have data structures that have variables in them as "holes" to be filled in later.

An example is the **open list**, a list ending in a variable.

```
L = [a,b|X].
L = [a,b|X], X=[c|Y].
```

- Note how the binding of `X` effectively appends to `L`, something that normally we can't do.
- It's almost like an assignment to an existing variable.

---

### 2.3.1 A Queue in Prolog

We can implement queues in Prolog as `q(X,Y)` where `X` is an open list and `Y` is the final variable in that open list.

```
setup(q(X,X)).

enter(A, q(X,Y), q(X,Z)) :- Y = [A|Z].

front(A, q(X,X)) :- !, fail.
front(A, q([A|_],Y)).

leave(q(X,X), _) :- !, fail.
leave(q([_|X],Y), q(X,Y)).
```

Sample use:

```
setup(Q), enter(a, Q, Q2),
  enter(b, Q2, Q3), front(X,Q3).
```

---

# 3 Closing thoughts

- Prolog offers an interesting model that works in some very unconventional situations
- Does not scale up well to large projects.
- Very useful in contexts where some/all of the code will be machine-generated.
  - examples: software testing
- Prolog variants add pure numerics, reduce order dependencies, etc.