Introduction to Prolog

•Useful references:

–Clocksin, W.F. and Mellish, C.S., Programming in Prolog: Using the ISO Standard (5th edition), 2003.

–Bratko, I., Prolog Programming for Artificial Intelligence (3rd edition), 2001.

–Sterling, L. and Shapiro, E., The Art of Prolog (Second edition), 1994.

PROLOG

Reviews

- A Prolog program consists of predicate definitions.
- A predicate denotes a property or relationship between objects.
- Definitions consist of clauses.
- A clause has a head and a body (Rule) or just a head (Fact).
- A head consists of a predicate name and arguments.
- A clause body consists of a conjunction of terms.
- Terms can be constants, variables, or compound terms.
- We can set our program goals by typing a command that unifies with a clause head.
- A goal unifies with clause heads in order (top down).
- Unification leads to the instantiation of variables to values.
- If any variables in the initial goal become instantiated this is reported back to the user.

PROLOG

Tests

- When we ask Prolog a question we are asking for the interpreter to prove that the statement is true.
  `?- 5 < 7, integer(bob).`
  `yes` = the statement can be proven.
  `no` = the proof failed because either
    - the statement does not hold, or
    - the program is broken.

  `Error` = there is a problem with the question or program.
  nothing = the program is in an infinite loop.
- We can ask about:
  - Properties of the database: `mother(jane,alan).`
  - Built-in properties of individual objects: `integer(bob).`
  - Relationships between objects:
    - Unification
    - Arithmetic relationships: <, >, =<, >=, =\=, +, -, *, /

PROLOG

Arithmetic Operators

- Operators for arithmetic and value comparisons are built-in to Prolog
  = always accessible / don't need to be written
- Comparisons: <, >, =<, >=, =:=, =\= (not equals)
  = Infix operators: go between two terms.
  =< is used
    - `5 =< 7.` (infix)
    - `=<(5,7).` (prefix) ← all infix operators can also be prefixed
- Equality is different from unification
  = checks if two terms unify
  =:= compares the arithmetic value of two expressions

```
?- X=Y.    ?- X=:=Y.               ?-X=4,Y=3, X+2 =:= Y+3.
yes        Instantiation error      X=4, Y=3? yes
```

PROLOG

Arithmetic Operators (2)

- Arithmetic Operators: +, -, *, /
  = Infix operators but can also be used as prefix.
  – Need to use `is` to access result of the arithmetic expression otherwise it is treated as a term:

```
 |?- X = 5+4.              |?- X is 5+4.
 X = 5+4 ?                  X = 9 ?
 yes                        yes
(Can X unify with 5+4?)    (What is the result of 5+4?)
```

- Mathematical precedence is preserved: /, *, before +,-
- Can make compound sums using round brackets
  – Impose new precedence
  – Inner-most brackets first

```
| ?- X is (5+4)*2.
X = 18 ?
yes
```

PROLOG

Tests within clauses

- These operators can be used within the body of a clause:
  – To manipulate values,

```
sum(X,Y,Sum):-
   Sum is X+Y.
```

  – To distinguish between clauses of a predicate definition

```
bigger(N,M):-
   N < M, write('The bigger number is '), write(M).
bigger(N,M):-
   N > M, write('The bigger number is '), write(N).
bigger(N,M):-
   N =:= M, write('Numbers are the same').
```

PROLOG

Backtracking

```
|?- bigger(5,4).

bigger(N,M):-
     N < M,
     write('The bigger number is '), write(M).
bigger(N,M):-
     N > M,
     write('The bigger number is '), write(N).
bigger(N,M):-
     N =:= M,
     write('Numbers are the same').
```


Backtracking

```
|?- bigger(5,4).

bigger(5,4):-
     5 < 4,      ← fails        Backtrack
     write('The bigger number is '), write(M).
bigger(N,M):-
     N > M,
     write('The bigger number is '), write(N).
bigger(N,M):-
     N =:= M,
     write('Numbers are the same').
```


Backtracking

```
|?- bigger(5,4).

bigger(N,M):-
     N < M,
     write('The bigger number is '), write(M).
bigger(5,4):-
     5 > 4,
     write('The bigger number is '), write(N).
bigger(N,M):-
     N =:= M,
     write('Numbers are the same').
```


Backtracking

```
|?- bigger(5,4).

   bigger(N,M):-
      N < M,
      write('The bigger number is '), write(M).
   bigger(5,4):-
      5 > 4,  ← succeeds, go on with body.
      write('The bigger number is '), write(5).

The bigger number is 5                 Reaches full-stop
yes                                    = clause succeeds
|?-
```


Backtracking

```
|?- bigger(5,5).  ← If our query only matches the final clause

   bigger(N,M):-
      N < M,
      write('The bigger number is '), write(M).
   bigger(N,M):-
      N > M,
      write('The bigger number is '), write(N).
   bigger(5,5):-
      5 =:= 5,  ← Is already known as the first two clauses failed.
      write('Numbers are the same').
```


Backtracking

```
|?- bigger(5,5).  ← If our query only matches the final clause

   bigger(N,M):-
      N < M,
      write('The bigger number is '), write(M).
   bigger(N,M):-
      N > M,
      write('The bigger number is '), write(N).
   bigger(5,5):-
                ← Satisfies the same conditions.
      write('Numbers are the same').

Numbers are the same
yes
```

Clauses should be ordered according to specificity
Most specific at top ⟷ Universally applicable at bottom

Reporting Answers

```
|?- bigger(5,4).           ← Question is asked
The bigger number is 5     ← Answer is written to terminal
yes                        ← Succeeds but answer is lost
```

- This is fine for checking what the code is doing but not for using the proof.

```
|?- bigger(6,4), bigger(Answer,5).
Instantiation error!
```

- To report back answers we need to
  - put an uninstantiated variable in the query,
  - instantiate the answer to that variable when the query succeeds,
  - pass the variable all the way back to the query.

PROLOG

Satisfying Subgoals

- Most rules contain calls to other predicates in their body. These are known as Subgoals.
- These subgoals can match:
  - facts,
  - other rules, or
  - the same rule = a recursive call

```
1) drinks(alan,beer).
2) likes(alan,coffee).
3) likes(heather,coffee).

4) likes(Person,Drink):-
       drinks(Person,Drink).    ← a different subgoal
5) likes(Person,Somebody):-
       likes(Person,Drink),     ← recursive subgoals
       likes(Somebody,Drink).   ←
```

PROLOG

Passing Back Answers

- To report back answers we need to
  1. put an uninstantiated variable in the query,

```
| ?- bigger(6,4,Answer),bigger(Answer,5,New_answer).

     bigger(X,Y,Answer):- X>Y, Answer = X.
     bigger(X,Y,Answer):- X=<Y, Answer = Y.
```

  2. instantiate the answer to that variable when the query succeeds,
  3. pass the variable all the way back to the query.

PROLOG

Central Ideas of Prolog

- SUCCESS/FAILURE
  - any computation can "**succeed**" or "**fail**", and this is used as a '**test**' mechanism.
- MATCHING
  - any two data items can be compared for similarity, and values can be bound to variables in order to allow a match to succeed.
- SEARCHING
  - the whole activity of the Prolog system is to search through various options to find a combination that succeeds.
    - Main search tools are backtracking and recursion
- BACKTRACKING
  - when the system fails during its search, it returns to previous choices to see if making a different choice would allow success.

PROLOG

Head Unification

- To report back answers we need to
  1. put an uninstantiated variable in the query,

```
| ?- bigger(6,4,Answer),bigger(Answer,5,New_answer).

  bigger(X,Y,X):- X>Y.
  bigger(X,Y,Y):- X=<Y.
```

Or, do steps 2 and 3 in one step by naming the variable in the head of the clause the same as the correct answer.

= head unification

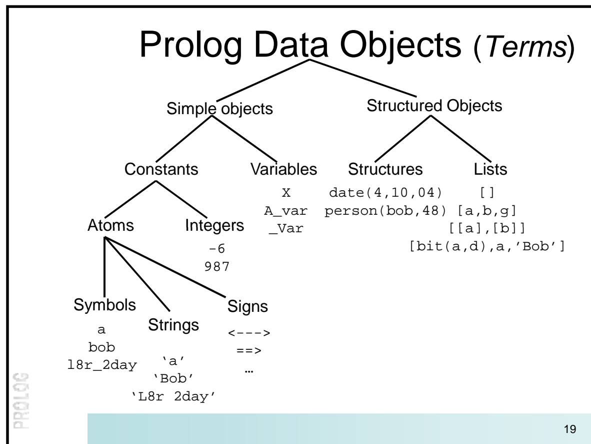
PROLOG

Why use recursion?

- It allows us to define very clear and elegant code.
  - Why repeat code when you can reuse existing code.
- Relationships may be recursive
  e.g. "X is my ancestor if X is my parent's ancestor."
- Data is represented recursively and best processed iteratively.
  - Grammatical structures can contain themselves
  - Ordered data: each element requires the same processing
- Allows Prolog to perform complex search of a problem space without any dedicated algorithms.

PROLOG

Prolog Data Objects (*Terms*)



PROLOG

Structures

- To create a single data element from a collection of related terms we use a *structure*.
- A structure is constructed from a *function name* (functor) and one of more components.

functor

```
somerelationship(a,b,c,[1,2,3])
```

- The components can be of any type: atoms, integers, variables, or structures.
- As functors are treated as data objects just like constants they can be unified with variables

```
|?- X = date(03,31,05).
X = date(03,31,05)?
yes
```

PROLOG

Lists

- A collection of ordered data.
- Has *zero* or more elements enclosed by square brackets ('[ ]') and separated by commas (',').

```
[a]                ← a list with one element
[]                 ← an empty list
```

1 2 3 (1 2)

```
[34,tom,[2,3]]     ← a list with 3 elements where the
                     3rd element is a list of 2 elements.
```

- Like any object, a list can be unified with a variable

```
|?- [Any, list, 'of elements'] = X.
X = [Any, list, 'of elements']?
yes
```

PROLOG

List Unification

- Two lists unify if they are the same length and all their elements unify.

```
|?-[a,B,c,D]=[A,b,C,d].
A = a,
B = b,
C = c,
D = d ?
yes
```

```
|?-[(a+X),(Y+b)]=[(W+c),(d+b)].
W = a,
X = c,
Y = d?
yes
```

```
|?- [[X,a]]=[b,Y].
no
```

Length 1 Length 2

```
|?-[[a],[B,c],[]]=[X,[b,c],Y].
B = b,
X = [a],
Y = [] ?
yes
```

PROLOG

Definition of a List

- Lists are *recursively defined* structures.

"An empty list, [], is a list.
A structure of the form [X, …] is a list if X is a term and […] is a list, possibly empty."

- This recursiveness is made explicit by the bar notation
  - `[Head|Tail]` ('|' = bottom left PC keyboard character)
- Head must unify with a single term.
- Tail unifies with a list of any length, including an empty list, [].
  - the bar notation turns everything after the Head into a list and unifies it with Tail.

PROLOG

Head and Tail

```
|?-[a,b,c,d]=[Head|Tail].
Head = a,
Tail = [b,c,d]?
yes
```

```
|?-[a,b,c,d]=[X|[Y|Z]].
X = a,
Y = b,
Z = [c,d];
yes
```

```
|?-[a] = [H|T].
H = a,
T = [];
yes
```

```
|?-[a,b,c]=[W|[X|[Y|Z]]].
W = a,
X = b,
Y = c,
Z = []? yes
```

```
|?-[] = [H|T].
no
```

```
|?-[a|[b|[c|[]]]]= List.
List = [a,b,c]?
yes
```

PROLOG

Identifying a list

- Last lecture we introduced lists: `[a,[],green(bob)]`
- We said that lists are *recursively defined* structures:

"An empty list, [ ], is a list.
A structure of the form [X, …] is a list if X is a term and […] is a list, possibly empty."

- This can be tested using the Head and Tail notation, [H|T], in a recursive rule.

```
is_a_list([]).            ← A term is a list if it is an empty list.
is_a_list([_|T]):-        ← A term is a list if it has two
    is_a_list(T).           elements and the second is a list.
```

PROLOG

Base and Recursive Cases

- A recursive definition, whether in prolog or some other language (including English!) needs two things.
- A definition of when the recursion *terminates*.
  - Without this the recursion would never stop!
  - This is called the *base case:* `is_a_list([]).`
  - *Almost always comes before recursive clause*
- A definition of how we can define the problem in terms of a similar, smaller problem.
  - This is called the *recursive case*: `is_a_list([_|T]):- is_a_list(T).`
- There might be more than one base or recursive case.

PROLOG

Focussed Recursion

- To ensure that the predicate terminates, the recursive case must move the problem closer to a solution.
  - If it doesn't it will loop infinitely.
- With list processing this means stripping away the Head of a list and recursing on the Tail.

```
is_a_list([_|T]):-
    is_a_list(T).
```

Head is replaced with an underscore as we don't want to use it.

- The same focussing has to occur when recursing to find a property or fact.

```
is_older(Ancestor,Person):-
    is_older(Someone,Person),
    is_older(Ancestor,Someone).
```

Doesn't focus → `is_older(Someone,Person),`

PROLOG

Focussed Recursion (2)

Given this program:

```
parent(tom,jim).
parent(mary,tom).

is_older(Old,Young):-
   parent(Old,Young).

is_older(Ancestor,Young):-
   is_older(Someone,Young),
   is_older(Ancestor,Someone).
```

- A query looking for all solutions will loop.

```
|?-is_older(X,Y).
  X = tom,
  Y = jim ? ;
  X = mary,
  Y - tom ? ;
  X = mary,
  Y = jim ? ;
  *loop*
```

It loops because the recursive clause does not focus the search it just splits it. If the recursive `is_older/2` doesn't find a parent it just keeps recursing on itself

PROLOG

Focussed Recursion (3)

The correct program:

```
parent(tom,jim).
parent(mary,tom).

is_older(Old,Young):-
   parent(Old,Young).

is_older(Ancestor,Young):-
   parent(Someone,Young),
   is_older(Ancestor,Someone).
```

- Can generate all valid matches without looping.

```
|?-is_older(X,Y).
  X = tom,
  Y = jim ? ;
  X = mary,
  Y - tom ? ;
  X = mary,
  Y = jim ? ;
  no
```

To make the problem space smaller we need to check that `Young` has a parent before recursion. This way we are not looking for something that isn't there.

PROLOG

Quick Aside: Tracing Prolog

- To make Prolog show you its execution of a goal type `trace.` at the command line.
  - Prolog will show you:
    - which goal is `Call`ed with which arguments,
    - whether the goal succeeds (`Exit`),
    - has to be `Redo`ne, or `Fail`s.
  - The tracer also indicates the level in the search tree from which a goal is being called.
    - The number next to the goal indicates the level in the tree (top level being 0).
    - The leftmost number is the number assigned to the goal (every new goal is given a new number).
- To turn off the tracer type `notrace`.

PROLOG

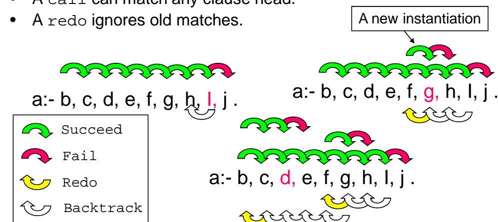
Tracing Member/2

```
| ?- trace.
| ?- member(ringo,[john,paul,ringo,george]).
  1     1 Call: member(ringo,[john,paul,ringo,george]) ?
  2     2 Call: member(ringo,[paul,ringo,george]) ?
  3     3 Call: member(ringo,[ringo,george]) ?
  3     3 Exit: member(ringo,[ringo,george]) ?
  2     2 Exit: member(ringo,[paul,ringo,george]) ?
  1     1 Exit: member(ringo,[john,paul,ringo,george]) ?
yes

| ?- member(stuart,[john,paul,ringo,george]).
  1     1 Call: member(ringo,[john,paul,ringo,george]) ?
  2     2 Call: member(ringo,[paul,ringo,george]) ?
  3     3 Call: member(ringo,[ringo,george]) ?
  4     4 Call: member(stuart,[george]) ?
  5     5 Call: member(stuart,[]) ?   ← [ ] does not match [H|T]
  5     5 Fail: member(stuart,[]) ?
  4     4 Fail: member(stuart,[george]) ?
  3     3 Fail: member(ringo,[ringo,george]) ?
  2     2 Fail: member(ringo,[paul,ringo,george]) ?
  1     1 Fail: member(ringo,[john,paul,ringo,george]) ?
no
```

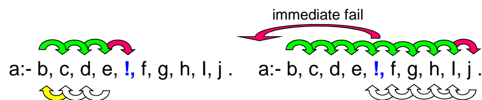

Prolog's Persistence

- When a sub-goal fails, Prolog will backtrack to the most recent successful goal and try to find another match.
- Once there are no more matches for this sub-goal it will backtrack again; retrying every sub-goal before failing the parent goal.
- A `call` can match any clause head.
- A `redo` ignores old matches.



Cut !

- If we want to restrict backtracking we can control which sub-goals can be redone using the cut **!**.
- We use it as a goal within the body of clause.
- It succeeds when `call`ed, but `fail`s the parent goal (the goal that matched the head of the clause containing the cut) when an attempt is made to `redo` it on backtracking.
- It commits to the choices made so far in the predicate.
  - unlimited backtracking can occur before and after the cut but no backtracking can go through it.



Failing the parent goal

a:- b, c, d, e, !, f, g, h, I, j .
a:- k.
a:- m .

a:- b, c, d, e, !, f, g, h, I, j .
Treated as if don't exist: a:- k. / a:- m .
This clause and these choices committed to

- The cut succeeds when it is `call`ed and commits the system to all choices made between the time the parent goal was invoked and the cut.
- This includes committing to the clause containing the cut.
  = the goal can only succeed if this clause succeeds.
- When an attempt is made to backtrack through the cut
  - the clause is immediately failed, and
  - no alternative clauses are tried.

Mutually Exclusive Clauses

- We should only use a cut if the clauses are mutually exclusive (if one succeeds the others won't).
- If the clauses are mutually exclusive then we don't want Prolog to try the other clauses when the first fails
  = redundant processing.
- By including a cut in the body of a clause we are committing to that clause.
  - Placing a cut at the start of the body commits to the clause as soon as head unification succeeds.
    ```
    a(1,X):- !, b(X), c(X).
    ```
  - Placing a cut somewhere within the body (even at the end) states that we cannot commit to the clause until certain sub-goals have been satisfied.
    ```
    a(_,X):- b(X), c(X), !.
    ```


Mutually Exclusive Clauses (2)

```
f(X,0):- X < 3.
f(X,1):- 3 =< X, X < 6.
f(X,2):- 6 =< X.
```

```
|?- trace, f(2,N).
   1      1 Call: f(2,_487) ?
   2      2 Call: 2<3 ?
   2      2 Exit: 2<3 ? ?
   1      1 Exit: f(2,0) ?
N = 0 ? ;
   1      1 Redo: f(2,0) ?
   3      2 Call: 3=<2 ?
   3      2 Fail: 3=<2 ?
   4      2 Call: 6=<2 ?
   4      2 Fail: 6=<2 ?
   1      1 Fail: f(2,_487) ?
no
```

Green Cuts !

```
f(X,0):- X < 3, !.
f(X,1):- 3 =< X, X < 6, !.
f(X,2):- 6 =< X.
```

If you reach this point don't bother trying any other clause.

```
|?- trace, f(2,N).
   1      1 Call: f(2,_487) ?
   2      2 Call: 2<3 ?
   2      2 Exit: 2<3 ? ?
   1      1 Exit: f(2,0) ?
N = 0 ? ;
no
```

- Notice that the answer is still the same, with or without the cut.
  - This is because the cut does not alter the logical behaviour of the program.
  - It only alters the procedural behaviour: specifying which goals get checked when.
- This is called a *green cut.* It is the correct usage of a cut.
- Be careful to ensure that your clauses are actually mutually exclusive when using green cuts!

Red Cuts !

```
f(X,0):- X < 3, !.
f(X,1):- 3 =< X, X < 6, !.
f(X,2):- 6 =< X.
```

Redundant?

```
| ?- f(7,N).
   1      1 Call: f(7,_475) ?
   2      2 Call: 7<3 ?
   2      2 Fail: 7<3 ?
   3      2 Call: 3=<7 ?
   3      2 Exit: 3=<7 ?
   4      2 Call: 7<6 ?
   4      2 Fail: 7<6 ?
   5      2 Call: 6=<7 ?
   5      2 Exit: 6=<7 ?
   1      1 Exit: f(7,2) ?
N = 2 ?
yes
```

- Because the clauses are mutually exclusive and ordered we know that once the clause above fails certain conditions must hold.
- We might want to make our code more efficient by removing superfluous tests.

Red Cuts !

```
f(X,0):- X < 3, !.
f(X,1):- X < 6, !.
f(X,2).
```

```
f(X,0):- X < 3.
f(X,1):- X < 6.
f(X,2).
```

```
| ?- f(7,N).
   1      1 Call: f(7,_475) ?
   2      2 Call: 7<3 ?
   2      2 Fail: 7<3 ?
   3      2 Call: 7<6 ?
   3      2 Fail: 7<6 ?
   1      1 Exit: f(7,2) ?
N = 2 ?
yes
```

```
| ?- f(1,Y).
   1      1 Call: f(1,_475) ?
   2      2 Call: 1<3 ?
   2      2 Exit: 1<3 ? ?
   1      1 Exit: f(1,0) ?
   Y = 0 ? ;
   1      1 Redo: f(1,0) ?
   3      2 Call: 1<6 ?
   3      2 Exit: 1<6 ? ?
   1      1 Exit: f(1,1) ?
   Y = 1 ? ;
   1      1 Redo: f(1,1) ?
   1      1 Exit: f(1,2) ?
   Y = 2 ?
   yes
```


Using the cut

- *Red cuts* change the logical behaviour of a predicate.
- TRY NOT TO USE RED CUTS!
- Red cuts make your code hard to read and are dependent on the specific ordering of clauses (which may change once you start writing to the database).
- If you want to improve the efficiency of a program use *green cuts* to control backtracking.
- Do not use cuts in place of tests.

To ensure a logic friendly cut either:

```
p(X):- test1(X), !, call1(X).
p(X):- test2(X), !, call2(X).
p(X):- testN(X), !, callN(X).
```

`testI` predicates are mutually exclusive.

```
p(1,X):- !, call1(X).
p(2,X):- !, call2(X).
p(3,X):- !, callN(X).
```

The mutually exclusive tests are in the head of the clause.

A larger example.

We'll define several versions of the disjoint partial map split.

split(*list of integers*, *non-negatives*, *negatives*).

1. A version not using cut. Good code (each can be read on its own as a fact about the program). Not efficient because choice points are retained. The first solution is desired, but we must ignore backtracked solutions.

```
split([], [], []).
split([H|T], [H|Z], R) :- H >= 0, split(T, Z, R).
split([H|T], R, [H|Z]) :- H < 0, split(T, R, Z).
```


A larger example.

2. A version using cut. Most efficient, but not the best 'defensively written' code. The third clause does not stand on its own as a fact about the problem. As in normal programming languages, it needs to be read in context. This style is often seen in practice, but is deprecated.

```
split([], [], []).
split([H|T], [H|Z], R) :- H >= 0, !, split(T, Z, R).
split([H|T], R, [H|Z]) :- split(T, R, Z).
```

Minor modifications (adding more clauses) may have unintended effects. Backtracked solutions invalid.

A larger example.

3. A version using cut which is also ‘safe’. The only inefficiency is that the goal H < 0 will be executed unnecessarily whenever H < 0.

```
split([], [], []).
split([H|T], [H|Z], R) :- H >= 0, !, split(T, Z, R).
split([H|T], R, [H|Z]) :- H < 0, split(T, R, Z).
```

Recommended for practical use. Hidden problem: the third clause does not capture the idea that
H < 0 is a committal. Here committal is the default because H < 0 is in the last clause. Some new compilers detect that H < 0 is redundant.

A larger example.

3. A version with unnecessary cuts

```
split([], [], []) :- !.
split([H|T], [H|Z], R) :- H >= 0, !, split(T, Z, R).
split([H|T], R, [H|Z]) :- H < 0, !, split(T, R, Z).
```

First cut unnecessary because anything matching first clause will not match anything else anyway. Most Prolog compilers can detect this.

Why is the third cut unnecessary? Because H<0 is in the last clause. Whether or not H<0 fails, there are no choices left for the caller of split. However, the above will work for any order of clauses.

Negation as Failure

Using cut together with the built-in predicate fail, we may define a kind of negation. Examples:

Mary likes any animals except reptiles:

```
likes(mary, X) :- reptile(X), !, fail.
likes(mary, X) :- animal(X).
```

A utility predicate meaning something like “not equals”:

```
different(X, X) :- !, fail.
different(_,_).
```


Negation as Failure

We can use the same idea of “cut fail” to define the predicate not, which takes a term as an argument. not will “call” the term, that is evaluate it as though it is a goal:

not(G) fails if G succeeds

not(G) succeeds if G does not succeed.

In Prolog,

```
not(G) :- call(G), !, fail.
not(_).
```

Call is a built-in predicate.

Negation as Failure

Most Prolog systems have a built-in predicate like not. SICStus Prolog calls it \+. Remember, not does not correspond to logical negation, because it is based on the success/failure of goals. It can, however, be useful:

```
likes(mary, X) :- not(reptile(X)).

different(X, Y) :- not(X = Y).
```


Negation as Failure can be Misleading

Once upon a time, a student who missed some of these lectures was commissioned to write a Police database system in Prolog. The database held the names of members of the public, marked by whether they are innocent or guilty of some offence. Suppose the database contains the following:

```
innocent(peter_pan).
innocent(X) :- occupation(X, nun).
innocent(winnie_the_pooh).
innocent(julie_andrews)
guilty(X) :- occupation(X, thief).
guilty(joe_bloggs).
guilty(rolf_harris).
```

Consider the following dialogue:

```
?- innocent(st_francis).
no.
```

Problem.

This cannot be right, beause everyone knows that St Francis is innocent. But in Prolog the above happens because st_francis is not in the database. Because the database is hidden from the user, the user will believe it because the computer says so.

How to solve this?

PROLOG

not makes things worse

Using not will not help you. Do not try to remedy this by defining:

```
guilty(X) :- not(innocent(X)).
```

This is useless, and makes matters even worse:

```
?- guilty(st_francis).
```

yes

It is one thing to show that st_francis cannot be demonstrated to be innocent. But it is quite another thing to incorrectly show that he is guilty.

PROLOG

Negation-by-failure can be non-logical

Some disturbing behaviour even more subtle than the innocent/guilty problem, and can lead to some extremely obscure programming errors. Here is a restaurant database:

```
good_standard(goedels).
good_standard(hilberts).
expensive(goedels).
reasonable(R) :- not(expensive(R)).
```

Consider the following dialogue:

?- good_standard(X), reasonable(X).

X = hilberts

But if we ask the logically equivalent question:

?- reasonable(X), good_standard(X).

no.

PROLOG

Question

Why do we get different answers for what seem to be logically equivalent queries?

The difference between the questions is as follows.

In the first question, the variable X is always instantiated when reasonable(X) is executed.

In the second question, X is not instantiated when reasonable(X) is executed.

The semantics of reasonable(X) differ depending on whether its argument is instantiated.

PROLOG

Not a Good Idea!

It is bad practice to write programs that destroy the correspondence between the logical and procedural meaning of a program without any good reason for doing so.

Negation-by-failure does not correspond to logical negation, and so requires special care.

PROLOG

How to fix it?

One way is to specify that negation is undefined whenever an attempt is made to negate a non-ground formula.

A formula is ‘ground’ if is has no unbound variables.

Some Prolog systems issue a run-time exception if you try to negate a non-ground goal.

PROLOG

Clauses and Databases

In a relational database, relations are regarded as tables, in which each element of an *n*-ary relation is stored as a row of the table having *n* columns.

supplier

jones	chair	red	10
smith	desk	black	50

Using clauses, a table can be represented by a set of unit clauses. An *n*-ary relation is named by an *n*-ary predicate symbol.

```
supplier(jones, chair, red, 10).
supplier(smith, desk, black, 50).
```


Clauses and Databases

Advantages of using clauses:

1. Rules as well as facts can coexist in the description of a relation.
2. Recursive definitions are allowed.
3. Multiple answers to the same query are allowed.
4. There is no role distinction between input and output.
5. Inference takes place automatically.

Negation and Representation

Like databases, clauses cannot represent negative information. Only true instances are represented.

The battle of Waterloo occurred in 1815.

How can we show that the battle of Waterloo did not take place in 1923? The database cannot tell us when something is not the case, unless we do one of the following:

1. 'Complete' the database by adding clauses to specify the battle didn't occur in 1814, 1813, 1812, ..., 1816, 1817, 1818,...
2. Add another clause saying the battle did not take place in another year (the battle occurred in *and only in* 1815).
3. Make the 'closed world assumption', implemented by 'negation by failure'.

Summary

- Controlling backtracking: the cut **!**
  - Efficiency: avoids needless REDO-ing which cannot succeed.
  - Simpler programs: conditions for choosing clauses can be simpler.
  - Robust predicates: definitions behave properly when forced to REDO.
- Green cut = cut doesn't change the predicate logic = **good**
- Red cut = without the cut the logic is different = **bad**
- Cut – fail: when it is easier to prove something is false than true.