




## Prolog

### Chapter 15 in your text

*also many books and articles on Prolog,  
if you're interested in doing more*



## Implementing Logical Systems

- ◆ Prolog is the most well-known logic-based programming language
- ◆ The syntax will look very familiar now that you've seen clause form, but it embodies most of what you can say in FOL.
- ◆ Prolog is available free for most platforms – sbprolog for unix (just type 'prolog'), amzi prolog for Windows or Linux
- ◆ We write files of logic code in Prolog, then consult them and query the logical system. Prolog uses resolution to prove our query, returning variables bound during this process.<sub>2</sub>

## Prolog

- ◆ We're going to do prolog in two parts: basic, and more advanced
- ◆ The motivation for seeing some prolog now is so that you can see the idea of symbol manipulation logically and do some basic work with it & move away from just doing stuff on paper
- ◆ However, there is still a lot to see as far as basic AI goes, and I don't want you to lose sight of it and think that logical representation is all there is to it!

## Prolog

- ◆ Programming in Logic - based on first-order logic)
- ◆ Illustrates the computational manipulation of symbols
- ◆ You could do all of this in other symbol manipulation languages like LISP
- ◆ We're looking at Prolog, however, because it lets you do basic things faster – it includes a resolution search algorithm, for example, so that you don't have to write one
- ◆ Concentrate on defining knowledge rather than building manipulation facilities

## Knowledge Rep. In Prolog

- ◆ The facts that are known about a problem domain are defined using predicates:  
male(bill).  
female(sally).  
parent(bill, sally).
- ◆ The order of the objects in a predicate obviously has meaning and must be used consistently
- ◆ Predicates are analogous to relations in relational database systems

5

## Predicates

- ◆ As A1 will hopefully show you, the selection of an appropriate set of predicates is not trivial
- ◆ a well organized set of predicates can make problem solving much easier than if a poorly organized set of predicates is defined  
parent(bill, jane, sally).
- ◆ bill is the father and jane is the mother of sally – not smart...saying too much in one predicate makes it much less extendable (and even less useful in many cases)
- ◆ Remember to focus on generality when doing prolog assignments!

6

## Predicates

- ◆ Much more suitable:  
father(bill, sally).  
mother(jane, sally).  
...and the addition of another fact that says that a parent in general can be a mother or a father.
- ◆ It is impossible to define all possible predicates for all combinations of objects in a complex system – that's why it's a model!
- ◆ Obviously much of our knowledge is in the form of more generally applicable rules, such as the parent rule we mentioned above

7

## Rules in Prolog

- ◆ When you think of the term *rule*, you usually think of an implication: if we have X, then we have Y
- ◆ No different in Prolog – we define rules using implication that make generic statements about the objects in a system  
e.g. the rule if (male(X) and parent(X,Y)) then father(X,Y)
- ◆ In Prolog this would look like:  
father(X,Y) :- male(X), parent(X,Y).
- ◆ In a logical system what would we have to do to X and Y?
- ◆ They would have to be universally quantified...

8

## Prolog Rules

- ◆ `father(X,Y) :- male(X), parent(X,Y).`
- ◆ All variables used in Prolog statements are assumed to be universally quantified (all part of clause form used internally!)
- ◆ The `:-` is implication in prolog, but we list the concluded implication before it, and the items that make the implication possible after
- ◆ This is done in a reverse manner because prolog cares about the goal we're looking for so keeps track of statements to that end
- ◆ The comma indicates an AND operation

9

## Deductions

- ◆ Prolog can make deductions by unifying variables and constants much as we did by hand:

```
parent(bill,sally).  
male(bill).  
father(X,Y) :- male(X), parent(X,Y).
```

```
∴ father(bill,sally).
```

10

## Prolog

- ◆ Prolog can also be used to mimic propositional logic if we remove the use of variables
- ◆ not terribly interesting then though: the programmer must explicitly define all possible fact relations

```
male(bill).  
female(sally).  
parent(bill, sally).  
parent(jane, sally).  
father(bill, sally).  
mother(jane,sally).  
sister(jane,sue).  
aunt(sue,sally).  
sister_in_law(bill,sue).  
...
```

11

## Prolog

- ◆ If facts and rules are used, then only the base facts are required

```
male(bill).  
female(sally).  
parent(bill, sally).  
parent(jane, sally).  
parent(jill, jane).  
parent(jill, sue).
```
- ◆ The additional information is then defined using rules

```
father(X,Y) :- male(X), parent(X,Y).
```

12

## Prolog Syntax

- ◆ **constant**: first letter lower case; represents a specific object or symbol
- ◆ **variable**: first letter upper case; represents a generic object or symbol
- ◆ **predicate**: lower case; defines a fact about an object or a relationship between/among objects

13

## Connectives

- `:-` implies ( $\leftarrow$ )
- `;` OR
- `,` AND
- `.` end-of-statement

`%` comment to end of line  
`/*` arbitrary comment `*/`

14

## Prolog Program

- ◆ Prolog programs are stored in text files and can be edited by any simple text editor

```
male(bill).  
female(sally).  
parent(bill,sally).  
father(X,Y):- male(X), parent(X,Y).
```

- ◆ Amzi prolog has a complete environment for editing

15

## Using Prolog

- ◆ The standard prompt in most versions of Prolog is `?-`
  - actually the operator for making a query – in some you have to type it
  - In those where it's the prompt everything you type there has to be a query
- ◆ The Prolog source file is loaded into the Prolog system using the `consult` command
  - `?- consult(myfile).` (single quotes around filename in sb-prolog; it's a menu choice in amzi)
  - ❖ You're actually giving the system a goal to consult the file (shades of procedural coding!)

16

## Using Prolog

- ◆ The Prolog system is terminated using the halt command: `?- halt.` (AMZI: `?- quit.`)
- ◆ Prolog source files may include an extension
- ◆ Unix (sb) prolog does not require an extension
  - `?- consult(test).`
  - `?- consult('test.pro').`
- ◆ Amzi prolog uses the extension `.pro` to associate the source file with the ALE (Amzi Logic Explorer) system

17

## Editing Files

- ◆ If you aren't using amzi, use a couple of terminal sessions to allow you to switch from Prolog to the editor to make changes to the source file
- ◆ Save the source file before returning to Prolog, then consult the file again
- ◆ There is a reconsult that is supposed to erase old predicates and put in new ones, but it is flaky in sb-prolog

18

## Amzi Prolog Basics

- ◆ Open it up
- ◆ start a listener: Select the Start command from the Listener menu
- ◆ Consult consults a file
- ◆ `halt.` kills the listener but does not stop the system
- ◆ Use the file exit menu command to stop the system

19

## Unix Prolog Basics

- ◆ SB Prolog is available on the Unix systems
  - `% prolog`
  - `?- consult(filename).`
  - `?- halt.`
- ◆ Use an X-windows terminal to edit the source file and run Prolog at the same time, or open up two separate terminal sessions using a terminal emulator
- ◆ Remember to save the source before consulting again when you make changes
- ◆ There are no man pages, but your text is a good reference. Lots of Prolog Books around too.

20

## Making Queries in Prolog

- ◆ Prolog can answer questions concerning the symbols and symbol expressions that have been defined
- ◆ Assume we have our earlier facts and rules encoded in Prolog, and ask:  
?- male(bill).
- ◆ Is bill male?
- ◆ Prolog examines the facts using resolution and responds with YES or NO
- ◆ In this case, the answer is YES

21

## Asking Questions

- ?- father(bill,sally).
- ◆ Is bill the father of sally?
  - ◆ Prolog examines the facts and can not find the explicit fact that indicates that bill is the father of sally
  - ◆ However, there is a rule that permits Prolog to conclude that bill is the father of sally
  - ◆ Prolog responds with YES

22

## Asking Questions

- ?- male(sally).
- ◆ Is sally male?  
No.
  - ◆ Prolog can not find a fact or a rule that indicates that Sally is male so Prolog concludes that the answer is No.
  - ◆ Note that Prolog does not prove that male(sally) is not true – *No only means it couldn't be proven given the knowledge in the system*

23

## Instantiation

- ?- father(Name,sally).
- ◆ Who is sally's father?
  - ◆ Name is a variable that does not have a value
  - ◆ Prolog is able to find a value for the variable Name (Name is instantiated) that causes the statement to be true  
Name = bill.
  - ◆ If there is no value that causes the statement to be true, Prolog returns the value No

24

## Instantiation and the Anonymous Vble

- ◆ Who has bill as a father?  
?-father(bill,Name).  
Name = sally.
  - prolog returns the substitution required to unify the solution with your query
- ◆ Does bill have a child?  
?-father(bill, \_).  
Yes.
- ◆ The underscore represents an “anonymous” variable that is instantiated by Prolog but the value is not displayed
- ◆ We use it when we don’t care about remembering what’s actually there, just if there’s something that fits or not!

25

## Expressions

- ◆ Who is sally's grandfather?
- ◆ I can make this query using the rules already in the system if I want to:  
?- father(X,Y), father(Y,sally).
- ◆ But it's useful enough to use in a rule!  
grandfather(X,Z):- father(X,Y), father(Y,Z).
- ◆ Here you see that rules can introduce intermediate symbols
- ◆ Notice that this rule is incomplete – it identifies only the paternal grandfather and ignores the maternal grandfather

26

## Two Possible Fixes

```
grandfather(X,Z):- (father(X,Y), father(Y,Z));  
                  (father(X,Y), mother(Y,Z)).
```

Better:

```
grandfather(X,Z):- father(X,Y), parent(Y,Z).  
parent(X,Y):-mother(X,Y). %or use or father here  
parent(X,Y):-father(X,Y).
```

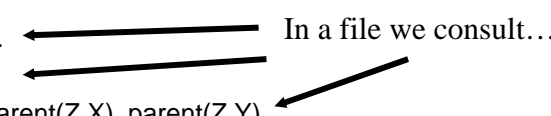
- ◆ The query now becomes:  
?- grandfather(Name,sally).

27

## More than One Result

- ◆ In situations where more than one object can fit a query, prolog returns the first. We can force it to keep going using the or operator typed after the result:  

```
parent(jill,jane).  
parent(jill,joe).  
sibling(X,Y):- parent(Z,X), parent(Z,Y).  
?- sibling(A,B).  
A=jane, B=joe;  
A=joe, B=jane;  
A=jane, B=jane;  
A=joe, B=joe  
-To stop the search, type . Instead of ;
```



28

## More than One Result

- ◆ You can see that prolog doesn't care about cases where the same object can fill more than one variable (makes sense, they unify!)
- ◆ That's because we haven't told it to care – it doesn't know that siblings in a different order are not relevant.
- ◆ We have to tell it:  
sibling(X,Y):- parent(Z,X), parent(Z,Y), X\=Y.  
– \= is not the same as not-equal – it says that X and Y do not unify – i.e. don't match, can't be bound to the same object. = is unification.

29

## Facts that Change

- ◆ The basic facts are permanently true
- ◆ Some information is temporal - true for a particular period in time  
married(bill, jane).
- ◆ If the marriage were to end, how would this knowledge be represented?

30

## Facts that Change

married(bill, jane).  
divorced(bill, jane).  
?- married(bill, jane).  
Yes

- ◆ Prolog does not “understand” that married and divorced are mutually exclusive
  - Prolog only manipulates symbols in the manner we tell it it can, and we haven't told it this fact!

31

## Temporal Knowledge

married(bill, jane).  
divorced(bill, jane).  
married(bill, sue).  
married\_to(X,Y) :- married(X,Y), not divorced(X,Y).  
?- married\_to(bill, jane).  
No.  
?- married\_to(bill, sue).  
Yes.

- ◆ This is one way of dealing with a knowledge representation problem – there are many specifics missing (when did they change from being married to not?) but it gives you the general idea

32



## Temporal Knowledge

married(bill, sue, date\_start).

married(bill, jane, date\_start, date\_end).

married\_to(X,Y) :- married(X, Y, Date\_start).

- this rule works if when a couple becomes divorced, the 3-parameter predicate is replaced by the 4-parameter predicate
  - ❖ They'd both exist, we'd just have to know and look for the 4-parameter predicate!
- This gives you a taste of some of the awkwardness we find in prolog sometimes!

33

## Non-Monotonic Systems

- ◆ Prolog as we have seen it thus far is a monotonic system – facts are added but never removed (monotone increasing = consistently increasing)
- ◆ To properly reflect real life we need non-monotonic features – removing facts as well as adding them
- ◆ Non-monotonic systems are difficult to manage; what we're doing here is dancing around that logically by handling change in a monotonic way

34

## Some Prolog Don'ts

- ◆ Do not nest predicates:

~~parent(parent(X,Y),Z) :- ...~~

- ◆ Do not place more than one term to the left of the implication ( :- )

~~parent(X,Y), father(X,Y) :-~~

- ◆ There are always ways around these!

35

## More Prolog Don'ts

- ◆ Do not use "not" to left of the :-

~~not female(X,Y) :- ...~~

Not in prolog is not a boolean not..

- ◆ Do not use a variable as a predicate name

~~A(X,Y) :- ...~~

We can't quantify over predicates (FOL!)

36

## Prolog's Search

- ◆ Prolog begins with our goal – it looks for rules concluding in our goal and then attempts to prove the preconditions of those rules
- ◆ This is done in a *depth-first* manner, starting with the first rule (or fact) whose conclusion matches
- ◆ In this way it will eventually search back to the initial facts we defined
- ◆ If at some point it doesn't find an initial fact, it will hit a dead end - a state that is not the goal state and to which no more rules can be applied
- ◆ In this case, Prolog "backtracks" to the last "choice point" and makes the next available choice (applies the next applicable rule)

37

## Prolog's Search

- ◆ That's why we have code that looks like this:  
parent(X,Y):-mother(X,Y).  
parent(X,Y):-father(X,Y).
- ◆ They're two different cases of somebody being a parent. If we ask if X is Y's parent, the first rule is tried and we see if we can prove mother(X,Y).
- ◆ If this fails, back up and try to find a different rule for concluding parent, and test father
- ◆ We'd actually look at facts before rules (makes sense!)

38

## Prolog's Search

- ◆ Suppose we wanted an ancestor predicate?
- ◆ A person who is the parent of someone is that person's ancestor  
  
ancestor(X,Y):- parent(X,Y).
- ◆ There's clearly more to it than this though, because we can go back to their parents and so on:

39

## Recursion

- ◆ A person who is the ancestor of an ancestor is also an ancestor  
  
ancestor(X,Y) :- ancestor(X,Z), ancestor(Z,Y).
- ◆ While this rule is logically correct, this does not work with Prolog
- ◆ Prolog's depth first search causes an infinite loop – there's nothing to stop this same rule from being applied over and over forever.

40

## A better way to think of it...

- ◆ A person who is the ancestor of a parent is also an ancestor  
    `ancestor(X,Y):- ancestor(X,Z), parent(Z,Y).`
- ◆ This rule on its own works in some cases but goes on infinitely in other cases
- ◆ The correct definition is:  
    `ancestor(X,Y) :- parent(X,Y).`  
    `ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).`
- ◆ Why? Think of how prolog would use these in it's search!

41

## Recursion

- ◆ The order of the two rules is important!
- ◆ The termination condition must be defined first (as it would in procedural code!)
- ◆ In general, a predicate may have any number of definitions; Prolog examines them in a depth-first manner *in the order in which we defined them*
- ◆ In some systems, ancestor is a system predicate, so be careful of the name

42

## Forcing Backtracking

- ◆ Recall that you can type ; (or) after a query result to tell prolog to keep going:
- ◆ `?-grandfather(Name,sally).`  
    `Name=fred ;`  
    `Name=joe ;`  
    `No.`
- ◆ What you're really doing here is forcing prolog to backtrack – making it treat the solution it found as if it were a dead end.
- ◆ “no” is returned last because looking for more at that point causes the search to fail

43

## Your ; response

- ◆ Is actually manually causing the search to fail – it's saying ignore what you found so far, and try to find something else
- ◆ This behaviour is often useful to reproduce internally. For example...

44

## Output

- ◆ The write() predicate can be used to display values on the monitor  
display\_a\_person :- person(X), write(X).
- ◆ You can insert blank spaces using the tab(n) predicate where n is the number of blanks
- ◆ The nl predicate generates a carriage return to the next line
- ◆ What if we want to print out all persons?

45

## Output

- ◆ display\_all\_persons:- person(X), write(X),nl.
- ◆ We could keep trying ; and eventually we'd get them, but we want them all at once
- ◆ display\_all\_persons:- person(X),write(X),nl, fail.
- ◆ fail is a system predicate that causes the which demands that the statement fail at that point – we've already written a value out, but prolog then fails and backtracks to try to find its goal another way
- ◆ Prolog finds another binding for person, prints the object, fails, etc.

46

## One more thing to be careful of

- ◆ Using NOT
- ◆ not does not mean logical negation – instead, it causes a predicate to fail if what the not is given succeeds, and vice versa
- ◆ sounds the same but it isn't...consider:
- ◆ mother(X,Y):-not(father(X,Y)),parent(X,Y).
- ◆ and assume I have:  
father(X,Y):-parent(X,Y),male(X).
- ◆ and parent(bill,sue).          parent(mary,sue).  
male(bill).                      female(sue).

47

## The trouble with not...

- ◆ Now suppose I ask ?-mother(X,sue).
- ◆ I get NO!
- ◆ mother(X,Y):-not(father(X,Y)),parent(X,Y).
  - father(X,sue) succeeds (X=bill).
  - So not(father(bill,X)) fails
  - ...so the rule fails!
- ◆ This rule is semantically stating that X is sue's mother if sue has no father and X is bill's parent
- ◆ That isn't what we intended, but it's what we got because of the semantics of not!

48

## End of Prolog Intro

- ◆ but more to come!

49

## Bagof

- ◆ Prolog contains a special system predicate that can be used to determine all objects that have a property  
`person_list(List):- bagof(Name,person(Name),List).`
- ◆ The result returned by this predicate is a list of all objects that satisfy the predicate **person**  
assuming we have `person(george).` `person(sam).` `person(jane).`, List gets bound to `[george, sam, jane]`
- ◆ Yes, you can manipulate lists in Prolog, we'll get there

50

## Bagof

`person_list(List):-bagof(Name,person(Name),List).`

- ◆ When processing this statement, Prolog attempts to find bindings (instantiations) for Name that satisfy the predicate `person(Name)`
- ◆ Its possible with some relations to have an item appear more than once (e.g. our problems with siblings!). So, there's also `setof`, which has each element appearing only once
- ◆ For each such instantiation, Prolog adds the value of the variable Name to the variable List
- ◆ A Prolog list is a collection of object constants separated by commas and enclosed in `[ ]` brackets  
`[george, sam, jane]`

51

## Prolog Lists

- ◆ An empty list consists of `[]`
- ◆ A list may contain sublists  
`[[a, b], c, [d]]`
- ◆ Prolog includes a special notation that permits the elements of a list to be extracted one at a time in a recursive manner
- ◆ You're used to the idea of an object being bound to a variable: `q(X)` unifies with `q(fred)` if we bind fred to X
- ◆ We can state that an object to be bound must be a list in the following manner...

52

## Matching Lists

- ◆  $[H | T]$ , where H and T are variables
- ◆ H is a variable that is bound to the first element in the list (the *head* - and there has to be something for this to unify – i. e. this will match only a list of at least one item!)
- ◆ T is a variable that is bound to the remaining elements in the list (the *tail* - this can be null, so this will still match to a one-item list)
- ◆ H & T can obviously be any variable name
- ◆ if you specified `[]` you'd be talking about the empty list constant (null)

53

## List Example

```
process([H|T]):- write(H), nl, write(T).  
/*split head & tail, write head, write tail*/  
?- process([a,b,c]).  
a  
[b,c]  
?- process([a]).  
a  
[]  
?- process([]).  
no. /*[H|T] won't match to []!*/
```

54

## Recursive List Example

- ◆ The elements of a list can be printed recursively
- ◆ in fact because  $[H|T]$  separates off only one element (the head), we *must* process things recursively.

```
print_list([]). /*terminate on a null list*/  
print_list([H|T]) :- write(H), print_list(T).
```

55

## A Member relation

- ◆ We can define a member relation to determine whether or not the first argument is a member of the second argument (which must be a list)  
member(X, [X|\_]).  
member(X, [H|T]) :- member(X,T).  
?- member(b,[a,b,c]).  
yes.  
?- member(e,[b,a,d]).  
no.

56

## User-Defined Functions

- ◆ The first parameter must appear at the “top level” of the second parameter  
?- member([a], [[b],[a],[d]]).  
yes.  
?- member([a],[b,[d,[a]]]).  
no.
- ◆ With the member function, it is not necessary to ensure that the second parameter is a list -  
- if a parameter is not a list, it will not be unifiable because of the [Head|Tail]  
?- member(a, a).  
no.

57

## Lists: a Membership relation

- ◆ this version of member “returns” 1 or 0 rather than success or failure – the concept of returning something doesn’t really work with predicates – but we can bind a 1 or a 0 to the third argument if it is a variable, which can be treated the same way  
member(X, [X|\_], 1).  
member(X, [], 0).  
member(X, [H|T], R) :- member(X, T, R).

58

## Variables

- ◆ Prolog does not support global variables
- ◆ Variables that are instantiated in a rule remain instantiated for the duration of the rule  
run:-bagof(Name, male(Name), List),  
write(List). %list is bound first, then written
- ◆ If the value of a variable is required in another rule/predicate, the variable must be passed to or from the predicate as a parameter
- ◆ run:-bagof(Name, male(Name), List),  
print\_list(List).
- ◆ Run(List):-bagof(Name, male(Name), List).

59

## Variables

- ◆ A variable can be given a value (instantiated) using the assignment operator (“is”)  
run:-X is 3, process(X).
- ◆ Once a variable has been given a value, the value cannot be modified  
run:-X is 3, process(X).  
process(X):-X is 4.  
?- run.  
no.
- ◆ So something like X is X+4 is impossible!
- ◆ Variables will still be made unbound when backtracking, but that is not the same as a traditional (imperative) assignment statement

60

## Variables

- ◆ If X had been instantiated to 4, then a subsequent X is 4 would have been successful
- ◆ If X had not been instantiated when process was called, the assignment would have been successful  
run:-process(X), write(X).  
process(X):-X is 4.  
?- run.  
4  
yes.

61

## Variables

- ◆ Since we can't modify a variable, we have to assign a changed value to a new variable
- ◆ process(X,Y):-Y is X+4.  
run:- X is 3, process(X,Y),  
write(X),tab(2), write(Y).  
?- run.  
3 7  
yes.

62

## Another use of Fail

- ◆ Mary likes all animals except snakes
- ◆ If X is a snake, then Mary does not like X; otherwise, if X is an animal, Mary likes X  
likes(mary,X) :- snake(X), fail.  
likes(mary,X) :- animal(X).
- ◆ This strategy does not work because Prolog will backtrack after the first rule fails, and use the second rule to deduce that Mary likes joe the snake because snakes are animals
- ◆ we want to it not just to fail, but to *quit looking*

63

## Cut

- ◆ we can insert ! (cut) to tell prolog not to backtrack: likes(mary,X) :- snake(X) , !, fail.
- ◆ When you place a cut in a rule, it tells prolog not to backtrack on anything to the left of it
- ◆ If there's stuff to prove to the right of it, backtracking will still occur in these clauses though
  - So it doesn't have to appear at the end of a rule, even though you see it there a lot!

64



## Be careful with Bagof

- ◆ When using the bagof (and setof) predicate, the internal predicate must either have only one parameter or, if there is more than one parameter, the additional parameter(s) must already be instantiated
- ◆ For example, attempting to determine the names of all parents using the statement below **will not work** correctly unless Child has already been instantiated by some other part of the rule. On it's own this will NOT work:

**X** bagof(Name, parent(Name, Child), List) **X**

65

## Existential Quantifier

- ◆ However, Prolog supports a feature that is similar to the existential quantifier in first-order logic that eliminates the need for an additional predicate  
bagof(Name, Child^parent(Name, Child), List)
- ◆ This statement locates all people for which there is a parent-child relationship but ignores the child in each relationship (this is not the same as an anonymous variable)

66

## Declarative/Procedural Knowledge

- ◆ Most Prolog programs for non-trivial applications will include both declarative knowledge and procedural knowledge
- ◆ The declarative knowledge defines what is known about the problem domain
- ◆ The procedural knowledge defines how to solve problems in the domain
- ◆ For example, procedural knowledge might define how to retrieve knowledge so that duplicates are removed
- ◆ good programming practice to keep the two types of knowledge separate

67

## Example

- ◆ Jack is a person.
- ◆ Sam and Janet are also persons.
- ◆ Jack likes Janet.  
person(jack).  
person(sam).  
person(janet).  
likes(jack, janet).

68

## Deductions

- ◆ Friends are people who like each other.  
friends(X,Y):- likes(X,Y),likes(Y,X),  
                  person(X), person(Y), X\=Y.  
  
?- friends(jack,janet).  
no.
- ◆ Prolog can make deductions only if all of the necessary knowledge has been defined
  - No info about janet liking jack here!

69

## Deductions

- likes(jack,janet).  
friends(X,Y):- likes(X,Y),likes(Y,X), X\=Y.  
friends(jack,janet).  
?- likes(jack,janet).  
yes.  
?- likes(janet,jack).  
no.
- ◆ Prolog can not (and should not) conclude that the premises are true even if the conclusion is true

70

## Deductions

- ◆ In general, do not define facts with predicates that are also used as the conclusions to rules
- ◆ Instead, add the fact:  
likes(janet,jack).
- ◆ Prolog can then conclude:  
friends(jack,janet).
- ◆ Prolog can also conclude:  
friends(janet,jack).

71

## Example

- ◆ Jack is married to Janet.  
married(jack,janet).
- ◆ We could agree that our intent is that a male must be the first participant in the predicate  
?- married(janet,jack).  
no.  
married(X,Y):-married(Y,X).
- ◆ This creates an infinite loop
- ◆ **Insisting on the correct use of the predicate is the best practice**

72

## Example

- ◆ However, there may be times when it is desirable to have the predicate bidirectional  
`married_to(X,Y) :- male(X), married(X,Y);  
                  female(X), married(Y,X).`
- ◆ This new predicate avoids loops but does generate duplicates
- ◆ Note that this predicate is defined in terms of the base predicate `married`

73

## Built-in Functions

- ◆ Prolog contains a variety of built-in functions
  - `nonvar(X)` returns true if X is currently bound to a value or if X is not a variable
  - `var(X)` returns true if X is not currently bound to a value
- ◆ These are useful when you know that supplying an unbound variable will cause problems to a definition, or where there is an easier way to calculate something depending on which variables are bound
  - Loads of these: look them up!

74

## User-Defined Functions

- ◆ If you want to declare your own functions, define them using meaningful predicates  
`min(X,Y,Z) :- X<Y, Z is X.`  
`min(X,Y,Z) :- Z is Y.`
- ◆ Prolog automatically backtracks until a solution is found
- ◆ Try this definition out with a few values and look at its behaviour!

75

## User-Defined Functions

- ◆ The previous version does not work correctly; for example, `min(3, 4, 4)` is true !?
- ◆ look back at the definition: `min(X,Y,Z):-Y is Z` will indeed be true if the same value for Y and Z is given on the query. Better definition:  
`min(X,Y,Z) :- X<Y, Z is X.`  
`min(X,Y,Z) :- Y=<X, Z is Y.`
- ◆ Note that each rule contains a precondition which defines the conditions under which the rule can be executed
- ◆ Prolog backtracks until a correct solution is found but Prolog will not backtrack to find incorrect solutions

76