Prolog

Tim Finin
University of Maryland
Baltimore County

9/19/01

Syllogisms

- "Prolog" is all about programming in logic.
  - Socrates is a man.
  - All men are mortal.
  - Therefore, Socrates is mortal.

UMBC
an Honors University in Maryland

Facts, rules, and queries

- Fact: Socrates is a man.
  `man(socrates).`
- Rule: All men are mortal.
  `mortal(X) :- man(X).`
- Query: Is Socrates mortal?
  `mortal(socrates).`

UMBC
an Honors University in Maryland

Running Prolog I

- Create your "database" (program) in any editor
- Save it as *text only,* with a **.pl** extension
- Here's the complete "program":

```
man(socrates).
mortal(X) :- man(X).
```

UMBC
an Honors University in Maryland

Running Prolog II

- Prolog is *completely interactive.*
- Begin by invoking the Prolog interpreter.
  sicstus
- Then load your program.
  consult('mortal.pl')
- Then, ask your question at the prompt:
  mortal(socrates).
- Prolog responds:
  Yes

On gl.umbc.edu

```
[finin@linux3 prolog]$ ls
mortal.pl
[finin@linux3 prolog]$ pl
Welcome to SWI-Prolog (Multi-threaded, Version 5.6.18)
...
For help, use ?- help(Topic). or ?- apropos(Word).
?- consult('mortal.pl').
% mortal.pl compiled 0.00 sec, 692 bytes
Yes
?- mortal(socrates).
Yes
?- mortal(X).
X = socrates
Yes
?-
```


Syntax I: Structures

- Example structures:
  - sunshine
  - man(socrates)
  - path(garden, south, sundial)
- <structure> ::=
  <name> | <name> (<arguments>)
- <arguments> ::=
  <argument> | <argument> , <arguments>

Syntax II: Base Clauses

- Example base clauses:
  - debug_on.
  - loves(john, mary).
  - loves(mary, bill).
- <base clause> ::= <structure> .

Syntax III: Nonbase Clauses

- Example nonbase clauses:
  - mortal(X) :- man(X).
  - mortal(X) :- woman(X)
  - happy(X) :- healthy(X), wealthy(X), wise(X).
- <nonbase clause> ::=
  <structure> :- <structures> .
- <structures> ::=
  <structure> | <structures> , <structure>

Syntax IV: Predicates

- A predicate is a collection of clauses with the same functor and arity.

  loves(john, mary).
  loves(mary, bill).
  loves(chuck, X) :- female(X), rich(X).
- <predicate> ::=
  <clause> | <predicate> <clause>
- <clause> ::=
  <base clause> | <nonbase clause>

Syntax V: Programs

- A program is a collection of predicates.
- Predicates can be in any order.
- Predicates are used in the order in which they occur.

Syntax VI: Assorted details

- Variables begin with a capital letter:
  X, Socrates, _result
- Atoms do not begin with a capital letter:
  x, socrates
- Other atoms must be enclosed in single quotes:
  - 'Socrates'
  - 'C:/My Documents/examples.pl'

Syntax VII: Assorted details

- In a quoted atom, a single quote must be quoted or backslashed: 'Can''t, or won\'t?'
- /* Comments are like this */
- Prolog allows some infix operators, such as :- (turnstile) and , (comma). These are syntactic sugar for the functors '÷ ' and ','.
- Example:
  '÷ '(mortal(X), man(X)).

UMBC

an Honors University in Maryland

13

Backtracking

loves(chuck, X) ÷ female(X), rich(X).
female(jane).
female(mary).
rich(mary).
---------- *Suppose we ask:* loves(chuck, X).
female(X) = female(jane), X = jane.
rich(jane) fails.
female(X) = female(mary), X = mary.
rich(mary) succeeds.

UMBC

an Honors University in Maryland

14

Additional answers

- female(jane).
  female(mary).
  female(susan).
- ? female(X).
- X = jane ;
- X = mary
- Yes

UMBC

an Honors University in Maryland

15

Common problems

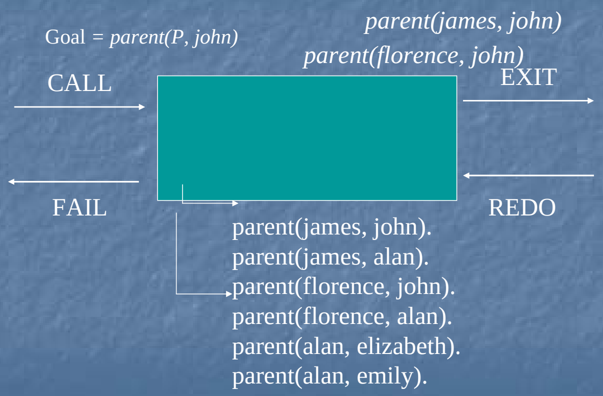
- Capitalization is *extremely* important!
  - Capitalized symbols are variables!
- No space between a functor and its argument list:
  man(socrates), *not* man (socrates).
- Don't forget the period! (But you can put it on the next line.)

UMBC

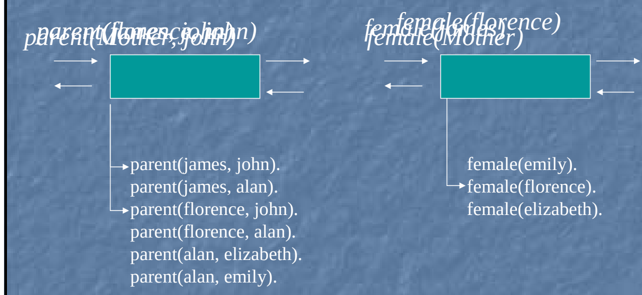
an Honors University in Maryland

16

Prolog Execution Model / Prolog Debugger



Execution Model (conjunctions)



Readings

- loves(chuck, X) :- female(X), rich(X).
- Declarative reading: Chuck loves X if X is female and rich.
- Approximate procedural reading: To find an X that Chuck loves, first find a female X, then check that X is rich.
- Declarative readings are almost always preferred.
- Try to write Prolog predicates so that the procedural and natural declarative reading give the same answers.

Logic is Monotonic

- Classical logic, anyway.
- Monotonic ~= never gets smaller
- In logic, a thing is true or false.
  - 3>2 is true
- If something is true, it's true for all time
  - 3>2 always was and always will be true
- us_president('George W. Bush') ?
- loves('Tom Cruse', 'Katie Holms') ?

Non-Monotonic Logic

- A non-monotonic logic is one in which a proposition's true value can change in time
- Learning a new fact may cause the number of true propositions to decrease.
- Prolog is non-monotonic for two reasons:
  - You can assert **and retract** clauses
  - Prolog uses "negation as failure"

UMBC
an Honors University in Maryland

21

Assert and Retract

- Normally we assert and retract facts (i.e., base clauses)
  - assert(loves(tom,nicole)).
  - retract(loves(tom,nicole)).
  - assert(loves(tom,katie)).
  - retract(loves(tom,X)).
  - retractall(loves(tom,X)).
- You can assert/retract any clause
  - assert(loves(X,Y) :- spouse(X,Y)).
- Static vs. dynamic predicates

UMBC
an Honors University in Maryland

22

Negation as failure

- NOT is basic to logic
- How can we prove that something is false?
- Pure prolog only supports positive proofs
- Handling negation is much more difficult
  - Quickly leads to undecidability
- Yet…

  ?- man(tom).

  No
- In Prolog, we often use our inability to prove P to be a prove that P is false.
- This is the semantics databases assume

UMBC
an Honors University in Maryland

23

not is Prolog's NAF operator

- Birds can fly, except for penguins.

  canFly(X) :- bird(X), not(penguin(X)).

  bird(eagle). bird(wren). bird(penguin). bird(emu).
- Birds can fly unless we know them to be flightless

  canFly(X) :- bird(X), not(cantFly(X)).

  cantFly(penguin). cantFly(emu).
- What does this mean?

  not(bird(X))
- The 'standard not operator is \+.

UMBC
an Honors University in Maryland

24

A Simple Prolog Model

- Imagine prolog as a system which has a database composed of two components:
  - FACTS - statements about true relations which hold between particular objects in the world. For example:

    parent(adam,able): adam is a parent of able

    parent(eve,able): eve is a parent of able

    male(adam): adam is male.
  - RULES - statements about true relations which hold between objects in the world which contain generalizations, expressed through the use of variables. For example, the rule

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

  might express:

    for any X and any Y, X is the father of Y if X is a parent of Y and X is male.

UMBC

an Honors University in Maryland

Nomenclature and Syntax

- A prolog rule is called a **clause**.
- A clause has a head, a neck and a body:

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

  head *neck* *body*
- the head is a rule's conclusion.
- The body is a rule's premise or condition.
- note:
  - read :- as IF
  - read , as AND
  - a . marks the end of input

UMBC

an Honors University in Maryland

Prolog Database

parent(adam,able)
parent(adam,cain)
male(adam)
...

Facts comprising the "extensional database"

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

Rules comprising the "intensional database"

UMBC

an Honors University in Maryland

Extensional vs. Intensional

The terms *extensional* and *intensional* are borrowed from the language philosophers use for *epistemology*.

- *Extension* refers to whatever *extends*, i.e., "is quantifiable in space as well as in time".
- *Intension* is an antonym of extension, referring to "that class of existence which may be quantifiable in time but not in space."
- NOT *intentional* with a "t", which has to do with "will, volition, desire, plan, …"

For KBs and DBs we use

- *extensional* to refer to that which is explicitly represented (e.g., a fact), and
- *intensional* to refer to that which is represented abstractly, e.g., by a rule of inference.

Prolog Database

parent(adam,able)
parent(adam,cain)
male(adam)
...

Facts comprising the "extensional database"

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

Rules comprising the "intensional database"

Epistemology is "a branch of philosophy that investigates the origin, nature, methods, and limits of knowledge"

UMBC

an Honors University in Maryland

A Simple Prolog Session

```
| ?-
  assert(parent(adam,able))
  .
yes
| ?-
  assert(parent(eve,able)).
yes
| ?- assert(male(adam)).
yes
| ?- parent(adam,able).
yes
| ?- parent(adam,X).
X = able
yes
```

```
| ?- parent(X,able).
X = adam ;
X = eve ;
no
| ?- parent(X,able) ,
   male(X).
X = adam ;
no
```

UMBC
an Honors University in Maryland

A Prolog Session

```
| ?- [user].
| female(eve).
| parent(adam,cain).
| parent(eve,cain).
| father(X,Y) :-
    parent(X,Y), male(X).
| mother(X,Y) :-
    parent(X,Y), female(X).
| ^Zuser consulted 356
    bytes 0.0666673 sec.
yes
| ?- mother(Who,cain).
Who = eve
yes
```

```
| ?- mother(eve,Who).
Who = cain
yes
| ?- trace, mother(Who,cain).
  (2) 1 Call: mother(_0,cain) ?
  (3) 2   Call: parent(_0,cain) ?
  (3) 2   Exit: parent(adam,cain)
  (4) 2   Call: female(adam) ?
  (4) 2   Fail: female(adam)
  (3) 2   Back to: parent(_0,cain) ?
  (3) 2   Exit: parent(eve,cain)
  (5) 2   Call: female(eve) ?
  (5) 2   Exit: female(eve)
  (2) 1 Exit: mother(eve,cain)
Who = eve
yes
```

UMBC
an Honors University in Maryland


```
| ?- [user].
| sibling(X,Y) :-
|   father(Pa,X),
|   father(Pa,Y),
|   mother(Ma,X),
|   mother(Ma,Y),
|   not(X=Y).
^Zuser consulted 152 bytes
   0.0500008 sec.
yes
| ?- sibling(X,Y).
X = able
Y = cain ;
X = cain
Y = able ;
```

```
trace,sibling(X,Y).
(2) 1 Call: sibling(_0,_1) ?
(3) 2  Call: father(_65643,_0) ?
(4) 3   Call: parent(_65643,_0) ?
(4) 3   Exit: parent(adam,able)
(5) 3   Call: male(adam) ?
(5) 3   Exit: male(adam)
(3) 2  Exit: father(adam,able)
(6) 2  Call: father(adam,_1) ?
(7) 3   Call: parent(adam,_1) ?
(7) 3   Exit: parent(adam,able)
(8) 3   Call: male(adam) ?
(8) 3   Exit: male(adam)
(6) 2  Exit: father(adam,able)
(9) 2  Call: mother(_65644,able) ?
(10) 3  Call: parent(_65644,able) ?
(10) 3  Exit: parent(adam,able)
(11) 3  Call: female(adam) ?
(11) 3  Fail: female(adam)
(10) 3  Back to: parent(_65644,able)
        ?
(10) 3  Exit: parent(eve,able)
(12) 3  Call: female(eve) ?
(12) 3  Exit: female(eve)
(9) 2  Exit: mother(eve,able)
(13) 2 Call: mother(eve,able) ?
(14) 3  Call: parent(eve,able) ?
(14) 3  Exit: parent(eve,able)
(15) 3  Call: female(eve) ?
(15) 3  Exit: female(eve)
(13) 2 Exit: mother(eve,able)
(16) 2 Call: not able=able ?
(17) 3  Call: able=able ?
(17) 3  exit: able=able
(16) 2 Back to: not able=able ?
(16) 2 Fail: not able=able
(15) 3  Back to: female(eve) ?
(15) 3  Fail: female(eve)
```

```
(14) 3  Back to: parent(eve,able) ?
(14) 3  Fail: parent(eve,able)
(13) 2 Back to: mother(eve,able) ?
(13) 2 Fail: mother(eve,able)
(12) 3  Back to: female(eve) ?
(12) 3  Fail: female(eve)
(10) 3  Back to: parent(_65644,able)
        ?
(10) 3  Fail: parent(_65644,able)
(9) 2  Back to: mother(_65644,able) ?
(9) 2  Fail: mother(_65644,able)
(8) 3   Back to: male(adam) ?
(8) 3   Fail: male(adam)
(7) 3   Back to: parent(adam,_1) ?
(7) 3   Exit: parent(adam,cain)
(18) 3  Call: male(adam) ?
(18) 3  Exit: male(adam)
(6) 2  Exit: father(adam,cain)
(19) 2 Call: mother(_65644,able) ?
(20) 3  Call: parent(_65644,able) ?
(20) 3  Exit: parent(adam,able)
(21) 3  Call: female(adam) ?
(21) 3  Fail: female(adam)
(20) 3  Back to: parent(_65644,able)
        ?
(20) 3  Exit: parent(eve,able)
(22) 3  Call: female(eve) ?
(22) 3  Exit: female(eve)
(19) 2 Exit: mother(eve,able)
(23) 2 Call: mother(eve,cain) ?
(24) 3  Call: parent(eve,cain) ?
(24) 3  Exit: parent(eve,cain)
(25) 3  Call: female(eve) ?
(25) 3  Exit: female(eve)
(23) 2 Exit: mother(eve,cain)
(26) 2 Call: not able=cain ?
(27) 3  Call: able=cain ?
(27) 3  Fail: able=cain
(26) 2 Exit: not able=cain
(2) 1 Exit: sibling(able,cain)
X = able
Y = cain
yes no
| ?-
```

UMBC
an Honors University in Maryland

How to Satisfy a Goal

Here is an informal description of how Prolog satisfies a goal (like father(adam,X)). Suppose the goal is G:

- if G = P,Q then first satisfy P, carry any variable bindings forward to Q, and then satiety Q.
- if G = P;Q then satisfy P. If that fails, then try to satisfy Q.
- if G = not(P) then try to satisfy P. If this succeeds, then fail and if it fails, then succeed.
- if G is a simple goal, then look for a fact in the DB that unifies with G look for a rule whose conclusion unifies with G and try to satisfy its body

UMBC
an Honors University in Maryland

Note

- two basic conditions are true, which always succeeds, and fail, which always fails.
- A comma (,) represents conjunction (i.e. and).
- A semi-colon represents disjunction (i.e. or), as in:
  grandParent(X,Y) :- grandFather(X,Y); grandMother(X,Y).
- there is no real distinction between RULES and FACTS. A FACT is just a rule whose body is the trivial condition true. That is *parent(adam,cain)* and *parent(adam,cain) :- true.* are equivalent
- Goals can usually be posed with any of several combination of variables and constants:
  - parent(cain,able) - is Cain Able's parent?
  - parent(cain,X) - Who is a child of Cain?
  - parent(X,cain) - Who is Cain a child of?
  - parent(X,Y) - What two people have a parent/child relationship?

Prolog Terms

- The term is the basic data structure in Prolog.
- The term is to Prolog what the s-expression is to Lisp.
- A term is either:
  - a constant
    - john , 13, 3.1415, +, 'a constant'
  - a variable
    - X, Var, _, _foo
  - a compound term
    - part(arm,body)
    - part(arm(john), body(john))
- The reader and printer support operators:
  - - X is read as '-'(X).
  - 5 + 2 is read as '+'(5,2).
  - a:-b,c,d. read as ':-'(a ','(b,','(c,d))).

Compound Terms

- A compound term can be thought of as a relation between one or more terms:
  - part_of(finger,hand)

and is written as:

1. the relation name (*principle functor*) which must be a constant.
2. An open parenthesis
3. The arguments - one or more terms separated by commas.
4. A closing parenthesis.

- The number of arguments of a compound terms is called its arity.

Term	arity
f	0
f(a)	1
f(a,b)	2
f(g(a),b)	2

The Notion of Unification

- Unification is when two things "become one"
- Unification is kind of like assignment
- Unification is kind of like equality in algebra
- Unification is mostly like pattern matching
- Example:
  - loves(john, X) unifies with loves(john, mary)
  - and in the process, X gets unified with mary

Unification I

- Any value can be unified with itself.
  - weather(sunny) = weather(sunny)
- A variable can be unified with another variable.
  - X = Y
- A variable can be unified with ("instantiated to") any Prolog term.
  - Topic = weather(sunny)

Unification II

- Two different structures can be unified if their constituents can be unified.
  - mother(mary, X) = mother(Y, father(Z))
- In Prolog, a variable can be unified with a structure containing that same variable.
- This is usually a Bad Idea.
- Unifying X and f(X) binds X to a circular structure which Prolog can not print.
  - X = f(f(f(f(f(…

Explicit Unification

- The explicit unification operator is =
- Unification is symmetric:

  Cain = father(adam)

  means the same as

  father(adam) = Cain
- Most unification happens implicitly, as a result of parameter transmission.
  - E.g., Prolog trys to prove older(X, bob) by unifying it with the fact older(zeus,_).

Scope of Names

- The scope of a variable is the single clause in which it appears.
- The scope of the "anonymous" ("don't care") variable (eg _ or _foo) is itself.
  - loves(_, _) = loves(john, mary)
- A variable that only occurs once in a clause is a useless *singleton;* replace it with an anonymous variable.
  - Most Prolog interpreters will issue warnings if you have rules with singleton variables
  - isFather(X) :- male(X), parent(X,_child).

Writing Prolog Programs

- Suppose the database contains
  loves(chuck, X) :- female(X), rich(X).
  female(jane).
  and we ask who Chuck loves,
  ?- loves(chuck, Woman).
- female(X) *finds* a value for X , say, jane
- rich(X) then *tests* whether Jane is rich

UMBC
an Honors University in Maryland

Clauses as Cases

- A predicate consists of multiple clauses whose *heads* have the same principle functor and arity.
- Each clause represents a "case".
  grandfather(X,Y) :- father(X,Z), father(Z,Y).
  grandfather(X,Y) :- father(X,Z), mother(Z,Y).
  abs(X, Y) :- X < 0, Y is -X.
  abs(X, X) :- X >= 0.
- Clauses with heads having different airty are unrelated.
  - Like methods in OO languages

UMBC
an Honors University in Maryland

Ordering

- Clauses are always tried in order
  buy(X) :- good(X).
  buy(X) :- cheap(X).
  cheap('Java 2 Complete').
  good('Thinking in Java').
- What will buy(X) choose first?

UMBC
an Honors University in Maryland

Ordering II

- Try to handle more specific cases (those having more variables instantiated) first.

dislikes(john, bill).

dislikes(john, X) :- rich(X).

dislikes(X, Y) :- loves(X, Z), loves(Z, Y).

UMBC
an Honors University in Maryland

Ordering III

- Some "actions" cannot be undone by backtracking over them:
  - write, nl, assert, retract, consult
- Do tests before you do undoable actions:
  - take(A) :-
    at(A, in_hand),
    write('You\'re already holding it!'),
    nl.

Recursion

- Prolog makes avoiding infinite recursion the programmer's responsibility.
- But it always tries clauses in order and processes conditions in a clause from left to right.
- So, handle the base cases first, recur only with a simpler case, use right recursion.

```
ancestor(P1,P2) :- parent(P1,P2).
ancestor(P1,P2) :- parent(P1,X), ancestor(X,P2).
```

- But not:

```
ancestor(P1,P2) :- parent(P1,P2).
ancestor(P1,P2) :- ancestor(P1,X), parent(X,P2).
```


Facts and Rules

- Designing a Prolog knowledge base usually starts with deciding which predicates will be provided as facts and which will be defined by rules.

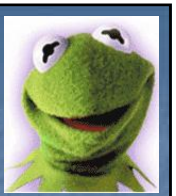
```
parent(Adam,cain).
child(X,Y) :- parent(Y,X).
```

- We don't have to worry about this in logic and in some logic programming languages:

  parent(X,Y) ⇔ child(Y,X)

- Of course, it's common for a predicate to be defined using both facts and rules.
  - Example: int(0). int(suc(X)) :- int(X).
  - What's at issue is really avoiding non-terminating reasoning.

Choosing predicates



- Designing a set of predicates (an ontology) requires knowledge of the domain and how the representation will be used.
- Example: representing an object's color.
  - green(kermit)
  - color(kermit,green)
  - value(kermit,color,green)
  - attribute(kermit,color,value,green)
- Which of these is best?

Issues in choosing predicates

green(kermit)
color(kermit,green)
value(kermit,color,green)
attribute(kermit,color,value,gree

- **What queries can be asked?**
  - A principle functor can not be a variable, e.g., can't do: Relation(john,mary)
  - Which can we use to answer:
    - Is kermit green?
    - What color is Kermit?
    - What do we know about Kermit?
    - What is the range of the color attribute?
- **How efficient is retrieval of facts and rules.**
  - Let a term's signature be its principle functor and arity.
  - Prolog indexes a fact or rule head on its signature and the signature of its first argument.
  - This is done for efficiency

UMBC
an Honors University in Maryland

Cut and Cut-fail

- The cut, !, is a commit point. It commits to:
  - the clause in which it occurs (can't try another)
  - everything up to that point in the clause
- Example:
  - loves(chuck, X) :- female(X), !, rich(X).
  - Chuck loves the *first* female in the database, but only if she is rich.
- Cut-fail, (!, fail), means give up *now* and don't even try for another solution.
- More on this later

UMBC
an Honors University in Maryland

Arithmetic: Built-In is/2

- Arithmetic expressions aren't normally evaluated in Prolog.
- Built-In *infix operator* **is/2** evaluates it's 2nd argument, and unifies the result with it's 1st argument.

```
| ?- X = 5 + 2.
X = 5+2?
yes
| ?- X is 5 + 2.
X = 7 ?
yes
```

- Any variables in the right-hand side of **is**/2 must be instantiated when it is evaluated.
- More on this later

UMBC
an Honors University in Maryland

What you can't do

- There are no functions, only predicates
- Prolog is programming in logic, therefore there are few control structures
- There are no assignment statements; the *state* of the program is what's in the database

UMBC
an Honors University in Maryland

Workarounds II

- There are few control structures in Prolog, BUT…
- You don't need IF because you can use multiple clauses with "tests" in them
- You seldom need loops because you have recursion
- You can, if necessary, construct a "fail loop"

UMBC
an Honors University in Maryland
53

Fail Loops

```
notice_objects_at(Place) :-
      at(X, Place),
      write('There is a '), write(X),
      write(' here.'), nl,
      fail.
notice_objects_at(_).
```

- Use fail loops sparingly, if at all.

UMBC
an Honors University in Maryland
54

Workarounds II

- There are no functions, only predicates, BUT…
- A call to a predicate can instantiate variables: female(X) can either
  - look for a value for X that satisfies female(X), or
  - if X already has a value, test whether female(X) can be proved true
- By convention, output variables come last
  - Square(N,N2) :- N2 is N*N.

UMBC
an Honors University in Maryland
55

Workarounds II

- Functions are a subset of relations, so you can define a function like factorial as a relation

```
factorial(N,0) :- N<1.
factorial(1,1).
factorial(N,M) :-
   N2 is N-1,
   factorial(N2,M2),
   M is N*M2.
```

- The last argument to the relation is used for the value that the function returns.
- How would you define:
  fib(n)=fib(n-1)+fib(n-2) where fib(0)=0 and fib(1)=1

UMBC
an Honors University in Maryland
56

Workarounds III

- There are no assignment statements, BUT…
- the Prolog database keeps track of program state

```
bump_count :-
      retract(count(X)),
      Y is X + 1,
      assert(count(Y)).
```

- Don't get carried away and misuse this!

UMBC

an Honors University in Maryland

57

Lists in Prolog

- Prolog has a simple universal data structure, the term, out of which others are built.
- Prolog lists are important because
  - They are useful in practice
  - They offer good examples of writing standard recursive predicates
  - They show how a little syntactic sugar helps

UMBC

an Honors University in Maryland

58

Linked Lists

- Prolog allows a special syntax for lists:
  - [a,b,c] is a list of 3 elements
  - [] is a special atom indicating a list with 0 elements
- Internally, Prolog lists are regular Prolog terms with the functor '.' (so called "dotted pairs")
  [a,b,c] = '.'(a, '.'(b, '.'(c, []))).
- The symbol | in a list indicates "rest of list", or the term that is a dotted pair's 2nd argument.
  [a,b,c] = [a|[b,c]].
- [Head|Tail] is a common expression for dividing a list into its 1st element (Head) and the rest of the list (Tail).

UMBC

an Honors University in Maryland

59

Example: list/1

```
% list(?List) succeeds if its arg is a well formed list.
list([]).
list([_Head|Tail]):-
   list(Tail).
```

- Since Prolog is untyped, we don't have to know anything about **Head** except that it is a term.
- The list can have terms of any type
  [1, foo, X, [sub, list], 3.14]

UMBC

an Honors University in Maryland

60

Example: member/2

% member(?Element, ?List) is true iff Element is a
% top-level member of the list List.
member(Element, [Element|_Tail]).
member(Element, [_Head|Tail]):- member(Element, Tail).

- This is a standard recursive definition of member:
  - (1) If the list has some elements, is what we're looking for the first one?
  - (2) If the list has some elements, is what we're looking for in the rest of the list?
  - (3) The answer is no.

UMBC
an Honors University in Maryland

Member has several uses

% member(+,+) checks
% membership.

```
| ?- member(b,[a,b,c]).
yes
| ?- member(x,[a,b,c]).
no
```

% member(-,+) generates
% members.

```
| ?- member(X,[a,b,c]).
X = a ? ;
X = b ? ;
X = c ? ;
no
| ?- member(X,[a,b,c,1,d,e,2]),
    integer(X).
X = 1 ? ;
X = 2 ? ;
no
```

% member(+,-) generates lists.

```
| ?- member(a,L).
L = [a|_A] ? ;
L = [_A,a|_B] ? ;
L = [_A,_B,a|_C] ?
yes
```

% member(-,-) generates lists.

```
| ?- member(X,L).
L = [X|_A] ? ;
L = [_A,X|_B] ? ;
L = [_A,_B,X|_C] ?
yes
| ?-
```

UMBC
an Honors University in Maryland

Using member to test list elements

- Does a list L have a negative number in it?
  - member(X,L), number(N), N<0.
- Are all of the elements of L numbers between 1 and 10?
  - not(member(X,L) ,
            not(number(X) ; X<1 ; X>10))

UMBC
an Honors University in Maryland

Example: delete/3

% delete(+Element, +List, -NewList)
% delete/3 succeeds if NewList results from
% removing one occurrence of Element from List.

delete(Element, [Element|Tail], Tail).
delete(Element, [Head|Tail], [Head|NewTail]):-
    delete(Element, Tail, NewTail).

UMBC
an Honors University in Maryland

Example: append/3

% append(?List1, ?List2, ?List3)

% append/3 succeeds if List3 contains all the
% elements of List1, followed by all the elements
% of List2.

```
append([], List2, List2).
append([Head|List1], List2, [Head|List3]):-
    append(List1, List2, List3).
```


Append is amazing

```
% append(+,+,+) checks
| ?- append([1,2],[a,b],[1,2,a,x]).
no

% append(+,+,-) concatenates
| ?- append([1,2],[a,b],L).
L = [1,2,a,b] ?
yes

% append(+,-,+) removes prefix.
| ?- append([1,2],L,[1,2,a,b]).
L = [a,b] ?
yes

% append(-,+,+) removes suffix.
| ?- append(X,[a,b],[1,2,a,b]).
X = [1,2] ?
yes
```

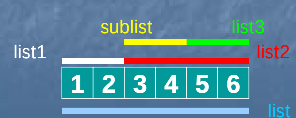
```
% append(-,-,+) generates all
%  ways to split a list into a
%  prefix and suffix.
| ?- append(X,Y,[1,2,a,b]).
X = [],
Y = [1,2,a,b] ? ;

X = [1],
Y = [2,a,b] ? ;

X = [1,2],
Y = [a,b] ? ;

X = [1,2,a],
Y = [b] ? ;

X = [1,2,a,b],
Y = [] ? ;
no
```


Example: sublist/3

% sublist(?SubList, +List). Note: The 1st append
% finds a beginning point for the sublist and the
% 2nd append finds an end point

```
sublist(SubList, List):-
    append(_List1, List2, List),
    append(SubList, _List3, List2).
```

% example: sublist([3,4],[1,2,3,4,5,6])



Example: sublist/3 (cont)

% here's another way to write sublist/2

```
sublist1(SubList, List):-
    append(List1, _List2, List),
    append(_List3, SubList, List1).
```

Example: “naïve” reverse

```
% nreverse(?List, ?ReversedList) is true iff the
% result of reversing the top-level elements of
% list List is equal to ReversedList.
  nreverse([], []).
  nreverse([Head|Tail], ReversedList):-
     nreverse(Tail, ReversedTail),
     append(ReversedTail, [Head], ReversedList).
```

- this is simple but inefficient
  - It’s not tail recursive
  - Append is constantly copying and recopying lists
- it’s a traditional benchmark for Prolog.

Example: efficient reverse/3

```
% reverse(+List, -ReversedList) is a “tail recursive”
% version of reverse.
reverse(List, ReversedList) :-
   reverse1(List, [], ReversedList).

reverse1([], ReversedList, ReversedList).
reverse1([Head|Tail], PartialList, ReversedList):-
   reverse1(Tail, [Head|PartialList], ReversedList).
```


reverse and nreverse

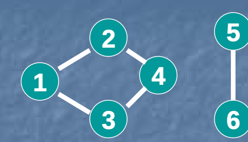
```
| ?- trace.
{The debugger will … everything (trace)}
yes
| ?- nreverse([1,2,3],L).
     1      1 Call: nreverse([1,2,3],_204) ?
     2       2 Call: nreverse([2,3],_712) ?
     3        3 Call: nreverse([3],_1122) ?
     4         4 Call: nreverse([],_1531) ?
     4         4 Exit: nreverse([],[]) ?
     5         4 Call: append([],[3],_1122) ?
     5         4 Exit: append([],[3],[3]) ?
     3        3 Exit: nreverse([3],[3]) ?
     6        3 Call: append([3],[2],_712) ?
     7         4 Call: append([],[2],_3800) ?
     7         4 Exit: append([],[2],[2]) ?
     6        3 Exit: append([3],[2],[3,2]) ?
     2       2 Exit: nreverse([2,3],[3,2]) ?
     8       2 Call: append([3,2],[1],_204) ?
     9         3 Call: append([2],[1],_5679) ?
    10          4 Call: append([],[1],_6083) ?
    10          4 Exit: append([],[1],[1]) ?
     9        3 Exit: append([2],[1],[2,1]) ?
     8       2 Exit: append([3,2],[1],[3,2,1]) ?
     1      1 Exit: nreverse([1,2,3],[3,2,1]) ?
L = [3,2,1] ?
yes
```

```
| ?- reverse([1,2,3],L).
     1      1 Call: reverse([1,2,3],_204) ?
     2       2 Call: reverse1([1,2,3],[],_204) ?
     3        3 Call: reverse1([2,3],[1],_204) ?
     4         4 Call: reverse1([3],[2,1],_204) ?
     5           5 Call: reverse1([],[3,2,1],_204) ?
     5           5 Exit: reverse1([],[3,2,1],[3,2,1]) ?
     4         4 Exit: reverse1([3],[2,1],[3,2,1]) ?
     3        3 Exit: reverse1([2,3],[1],[3,2,1]) ?
     2       2 Exit: reverse1([1,2,3],[],[3,2,1]) ?
     1      1 Exit: reverse([1,2,3],[3,2,1]) ?

L = [3,2,1] ?

yes
```

Note: calling trace/0 turns on tracing. Calling notrace/0 turns it off.

Finding Paths

```
adj(1,2). adj(1,3).
adj(2,3). adj(2,4).
adj(3,4). adj(5,6).

adjacent(N1,N2) :- adj(N1,N2).
adjacent(N1,N2) :- adj(N2,N1).

connected(From,To) :-
    go(From,To,[From]).

go(From,To,Passed) :-
  adjacent(From,To),

  not(member(To,Passed)).

go(From,To,Passed) :-
  adjacent(From,Next),
  not(member(Next,Passed)),
  go(Next,To,[Next|Passed]).
```



1 IS CONNECTED TO 2,3,4

Paths:
12
13
123
124
132
134
1243
1234
1342
1324

“Pure Prolog” and non-logical built-ins

- All the examples so far have been “pure Prolog”
  - Contain no built-ins with non-logical side-effects
- Prolog has many built-in predicates that have such side-effects:
  - Type checking of terms
  - Arithmetic
  - Control execution
  - Input and output
  - Modify the program during execution (assert, retract, etc.)
  - Perform aggregation operations
- Use of non-logical built-in predicates usually effects the reversability of your program.

UMBC
an Honors University in Maryland

The End

UMBC
an Honors University in Maryland