Logic Programming Languages

BBM 301 – Programming Languages

Introduction

- Programs in logic languages are expressed in a form of symbolic logic
- Use a logical inferencing process to produce results
- *Declarative* rather that *procedural*:
  - Only specification of *results* are stated (not detailed *procedures* for producing them)

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Proposition

- A logical statement that may or may not be true
  - Consists of objects and relationships of objects to each other

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Symbolic Logic

- Logic which can be used for the basic needs of formal logic:
  - Express propositions
  - Express relationships between propositions
  - Describe how new propositions can be inferred from other propositions
- Particular form of symbolic logic used for logic programming called *predicate calculus*

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Object Representation

- Objects in propositions are represented by simple terms: either constants or variables
- *Constant*: a symbol that represents an object
- *Variable*: a symbol that can represent different objects at different times
  - Different from variables in imperative languages

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Compound Terms

- *Atomic propositions* consist of compound terms
- *Compound term*: one element of a mathematical relation, written like a mathematical function
  - Mathematical function is a mapping
  - Can be written as a table

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Parts of a Compound Term

- Compound term composed of two parts
  - Functor: function symbol that names the relationship
  - Ordered list of parameters (tuple)
- Examples:

```
student(jon)
like(seth, OSX)
like(nick, windows)
like(jim, linux)
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Forms of a Proposition

- Propositions can be stated in two forms:
  - *Fact*: proposition is assumed to be true
  - *Query*: truth of proposition is to be determined
- Compound proposition:
  - Have two or more atomic propositions
  - Propositions are connected by operators

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Logical Operators

Name	Symbol	Example	Meaning
negation	¬	¬ a	not a
conjunction	∩	a ∩ b	a and b
disjunction	∪	a ∪ b	a or b
equivalence	≡	a ≡ b	a is equivalent to b
implication	⊃ ⊂	a ⊃ b a ⊂ b	a implies b b implies a

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Quantifiers

Name	Example	Meaning
universal	$\forall X.P$	For all X, P is true
existential	$\exists X.P$	There exists a value of X such that P is true

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Clausal Form

- Too many ways to state the same thing
- Use a standard form for propositions
- *Clausal form*:
  - $B_1 \cup B_2 \cup \ldots \cup B_n \subset A_1 \cap A_2 \cap \ldots \cap A_m$
  - means if all the As are true, then at least one B is true
- *Antecedent*: right side
- *Consequent*: left side
- All predicate calculus propositions can be algorithmically converted to clausal form.

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Example Clausal Forms

$$likes(bob, trout) \subset likes(bob, fish) \cap fish(trout)$$

if bob likes fish and a trout is a fish, then bob likes trout.

$$father(louis, al) \cup father(louis, violet) \subset father(al, bob) \cap mother(violet, bob) \cap grandfather(louis, bob)$$

if al is bob's father and violet is bob's mother and louis is bob's grandfather, then louis is either al's father or violet's father

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Predicate Calculus and Proving Theorems

- A use of propositions is to discover new theorems that can be inferred from known axioms and theorems
- *Resolution*: an inference principle that allows inferred propositions to be computed from given propositions

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Concept of Resolution

Suppose there are two propositions of the form:

$$P_1 \subset P_2$$
$$Q_1 \subset Q_2$$

Suppose P1 is identical to Q2

$$T \subset P_2$$
$$Q_1 \subset T$$

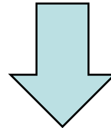
We can write

$$Q_1 \subset P_2$$

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Concept of Resolution

older(joanne, jake) ⊂ mother(joanne, jake)
wiser(joanne, jake) ⊂ older(joanne, jake)



wiser(joanne, jake) ⊂ mother(joanne, jake)

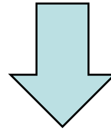
The mechanics:

- **Terms of the left sides are OR'd**
- **Terms of the right sides are AND'd.**
- **Any term that appears on both sides is removed.**

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Concept of Resolution

$$\text{father(bob, jake)} \cup \text{mother(bob, jake)} \subset \text{parent(bob, jake)}$$
$$\text{grandfather(bob, fred)} \subset \text{father(bob, jake)} \cap \text{father(jake, fred)}$$



$$\text{mother(bob, jake)} \cup \text{grandfather(bob, fred)} \subset \text{parent(bob, jake)} \cap \text{father(jake, fred)}$$

The mechanics:

- **Terms of the left sides are OR'd**
- **Terms of the right sides are AND'd.**
- **Any term that appears on both sides is removed.**

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Resolution

- *Unification*: finding values for variables in propositions that allows matching process to succeed
- *Instantiation*: assigning temporary values to variables to allow unification to succeed
- After instantiating a variable with a value, if matching fails, may need to *backtrack* and instantiate with a different value

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Proof by Contradiction

- *Hypotheses*: a set of pertinent propositions
- *Goal*: negation of theorem stated as a proposition
- Theorem is proved by finding an inconsistency

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Theorem Proving

- Basis for logic programming
- When propositions used for resolution, only restricted form can be used
- *Horn clause* - can have only two forms
  - *Headed*: single atomic proposition on left side
  - *Headless*: empty left side (used to state facts)
- Most propositions can be stated as Horn clauses

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Overview of Logic Programming

- Declarative semantics
  - There is a simple way to determine the meaning of each statement
  - Simpler than the semantics of imperative languages
- Programming is nonprocedural
  - Programs do not state now a result is to be computed, but rather the form of the result
  - *Programming in both imperative and functional languages is procedural*, which means that the programmer instructs the computer on exactly how the computation is to be done.

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Example: Sorting a List

- Describe the characteristics of a sorted list, not the process of rearranging a list

sort(old_list, new_list) $\subset$ permute (old_list, new_list) $\cap$ sorted (new_list)

sorted (list) $\subset$ $\forall_j$ such that $1 \leq j < n$, list(j) $\leq$ list (j+1)

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

The Origins of Prolog

- University of Aix-Marseille (Calmerauer & Roussel)
  - Natural language processing
- University of Edinburgh (Kowalski)
  - Automated theorem proving

This book uses the Edinburgh syntax of Prolog

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Terms

- *Term*: a constant, variable, or structure
- *Constant*: an atom or an integer
- *Atom*: symbolic value of Prolog
- Atom consists of either:
  - a string of letters, digits, and underscores beginning with a lowercase letter
  - a string of printable ASCII characters delimited by apostrophes

  e.g: elephant, xYZ, a_123, ‘Another atom’

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Terms

- *Variable*: any string of letters, digits, and underscores beginning with an uppercase letter
- e.g: X, Elephant, _G10, MyVariable, _

- *Instantiation*: binding of a variable to a value
  - Lasts only as long as it takes to satisfy one complete goal, which involves the proof or disproof of one proposition

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Terms

- *Structure*: represents atomic proposition

  functor (*parameter list*)

  – Functor is an atom, parameter list can be atoms, variables or other structures

  – sibling(jack, jill).

  – F(g(Alpha,)), 7).

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Fact Statements

- Used for the hypotheses / database
  - To define something as being unconditionally true
- Headless Horn clauses

```
female(shelley).
male(bill).
father(bill, jake).
father(bill, shelley).
mother(mary, jake).
mother(mary, shelley).
```

Notice that every Prolog statement is terminated by a period.

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Rule Statements

Headed Horn clause

The general form of the Prolog headed Horn clause statement:

head :- body.

Head of a rule is true if all predicates in the body can be proved as true

consequence :- antecedent_expression.

Right side: *antecedent* (***if*** part)

May be single term or conjunction

Left side: *consequent* (***then*** part)

Must be single term

"consequence can be concluded if the antecedent expression is true or can be made to be true by some instantiation of its variables."

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Example Rules

```
ancestor(mary,shelley):- mother(mary,shelley).
```

- *Conjunction*: multiple terms separated by logical AND operations (implied)

```
doughter(shelley):-female(shelley), child(shelley).
```

- Can use variables (*universal objects*) to generalize meaning:

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

grandparent(X,Z):- parent(X,Y), parent(Y,Z).
```

Here X, Y, Z are variables (they start with uppercase letters)

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Goal Statements

- For theorem proving, theorem is in form of proposition that we want system to prove or disprove – *goal statement*
- Same format as headless Horn

  `man(fred)`

- `yes` means that the system has proved the goal was true under the given database of facts and relationships
- `No` means that goal was determined to be false or the system was simply unable to prove it.

Conjunctive propositions and propositions with variables also legal goals.

- When variables are present, the system not only asserts the validity of the goal but also identifies the instantiations of the variables that make the goal true.

`father(X, mike).`

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Inferencing Process of Prolog

- Queries are called **goals**
- If a goal is a compound proposition, each of the facts is a **subgoal**
- To prove a goal is true must find a chain of inference rules and/or facts. For goal Q,

  either Q must be found as a fact in the database or

  the inferencing process must find a fact P1 and a sequence of propositions P2, P3, c, Pn such that

  P_2 :- P_1

  P_3 :- P_2

  …

  Q :- P_n
- Process of proving a subgoal called **matching**, satisfying, or resolution

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Query : `man(bob).`

1:

```
man(bob).
```

2:

```
father(bob).
man(X) :- father(X).
```

What happens if the query is

`man(X).`

Approaches

- *Matching* is the process of proving a proposition
- Proving a subgoal is called *satisfying* the subgoal
- *Bottom-up resolution, forward chaining*
  - Begin with facts and rules of database and attempt to find sequence that leads to goal
  - Works well with a large set of possibly correct answers
- *Top-down resolution, backward chaining*
  - Begin with goal and attempt to find sequence that leads to set of facts in database
  - Works well with a small set of possibly correct answers
- Prolog implementations use backward chaining

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Subgoal Strategies

- When goal has more than one subgoal, can use either
  - Depth-first search: find a complete proof for the first subgoal before working on others
  - Breadth-first search: work on all subgoals in parallel
- Prolog uses depth-first search
  - Can be done with fewer computer resources

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Backtracking

- With a goal with multiple subgoals, if fail to show truth of one of subgoals, reconsider previous subgoal to find an alternative solution: *backtracking*
- Begin search where previous search left off
- Can take lots of time and space because may find all possible proofs to every subgoal

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

```
male(X), parent(X, shelley).

male(bill).
male(dan).
male(mike).
Parent(mike, shelley).
```

Prolog finds the first fact in the database with male as its functor. It then instantiates X to the parameter of the found fact, say mike. Then, it attempts to prove that parent(mike, shelley) is true. If it fails, it backtracks to the first subgoal, male(X), and attempts to resatisfy it with some alternative instantiation of X.

example goal might be processed more efficiently if the order of the two subgoals were reversed. Then, only after resolution had found a parent of shelley would it try to match that person with the male subgoal. This is more efficient if shelley has fewer parents than there are males in the database

Simple Arithmetic

- Prolog supports integer variables and integer arithmetic
- **`is`** operator: takes an arithmetic expression as right operand and variable as left operand

```
A is B / 17 + C
```

- Not the same as an assignment statement!
  - The following is illegal (Left side variable cannot be previously instantiated)

```
Sum is Sum + Number.
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Example

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).
distance(X,Y) :-   speed(X,Speed),
                      time(X,Time),
                      Y is Speed * Time.
```

A query: `distance(chevy, Chevy_Distance).`

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Trace

- Built-in structure that displays instantiations at each step
- *Tracing model* of execution - four events:
  – *Call* (beginning of attempt to satisfy goal)
  – *Exit* (when a goal has been satisfied)
  – *Redo* (when backtrack occurs)
  – *Fail* (when goal fails)

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

```
trace.
distance(chevy, Chevy_Distance).
(1) 1 Call: distance(chevy, _0)?
(2) 2 Call: speed(chevy, _5)?
(2) 2 Exit: speed(chevy, 105)
(3) 2 Call: time(chevy, _6)?
(3) 2 Exit: time(chevy, 21)
(4) 2 Call: _0 is 105*21?
(4) 2 Exit: 2205 is 105*21
(1) 1 Exit: distance(chevy, 2205)
Chevy_Distance = 2205
```

Example

```
likes(jake,chocolate).
likes(jake, apricots).
likes(darcie, licorice).
likes(darcie, apricots).

trace.
likes(jake, X), likes(darcie, X).
(1)  1 Call: likes(jake, _0)?
(1) 1 Exit: likes(jake, chocolate)
(2) 1 Call: likes(darcie, chocolate)?
(2) 1 Fail: likes(darcie, chocolate)
(1)  1 Redo: likes(jake, _0)?
(1) 1 Exit: likes(jake, apricots)
(3) 1 Call: likes(darcie, apricots)?
(3) 1 Exit: likes(darcie, apricots)
X = apricots
```

Call Fail

likes (jake, X)

Exit Redo

Call Fail

likes (darcie, X)

Exit Redo

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Recall

Programs are list of facts and rules

A fact declares something as being true

A rule states conditions for a statement being true

Answering a query means proving that the goal represented by that query can be satisfied.

- if a goal matches with a fact then it is satisfied
- If a goal matches the head of a rule, then it is satisfied if the goal represented by the rule's body is satisfied
- If a rule consists of several subgoals separated by commas, then it is satisfied if all its subgoals are satisfied

Adapted from Ulle Endriss(University of Amsterdam)

Example

Consider the following argument

All men are mortal

Socrates is a man

Hence, Socrates is mortal.

It has two premises and a conclusion

The corresponding Prolog program is :

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

?-mortal(socrates).
Yes.
```

Adapted from Ulle Endriss(University of Amsterdam)

Examples

```
bigger(elephant, horse).
bigger(horse,donkey).
bigger(donkey, dog).
bigger(donkey, monkey).

Queries:
?-bigger(donkey,dog).
Yes
?-bigger(monkey, elephant).
No

?-bigger(elephant, monkey).
No
```

Adapted from Ulle Endriss(University of Amsterdam)

Examples

```
bigger(elephant, horse).
bigger(horse,donkey).
bigger(donkey, dog).
bigger(donkey, monkey).

is_bigger(X,Y) :- bigger(X,Y).
is_bigger(X,Y) :- bigger(X,Z), is_bigger(Z,Y).

?-is_bigger(elephant, monkey).
Yes

?-is_bigger(X, donkey).
X=horse;
X=elephant;
No
```

Adapted from Ulle Endriss(University of Amsterdam)

Examples

```
bigger(elephant, horse).
bigger(horse,donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
is_bigger(X,Y) :- bigger(X,Y).
is_bigger(X,Y) :- bigger(X,Z), is_bigger(Z,Y).

?-is_bigger(donkey, X), is_bigger(X, monkey).
No

?-is_bigger(horse, X), is_bigger(X, dog).
X= donkey
No
```

Adapted from Ulle Endriss(University of Amsterdam)

More examples

```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

```
?-food(carrot).
Yes
?-food(pickle).
No
?-food(Edible).
Edible = apple ;
Edible = broccoli ;
Edible = carrot ;
Edible = lettuce ;
Edible = rice ;
No
```

Adapted from William Mitchell (University of Arizona)

More examples

```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).

color(sky, blue).
color(soil, brown).
color(grass, green).
color(broccoli, green).
color(lettuce, green).
color(apple, red).
color(carrot, orange).
color(rice, white).
color(rose, red).
color(tomato, red).
```

```
?- food(F), color(F, green).
F = broccoli ;
F = lettuce ;
No

?- color(F, blue), food(F).
No
```

Adapted from William Mitchell (University of Arizona)

More examples

```
food(apple).
…
color(sky, blue).
…
likes(bob, carrot).
likes(bob, apple).
likes(joe, lettuce).
likes(mary, broccoli).
likes(mary, tomato).
likes(bob, mary).
likes(mary, joe).
likes(joe, baseball).
likes(mary, baseball).
likes(jim, baseball).
```

```
Who likes baseball
?- likes(Who, baseball).

Whol likes a food
?- likes(Who, X), food(X).

Who likes green foods
?- likes(Who,X),food(X),color(X, green).

Who likes foods with the same color as
foods that Mary likes
?- likes(mary,F1), food(F1), color(F1,C),
   likes(Who,F2), food(F2), color(F2,C),
```

Adapted from William Mitchell (University of Arizona)

More examples

```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).

color(sky, blue).
color(soil, brown).
color(grass, green).
color(broccoli, green).
color(lettuce, green).
color(apple, red).
color(carrot, orange).
color(rice, white).
color(rose, red).
color(tomato, red).
```

Are there two foods with the same color?

```
?- food(F1), food(F2), color(F1,C),
color(F2,C).
F1 = apple
F2 = apple
C = red;

F1 = broccoli
F2 = broccoli
C = green;

?- food(F1), food(F2), F1 \== F2,
color(F1,C), color(F2,C).
F1 = broccoli
F2 = lettuce
C = green
```

Adapted from William Mitchell (University of Arizona)

Matching/Unification

Recall that two terms match if they are either identical or if they ca be made identical by substituting their variables with suitable ground terms.

We can explicitly ask Prolog whether two given terms match by using equality predicate = (which we will read as unify)

```
?-born(mary, newyork) = born(mary, X).
X = newyork
Yes
```

```
?- f(a, g(X,Y)) = f(X,Z), Z = g(W,h(X)).
X = a
Y = h(a)
Z = g(a, h(a))
W = a
Yes
```

Adapted from Ulle Endriss(University of Amsterdam)

Matching

```
?-p(X,2,2) = p(1,Y,X).
No

?- p(_,2,2) = p(1,Y,_).
Y=2
Yes
```

_ (underscore) is called the anonymous variable

Every occurrence of _ represents a different variable

The instantiations are not being reported

Adapted from Ulle Endriss(University of Amsterdam)

“=” and “is”

```
| ?- p(X, b, f(c,Y)) = p(a, U, f(V,U)).
U = b,
V = c,
X = a,
Y = b ?
yes
| ?- X is 2*(3+6).
X = 18 ?
yes
| ?- X = 2*(3+6).
X = 2*(3+6) ?
yes
| ?- X is 2*(3+6), Y is X/3.
X = 18,
Y = 6.0 ?
yes
| ?- Y is X/3, X is 2*(3+6).
! Instantiation error in argument 2 of is/2
! goal:  _76 is _73/3
```

Adapted from Stuart C. Shapiro (SUNY, Buffalo)

Unification, continued

We can think of the query

```
?- food(carrot).
```

as a search for facts that can be unified with `food(apple)`.

Here's a way to picture how Prolog considers the first fact, which is `food(apple)`.

```
?- Fact = food(apple), Query = food(carrot), Fact = Query.
No
```

The demands of the three unifications cannot be satisfied simultaneously . The same is true for the second fact, `food(broccoli)`.

The third fact produces a successful unification:

```
?- Fact = food(carrot), Query = food(carrot), Fact = Query.
Fact = food(carrot)
Query = food(carrot)
Yes
```

The instantiations for `Fact` and `Query` are shown, but are no surprise.

Adapted from William Mitchell (University of Arizona)

Unification, continued

Things are more interesting when the query involves a variable, like `?- food(F).`

```
?- Fact = food(apple), Query = food(F), Query = Fact.
Fact = food(apple)
Query = food(apple)
F = apple
```

The query succeeds and Prolog shows that `F` has been instantiated to `apple`.

Unification, continued

Consider again this interaction:

```
?- food(F).
F = apple ;
F = broccoli ;
F = carrot ;
F = lettuce ;
F = rice ;
No
```

Prolog first finds that `food(apple)` can be unified with `food(F)` and shows that `F` is instantiated to `apple`.

When the user types semicolon `F` is uninstantiated and the search for another fact to unify with `food(F)` resumes.

`food(broccoli)` is unified with `food(F)`, `F` is instantiated to `broccoli`, and the user is presented with `F = broccoli`.

The process continues until Prolog has found all the facts that can be unified with `food(F)` or the user is presented with a value for `F` that is satisfactory.

Adapted from William Mitchell (University of Arizona)

Unification, continued

Following an earlier example, here's how we might view successful unifications with the query `?- food(F), color(F,C)`:

```
?- Fact1 = food(lettuce), Fact2 = color(lettuce,green),
   Query1 = food(F), Query2 = color(F,C),
   Fact1 = Query1, Fact2 = Query2.

C = green
F = lettuce
...
```

Only the interesting instantiations, for `F` and `C`, are shown above

What we see is that unifying `Fact1` with `Query1` causes `F` to be instantiated to `lettuce`.

`Query2`, which due to the value of `F` is effectively `color(lettuce,C)`, can be unified with `Fact2` if `C` is instantiated to `green`.

Unification and variable instantiation are cornerstones of Prolog.

Adapted from William Mitchell (University of Arizona)

Query execution

Goals, like `food(fries)` or `color(What, Color)` can be thought of as having four *ports*:

In the `Active Prolog Tutor`, Dennis Merritt describes the ports in this way:

call: Using the current variable bindings, begin to search for the facts which unify with the goal.

exit: Set a place marker at the fact which satisfied the goal. Update the variable table to reflect any new variable bindings. Pass control to the right.

redo: Undo the updates to the variable table [that were made by this goal]. At the place marker, resume the search for a clause which unifies with the goal.

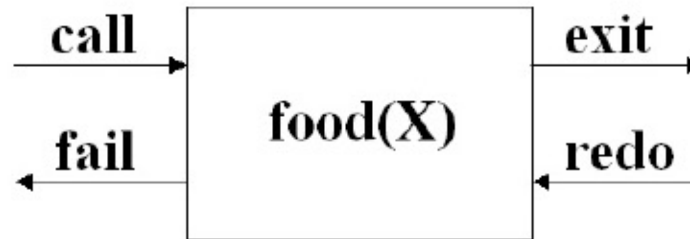
fail: No (more) clauses unify, pass control to the left.

Adapted from William Mitchell (University of Arizona)

Query execution, continued

Example:

```
?- food(X).
X = apple ;
X = broccoli ;
X = carrot ;
X = lettuce ;
X = rice ;
No
```



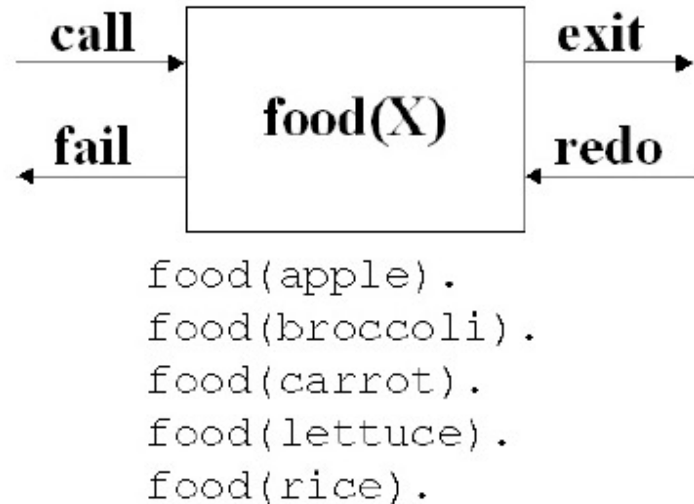

```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

Adapted from William Mitchell (University of Arizona)

Query execution, continued

The goal `trace/0` activates "tracing" for a query. Here's what it looks like:

```
?- trace, food(X).
   Call: food(_G410) ? <CR>
   Exit: food(apple) ? <CR>
X = apple ;
   Redo: food(_G410) ? <CR>
   Exit: food(broccoli) ? <CR>
X = broccoli ;
   Redo: food(_G410) ? <CR>
   Exit: food(carrot) ? <CR>
X = carrot ;
```




```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

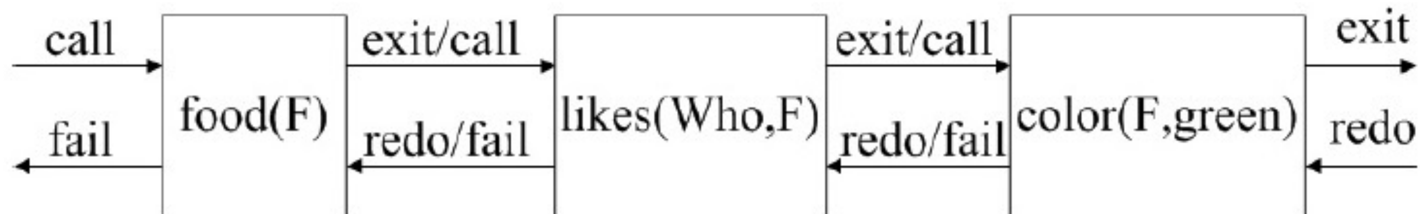
Tracing shows the transitions through each port. The first transition is a call on the goal `food(X)`. The value shown, `_G410`, stands for the uninstantiated variable `X`. We next see that goal being exited, with `X` instantiated to `apple`. The user isn't satisfied with the value and by typing a semicolon forces the redo port to be entered, which causes X, previously bound to `apple`, to be uninstantiated. The next food fact, `food(broccoli)` is tried, instantiating `X` to `broccoli`, exiting the goal, and presenting `X = broccoli` to the user. (etc.)

Adapted from William Mitchell (University of Arizona)

Query execution, continued

Query: Who likes green foods?

```
?- food(F), likes(Who,F), color(F, green).
```



Facts:

```
food(apple).       likes(bob, carrot).      color(sky, blue).
food(broccoli).    likes(bob, apple).       color(dirt, brown).
food(carrot).      likes(joe, lettuce).     color(grass, green).
food(lettuce).     likes(mary, broccoli).   color(broccoli,green).
food(rice).        likes(mary, tomato).     color(lettuce, green).
                   likes(bob, mary).        color(apple, red).
                   likes(mary, joe).        color(tomato, red).
                   likes(joe, baseball).    color(carrot, orange).
                   likes(mary, baseball).   color(rose, red).
                   likes(jim, baseball).    color(rice, white).
```

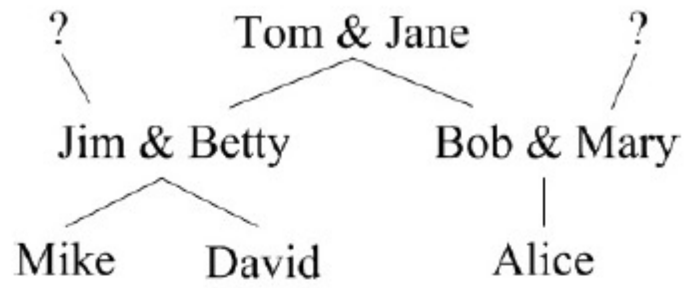
Try tracing it!

Adapted from William Mitchell (University of Arizona)

Another example of rules

Here is a set of facts describing parents and children:

```
male(tom).        parent(tom,betty).
male(jim).        parent(tom,bob).
male(bob).        parent(jane,betty).
male(mike).       parent(jane,bob).
male(david).      parent(jim,mike).
                  parent(jim,david).
female(jane).     parent(betty,mike).
female(betty).    parent(betty,david).
female(mary).     parent(bob,alice).
female(alice).    parent(mary,alice).
```



`parent(P,C)` is read as "P is a parent of C".

Problem: Define rules for `father(F,C)` and `grandmother(GM,GC)`.

Adapted from William Mitchell (University of Arizona)

Another example, continued

```
father(F,C) :- parent(F,C), male(F).
mother(M,C) :- parent(M,C), female(M).

grandmother(GM,GC) :- parent(P,GC), mother(GM,P).
```

Who is Bob's father?

For who is Tom the father?

What are all the father/child relationships?

What are all the father/daughter relationships?

What are the grandmother/grandchild relationships?

Problems: Define `sibling(A,B)`, such that "`A` is a sibling of `B`".
Using `sibling`, define `brother(B,A)` such that "`B` is `A`'s brother".

Adapted from William Mitchell (University of Arizona)

Another example, continued

```
sibling(A,B) :- father(F,A), mother(M,A),
                father(F,B), mother(M,B), A \== B.
```

Queries:

Is Mike a sibling of Alice?

What are the sibling relationships?

Who is somebody's brother?

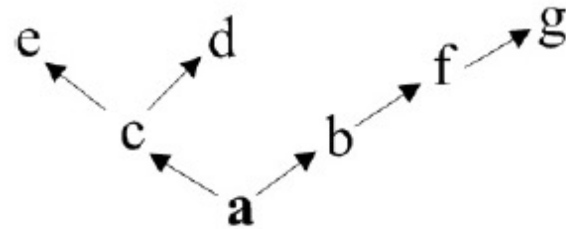
Is the following an equivalent definition of `sibling`?

```
sibling2(S1,S2) :- parent(P,S1), parent(P,S2), S1 \== S2.
```

Recursive predicates

Consider an abstract set of parent/child relationships:

```
parent(a,b).   parent(c,d).
parent(a,c).   parent(b,f).
parent(c,e).   parent(f,g).
```



If a predicate contains a goal that refers to itself the predicate is said to be recursive.

```
ancestor(A,X) :- parent(A, X).
ancestor(A,X) :- parent(P, X), ancestor(A,P).
```

"A is an ancestor of X if A is the parent of X or P is the parent of X and A is an ancestor of P."

```
?- ancestor(a,f).                % Is a an ancestor of f?
Yes

?- ancestor(c,b).                % Is c an ancestor of b?
No

?- ancestor(c,Descendant).       % What are the descendants of b?
Descendent = e ;
Descendent = d ;
No
```

Adapted from William Mitchell (University of Arizona)

Computing with structures

The Prolog structure, which is the notation of a predicate symbol with arguments, can be treated as a data structure, and run computations on it. Consider the following arithmetic.

We introduce zero as a symbol zero, and the number X's successor as s(X).

This is called the Peano arithmetic.

For example, the number 5 is written as: `s(s(s(s(s(zero))))).`

We want to define addition with the sum(S1, S2, S3) predicate, which would be true if, and only if, the S3 argument was the sum of the first two arguments:

```
sum(zero, S1, S1).
sum(s(X), S1, s(S2)) :-sum(X, S1, S2).
```

Now we can conduct computations in this arithmetic, eg. to compute 3+4:

```
?-sum(s(s(s(zero))), s(s(s(s(zero)))), X).
X = s(s(s(s(s(s(s(zero)))))))
```

List Structures

- Other basic data structure (besides atomic propositions we have already seen): list
- *List* is a sequence of any number of elements
- Elements can be atoms, atomic propositions, or other terms (including other lists)

```
[apple, prune, grape, kumquat]
[]              (empty list)
[X | Y]         (head X and tail Y)
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

this_is_a_list([apricot, peach, pear]).

this_is_a_list([apple, apricot, peach, pear]).

In query mode, one of the elements of new_list can be dismantled into head and tail with

```
this_is_a_list([New_List_Head | New_List_Tail]).
```

the following are equivalent:

```
[apricot, peach, pear | []]
[apricot, peach | [pear]]
[apricot | [peach, pear]]
```

```
[a]

[X,Y]

[1,2,3,4]

[a,[1,X],[],[],a,[a]]

[1|[2|[3|[4|[]]]]] /* exactly equal to the list [1,2,3,4] */

[1|[2|[3|[4]]]] /* another way of writing the list [1,2,3,4] */

[1,2|[3,4]] /* this is also allowed and is the same [1,2,3,4] */
```

http://www.inf.ed.ac.uk/teaching/courses/aipp/

```
add_element([E|L], E, L).

?-add_element(L, apple , [apricot, peach]).
L = [apple, apricot, peach]
?-add_element(L, apple , []).
L = [apple]

get_element(E, [E|L]).

?-get_element(E, [apricot, peach]).
E = apricot
?-get_element(E, [apricot]).
E = apricot
```

Append Example

```
append([], List, List).
append([Head | List_1], List_2, [Head | List_3]) :-
        append (List_1, List_2, List_3).
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

```
trace.
append([bob, jo], [jake, darcie], Family).
(1) 1 Call: append([bob, jo], [jake, darcie], _10)?
(2) 2 Call: append([jo], [jake, darcie], _18)?
(3) 3 Call: append([], [jake, darcie], _25)?
(3) 3 Exit: append([], [jake, darcie], [jake, darcie])
(2) 2 Exit: append([jo], [jake, darcie], [jo, jake, darcie])
(1) 1 Exit: append([bob, jo], [jake, darcie], [bob, jo, jake, darcie])
Family = [bob, jo, jake, darcie]
yes
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

```
?-append(X, Y, [a, b, c]).
X = []
Y = [a, b, c]
;
X = [a]
Y = [b, c]
;
X = [a, b]
Y = [c]
;
X = [a, b, c]
Y = []
```

More Examples

```
member(Element, [Element | _]).
member(Element, [_ | List]) :-
                        member(Element, List).
```

The underscore character means an anonymous variable—it means we do not care what instantiation it might get from unification

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

- trace.
- member(a, [b, c, d]).
- (1) 1 Call: member(a, [b, c, d])?
- (2) 2 Call: member(a, [c, d])?
- (3) 3 Call: member(a, [d])?
- (4) 4 Call: member(a, [])?
- (4) 4 Fail: member(a, [])
- (3) 3 Fail: member(a, [d])
- (2) 2 Fail: member(a, [c, d])
- (1) 1 Fail: member(a, [b, c, d])
- no
- member(a, [b, a, c]).
- (1) 1 Call: member(a, [b, a, c])?
- (2) 2 Call: member(a, [a, c])?
- (2) 2 Exit: member(a, [a, c])
- (1) 1 Exit: member(a, [b, a, c])
- yes

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

More Examples

```
reverse([], []).
reverse([Head | Tail], List) :-
  reverse (Tail, Result),
           append (Result, [Head], List).
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

- trace.
- reverse([a, b, c], Q).
- (1) 1 Call: reverse([a, b, c], _6)?
- (2) 2 Call: reverse([b, c], _65636)?
- (3) 3 Call: reverse([c], _65646)?
- (4) 4 Call: reverse([], _65656)?
- (4) 4 Exit: reverse([], [])
- (5) 4 Call: append([], [c], _65646)?
- (5) 4 Exit: append([], [c], [c])
- (3) 3 Exit: reverse([c], [c])
- (6) 3 Call: append([c], [b], _65636)?
- (7) 4 Call: append([], [b], _25)?
- (7) 4 Exit: append([], [b], [b])
- (6) 3 Exit: append([c], [b], [c, b])
- (2) 2 Exit: reverse([b, c], [c, b])
- (8) 2 Call: append([c, b], [a], _6)?
- (9) 3 Call: append([b], [a], _32)?
- (10) 4 Call: append([], [a], _39)?
- (10) 4 Exit: append([], [a], [a])
- (9) 3 Exit: append([b], [a], [b, a])
- (8) 2 Exit: append([c, b], [a], [c, b, a])
- (1) 1 Exit: reverse([a, b, c], [c, b, a])

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Backtracking

```
bird(tweety).
bird(oscar).
bird(X) :- feathered(X).
feathered(maggie).
large(oscar).
ostrich(X) :- bird(X), large(X).
```

trace, (*ostrich*(oscar)).

Call:*ostrich*(oscar)
Call:*bird*(oscar)
Exit:*bird*(oscar)
Call:*large*(oscar)
Exit:*large*(oscar)
Exit:*ostrich*(oscar)

trace, (*ostrich*(X)).

Call:*ostrich*(_4470)
Call:*bird*(_4470)
Exit:*bird*(tweety)
Call:*large*(tweety)
Fail:*large*(tweety)
Redo:*bird*(_4470)
Exit:*bird*(oscar)
Call:*large*(oscar)
Exit:*large*(oscar)
Exit:*ostrich*(oscar)
X = oscar

Unwanted Backtracking & Cut

```
bird(tweety).
bird(oscar).
bird(X) :- feathered(X).
feathered(maggie).
large(oscar).
ostrich(X) :- bird(X), large(X).
```

trace, (*ostrich*(tweety)).

Call:*ostrich*(tweety)
Call:*bird*(tweety)
Exit:*bird*(tweety)
Call:*large*(tweety)
Fail:*large*(tweety)
Redo:*bird*(tweety)
Call:*feathered*(tweety)
Fail:*feathered*(tweety)
Fail:*bird*(tweety)
Fail:*ostrich*(tweety)
false

No need to try tweety again

```
bird(tweety).
bird(oscar).
bird(X) :- feathered(X).
feathered(maggie).
large(oscar).
ostrich(X) :- bird(X), !, large(X).
```

trace, (*ostrich*(tweety)).

Call:*ostrich*(tweety)
Call:*bird*(tweety)
Exit:*bird*(tweety)
Call:*large*(tweety)
Fail:*large*(tweety)
Fail:*ostrich*(tweety)
false

Fail : Forcing failure

```
bird(tweety).
bird(willy).
canary(tweety).
swims(willy).
penguin(X) :- canary(X), !, fail.
penguin(X) :- bird(X), swims(X).
```

If something is a canary it is not a penguin

trace, (*penguin*(willy)).

Call:*penguin*(willy)
Call:*canary*(willy)
Fail:*canary*(willy)
Redo:*penguin*(willy)
Call:*bird*(willy)
Exit:*bird*(willy)
Call:*swims*(willy)
Exit:*swims*(willy)
Exit:*penguin*(willy)
*1***true**

trace, (*penguin*(tweety)).

Call:*penguin*(tweety)
Call:*canary*(tweety)
Exit:*canary*(tweety)
Call:fail
Fail:fail
Fail:*penguin*(tweety)
false

Use cut when seeing if a ground atom is satisfied, not for generating satisfying instances

trace, (*penguin*(X)).
Call:*penguin*(_4902)
Call:*canary*(_4902)
Exit:*canary*(tweety)
Call:fail
Fail:fail
Fail:*penguin*(_4902)
false

Rule order

```
penguin(X) :- bird(X), swims(X).
penguin(X) :- canary(X), !, fail.
bird(X) :- canary(X).
bird(willy).
canary(tweety).
swims(willy).
```

bad

trace, (*penguin*(tweety)).

Call:*penguin*(tweety)
Call:*bird*(tweety)
Call:*canary*(tweety)
Exit:*canary*(tweety)
Exit:*bird*(tweety)
Call:*swims*(tweety)
Fail:*swims*(tweety)
Redo:*penguin*(tweety)
Call:*canary*(tweety)
Exit:*canary*(tweety)
Call:fail
Fail:fail
Fail:*penguin*(tweety)

```
penguin(X) :- canary(X), !, fail.
penguin(X) :- bird(X), swims(X).
bird(X) :- canary(X).
bird(willy).
canary(tweety).
swims(willy).
```

good

trace, (*penguin*(tweety)).

Call:*penguin*(tweety)
Call:*canary*(tweety)
Exit:*canary*(tweety)
Call:fail
Fail:fail
Fail:*penguin*(tweety)

Avoid left recursive rules

```
parent(a,b).
parent(b,c).
ancestor(X,Y):-parent(X,Y).
ancestor(X,Y):-parent(X,Z), ancestor(Y,Z).
```

Do not use the following:

ancestor(X,Y):- **ancestor(X,Z), parent(Y,Z).**

ancestor(X,Y):- **ancestor(X,Z), ancestor(Y,Z).**

Deficiencies of Prolog

- Resolution order control
  - In a pure logic programming environment, the order of attempted matches is nondeterministic and all matches would be attempted concurrently
- The closed-world assumption
  - The only knowledge is what is in the database
- The negation problem
  - Anything not stated in the database is assumed to be false
- Intrinsic limitations
  - It is easy to state a sort process in logic, but difficult to actually do—it doesn't know how to sort

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Applications of Logic Programming

- Relational database management systems
- Expert systems
- Natural language processing

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Summary

- Symbolic logic provides basis for logic programming
- Logic programs should be nonprocedural
- Prolog statements are facts, rules, or goals
- Resolution is the primary activity of a Prolog interpreter
- Although there are a number of drawbacks with the current state of logic programming it has been used in a number of areas

Copyright © 2017 Pearson Education, Ltd. All rights reserved.