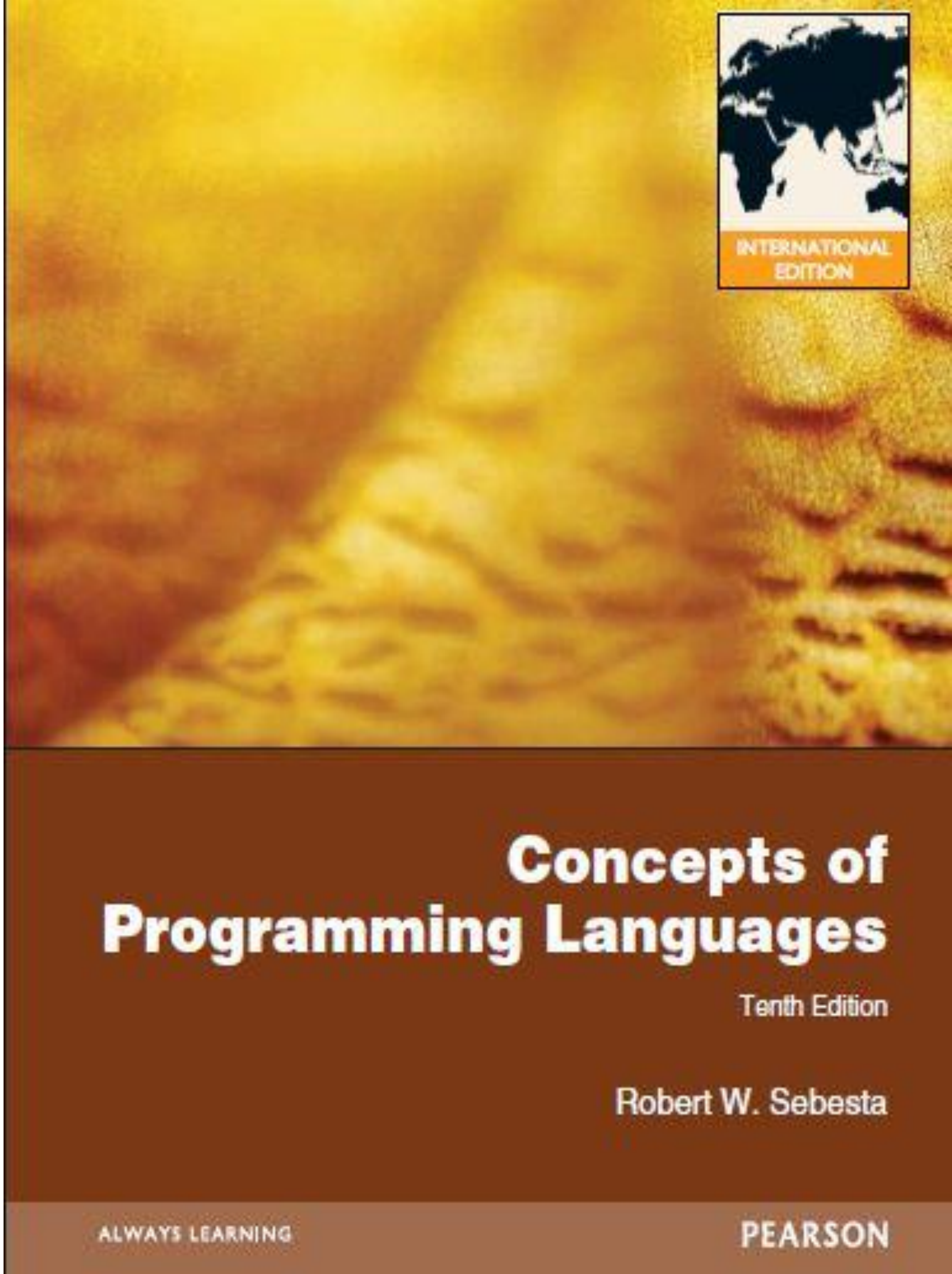
# Programming Language

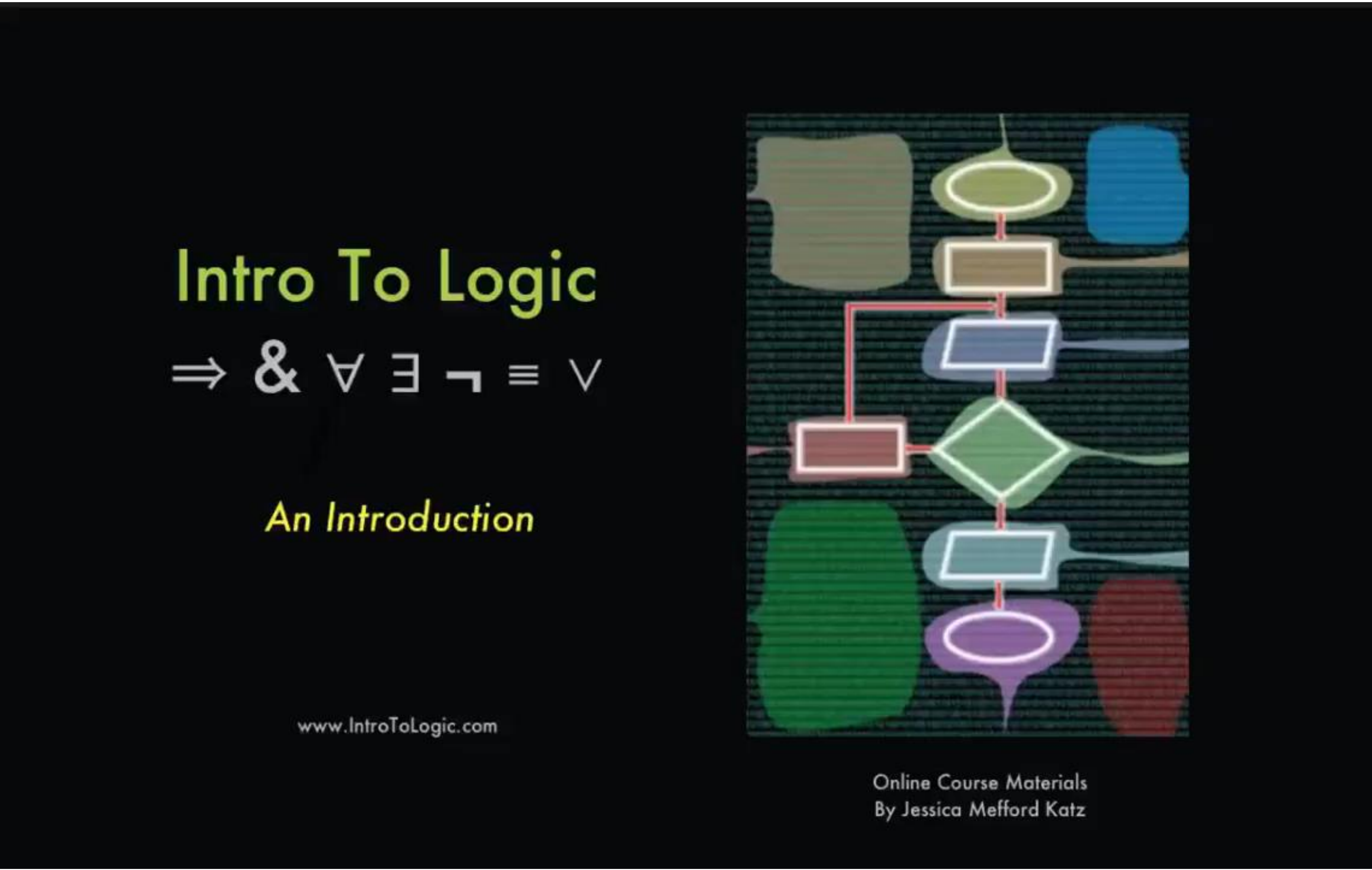



Instructor:
Min-Chun Hu
anita_hu@mail.ncku.edu.tw

# Lecture 6
# Logic Programming Languages

- Introduction
- A Brief Introduction to Predicate Calculus
- Predicate Calculus and Proving Theorems
- An Overview of Logic Programming
- The Origins of Prolog
- The Basic Elements of Prolog
- Deficiencies of Prolog
- Applications of Logic Programming

# Introduction



# 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
- Programs in logic programming languages are collections of facts and rules.
  - The program is used by asking it questions, and the program answers the question by consulting the facts and rules.

# Introduction

- Example: Sorting a list using a logic language
  - 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)

# Predicate Calculus

- *Predicate calculus*
  - A particular form of symbolic logic used for logic programming
  - Formally expresses logic statements
- *Proposition*
  - A logic statement that is either true or false
  - Consists of objects and relationships of objects to each other
- Translate logic statements into predicate calculus:
  - 0 is a natural number → Natural(0)
  - 2 is a natural number → Natural(2)
  - For all x, if x is a natural number, then so is the successor of x. → For all x, natural (x) ⊃ natural (successor (x))

# Logic and Logic Programs

- Axioms are logic statements that are assumed to be true
  - Natural (2)
- Symbolic logic is used for the basic needs of formal logic:
  - Express propositions
  - Express relationships between propositions
  - Describe how new propositions can be inferred from other propositions

# Objects and Connectives

- Objects in propositions are represented by simple terms: either constants or variables
  - *Constant* : A symbol that represents an object
    - natural(0) : constants are 0 and natural
  - *Variable* : A symbol that can represent different objects at different times (different from variables in imperative languages)
    - successor(x): x is a variable
- Connectives: indicate Boolean operations such as and, or, imply

# Compound Terms

- *Compound term*: one element of a mathematical relation, written like a mathematical function
  - Composed of function symbol (functor) that names the relationship and ordered list of parameters (tuple)
- Examples:

```
student(jon)
man(jake)
like(nick, linux)
```

# Forms of a Proposition

- Propositions can be stated in two forms:
  - *Fact*: proposition is assumed to be true
    - Eg: father(bob, bill).
  - *Query*: truth of proposition is to be determined
    - Eg: ?–father(bob, bill).

# Compound Propositions

- *Atomic propositions* : consists of compound terms, and the truth or falsity of the proposition does not depend on that of any other proposition.
  - man(jake) : 1-tuple compound terms
  - likes(bill, flower) : 2-tuple compound terms
- *Compound propositions* : two or more atomic propositions connected by logic connectors.
  - For all x, natural (x) ⊃ natural (successor (x))
  - likes(john, trout) ⊂ likes(john,fish) ∩ fish(trout)

# Logical Connectors/Operators

| Name | Symbol | Example | Meaning |
|---|---|---|---|
| negation | $\neg$ | $\neg$ a | not a |
| conjunction | $\cap$ | a $\cap$ b | a and b |
| disjunction | $\cup$ | a $\cup$ b | a or b |
| equivalence | $\equiv$ | a $\equiv$ b | a is equivalent to b |
| implication | $\supset$<br>$\subset$ | a $\supset$ b<br>a $\subset$ b | a implies b<br>b implies a |

Ex: $a \cap b \subset c$

Ex: $a \cap \neg b \subset d$

# 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 |

Ex: $\forall X.(\text{woman}(X) \supset \text{human}(X))$

$\exists X.(\text{mother}(\text{Mary}, X) \cap \text{male}(X))$

# Example

| Logic Statement | Predicate Calculus |
|---|---|
| A horse is a mammal. | mammal(horse) |
| A human is a mammal. | mammal(human) |
| A horse has no arms. | arms (horse,0) |
| Mammals have four legs and no arm, or two legs and two arms. | mammal(x) ⊃ ( legs(x,4) ∩ arm (x,0) ) ∪ ( legs(x,2) ∩ arm (x,2) ) |

# Clausal Form

- All predicate calculus propositions can be converted to *Clausal form*:
  - $B_1 \cup B_2 \cup \ldots \cup B_n \subset A_1 \cap A_2 \cap \ldots \cap A_m$
    means that if all the A's are true, then at least one B is true
  - *Antecedent*: right side; *Consequent*: left side
  - B is the head of the clause, and A's are the body of the clause
  - Example:

```
father(louis,al) ∪ father(louis,violet) ⊂
father(al,bob)∩ mother(violet,bob)∩ grandfather(louis,bob)
```

# Horn Clauses

- Horn *clauses:*
  - *Headed*: single atomic proposition on left side (used to state relationship)
    - `likes(bob,trout) ⊂ likes(bob,fish) ∩ fish(trout)`
  - *Headless*: empty left side (used to state facts)
    - `father(bob,jake)`
  - Most, but not all propositions can be stated as Horn clauses

# 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
  - $P1 \subset P2 \qquad Q1 \subset Q2$

    If $(P1 == Q2) \Rightarrow Q1 \subset P2$
  - Example:

```
    older(joanne,jake) ⊂ mother(joanne,jake)
    wiser(joanne,jake) ⊂ older(joanne,jake)
=> wiser(joanne,jake) ⊂ mother(joanne,jake)
```

# Predicate Calculus and Proving Theorems

- Example of resolution:

  `father(bob,jake)` $\cup$ `mother(bob,jake)` $\subset$ `parent(bob,jake)`

  `grandfather(bob,fred)` $\subset$ `father(bob,jake)` $\cap$

  `father(jake,fred)`

  `mother(bob,jake)` $\cup$ `grandfather(bob,fred)` $\subset$ `parent(bob,jake)` $\cap$ `father(jake,fred)`

# Unification

- *Unification*: find values for variables in propositions that allows matching process to succeed

    eats(Frank, apple)

    ?-eats(Frank,X)

    X=apple

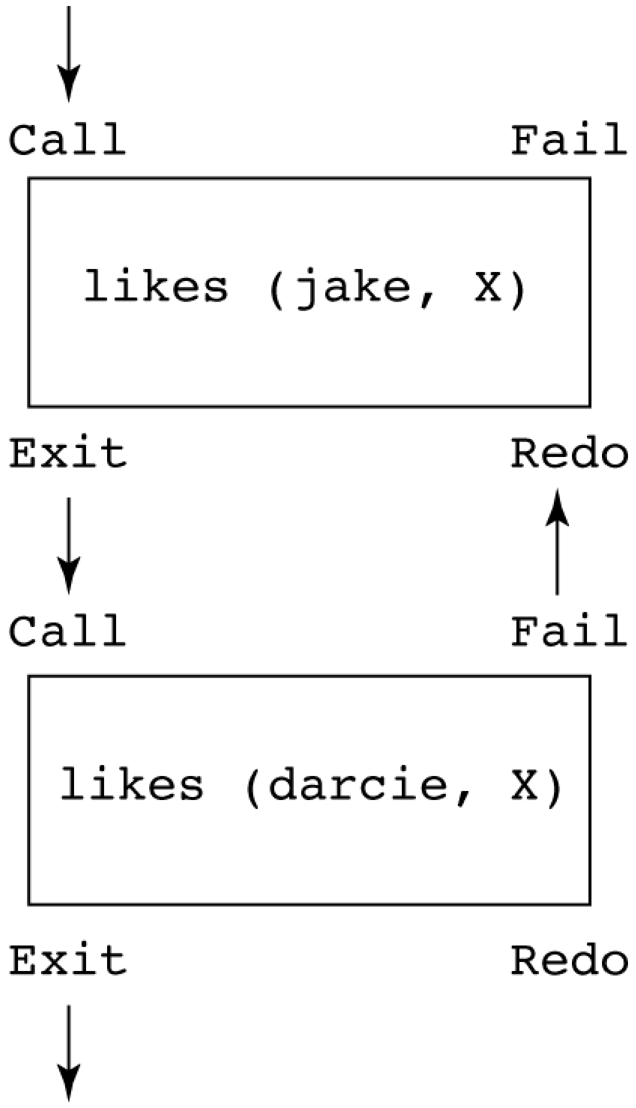
    Yes

- *Instantiation*: assign 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

# 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
```



# Proof by Contradiction

- *Hypotheses*:
  - a set of pertinent propositions
- *Goal*:
  - negation of theorem stated as a proposition
- Theorem is proved by finding an inconsistency
- Proving a theorem by contradiction results in high time complexity.

# Introduction of Prolog

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

# Terms

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

# Terms (Cont.)

- *Variable*: any string of letters, digits, and underscores beginning with an uppercase letter or an underscore (_)
- *Instantiation*: binding of a variable to a value
  - Lasts only as long as it takes to satisfy one complete goal, involving proof or disproof of one proposition
- *Structure*: represents atomic proposition
  - State relationships among terms
  - General form:

    functor (*parameter list*)

# Fact Statements

- Used for the hypotheses
- Headless Horn clauses

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

# Rule Statements

- Used for the hypotheses
- Headed Horn clause
  - Right side: *antecedent* (*if* part)
    - May be single term or conjunction
  - Left side: *consequent* (*then* part)
    - Must be single term
  - *Conjunction*: multiple terms separated by logical AND operations (implied)
    - Example: `Female(shelly), child(shelly).`
- General form:
  - Consequence :- antecedent_expression.
    - Example:

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

# Example Rules

- 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).
```

# Goal Statements

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

```
man(fred).
```

- Conjunctive propositions and propositions with variables are also legal goals

```
father(X, mike).
```

# 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:

```
P2 :- P1
P3 :- P2
...
Q :- Pn
```

- Process of proving a subgoal called matching, satisfying, or resolution

# Inferencing Process of Prolog

- Consider the following query: `man(bob).`
  - If the database includes the same fact, the proof is trivial.
  - If the database contains:

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

Prolog would use them to infer the truth of the goal and this would instantiate `X` temporarily to `bob`.

# Approaches of Matching

- *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

# Backtracking

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

# 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

# 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)

# Example: Factorial

```
factorial(0,1).
```

Clause 1 (a unit clause)

```
factorial(N,Result) :-
    N>0,
    N1 is N-1,
    factorial(N1,Result1),
    Result is N * Result1.
```

Body

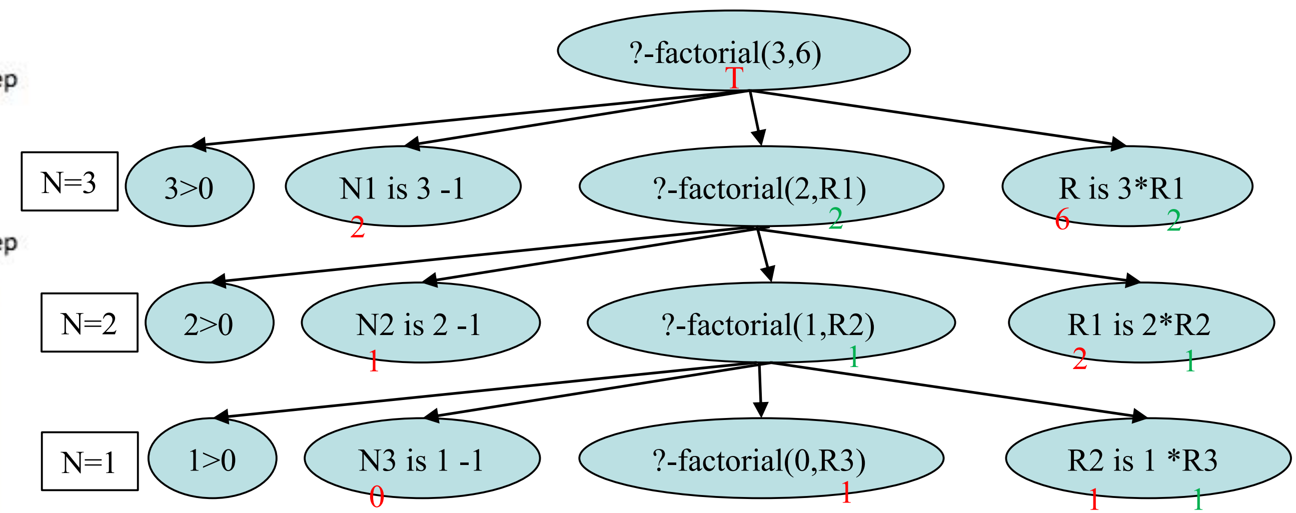
Clause 2 (a rule)

?- factorial(3,W).
W=6
?- factorial(3,6).
yes
?- factorial(5,2).
no

# Trace Example

```
factorial(N,Result) :-
    N>0,
    N1 is N-1,
    factorial(N1,Result1),
    Result is N * Result1.
```

```
[trace]  ?- factorial(3,6).
   Call: (6) factorial(3, 6) ? creep
   Call: (7) 3>0 ? creep
   Exit: (7) 3>0 ? creep
   Call: (7) _G2679 is 3+ -1 ? creep
   Exit: (7) 2 is 3+ -1 ? creep
   Call: (7) factorial(2, _G2680) ? creep
   Call: (8) 2>0 ? creep
   Exit: (8) 2>0 ? creep
   Call: (8) _G2682 is 2+ -1 ? creep
   Exit: (8) 1 is 2+ -1 ? creep
   Call: (8) factorial(1, _G2683) ? creep
   Call: (9) 1>0 ? creep
   Exit: (9) 1>0 ? creep
   Call: (9) _G2685 is 1+ -1 ? creep
   Exit: (9) 0 is 1+ -1 ? creep
   Call: (9) factorial(0, _G2686) ? creep
   Exit: (9) factorial(0, 1) ? creep
   Call: (9) _G2688 is 1*1 ? creep
   Exit: (9) 1 is 1*1 ? creep
   Exit: (8) factorial(1, 1) ? creep
   Call: (8) _G2691 is 2*1 ? creep
   Exit: (8) 2 is 2*1 ? creep
   Exit: (7) factorial(2, 2) ? creep
   Call: (7) 6 is 3*2 ? creep
   Exit: (7) 6 is 3*2 ? creep
   Exit: (6) factorial(3, 6) ? creep
true
```



# Simple Arithmetic

- Prolog supports integer variables and integer arithmetic
  - Eg. sum of 7 and the variable x:  `+(7,X)`
- **`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:

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

# 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).`

# 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)*

# Member Example

```
member(X,[X|List]).
member(X,[Y|List]) :- member(X,List).
```

or

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

(Not having to bind values to anonymous variables saves a little run-space and run-time.)

?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
No

?- member([3,Y], [[1,a],[2,m],[3,z],[4,v],[3,p]]).
Y = z ;
Y = p ;
No

?- member(X,[23,45,67,12,222,19,9,6]), Y is X*X, Y < 100.
X = 9   Y = 81 ;
X = 6   Y = 36 ;
No

# Append Example

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

?- append([1,2,3],[4,5],[1,2,3,4,5]).
Yes

?- append([1,2,3],[4,5],A).
A = [1,2,3,4,5]

?- append([1,2,3],W,[1,2,3,4,5]).
W = [4,5]

# Append Example

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

?- append([1,2,3],[4,5],A).
A = [1,2,3,4,5]

append([1,2,3],[4,5],_G1)

append([2,3],[4,5],_G2)
_G1=[1| _G2]

append([3],[4,5],_G3)
_G2=[2| _G3]

append([],[4,5],_G4)
_G3=[3| _G4]

_G4=[4,5]
append([],[4,5],[4,5]) T

_G3=[3,4,5]
append([3],[4,5],[3,4,5]) T

_G2=[2,3,4,5]
append([2,3],[4,5],[2,3,4,5]) T

_G1=[1,2,3,4,5]
append([1,2,3],[4,5],[1,2,3,4,5]) T

# Reverse Example

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

or

**Reverse using an accumulator**

```
reverse([H|T],A,R) :-reverse(T,[H|A],R).
reverse([],A,A).
```

| | |
|---|---|
| List: [a,b,c,d] | Accumulator: [] |
| List: [b,c,d] | Accumulator: [a] |
| List: [c,d] | Accumulator: [b,a] |
| List: [d] | Accumulator: [c,b,a] |
| List: [] | Accumulator: [d,c,b,a] |

# Additional Prolog Examples

Defining Max:

```
max(X,Y,M) :- X > Y, M is X.
max(X,Y,M) :- Y >= X, M is Y.
```

Defining GCD:

```
gcd(X,Y,D) :- X=Y, D is X.
gcd(X,Y,D) :- X<Y, Y1 is Y - X, gcd(X, Y1, D).
gcd(X,Y,D) :- X>Y, gcd(Y, X, D).
```

Two List examples

Defining Length:

```
length([ ], 0).                    // empty list has a length of 0
length([ _ | Tail, N) :- length(Tail, N1), N is 1 + N1.  // a list that has an
// item _ and a Tail is length N if the length of Tail is N1 where N = 1 + N1
```

Sum of the items in a list:

```
sum([ ], 0).          // sum of an empty list is 0
sum([X | Tail], S) :- sum(Tail, S1), S is X + S1.
```

# 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

# Applications of Logic Programming

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