

# **COP4020**

# **Programming**

# **Languages**

## **Prolog**

*Chris Lacher*

*Based on Robert van Engelen*



# Overview

- Logic programming principles
- Prolog

# Logic Programming

- Logic programming is a form of declarative programming
- A program is a *collection of axioms*
- Each axiom is a *Horn clause* of the form:

$$H :- B_1, B_2, \dots, B_n.$$

where  $H$  is the head term and  $B_i$  are the body terms

- Meaning:  $H$  is true if all  $B_i$  are true
- A user states a *goal* (a theorem) to be proven
- The logic programming system uses *inference steps* to prove the goal (theorem) is true, using a logical resolution strategy

# Resolution Strategies

- To *deduce* a goal (theorem), the programming system searches axioms and combines sub-goals using a resolution strategy
- For example, given the axioms:  
     $C :- A, B.$   
     $D :- C.$
- *Forward chaining* deduces first that  $C$  is true:  
     $C :- A, B$   
    and then that  $D$  is true:  
     $D :- C$
- *Backward chaining* finds that  $D$  can be proven if sub-goal  $C$  is true:  
     $D :- C$   
    the system then deduces that the sub-goal is  $C$  is true:  
     $C :- A, B$   
    since the system could prove  $C$  it has proven  $D$

# Prolog

- Prolog uses backward chaining, which is more efficient than forward chaining for larger collections of axioms
- Prolog is interactive (mixed compiled/interpreted)
- Example applications:
  - Expert systems
  - Artificial intelligence
  - Natural language understanding
  - Logical puzzles and games
- Popular system: SWI-Prolog
  - Login `linprog.cs.fsu.edu`
  - `p1` (or `swip1`) to start SWI-Prolog
  - `halt.` to halt Prolog (period is the Prolog command terminator)

# Definitions: Prolog Clauses

- A program consists of a collection of *Horn clauses*
- Each clause consists of a *head predicate* and *body predicates*:

$$H :- B_1, B_2, \dots, B_n.$$

- A clause is either a *rule*, e.g.

**snowy(X) :- rainy(X), cold(X) .**

meaning: "If X is rainy and X is cold then this implies that X is snowy"

- Or a clause is a *fact*, e.g.

**rainy(rochester) .**

meaning "Rochester is rainy."

- This fact is identical to the rule with **true** as the body predicate:

**rainy(rochester) :- true .**

- A predicate is a term (an atom or a structure), e.g.

- **rainy(rochester)**

- **member(X,Y)**

- **true**

# Definitions: Queries and Goals

- *Queries* are used to "execute" goals
- A query is interactively entered by a user after a program is loaded
  - A query has the form
$$?- G_1, G_2, \dots, G_n.$$
where  $G_i$  are goals (predicates)
- A goal is a predicate to be proven true by the programming system
  - Example program with two facts:
    - `rainy(seattle) .`
    - `rainy(rochester) .`
  - Query with one goal to find which city  $C$  is rainy (if any):
$$?- \text{rainy}(C) .$$
  - Response by the interpreter:
$$C = \text{seattle}$$
  - Type a semicolon ; to get next solution:
$$C = \text{rochester}$$
  - Typing another semicolon does not return another solution

# Example

- Consider a program with three facts and one rule:

- `rainy(seattle) .`
- `rainy(rochester) .`
- `cold(rochester) .`
- `snowy(X) :- rainy(X), cold(X) .`

- Query and response:

`?- snowy(rochester) .`  
`yes`

- Query and response:

`?- snowy(seattle) .`  
`no`

- Query and response:

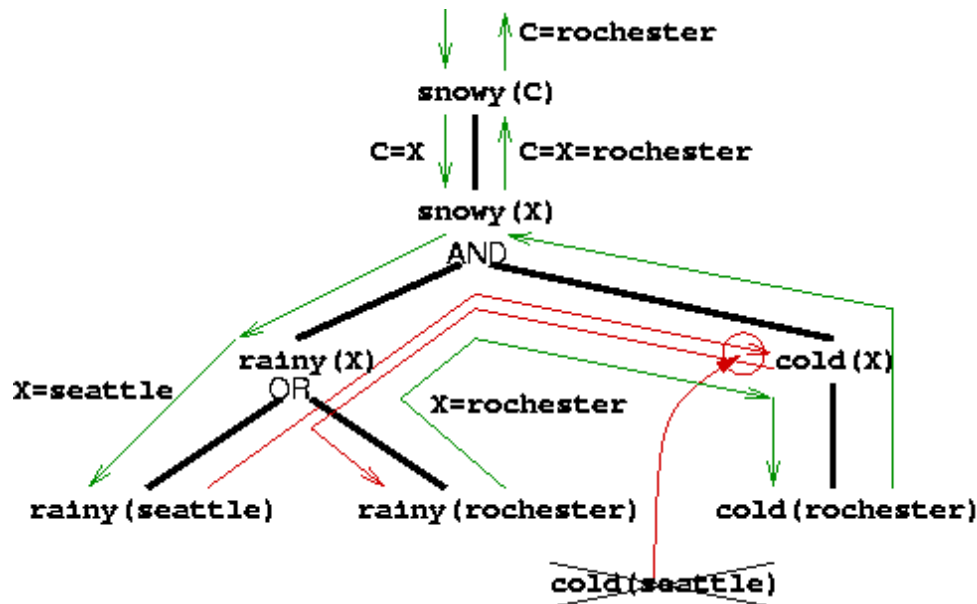
`?- snowy(paris) .`  
`no`

- Query and response:

`?- snowy(C) .`  
`C = rochester`

because `rainy(rochester)` and `cold(rochester)` are sub-goals  
that are both true facts

# Backward Chaining with Backtracking



An unsuccessful match forces backtracking in which alternative clauses are searched that match (sub-)goals

- Consider again:  
`?- snowy(C) .`  
`C = rochester`
- The system first tries `C=seattle`:  
`rainy(seattle)`  
`cold(seattle) fail`
- Then `C=rochester`:  
`rainy(rochester)`  
`cold(rochester)`
- When a goal fails, *backtracking* is used to search for solutions
- The system keeps this execution point in memory together with the *current variable bindings*
- Backtracking unwinds variable bindings to establish new bindings

# Example: Family Relationships

## ■ Facts:

- `male(albert) .`
- `male(edward) .`
- `female(alice) .`
- `female(victoria) .`
- `parents(edward, victoria, albert) .`
- `parents(alice, victoria, albert) .`

## ■ Rule:

`sister(X,Y) :- female(X), parents(X,M,F), parents(Y,M,F) .`

## ■ Query: `?- sister(alice, Z) .`

## ■ The system applies backward chaining to find the answer:

1. `sister(alice,Z)` matches 2nd rule: `X=alice, Y=Z`
2. New goals: `female(alice), parents(alice,M,F), parents(Z,M,F)`
3. `female(alice)` matches 3rd fact
4. `parents(alice,M,F)` matches 2nd rule: `M=victoria, F=albert`
5. `parents(Z,victoria,albert)` matches 1st rule: `Z=edward`

# Example: Murder Mystery

```
% the murderer had brown hair:
    murderer(X) :- hair(X, brown).
% mr_holman had a ring:
    attire(mr_holman, ring).
% mr_pope had a watch:
    attire(mr_pope, watch).
% If sir_raymond had tattered cuffs then mr_woodley had the pincenez:
    attire(mr_woodley, pincenez) :-
        attire(sir_raymond, tattered_cuffs).
% and vice versa:
    attire(sir_raymond,pincenez) :-
        attire(mr_woodley, tattered_cuffs).
% A person has tattered cuffs if he is in room 16:
    attire(X, tattered_cuffs) :- room(X, 16).
% A person has black hair if he is in room 14, etc:
    hair(X, black) :- room(X, 14).
    hair(X, grey) :- room(X, 12).
    hair(X, brown) :- attire(X, pincenez).
    hair(X, red) :- attire(X, tattered_cuffs).
% mr_holman was in room 12, etc:
    room(mr_holman, 12).
    room(sir_raymond, 10).
    room(mr_woodley, 16).
    room(X, 14) :- attire(X, watch).
```

# Example (cont'd)

- Question: who is the murderer?

`?- murderer(X) .`

- Execution trace (indentation shows nesting depth):

```
murderer(X)
  hair(X, brown)
    attire(X, pincenez)
      X = mr_woodley
      attire(sir_raymond, tattered_cuffs)
        room(sir_raymond, 16)
          FAIL (no facts or rules)
        FAIL (no alternative rules)
      REDO (found one alternative rule)
      attire(X, pincenez)
        X = sir_raymond
        attire(mr_woodley, tattered_cuffs)
          room(mr_woodley, 16)
            SUCCESS
          SUCCESS: X = sir_raymond
        SUCCESS: X = sir_raymond
      SUCCESS: X = sir_raymond
    SUCCESS: X = sir_raymond
```

# Unification and Variable Instantiation

- In the previous examples we saw the use of variables, e.g. C and X
- A variable is *instantiated* to a term as a result of *unification*, which takes place when goals are matched to head predicates
  - Goal in query: `rainy(C)`
  - Fact: `rainy(seattle)`
  - Unification is the result of the goal-fact match: `C=seattle`
- Unification is recursive:
  - An uninstantiated variable unifies with anything, even with other variables which makes them identical (aliases)
  - An atom unifies with an identical atom
  - A constant unifies with an identical constant
  - A structure unifies with another structure if the functor and number of arguments are the same and the arguments unify recursively
- Once a variable is instantiated to a non-variable term, it cannot be changed: “proofs cannot be tampered with”

# Examples of Unification

- The built-in predicate  $=(A,B)$  succeeds if and only if  $A$  and  $B$  can be unified, where the goal  $=(A,B)$  may be written as  $A = B$ 
  - `?- a = a.`  
`yes`
  - `?- a = 5.`  
`No`
  - `?- 5 = 5.0.`  
`No`
  - `?- a = X.`  
`X = a`
  - `?- foo(a,b) = foo(a,b) .`  
`Yes`
  - `?- foo(a,b) = foo(X,b) .`  
`X = a`
  - `?- foo(X,b) = Y.`  
`Y = foo(X,b)`
  - `?- foo(Z,Z) = foo(a,b) .`  
`no`

# Definitions: Prolog Terms

- *Terms* are symbolic expressions that are Prolog's building blocks
- A Prolog program consists of Horn clauses (axioms) that are terms
- Data structures processed by a Prolog program are terms
- A term is either
  - a *variable*: a name beginning with an upper case letter
  - a *constant*: a number or string
  - an *atom*: a symbol or a name beginning with a lower case letter
  - a *structure* of the form:  
$$\text{functor}(\text{arg}_1, \text{arg}_2, \dots, \text{arg}_n)$$
where *functor* is an atom and  $\text{arg}_i$  are terms
- Examples:
  - **x**, **Y**, **ABC**, and **Alice** are variables
  - 7, 3.14, and "hello" are constants
  - **foo**, **barFly**, and **+** are atoms
  - **bin\_tree(foo, bin\_tree(bar, glarch))**  
and **+(3,4)** are structures

# Term Manipulation

- Terms can be analyzed and constructed
  - Built-in predicates **functor** and **arg**, for example:
    - `?- functor(foo(a,b,c), foo, 3).`  
`yes`
    - `?- functor(bar(a,b,c), F, N).`  
`F = bar`  
`N = 3`
    - `?- functor(T, bee, 2).`  
`T = bee(_G1,_G2)`
    - `?- functor(T, bee, 2), arg(1, T, a), arg(2, T, b).`  
`T = bee(a,b)`
  - The “univ” operator `=..`
    - `?- foo(a,b,c) =.. L`  
`L = [foo,a,b,c]`
    - `?- T =.. [bee,a,b]`  
`T = bee(a,b)`

# Prolog Lists

- A list is of the form:

$$[elt_1, elt_2, \dots, elt_n]$$

where  $elt_i$  are terms

- The special list form

$$[elt_1, elt_2, \dots, elt_n \mid tail]$$

denotes a list whose tail list is *tail*

- Examples

- $?- [a, b, c] = [a \mid T].$   
 $T = [b, c]$

- $?- [a, b, c] = [a, b \mid T].$   
 $T = [c]$

- $?- [a, b, c] = [a, b, c \mid T].$   
 $T = []$

# List Operations:

## List Membership

- List membership definitions:

`member(X, [X|T]) .`

`member(X, [H|T]) :- member(X, T) .`

- ?- `member(b, [a,b,c]) .`

- Execution:

`member(b, [a,b,c])` does not match `member(X, [X|T])`

- `member(b, [a,b,c])` matches predicate `member(X1, [H1|T1])` with `X1=b`, `H1=a`, and `T1=[b,c]`

- Sub-goal to prove: `member(b, [b,c])`

- `member(b, [b,c])` matches predicate `member(X2, [X2|T2])` with `X2=b` and `T2=[c]`

- The sub-goal is proven, so `member(b, [a,b,c])` is proven (deduced)

- Note: variables can be "local" to a clause (like the formal arguments of a function)

- Local variables such as `x1` and `x2` are used to indicate a match of a (sub)-goal and a head predicate of a clause

# Predicates and Relations

- Predicates are *not* functions with distinct inputs and outputs
- Predicates are more general and define *relationships* between objects (terms)
  - `member(b, [a,b,c])` relates term `b` to the list that contains `b`
  - `?- member(X, [a,b,c]).`  
`X = a ;`      % *type ';' to try to find more solutions*  
`X = b ;`      % *... try to find more solutions*  
`X = c ;`      % *... try to find more solutions*  
`no`
  - `?- member(b, [a,Y,c]).`  
`Y = b`
  - `?- member(b, L).`  
`L = [b|_G255]`  
where `L` is a list with `b` as head and `_G255` as tail, where `_G255` is a new variable

# List Operations: List Append

- List append predicate definitions:

```
append([], A, A).
```

```
append([H|T], A, [H|L]) :- append(T, A, L).
```

- ?- append([a,b,c], [d,e], X).

X = [a,b,c,d,e]

- ?- append(Y, [d,e], [a,b,c,d,e]).

Y = [a,b,c]

- ?- append([a,b,c], Z, [a,b,c,d,e]).

Z = [d,e]

- ?- append([a,b], [], [a,b,c]).

No

- ?- append([a,b], [X|Y], [a,b,c]).

X = c

Y = []

# Example: Bubble Sort

```
    bubble(List, Sorted) :-
        append(InitList, [B,A|Tail], List),
        A < B,
        append(InitList, [A,B|Tail], NewList),
        bubble(NewList, Sorted).
    bubble(List, List).
?- bubble([2,3,1], L).
    append([], [2,3,1], [2,3,1]),
    3 < 2,                                % fails: backtrack
    append([2], [3,1], [2,3,1]),
    1 < 3,
    append([2], [1,3], NewList1),         % this makes: NewList1=[2,1,3]
    bubble([2,1,3], L).
        append([], [2,1,3], [2,1,3]),
        1 < 2,
        append([], [1,2,3], NewList2), % this makes: NewList2=[1,2,3]
        bubble([1,2,3], L).
            append([], [1,2,3], [1,2,3]),
            2 < 1,                        % fails: backtrack
            append([1], [2,3], [1,2,3]),
            3 < 2,                        % fails: backtrack
            append([1,2], [3], [1,2,3]), % does not unify: backtrack
            bubble([1,2,3], L).          % try second bubble-clause which makes L=[1,2,3]
        bubble([2,1,3], [1,2,3]).
    bubble([2,3,1], [1,2,3]).
```

# Imperative Features

- Prolog offers built-in constructs to support a form of control-flow
  - `\+ G` negates a (sub-)goal `G`
  - `!` (cut) terminates backtracking for a predicate
  - `fail` always fails to trigger backtracking
- Examples
  - `?- \+ member(b, [a,b,c]).`  
`no`
  - `?- \+ member(b, []).`  
`yes`
  - Define:  
`if(Cond, Then, Else) :- Cond, !, Then.`  
`if(Cond, Then, Else) :- Else.`
  - `?- if(true, X=a, X=b).`  
`X = a ; % type ';' to try to find more solutions`  
`no`
  - `?- if(fail, X=a, X=b).`  
`X = b ; % type ';' to try to find more solutions`  
`no`

# Example: Tic-Tac-Toe

1	2	3
4	5	6
7	8	9

- Rules to find line of three (permuted) cells:

- `line(A,B,C) :- ordered_line(A,B,C) .`
- `line(A,B,C) :- ordered_line(A,C,B) .`
- `line(A,B,C) :- ordered_line(B,A,C) .`
- `line(A,B,C) :- ordered_line(B,C,A) .`
- `line(A,B,C) :- ordered_line(C,A,B) .`
- `line(A,B,C) :- ordered_line(C,B,A) .`

# Example: Tic-Tac-Toe

1	2	3
4	5	6
7	8	9

## ■ Facts:

- `ordered_line(1,5,9).`
- `ordered_line(3,5,7).`
- `ordered_line(1,2,3).`
- `ordered_line(4,5,6).`
- `ordered_line(7,8,9).`
- `ordered_line(1,4,7).`
- `ordered_line(2,5,8).`
- `ordered_line(3,6,9).`

# Example: Tic-Tac-Toe

- How to make a good move to a cell:
  - `move(A) :- good(A), empty(A).`
- Which cell is empty?
  - `empty(A) :- \+ full(A).`
- Which cell is full?
  - `full(A) :- x(A).`
  - `full(A) :- o(A).`

## Example: Tic-Tac-Toe

- Which cell is best to move to? (check this in this order)
  - ☐ `good(A) :- win(A).`                   % a cell where we win
  - ☐ `good(A) :- block_win(A).`   % a cell where we block the opponent from a win
  - ☐ `good(A) :- split(A).`           % a cell where we can make a split to win
  - ☐ `good(A) :- block_split(A).` % a cell where we block the opponent from a split
  - ☐ `good(A) :- build(A).`           % choose a cell to get a line
  - ☐ `good(5).`                       % choose a cell in a good location
  - ☐ `good(1).`
  - ☐ `good(3).`
  - ☐ `good(7).`
  - ☐ `good(9).`
  - ☐ `good(2).`
  - ☐ `good(4).`
  - ☐ `good(6).`
  - ☐ `good(8).`

# Example: Tic-Tac-Toe

O		
	X	O
	X	X

*split*

- How to find a winning cell:
  - `win(A) :- x(B), x(C), line(A,B,C) .`
- Choose a cell to block the opponent from choosing a winning cell:
  - `block_win(A) :- o(B), o(C), line(A,B,C) .`
- Choose a cell to split for a win later:
  - `split(A) :- x(B), x(C), \+ (B = C),  
line(A,B,D), line(A,C,E), empty(D), empty(E) .`
- Choose a cell to block the opponent from making a split:
  - `block_split(A) :- o(B), o(C), \+ (B = C),  
line(A,B,D), line(A,C,E), empty(D), empty(E) .`
- Choose a cell to get a line:
  - `build(A) :- x(B), line(A,B,C), empty(C) .`

# Example: Tic-Tac-Toe

O		
X	O	
X		

- Board positions are stored as facts:
  - `x(7) .`
  - `o(5) .`
  - `x(4) .`
  - `o(1) .`
- Move query:
  - `?- move(A) .`  
`A = 9`

# Prolog Arithmetic

- Arithmetic is needed for computations in Prolog
- Arithmetic is not relational
- The `is` predicate evaluates an arithmetic expression and instantiates a variable with the result
- For example
  - `X is 2*sin(1)+1`  
instantiates `X` with the results of `2*sin(1)+1`

# Examples with Arithmetic

- A predicate to compute the length of a list:
  - `length([], 0).`
  - `length([H|T], N) :- length(T, K), N is K + 1.`
- where the first argument of `length` is a list and the second is the computed length
- Example query:
  - `?- length([1,2,3], X).`  
`X = 3`
- Defining a predicate to compute GCD:
  - `gcd(A, A, A).`
  - `gcd(A, B, G) :- A > B, N is A-B, gcd(N, B, G).`
  - `gcd(A, B, G) :- A < B, N is B-A, gcd(A, N, G).`

# Database Manipulation

- Prolog programs (facts+rules) are stored in a database
- A Prolog program can manipulate the database
  - Adding a clause with **assert**, for example:  
**assert(rainy(syracuse))**
  - Retracting a clause with **retract**, for example:  
**retract(rainy(rochester))**
  - Checking if a clause is present with **clause(Head, Body)** for example:  
**clause(rainy(rochester), true)**
- Prolog is *fully reflexive*
  - A program can reason about all its aspects (code+data)
  - A meta-level (or metacircular) interpreter is a Prolog program that executes (another) Prolog program, e.g. a tracer can be written in Prolog

# A Meta-level Interpreter

- ```
clause_tree(G) :- write_ln(G), fail. % just show goal
clause_tree(true) :- !.
clause_tree((G,R)) :-
    !,
    clause_tree(G),
    clause_tree(R).
clause_tree((G;R)) :-
    !,
    ( clause_tree(G)
    ; clause_tree(R)
    ).
clause_tree(G) :-
    ( predicate_property(G,built_in)
    ; predicate_property(G,compiled)
    ), !,
    call(G). % let Prolog do it
clause_tree(G) :- clause(G,Body), clause_tree(Body).
```
- ```
?- clause_tree((X is 3, X<1; X=4)).
_G324 is 3, _G324<1 ; _G324=4
_G324 is 3, _G324<1
_G324 is 3
3<1
_G324=4
X = 4
```