COP4020
Programming Languages

Prolog

Prof. 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:

  \[ H :\! -\! B_1, B_2, \ldots, B_n. \]

  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:
  
  \[
  \begin{align*}
  C & : - A, B. & A & : - \text{true}. \\
  D & : - C. & B & : - \text{true}. 
  \end{align*}
  \]

- *Forward chaining* starts with the facts A and B then deduces 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`
  - `pl` (or `swipl`) 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*:
  \[ H : - B_1, B_2, \ldots, B_n. \]

- A clause is either a *rule*, e.g.
  \[
  \text{snowy}(X) : - \text{rainy}(X), \text{cold}(X).
  \]
  meaning: "If \( X \) is rainy and \( X \) is cold then this implies that \( X \) is snowy"
- Or a clause is a *fact*, e.g.
  \[
  \text{rainy}(\text{rochester}).
  \]
  meaning "Rochester is rainy."
- This fact is identical to the rule with *true* as the body predicate:
  \[
  \text{rainy}(\text{rochester}) : - \text{true}.
  \]

- A predicate may or may not have arguments, e.g.
  \[
  \text{rainy}(\text{rochester}), \text{member}(X,Y), \text{true}
  \]
- A variable name must start with an upper case letter, e.g. \( X \)
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
    
    \[- G_1, G_2, ..., G_n.\]
  
  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):
    
    \[- rainy(C).\]
  
  - Response by the interpreter:
    
    \[C = seattle\]
  
  - Type a semicolon ; to get next solution:
    
    \[C = rochester\]
  
  - 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

Consider again:

\[-\text{snowy}(C)\]
\[C = \text{rochester}\]

The system first tries \(C=\text{seattle}\):

\[-\text{rainy}(\text{seattle})\]
\[-\text{cold}(\text{seattle})\]
fail

Then \(C=\text{rochester}\):

\[-\text{rainy}(\text{rochester})\]
\[-\text{cold}(\text{rochester})\]

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.

An unsuccessful match forces backtracking in which alternative clauses are searched that match (sub-)goals.
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 the rule and unifies X=alice, Y=Z
2. New goals: female(alice), parents(alice, M, F), parents(Z, M, F)
3. female(alice) matches the 3rd fact
4. parents(alice, M, F) matches the 6th fact, so M=victoria, F=albert
5. parents(Z, victoria, albert) matches the 5th fact: 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: \texttt{rainy(C)}
- Fact: \texttt{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
  
  - \(?- a = X.\)
    \(X = a\)
  
  - \(?- \text{foo}(a,b) = \text{foo}(a,b).\)
    Yes
  
  - \(?- \text{foo}(a,b) = \text{foo}(X,b).\)
    \(X = a\)
  
  - \(?- \text{foo}(X,b) = Y.\)
    \(Y = \text{foo}(X,b)\)
  
  - \(?- \text{foo}(Z,Z) = \text{foo}(a,b).\)
    no
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:
    \[ \text{functor}(\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n) \]
    where \text{functor} is an atom and \text{arg}_i are sub-terms
  - a *list* (also a structure with a functor) of the form \[\text{term}_1, \text{term}_2, \ldots, \text{term}_n\]
- 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 constructed and unified with other terms
  - 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:

  \[ [elt_1, elt_2, ..., elt_n] \]

  where \( elt_i \) are terms

- The special list form

  \[ [elt_1, elt_2, ..., elt_n \mid tail] \]

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:
  
  \[
  \text{member}(X, [X|T]). \\
  \text{member}(X, [H|T]) \leftarrow \text{member}(X, T).
  \]

- \(?- \text{member}(b, [a,b,c]).\)

  - Execution:
    
    \[
    \text{member}(b, [a,b,c]) \text{ does not match } \text{member}(X, [X|T]).
    \]

    \[
    \text{member}(b, [a,b,c]) \text{ matches predicate } \text{member}(X_1, [H_1|T_1])
    \]
    
    \[
    \text{with } X_1=b, \ H_1=a, \text{ and } T_1=[b,c].
    \]

  - Sub-goal to prove: \(\text{member}(b, [b,c]).\)

  - \(\text{member}(b, [b,c]) \text{ matches predicate } \text{member}(X_2, [X_2|T_2]).\)
    
    \[
    \text{with } X_2=b \text{ and } T_2=[c].
    \]

  - The sub-goal is proven, so \(\text{member}(b, [a,b,c])\) is proven (deduced)

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

- Local variables such as \(X_1\) and \(X_2\) 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

\[
\text{bubble}(\text{List}, \text{Sorted}) :- \\
\text{append}(\text{InitList}, [B,A|\text{Tail}], \text{List}), \\
A < B, \\
\text{append}(\text{InitList}, [A,B|\text{Tail}], \text{NewList}), \\
\text{bubble}(\text{NewList}, \text{Sorted}). \\
\text{bubble}(\text{List}, \text{List}).
\]

?- \text{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], \text{NewList1}), % this makes: \text{NewList1}=[2,1,3]
\text{bubble}([2,1,3], L).
append([], [2,1,3], [2,1,3]),
1 < 2,
append([], [1,2,3], \text{NewList2}), % this makes: \text{NewList2}=[1,2,3]
\text{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
\text{bubble}([1,2,3], L). % try second bubble-clause which makes \text{L}=[1,2,3]
\text{bubble}([2,1,3], [1,2,3]).
bubble([2,3,1], [1,2,3]).
Control 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 (so to trigger backtracking)

- Examples
  - ?- \+ member(b, [a,b,c]).
    no
  - ?- \+ member(b, []).
    yes
  - We can (re)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
Control Features

- Prolog offers built-in constructs to support a form of control-flow
  - $G_1 ; G_2$ forms an “or”: try $G_1$ then $G_2$ if $G_1$ failed
  - $G_1 \rightarrow G_2 ; G_3$ forms an if-then-else: if $G_1$ then $G_2$ else $G_3$
  - true is a true goal and acts like a no-op
  - repeat always succeeds (for looping)

- Examples:
  - We can use an “or” instead of two clauses:
    rainy(C) :- ( C = rochester ; C = seattle ).
  - Using an if-then-else:
    guess(X) :- ( X = heads
    -> format("~s~n", ["Guessed it!"])
    ; format("~s~n", ["Sorry, no luck"]) )
  - Repeat drawing a random X until X=1 (and then cuts backtracking):
    do_until_one :- repeat, X is random(10), X = 1, !.
Meta-Logical

- Meta-logical predicates perform operations that require reasoning about terms
  - \( X == Y \) true if \( X \) is identical to \( Y \) (vice versa \( X \neq Y \))
  - \( \text{var}(X) \) true if \( X \) is uninstantiated (vice versa \( \text{nonvar}(Y) \))
  - \( \text{ground}(X) \) true if \( X \) is a ground term, has no variables
  - \( \text{atom}(X) \) true if \( X \) is an atom
  - \( \text{functor}(X, F, N) \) true if \( X \) is a functor \( F \) with \( N \) arguments
  - \( X =.. [F,A1,A2,\ldots,A_n] \) true if \( X \) a functor \( F \) with arguments

- Impact:
  - Pure logic has no ordering constraints, e.g. \( X \) “or” \( Y = Y \) “or” \( X \)
  - Meta-logical predicates are order sensitive and influence control
  - Theoretically, cannot be expressed using predicate definitions with a finite number of clauses
Example: Tic-Tac-Toe

- Cells on the grid are numbered

- 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

- 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

- How to make a good move to a cell:
  - \texttt{move(A)} :- \texttt{good(A)}, \texttt{empty(A)}.

- Which cell is empty?
  - \texttt{empty(A)} :- \texttt{\neg full(A)}.

- Cell has an X or O placed in it?
  - \texttt{full(A)} :- \texttt{x(A)}.
  - \texttt{full(A)} :- \texttt{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

- 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

Board positions are stored as facts:
- \( x(7) \).
- \( o(5) \).
- \( x(4) \).
- \( o(1) \).

Move query:
- \(?- \text{move}(A). \quad A = 9\)
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, B, G) :- A > B, N is A-B, gcd(N, B, 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 if 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)) :-` ![image](image)
- `clause_tree((G;R)) :-` ![image](image)
- `clause_tree(G) :-` ![image](image)
  - `( predicate_property(G,built_in)` ![image](image)
  - `; predicate_property(G,compiled)` ![image](image)
  - `), !,` ![image](image)
  - `call(G). % let Prolog do it`
- `clause_tree(G) :- clause(G,Body), clause_tree(Body).`

?- `clause_tree((X is 3, X<1; X=4)).` ![image](image)

- `_G324 is 3, _G324<1 ; _G324=4` ![image](image)
- `_G324 is 3, _G324<1` ![image](image)
- `_G324 is 3` ![image](image)
- `3<1` ![image](image)
- `_G324=4` ![image](image)
- `X = 4` ![image](image)