

10. Logic Programming With Prolog

Overview

- Logic Programming
- Prolog

Note: Study Section 11.3 of the textbook, excluding 11.3.2.

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 of the program states a *goal* (a theorem) to be proven
 - The logic programming system attempts to find axioms using *inference steps* that imply the goal (theorem) is true

Resolution

- To deduce a goal (theorem), the logic programming system searches axioms and combines sub-goals
- For example, given the axioms:

$$C :- A, B.$$

$$D :- C.$$
- To deduce goal D given that A and B are true:
 - *Forward chaining* 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

- Uses backward chaining
 - More efficient than forward chaining for larger collections of axioms
- Interactive (hybrid compiled/interpreted)
- Applications: expert systems, artificial intelligence, natural language understanding, logical puzzles and games
- Popular system: SWI-Prolog
 - Login: **program.cs.fsu.edu**
 - Type: **pl** to start SWI-Prolog
 - Type: **halt.** to halt Prolog (note that a period is used as a command terminator)

Prolog Terms

- Terms are symbolic expressions that form the building blocks of Prolog
 - A Prolog program consists of *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:
 $functor(arg_1, arg_2, \dots, arg_n)$
where *functor* is an atom and arg_i are terms
- Examples:
 - **X, Y, ABC**, and **Alice** are variables
 - **7, 3.14**, and **"hello"** are constants
 - **foo, bAR**, and **+** are atoms
 - **bin_tree(foo, bin_tree(bar, glarch))** and **+(3,4)** are structures

Prolog Clauses

- A program consists of a database 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* (must be an atom or a structure)
 - **rainy(rochester)**
 - **member(X,Y)**
 - **true**

Queries and Goals

- *Queries* are used to "execute" *goals*
 - A *query* is interactively entered by a user after a program is loaded and stored in the database
 - A *query* has the form
 $?- G_1, G_2, \dots, G_n$
where G_i are *goals*
- 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
 - Type another semicolon ;:
no
(no more solutions)

Example

- 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

Example: Murder Mystery (cont'd)

- Question: who is the murderer?
- ?- murderer(X).
- Trace (indentation showing 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 Variables

- 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*
- *Unification* takes place when goals are matched to head predicates of rules and facts
 - Goal in query: **rainy(C)**
 - Fact in database: **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 corresponding arguments unify recursively
- Once a variable is instantiated to a non-variable term, it cannot be changed and cannot be instantiated with a term that has a different structure

Unification Examples

- The built-in predicate $=(A,B)$ succeeds if and only if A and B can be unified
- 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

Lists

- A *list* is of the form:
[elt₁,elt₂, ..., elt_n]
where elt_i are terms
- The special list form
[elt₁,elt₂, ..., elt_n | tail]
denotes a list whose tail list is *tail*
 - ?- [a,b,c] = [a|T].
 T = [b,c]
 - ?- [a,b,c] = [a,b|T].
 T = [c]
 - ?- [a,b,c] = [a,b,c|T].
 T = []

List Membership

- List membership is tested with the **member** predicate, defined by
member(X, [X|T]).
member(X, [_|T]) :- member(X, T).
- ?- **member(b, [a,b,c]).**
- Execution:
 - member(b, [a,b,c])** does not match predicate **member(X₁, [X₁|T₁])**
 - 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(X₁, T₁)** with **X₁ = b** and **T₁ = [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 are "local" to a clause (just 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 are 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]
therefore, **L** is a list with **b** as head and **_G255** as tail, where **_G255** is a new variable
- List appending predicate:
 - 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]

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]).
```

- Prolog offers a few built-in constructs to support a form of control-flow
 - \+ G** negates a (sub-)goal **G**
 - !** (cut) terminates backtracking *for a predicate and within the body of the clause of that predicate*
 - fail** always fails
- 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
 - ?- **if(true, a=b, X=b).**
no
 - The cut makes sure that the **Cond** is not executed again upon backtracking and that the second **if**-clause is not executed when **Cond** is true when backtracking
 - Therefore, this example would not work without the cut when backtracking

Imperative Control Flow

Example: Tic-Tac-Toe

- Board layout:

1	2	3
4	5	6
7	8	9

- Facts:

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).
ordered_line(1,5,9).
ordered_line(3,5,7).

Note: You can download the program from here
(instructions are included in the source).

Example: Tic-Tac-Toe (cont'd)

- 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).

- 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).

- 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 (cont'd)

- 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).

O		
	X	O
	X	X

- 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 (cont'd)

- Board positions:

O		
X	O	
X		

- Are stored as facts in the database:

x(7).

o(5).

x(4).

o(1).

- Move query:

?- move(A).

A = 9

Arithmetic

- Arithmetic is useful for many computations in Prolog
- 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**

Arithmetic Examples

- A predicate to compute the length of a list:
length([], 0).
length([_|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
- 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).