Prolog

Aditya Thakur

ECS 140A Programming Languages – Winter 2019

Outline

- Defining facts, and rules
- Solving goals
- Lists in Prolog
- Unification
- Arithmetic, logical operators, and comparison operators
- Control in Prolog (goal and rule ordering)
- Search and backtracking
  - Deriving search trees (examples: mylength1, mylength2, prefix/suffix, etc.)
- Cuts (green and red)

Introduction

- What is logic programming?
  - Use facts and rules to represent information
  - Use logical deduction to answer queries
- It is based on first-order logic
  - Use logic to represent information
  - Use proof rules/techniques to derive more information
- Logic programming view:

  Algorithm = Logic + Control

  - Logic: What needs to be done (provided by the user/programmer)
  - Control: How it needs to be done (derived by system based on the logic)
- Programming in terms of <u>deduction</u> rather than <u>evaluation</u>
  - Functional programming: expression evaluates to a value
  - Imperative programming: program runs and updates some store

Key to Logic Programming

- You do not need to specify exactly how to compute a result
- But simply need to describe the form of the results
- The language system will determine how to compute the result

Building Blocks

- Everything in a Prolog program is built from Prolog terms

1. Constants

   Numbers: `1, 2, 1.4`

   Atoms: `a, b, c, parent, append`
   (Start with lowercase letter)

2. Variables

   `X, Y, Z`
   (Any name beginning with an uppercase letter or an underscore)

3. Compound Terms

   `x(Y,Z), parent(adam,seth)`
   (An atom followed by a parenthesized, comma-separated list of terms)

Prolog Language System

- Collection of *facts* and *rules* of inference
- Fact: term followed by a period
- Example of facts:

```
parent(kim,holly).
parent(margaret,kim).
parent(margaret,kent).
parent(esther,margaret).
parent(herbert,margaret).
parent(herbert,jean).
```

In this example, assume facts are saved in a file family.pl

Runtime System

http://www.swi-prolog.org/

https://swish.swi-prolog.org Web-based tool for running Prolog

```
% swipl
| ?- consult(family).
```

More on Facts

- Facts express unconditional information:

```
parent(margaret,kent).        % margaret is kent's parent
parent(fred,john).
```

- More examples

```
|- ? parent(margaret,kent).
true
|- ? parent(margaret,X).
X = kim ? ;
X = kent ?
true
|- ? parent(X,jean).
X = herbert
true
|- ? parent(margaret,X), parent(X,holly).   % means "anded" together
X = kim ? ;
false
```

; means to show more answers

Rules

- A rule is defined in the following way

```
<term> :- <term1>, <term2>, ..., <termn>
HEAD                 CONDITIONS
```

  - Means if <term1> AND <term2> AND ... AND <termn> then <term>

- A fact is just a special case of a rule: with no conditions

- Example:

```
grandparent(GP,GC) :- parent(GP,P), parent(P,GC).

|- ? grandparent(X,Y).
X = margaret,
Y = holly ? ;
…
```

Goals (Queries)

- Questions that we can ask a system
- Goal of the form: `g1,g2,...,gk`
  - The gi's are called subgoals

```
|- ? parent(X,kim), parent(X,kent).
|- ? parent(_,X).        % anybody with a parent
```

- Notice the closed world assumption
  - Everybody has a parent, but we only get whatever is represented in the database
  - Plus what can be deduced from the facts and rules
  - Nothing more

Lists in Prolog

- Lists:

```
[]              empty list
[a, b, c]       list with a, b, & c
```

- Head/tail notation

```
[a, b, c] = [a | [b | [c | []]]]
```

consider the same list in Lisp

```
(1 2 3) = (1 . (2 . (3 . nil)))
```

Append Predicate (1)

- Let's look at the append relationship on lists

`[]`	empty list
`[1,2,3]`	list with three elements

- `append(X,Y,Z)` is true if `append(X,Y) = Z`

```
append([], [1], [1]).
append([1], [], [1]).
append([1,2], [3], [1,2,3]).
...
...
```

- This is an infinite set

Append Predicate (2)

- We can ask some questions about append

```
append([],[],[]).                    true
append([],[1],[1]).                  true
append([1,2],[3], [1,2,3]).          true
append([1],[],[]).                   false
```

- What if we want to ask

```
append(X,[1],[1]).
append([1,2],X,[1,2,3]).
append(X,Y,[1,2]).
append(X,Y,Z).
```

Defining our own Append

```
% swipl
| ?- [user].
myappend([],L,L).
myappend([X|L1],L2,[X|L3]) :- myappend(L1,L2,L3).

user compiled, 3 lines read - 476 bytes written, 25598 ms

true
```

myappend

```
| ?- myappend([1],[2,3],X).

X = [1,2,3]

true
| ?- myappend(X,[2,3],[1,2,3]).

X = [1] ? ;

false
| ?- myappend(X,[2,3],[1,3,2]).

false
| ?-
```

myappend

```
| ?- myappend(X,Y,[1,2,3]).

X = []
Y = [1,2,3] ? ;

X = [1]
Y = [2,3] ? ;

X = [1,2]
Y = [3] ? ;

X = [1,2,3]
Y = [] ? ;

false
| ?-
```

Unification (1)

- How does prolog do deductions?
  - Based on a process called unification
- Unification is a process to match two or more terms
  - Similar to pattern matching
  - But pattern-matching goes only one way: variable to a pattern
  - Unification can go both ways
- Example
  - foo(X,b) and foo(a,Y) matches if X = a, Y = b
  - f(a,b) is an instance of both
- Unification attempts to find if there is a common instance of two terms
- Definition: *Two terms t1 and t2 unify, if they have a common instance*

Unification (2)

- When does unification happen?
  - When two terms are checked for equality

```
|- ? foo(X,b) = foo(a,Y).
```

    - Example:

```
| ?- foo(X,b)=foo(a,Y).
X = a
Y = b
true
```

  - When a rule is applied

```
| ?- [user].
same(X,X).
true
| ?- same(foo(X,b), foo(a,Y)).
X = a
Y = b
true
```

Unification Checks with Lists

Consider the following unification checks

```
[H | T] = [a, b, c].
H = a
T = [b,c]
[a | T] = [H, b, c].
H = a
T = [b,c]
[a, b] = [a | X].
X = [b]
[a | b] = [X | Y].
X = a
Y = b
```

Control in Prolog (1)

- How is a prolog program evaluated?
  - Goal order: subgoals are processed <u>left-to-right</u>
  - Rule order: rules are applied <u>top-to-bottom</u>
- Answer to a query is affected by
  - Goal order in the query
  - Rule order in the database of facts and rules

Control in Prolog (2)

```
current_goal := query
  while current_goal is nonempty do
     assume current_goal = g1,...,gk
     choose the leftmost goal g1
     if there is a rule that applies to g1 then
        select first such rule H :- C1,...,Cn
        let s be the most general unifier of H and g1
        current_goal := s(C1), ..., s(Cn), s(g2), ..., s(gk)
     else
        backtrack
  end-while
success
```

Ignore "most general unifier"

Just think that it is a solution to a unification equation

Arithmetic in Prolog

- Prolog has the usual `+, -, *`, etc. operators for arithmetic
  - And each implementation may have many more
  - Use infix notation: `3 * 4`
- Arithmetic expressions not evaluated unless forced to

```
| ?- X=2+3.
X = 2+3
```

NOT `X = 5`

- Use "is" to force their evaluations

```
| ?- X is 2+3.
X = 5
```

Numeric Comparisons

- Six special operators for comparing numeric values

  `<, >, =<, >=, =:=`

- What is the difference between the following:

```
X = Y.
X is Y.
X =:= Y.
```

More Unification Examples

```
| ?- X = 2 + 3.
X = 2+3
| ?- 5 = 2 + 3.
false
| ?- 2 + 3 = 2 + X.
X = 3
| ?- 2 * X = Y * (3 + Y).
X = 3+2
Y = 2
| ?- X = X + 2.
X=X+2.
```

- How unification is done?
  - Unification algorithm (Robinson's)
  - Union-find
  - We will not cover here

Logical Operators (1)

- `true`
  - Goal always succeeds
- `fail`
  - Always fails
- `=` (equality)

  t1 = t2 succeeds if t1 and t2 <u>unifies</u>
  - Examples

    ```
    | ?- 5 = 5        succeeds
    | ?- 5 = 2+3      fails
    ```
- `\=` (inequality)
  - Negation of =

Logical Operators (2)

- `not` (negation)
  - `not(X)` succeeds if fails to satisfy X
  - order matters
    - `X=2, not(X=1)` succeeds **because**
      - **To satisfy `X=2`, we can unify `X` with `2`**
      - And then `2=1` fails, thus `not(X=1)` is satisfied
    - `not(X=1), X=2` fails because
      - `X=1` succeeds, thus `not(X=1)` fails
      - Then the whole goal fails
    - Different from the logical interpretation where both are equivalent
- `g1; g2`: disjunction of goals
  - Its meaning is: g1 OR g2
  - The goal succeeds if g1 succeeds or g2 succeeds

Other Predicates for Lists

- Provable if the list Y contains the element X

```
member(X,Y).
```

- Provable if X is a list of length Y

```
length(X,Y).
```

- Provable if Y is a list that contains the elements of list X in reverse order

```
reverse(X,Y).
```

- Provable when List1, with Elem removed, results in List2.

```
select(Elem,List1,List2).
```

Example: Length of a List

- length of a list (correct and buggy versions):

```
mylength1([],0).
mylength1([_|T],L) :- mylength1(T, K), L is K+1.

mylength2([],0).
mylength2([_|T],L) :- K is L-1, mylength2(T, K).
```

What happens when we pose the following queries?

```
mylength1([1,2,3],3).   ✓
mylength2([1,2,3],3).   ✓
```

Show the search tree

How about the following queries?

```
mylength1([1,2,3],X).   ✓
mylength2([1,2,3],X).   X
```

WHY?

Show the search tree

Trace for mylength1

```
| ?- mylength1([1,2,3],3).
     1    1  Call: mylength1([1,2,3],3) ?
     2    2  Call: mylength1([2,3],_348) ?
     3    3  Call: mylength1([3],_372) ?
     4    4  Call: mylength1([],_396) ?
     4    4  Exit: mylength1([],0) ?
     5    4  Call: _424 is 0+1 ?
     5    4  Exit: 1 is 0+1 ?
     3    3  Exit: mylength1([3],1) ?
     6    3  Call: _453 is 1+1 ?
     6    3  Exit: 2 is 1+1 ?
     2    2  Exit: mylength1([2,3],2) ?
     7    2  Call: 3 is 2+1 ?
     7    2  Exit: 3 is 2+1 ?
     1    1  Exit: mylength1([1,2,3],3) ?
(1 ms) true
```

Trace for mylength2

```
| ?- mylength2([1,2,3],3).
      1    1  Call: mylength2([1,2,3],3) ?
      2    2  Call: _351 is 3-1 ?
      2    2  Exit: 2 is 3-1 ?
      3    2  Call: mylength2([2,3],2) ?
      4    3  Call: _403 is 2-1 ?
      4    3  Exit: 1 is 2-1 ?
      5    3  Call: mylength2([3],1) ?
      6    4  Call: _455 is 1-1 ?
      6    4  Exit: 0 is 1-1 ?
      7    4  Call: mylength2([],0) ?
      7    4  Exit: mylength2([],0) ?
      5    3  Exit: mylength2([3],1) ?
      3    2  Exit: mylength2([2,3],2) ?
      1    1  Exit: mylength2([1,2,3],3) ?
(1 ms) true
```

Practice Exercise

- reversing a list:

```
myreverse([], []).
myreverse([H|T], L1) :- myreverse(T,L2),myappend(L2,[H],L1).
```

Another Example

- Example :

```
append([],Y,Y).
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
prefix(X,Z) :- append(X,Y,Z).
suffix(Y,Z) :- append(X,Y,Z).
sublist1(S,L) :- prefix(X,L), suffix(S,X).
sublist2(S,L) :- suffix(S,X), prefix(X,L).
```

- A graphical representation of the logic of sublist1 and sublist2

```
|<-- X  -->|
+-------+-+-----+
|       |S|     |
+-------+-+-----+
|<--    L    -->|
```

How Does Search Work - Example 1

```
| ?- suffix([a],L), prefix(L,[a,b,c]).
L = [a].
```

- Search tree with no backtracking (_i fresh variables)

`suffix([a],L), prefix(L,[a,b,c])`

pick this subgoal and match rule: `suffix(Y,Z) :- append(X,Y,Z).`

`unifier: Y = [a], Z = L, X = _1` (X can be anything, let's use _1)

`append(_1,[a],L), prefix(L,[a,b,c])`

pick this subgoal and match the fact: `append([], Y, Y).`

`unifier: Y = [a], L = [a]`

`prefix([a], [a,b,c])`

pick this subgoal and match the rule: `prefix(X, Z) :- append(X, Y, Z).`

`unifier: X = [a], Z = [a,b,c], Y = _2` (call Y _2)

How Does Search Work - Example 1

```
append([a], _2, [a,b,c])
```

pick this subgoal, match rule:

```
append([H|X], Y, [H|Z]) :- append(X,Y,Z).
unifier: H = a, X = [], Z = [b,c], Y = _2
```
(still anything)

```
append([], _2, [b,c])
```

pick this subgoal and match the fact: `append([], Y, Y).`

```
unifier: Y = [b,c]
```

```
yes
```

How Does Search Work - Example 2

```
| ?- suffix([b],L), prefix(L,[a,b,c]).
L = [a,b].
```

- Search tree with backtracking

```
suffix([b],L), prefix(L,[a,b,c]).
```

matches rule `suffix(Y, Z) :- append(X, Y, Z).`

`unifier: Y = [b], Z = L, X = _1`

```
append(_1, [b], L),        prefix(L, [a,b,c])
```

matches rule `append([], Y, Y)`

`unifier: Y = L = [b]`

```
prefix([b], [a,b,c])
```

matches `prefix(X, Z) :- append(X, Y, Z).`

`unifier: X = [b], Z = [a,b,c], Y = _2`

How Does Search Work - Example 2

```
append([b], _2, [a,b,c])
          matches no rule or fact, backtrack TO:
append(_1, [b], L),      prefix(L, [a,b,c])
          match append([H|X], Y, [H|Z]) :- append(X,Y,Z)
          unifier: _1 = [_3|_4], Y = [b], L =[_3|_5]

append(_4, [b], _5),        prefix([_3|_5], [a,b,c])
         match append([], Y, Y)
         unifier: _4 = [], Y = [b] = _5

prefix([_3, b], [a,b,c])
        match prefix(X, Z) :- append(X, Y, Z)
        unifier: X = [_3,b], Y = _6, Z = [a,b,c]
```

How Does Search Work - Example 2

```
append([_3,b], _6, [a,b,c])
   match append([H|X], Y, [H|Z]) :- append(X, Y, Z).
    unifier: H = a = _3, Z = [b,c], Y = _6, X = [b]

append([b], _6, [b,c])
   match append([H|X], Y, [H|Z]) :- append(X, Y, Z).
    unifier: H = b, Z = [c], Y = _6, X = []

append([], _6, [c])
      match append([], Y, Y).
      unifier: Y = _6 = [c]

yes
```

Order in the Goals and Rules (1)

- What's the difference between the the sublist predicates?

```
sublist2(S,L) :- suffix(S,X), prefix(X,L).
sublist1(S,L) :- prefix(X,L), suffix(S,X).
```

Hint: use same example where suffix is [b] and the list is [a,b,c].

```
suffix([b],L), prefix(L,[a,b,c]). ✗
prefix(L,[a,b,c],suffix([b],L). ✓
```

WHY?

Order in the Goals and Rules (2)

- What's the difference between the following definitions?

```
ancestor1(X,Y) :- parent(X,Y).
ancestor1(X,Y) :- parent(Z,Y),ancestor1(X,Z).
```

```
ancestor2(X,Y) :- parent(Z,Y),ancestor2(X,Z).
ancestor2(X,Y) :- parent(X,Y).
```

Minor difference: same solutions but in different order
Sample query:

```
-? ancestor1(X,holly).
```

Order in the Goals and Rules (3)

- What's the difference between the following definitions?

```
ancestor3(X,Y) :- ancestor3(X,Z),parent(Z,Y).
ancestor3(X,Y) :- parent(X,Y).
```

```
ancestor4(X,Y) :- parent(X,Y).
ancestor4(X,Y) :- ancestor4(X,Z),parent(Z,Y).
```

Very different to ancestor1 and ancestor2
Non-terminating in all or some cases (left recursive!)
Sample query:

```
-? ancestor3(kim,holly).
```

Cuts

- The cut, in Prolog, is a goal, written as !, which always succeeds, but cannot be backtracked past.
  - Cuts allow you to prune out or "cut out" an unexplored part of a prolog search tree
- Cuts are written as

```
C :- A_1, A_2, ..., A_m, !, B_1, B_2, ..., B_n.
```

  - Semantics: tells control to backtrack past A_1, ..., A_m without considering any of the rules for them

Cuts

- Rules for max:

```
max(A,B,B) :- A =< B.
max(A,B,A) :- A > B.
```

- Optimize the code:

```
max(A,B,B) :- A =< B,!. % no need to check other rule
max(A,B,A) :- A > B.    % if backtracking

-? max(1,3,M).
M = 3
```

The meaning of the program remains unchanged, but the program is more efficient

What happens if we remove the cut?

Cuts

- Rules for max:

```
max(A,B,B) :- A =< B.
max(A,B,A) :- A > B.
```

- Can simplify by removing redundant comparison:

```
max(A,B,B) :- A =< B.
max(A,B,A).

?- max(1,3,M).
M = 3;
M = 1;
```

Cuts

- Attempt 1:

```
max(A,B,B) :- A =< B,!.
max(A,B,A).

?- max(1,3,M).
M = 3

?- max(1,3,1).
true
```

- Attempt 2:

```
max(A,B,C) :- A =< B,!,B=C.
max(A,B,A).

?- max(1,3,1).
false
```

What happens if we remove the cut?

Cuts - Example

- DB and search tree for rule:

```
b :- c.
b :- d.
d.
e.
c :- 1 = 2.
a(1) :- b.
a(2) :- e.

?- a(X).
X = 1;
X = 2;
```

Cuts – Example

- If we change the first rule to `b :- !, c.`

```
b :- !, c.
b :- d.
d.
e.
c :- 1 = 2.
a(1) :- b.
a(2) :- e.

?- a(X).
X = 2;
```

Practice Exercise

- How to represent binary trees?
  - Use predicates, e.g., "tree”

`void`: empty tree

`tree(K,L,R)`: labeled with K, left subtree L, right subtree R

  - Examples:

```
tree(2,void,void).
tree(4,tree(2,void,void),tree(10,void,void)).
```

Practice Exercise

- tree height

```
height(void, 0).

height(tree(_, L, R), H) :- height(L, H1),
                            height(R, H2),
                            H1 > H2,
                            H is H1+1.

height(tree(_, L, R), H) :- height(L, H1),
                            height(R, H2),
                            H1 =< H2,
                            H is H2+1.
```

Practice Exercise

- Write a predicate **`insert`** to insert an element into a binary search tree

```
insert(K,void,tree(K,void,void)).      % base case

insert(K,tree(K,L,R),tree(K,L,R)).     % no duplicates

insert(K,tree(N,L,R),tree(N,Lnew,R)) :- K < N,
insert(K,L,Lnew).                      % to the left

insert(K,tree(N,L,R),tree(N,L,Rnew)) :- K > N,
insert(K,R,Rnew).                      % to the right
```

Practice Exercise

- Write a predicate **`member`** (binary search)

```
member(K, tree(K,_,_)).
member(K, tree(N,L,_)) :- K < N, member(K, L).
member(K, tree(N,_,R)) :- K > N, member(K, R).
```

- How does it work?

- Compare to what you have to do in other languages

- Here, it is nice because we don't even need to create a new datatype for tree, but simply how you want to traverse the tree

Additional Practice Exercises

- factorial: fac(N,N!)

```
fac(0,1).
fac(N,R) :- N > 0,
      M is N-1,
      fac(M,S),
      R is S*N.
```

- Fibonacci: fib(N,M) -- M is the Nth Fibonacci number

```
fib(0,0).
fib(1,1).
fib(N,M) :- N > 1,
      N1 is N-1,
      N2 is N-2,
      fib(N1,M1),
      fib(N2,M2),
      M is M1+M2.
```

Additional Practice Exercises

- `quicksort(L,S)`: L is the input, and S is the sorted list
- `split(X,L,K,M)`: X is the value to be splitted upon

  L is the input list

  K is the smaller list

  M is the larger list

```
quicksort([],[]).
quicksort([X|L],K) :- split(X,L,L1,L2),
                      quicksort(L1,K1),
                      quicksort(L2,K2),
                      append(K1,[X|K2],K).
split(_,[],[],[]).
split(X,[Y|L],K,[Y|M]) :- X < Y, split(X,L,K,M).
split(X,[Y|L],[Y|K],M) :- X >= Y, split(X,L,K,M).
```

Summary

- Computation through manipulation of relationships
- A Prolog program consists of
  - Facts: unconditional relationships among terms
  - Rules: relationships among terms that may be true under certain conditions
  - Queries that must be true give a set of facts and rules
- A prolog program merely specifies the various relationships among terms
- Prolog system defines how the computation is carried out
  - Unification of variables and terms: assignment of values to variables so that relationships match
  - Implement control by searching the different rules:

    Employ backtracking to search for all solutions