Logic Programming and Prolog

Keep reading: Scott, Chapter 12

1

1

Announcements

- Rainbow grades
  - You should see Quiz 1 and Quiz 2
  - Quiz answers are in forum
- HW1 was due last night
- HW2 is out today
  - I need a few days to set Submitty

2

Lecture Outline

- *Prolog concepts*
  - *Search tree and rule ordering, unification, backtracking, backward chaining*
- *Prolog programming*
  - *Lists*
  - *Programming with lists*
  - *Parsing and language processing*
  - *Arithmetic*

Logical Semantics

- Prolog program consists of facts and rules

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

Rules like **`snowy(X):-rainy(X),cold(X).`**

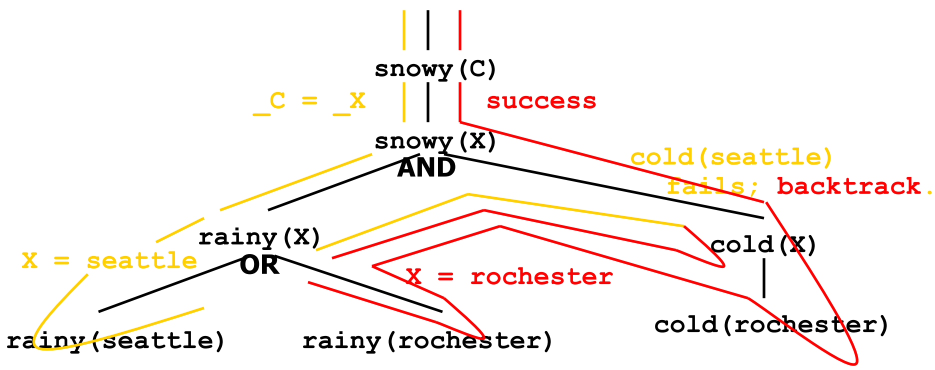
correspond to logical formulas:

$\forall$X[snowy(X) $\leftarrow$ rainly(X) ^ cold(X)]

/* For every X, X is snowy, if X is rainy and X is cold */

Procedural Semantics

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



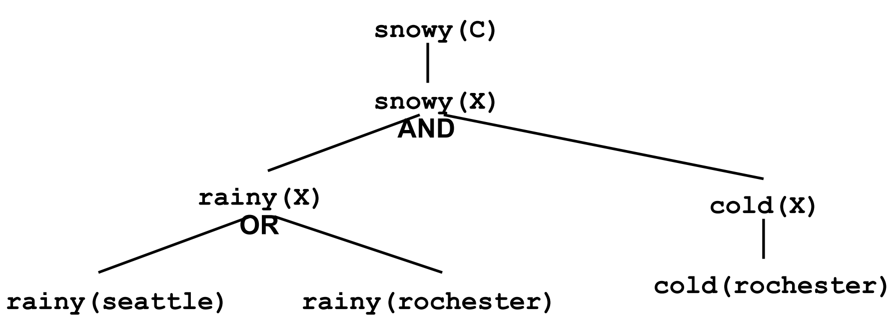
Prolog Concepts: Search Tree

OR levels:
- parent: goal (e.g., `rainy(X)`)
- children: heads-of-clauses (`rainy(…)`)
  - **ORDER:** from left to right

AND levels:
- parent: goal (e.g., `snowy(X)`)
- children: subgoals (`rainy(X), cold(X)`)
  - **ORDER:** from left to right

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

?- snowy(C).
```



Prolog Concepts: Unification

- At **OR** levels Prolog performs unification
  - Unifies parent (goal), with child (head-of-clause)
- E.g.,
  - **`snowy(C) =    snowy(X)`**
    - success, **`_C = _X`**
  - **`rainy(X) =    rainy(seattle)`**
    - success, **`X = seattle`**
  - **`parents(alice,M,F) = parents(edward,victoria,albert)`**
    - fail
  - **`parents(alice,M,F) = parents(alice,victoria,albert)`**
    - success, **`M = victoria, F = albert`**

In Prolog, = denotes unification, not assignment!

7

7

Prolog Concepts: Unification

- A constant unifies only with itself
  - E.g., **`alice=alice`**, but **`alice=edward`** fails
- Two structures unify if and only if (i) they have the same functor, (ii) they have the same number of arguments, and (iii) their arguments unify recursively
  - E.g., **`rainy(X) = rainy(seattle)`**
- A variable unifies with anything. If the other thing has a value, then variable is bound to that value. If the other thing is an unbound variable, then the two variables are associated and if either one gets a value, both do

8

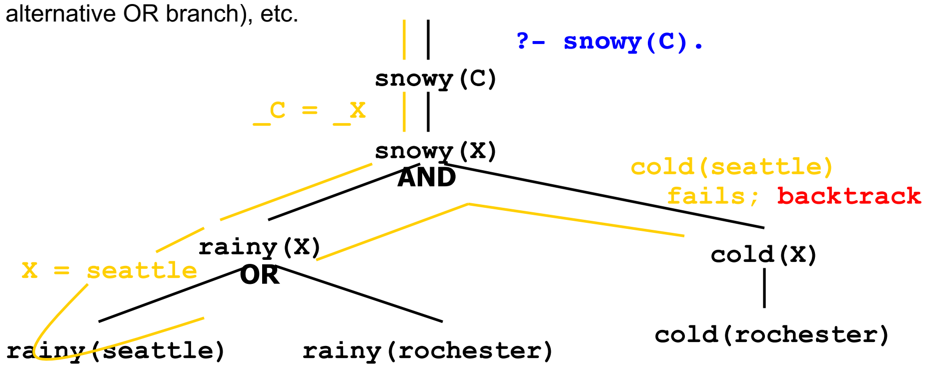
8

Prolog Concepts: Backtracking

If at some point, a goal fails, Prolog backtracks to the last goal (i.e., last unification point) where there is an untried binding, undoes current binding and tries new binding (an alternative OR branch), etc.

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

?- snowy(C).
```



Prolog Concepts: Backward Chaining

- Backward chaining: starts from goal, towards facts

```
? – snowy(rochester).

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

- Forward chaining: starts from facts towards goal

```
? – snowy(rochester).

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

Exercise

```
takes(jane, his).
takes(jane, cs).
takes(ajit, art).
takes(ajit, cs).
classmates(X,Y):-takes(X,Z),takes(Y,Z).

?- classmates(jane,C).
```

C = jane ; // Z = his
C = jane ; // Z = cs
C = ajit. // Z = cs

Draw search tree for query.
What are the bindings for **C**?

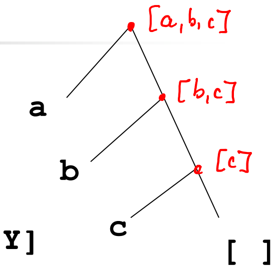
Lecture Outline

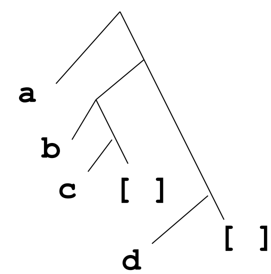
- *Prolog concepts*
  - *Search tree, unification, rule ordering, backtracking, backward chaining*
- *Prolog programming*
  - *Lists*
  - *Programming with lists*
  - *Parsing and language processing*
  - *Arithmetic*

Lists

list	head	tail
`[a,b,c]`	`a`	`[b,c]`
`[X,[cat],Y]`	`X`	`[[cat],Y]`
`[a,[b,c],d]`	`a`	`[[b,c],d]`
`[X	Y]`	`X`	`Y`

[a,b,c] is just a more convenient notation for [a | [b | [c]]].





Lists: Unification

- `[ H1 | T1 ] = [ H2 | T2 ]`
  - Head `H1` unifies with `H2`, possibly recursively
  - Tail `T1` unifies with `T2`, possibly recursively
- E.g., `[ a | [b, c] ] = [ X | Y ]`
  - `X = a`
  - `Y = [b, c]`
- NOTE: In Prolog, = denotes unification, not assignment!

Question

- `[X,Y,Z] = [john, likes, fish]`

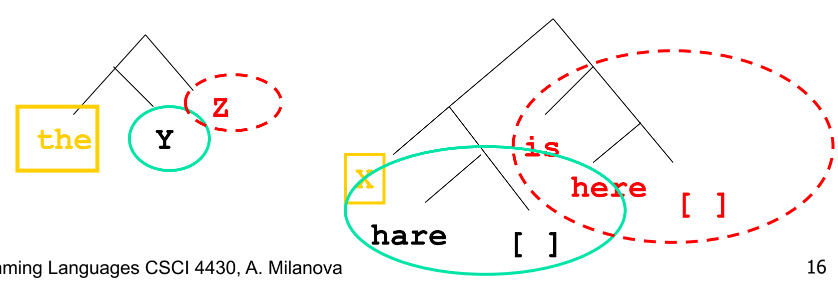
- `[cat] = [X | Y]`

- `[[the, Y]|Z] = [[X, hare]|[is,here]]`

List Unification

- Sequence of comma separated terms, or
- [first term | rest_of_list]

`[ [the | Y] | Z ]   = [ [X, hare] | [is, here] ]`



List Unification

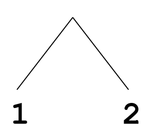
- Look at the trees to see how this works!

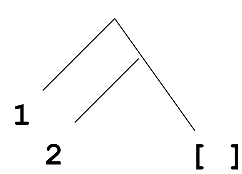
```
[ a, b, c ] = [ X | Y ]
  X = a, Y = [b,c].
```

```
[a | Z ] =? [ X | Y ]
  X = a, Y = Z.
```


Improper and Proper Lists

`[1 | 2]` versus `[1, 2]`





[1, 2|3] = [1|[2|3]]
is an improper list
as well.

Question. Can we unify these lists?

```
[abc, Y]    =?   [ abc | Y ]
```

Answer: No. There is no value binding for **`Y`** that makes these two trees isomorphic

Unification and the Occurs check

occurs_check checks if a variable occurs in a structure (potentially deeply).

```
?- Y = [Y].
Y = [Y].
```

By default, the occurs_check is turned to False. Therefore, unification of a variable with a structure always succeeds (provided the variable is not yet bound).

```
?- set_prolog_flag(occurs_check, true).
true.
```

```
?- Y = [Y].
false.
```

If we turn the occurs_check to True, unification of a variable with a structure that contains the variable, fails.

Lecture Outline

- *Prolog concepts*
  - *Search tree, unification, rule ordering, backtracking, backward chaining*
- *Prolog programming*
  - *Lists*
  - *Programming with lists*
  - *Parsing and language processing*
  - *Arithmetic*

Member_of

ELEMENT LIST

```
1. member(A, [A | B]).
2. member(A, [B | C]) :- member(A, C).
```

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

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

Member_of

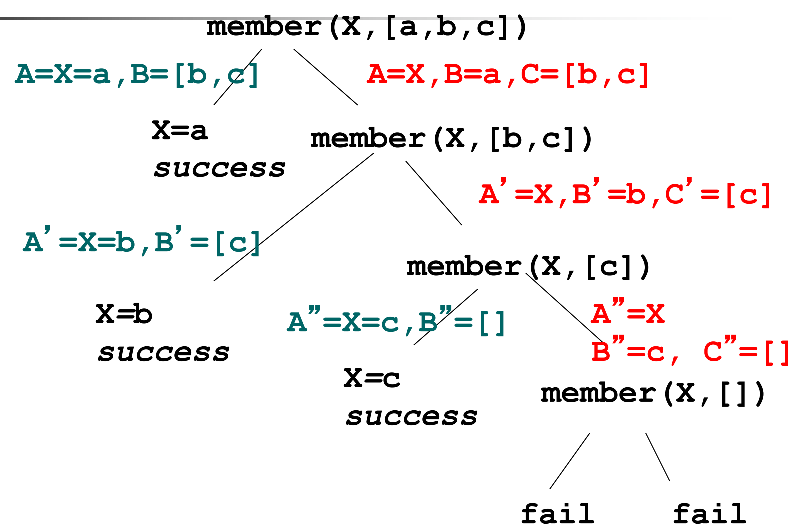
```
1. member(A,  [A | B]).
2. member(A,  [B | C]) :- member (A, C).
```

```
?- member(a,[a,b]).
true.
?- member(a,[b,c]).
false.
?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c.
?- member(a,[b,c,X]).
X = a.
```

23

23

Prolog Search Tree (OR levels only)




```
1. member(A,  [A | B]).
2. member(A,  [B | C]) :- member (A, C).
```

24

24

Member_of

```
member(A, [A|B]).
member(A, [B|C]) :- member(A,C).
```

logical semantics: For every **A,B** and **C**
`member(A,[B|C])` if `member(A,C)`

procedural semantics: Head of clause is procedure entry. Tail of clause is procedure body; subgoals correspond to calls

Procedural Interpretation

```
member(A, [A|B]).
member(A, [B|C]) :- member(A,C).
```

- `member` is a recursive "procedure"
- `member(A, [A|B]).` is the base case. "Returns" true if the element we are looking for, `A`, is the first element in the list. "Returns" false if we have reached the end of the list
- `member(A, [B|C]) :- member(A,C).` is the recursive case. "Returns" result of recursive call with arguments `A` and tail `C`

Question

```
1. member(A, [A | B]).
2. member(A, [B | C]) :- member(A, C).
```

Give all answers to the following query:

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

Answer:

```
  true ;
  X = a.
```


Question

```
1. member(A, [A | B]).
2. member(A, [B | C]) :- member(A, C).
```

Give all answers to the following query:

c

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

fail

member(a,a)

Answer:

unification a=[A|B] fails

a=[B|C] fails as well

```
  false.
```

a

Append

```
append([ ], A, A).
append([A|B], C, [A|D]) :- append(B,C,D).
```

- Build a list:

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

- Break a list into constituent parts:

```
?- append(X,Y,[a,b]).
X = [], Y = [a,b]; X = [a], Y = [b];
X = [a,b], Y = []; false.
```


More Append

```
append([ ], A, A).
append([A|B], C, [A|D]) :- append(B,C,D).
```

- Break a list into constituent parts

```
?- append(X,[b],[a,b]).
X = [ a ]
?- append([a],Y,[a,b]).
Y = [ b ]
```

More Append

```
? - append(X,Y,[a,b]).
X = [ ],
Y = [a,b] ;
X = [a],
Y = [b]   ;
X = [a,b],
Y = [ ]   ;
false.
```


Unbounded Arguments

- Generating an unbounded number of lists

```
?- append(X,[b],Y).
X = [ ],
Y = [b]  ;
X = [ _G604],
Y = [ _G604, b]   ;
X = [ _G604, _G610],
Y = [ _G604, _G610, b]  ;
Etc.
```

An underscore, "don't care" variable. Unifies with anything. E.g., `bad(Dog) :- bites(Dog,_).`

- Be careful when using append with two unbound variable arguments

Question

- What does this ”procedure” do:

```
p([],[]).
p([A|B],[[A]|Rest]) :- p(B,Rest).

  ?- p([a,b,c],Y).
  Y = [ [a],[b],[c] ]
```


Common Structure

- “Processing” a list:

```
p([],[]).
p([H|T],[H1|T1]) :- f(H,H1),p(T,T1).
```

- Base case: we have reached the end of list. Result for `[ ]` is `[ ]`.
- Recursive case: result is `[H1|T1]`. `H1` was obtained by calling `f(H,H1)` --- processes element `H` into result `H1`. `T1` is the result of the recursive call of `p` on `T`

Lecture Outline

- *Prolog concepts*
  - *Search tree, unification, rule ordering, backtracking, backward chaining*
- *Prolog programming*
  - *Lists*
  - *Programming with lists*
  - *Parsing and language processing*
  - *Arithmetic*

Horn Clause Grammars

- Horn clauses naturally encode grammar rules
- Natural language processing

Input sequence.
remainder.

```
?- noun_phrase([the,dog],[]).
true.

?- noun_phrase([goes,away],[]).
false.
```

INPUT
REMAINDER
A noun phrase is an article followed by a noun.

```
noun_phrase(S0,S):-article(A,S0,S1),noun(N,S1,S).
```

matches article in input sequence.

```
article(A,[A|S],S):-member(A,[a,an,the]).
noun(N,[N|S],S):-member(N,[child,dog]).
```

matches noun in input sequence.
If there is a match, break S0 and "return" tail S; otherwise, fail.

Parsing

- A Horn clause grammar rule

STRUCTURE INPUT REMAINDER

```
h(M,N,S0,S) :-
    b1(…,S0,S1),
    b2(…,S1,S2),
    b3(…,S2,S).
```

Think of h as parsing input S0 and producing some structured objects in M and N.

Parsing

$S \rightarrow$ **cAd**
$A \rightarrow$ **a | ab**

- Recursive descent with backtracking, naturally and easily!

Recursive descent structure:

```
S()
  match(a)
  A()
  match(d)
```

Prolog:

```
match(T, [T|S], S).
s_prod(S0, S) :-
    match(c, S0, S1),
    a_prod(S1, S2),
    match(S2, S).

a_prod(S0, S) :- match(a, S0, S).
a_prod(S0, S) :-
    match(a, S0, S1), match(b, S1, S).

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

Parsing

$S \rightarrow$ **a**S**b**S | **b**S**a**S | ε

- Naturally process ambiguous grammars

```
match(T,[T|S],S).
```

```
?- s(D,[a,b,a,b],[]).
D = [1, 2, 3, 3, 3] ;
D = [1, 3, 1, 3, 3] ;
false.
```

- How do I switch order of answers?

Lecture Outline

- *Prolog concepts*
  - *Search tree, unification, rule ordering, backtracking, backward chaining*
- *Prolog programming*
  - *Lists*
  - *Programming with lists*
  - *Parsing and language processing*
  - *Arithmetic*

Arithmetic

- Prolog has all arithmetic operators
- Built-in predicate **is**
  - `is(X, 1+3)` or more commonly we write
  - `X is 1+3`

  `is` forces evaluation of `1+3`:

  ```
  ?- X is 1+3
  X = 4
  ```
- `=` is unification not assignment!

  ```
  ?- X = 4-1.
  X = 4-1 % unifies X with 4-1!!!
  ```


Arithmetic: Pitfalls

- `is` is not invertible! That is, arguments on the right cannot be unbound!
  - `3 is 3 - X.`

  ```
  ERROR: is/2: Arguments are not
   sufficiently instantiated
  ```

- This doesn't work either:

  ```
  ?- X is 4, X = X+1.
  false.
  ```

  Why? What's going on here?

Exercise

- Write **sum**, which takes a list of integers and computes the sum of the integers. E.g.,

```
sum([1,2,3],R).
?- R = 6.
```

- How about if the integers are arbitrarily nested? E.g.,

```
sum([[1],[[[2]],3]],R).
?- R = 6.
```


Exercise

Exercise

- Write **plus10**, which takes a list of integers and computes another list, where all integers are shifted +10. E.g.,

```
plus10([1,2,3],R).
?- R = [11,12,13].
```

- Write **len**, which takes a list and computes the length of the list. E.g.,

```
len([1,[2],3],R).
?- R = 3.
```


Exercise

- Write **atoms**, which takes a list and computes the number of atoms in the list. E.g.,

```
atoms([a,[b,[[c]]]],R).
?- R = 3.
```

- Hint: built-in predicate `atom(X)` yields true if `X` is an atom (i.e., symbolic constant such as `x`, `abc`, `tom`).

Negation

```
not(member(a,[a,b])).

not(member(c,[a,b])).
```

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


The End
  ```