Logic Programming and Prolog

Finish reading: Scott, Chapter 12

Lecture Outline

- *Prolog programming*
  - *Parsing and language processing*
  - *Arithmetic*
  - *Imperative control flow*
  - *Negation by failure*
  - *Generate and test paradigm*

Horn Clause Grammars

- Horn clauses naturally represent grammar rules

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

STRUCTURED OBJECT INPUT SEQUENCE REMAINDER

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

Think of h corresponding to context-free grammar rule:

H → B1 B2 B3

H → A1 A2 A3

Horn Clause Grammars

- Example, simple natural language processing

```
?- noun_phrase([the,dog],[]).
true.
?- noun_phrase([iron,maiden],[]).
false.
?- noun_phrase([goes,away],[]).
false.
```

?- noun_phrase([the,dog,ate,my,homework],R).
R = [ate,my …

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

article(a,[a|S],S).
article(the,[the|S],S).

noun(child,[child|S],S).
noun(dog,[dog|S],S).
```

Parsing

$S \rightarrow$ **c**A**d**
$A \rightarrow$ **a** | **ab**

- Back to parsing formal languages
- Recursive descent with backtracking, naturally!

```
S()
  match(c)
  A()
  match(d)
```

Prolog

```
match(Term,[Term|S],S).
s(S0,S) :-
  match(c,S0,S1),
  a(S1,S2),
  match(d,S2,S).
a(S0,S) :- match(a,S0,S).
a(S0,S) :- match(a,S0,S1),match(b,S1,S).
?- s([c,a,b,d],R).
R=[].
```


Parsing

$S \rightarrow$ **a**S**b**S (1) | **b**S**a**S (2) | **ε** (3)

- Ambiguous grammars, naturally!

```
match(T,[T|S],S).
s(D,S0,S) :-
  match(a,S0,S1), s(D1,S1,S2), match(b,S2,S3), s(D2,S3,S),
  append([1|D1],D2,D).
s(D,S0,S) :- % analogous for (2)
s([3],S0,S0).
```

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

$S \Rightarrow^{(1)} aSbS \Rightarrow abSaSbS \Rightarrow^{*} abab$

Parsing

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

- How do we switch the order of answers?

```
match(T,[T|S],S).
s([3],S0,S0).
s(D,S0,S) :- match(b,S0,S1),s(D1,S1,S2),match(a,S2,S3),s(D2,S3,S),
             append([2|D1],D2,D).
s(D,S0,S) :- ... % analogous, match a, then S, then b, then S.

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


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.
```

% fails because 4 does not unify with +(X,1).

Why? What's going on here?

? X is 4, Y is X+1.
Y = 5. % succeeds

+(X,1) ?- 4 = +(X,1)
false. % fails

? X is 4, X is X+1.
false. 4 ≠ 5
% fails because 4 ≠ 5.

Exercise

- Write `sum`, which takes a flat list of integers and computes their sum. E.g.,

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

sum([],0).
sum([H|T],R) :- integer(H),
sum(T,R1), R is H+R1.

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

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

Exercise

```
sum2([],0).
sum2(H,H):- integer(H).
sum2([H|T],R):- sum2(H,R1), sum2(T,R2),
                R is R1+R2.
```

```
?- sum2([[[1]], [[[2]]]], R).
   R = 3.
```


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.
```

```
len([],0).
len([_|T],R):-
    len(T,R1), R is R1+1.
```

DON'T CARE VARIABLE

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.
```

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

Negation

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

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

female(alice).
parents(alice, victoria, albert).
parents(edward, victoria, albert).

-
```
sister_of(X,Y) :-
    female(X),parents(X,M,F),
    parents(Y,M,F), not(X=Y).
```

?- sister_of(alice, Y).
~~Y = alice ;~~
Y = edward.

Lecture Outline

- *Prolog programming*
  - *Parsing and language processing*
  - *Arithmetic*
  - *Imperative control flow*
  - *Negation by failure*
  - *Generate and test paradigm*

Imperative Control Flow

- Programmer has explicit control on backtracking process

```
cut (!)
```

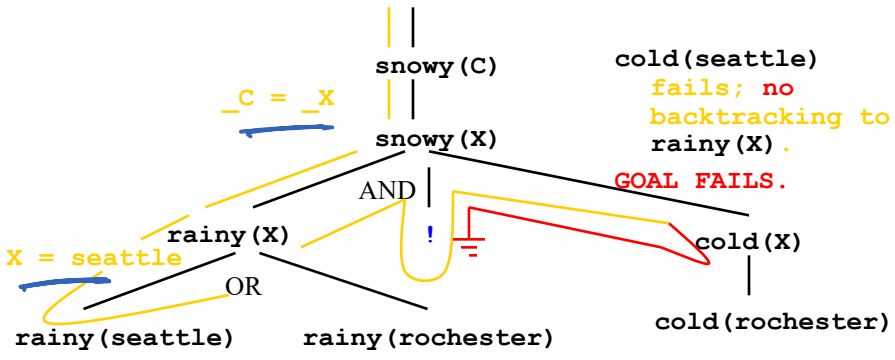
- `!` is a subgoal
- As a goal it succeeds, but with a side effect:
  - Commits interpreter to all bindings made since unifying left-hand side of current rule with parent goal

Cut (!) Example

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

?- snowy(C).
```


Cut (!) Example

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



Cut (!) Example 2

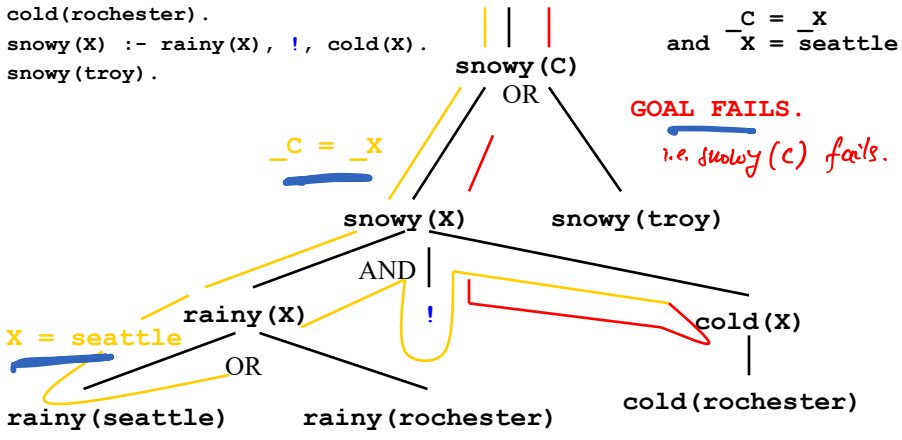
```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), !, cold(X).
snowy(troy).

?- snowy(C).
```


Cut (!) Example 2

?- snowy(rochester).
true.

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

2 committed OR bindings:
_C = _X
and _X = seattle

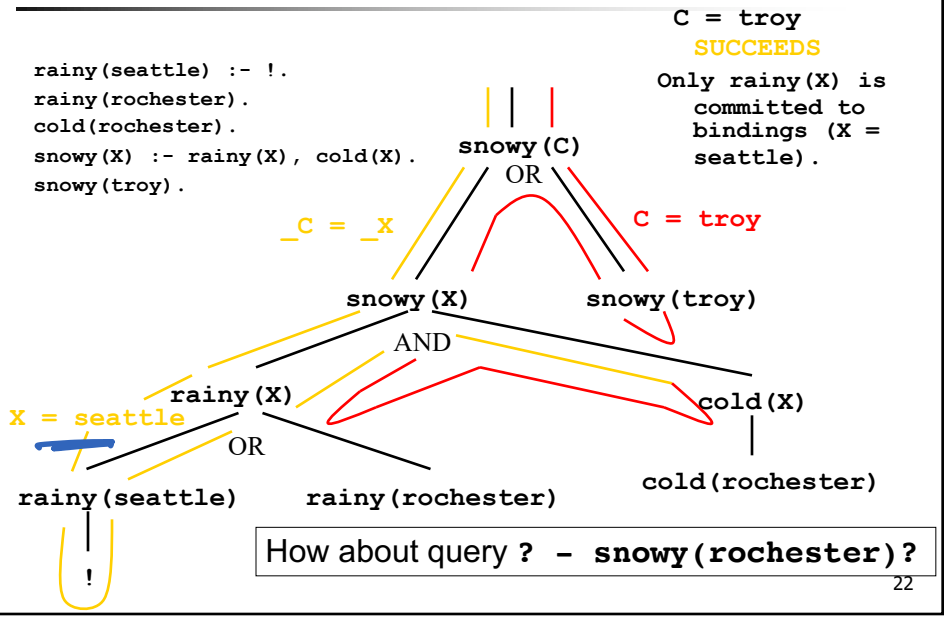


How about query ?- snowy(troy)?
true.

Cut (!) Example 3

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

?- snowy(C).
```


Cut (!) Example 3



How about query `? - snowy(rochester)?`

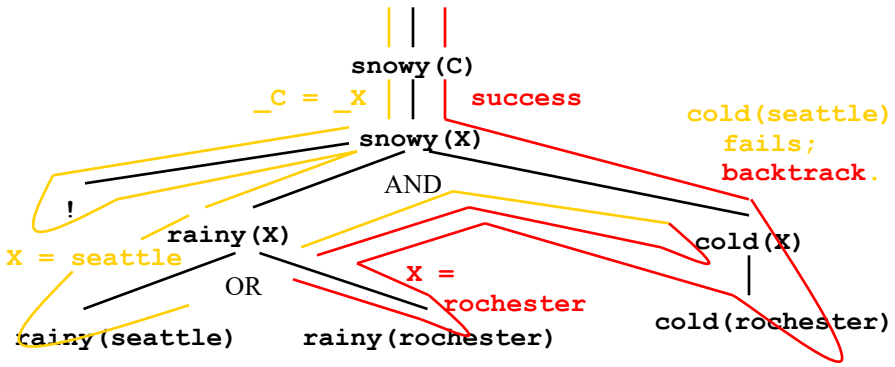
Cut (!) Example 4

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

?- snowy(C).
```


23

Cut (!) Example 4

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



24

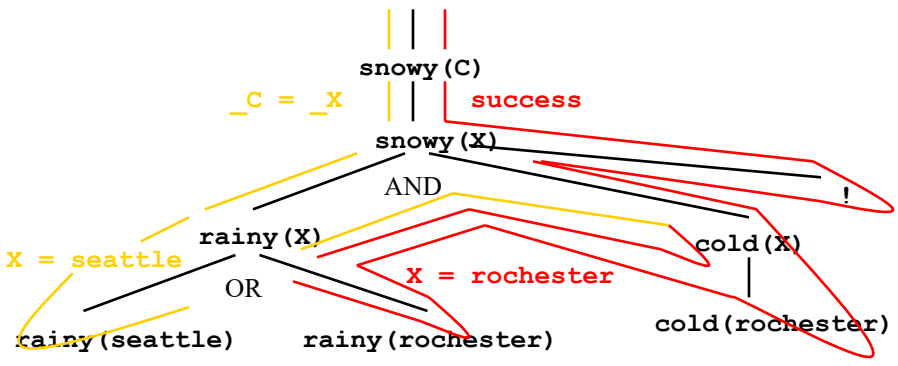
Cut (!) Example 5

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

?- snowy(C).
```


25

Cut (!) Example 5

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

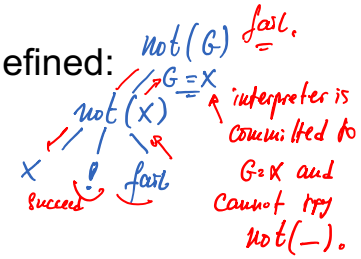


26

Negation by Failure: `not(X)`, `\+(X)`

- **`not(G)`** succeeds when goal `G` cannot be made to succeed
  - Called negation by failure, defined:

```
not(X) :- X,!,fail.
not(_).
```



- Not the same as negation in logic ¬G!

- In Prolog, we can assert that something is true, but we cannot assert that something is false

Exercise

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

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

What are the bindings of `Y`?

Exercise

```
p(X) :- q(X), not(r(X)).
r(X) :- w(X), not(s(X)).
q(a). q(b). q(c).
s(a). s(c).
w(a). w(b).
```

- Evaluate:
  - `?- p(a).`
  - `?- p(b).`
  - `?- p(c).`

Lecture Outline

- *Prolog programming*
  - *Parsing and language processing*
  - *Arithmetic*
  - *Imperative control flow*
  - *Negation by failure*
  - *Generate and test paradigm*

Generate and Test Paradigm

- Search in space
- Prolog rules to generate potential solutions
- Prolog rules to test potential solutions for desired properties
- Easy prototyping of search

```
solve(P) :- generate(P), test(P).
```


A Classical Example: n Queens

- Given an n by n chessboard, place each of n queens on the board so that no queen can attack another in one move
  - Queens can move either vertically,
  - horizontally, or
  - diagonally.
- A classical generate and test problem

n Queens

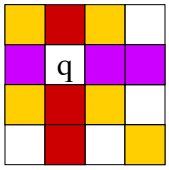
```
my_not(X):- X, !, fail.   %same as not
my_not(_).
in(H,[H|_]).              %same as member
in(H,[_|T]):- in(H,T).


nums(H,H,[H]).
nums(L,H,[L|R]):- L<H, N is L+1, nums(N,H,R).



queen_no(4).
```


n Queens (ii)

```
ranks(L):- queen_no(N), nums(1,N,L).
files(L):- queen_no(N), nums(1,N,L).


rank(R):- ranks(L), in(R,L).


file(F):- files(L), in(F,L).
```

n Queens (iii)

```
attacks((R,_),(R,_)).
attacks((_,F),(_,F)).  %a Prolog tuple
attacks((R1,F1),(R2,F2)):-
                    diagonal((R1,F1),(R2,F2)).
%%%can decompose a Prolog tuple by unification
  (X,Y)=(1,2) results in X=1,Y=2; tuples have fixed
  size and there is not head-tail type construct for
  tuples
```



same rank
same file
same diagonal

What is safe placement for next queen on board?

35

35

n Queens (iv)

```
%%% Two squares are on the same diagonal if the slope of
  the line between them is 1 or -1.  Since / is used, real
  number values for 1 and -1 are needed.

diagonal((X,Y),(X,Y)). %degenerate case
diagonal((X1,Y1),(X2,Y2)):-N is Y2-Y1,D is X2-X1,
  Q is N/D, Q is 1 . %diagonal needs bound
  arguments!
diagonal((X1,Y1),(X2,Y2)):-N is Y2-Y1,D is X2-X1,
  Q is N/D, Q is -1 .
%%%because of use of "is", diagonal is NOT invertible.
```


36

n Queens (v)

```
%%% This solution works by generating every list of
  squares, such that the length of the list is the same
  as the number of queens, and then checks every list
  generated to see if it represents a valid placement of
  queens to solve the N queens problem;
  assume list length function
queens(P):- queen_no(N), length(P,N),
  placement(P), ok_place(P).
```

"generate" code given first "test" code follows

n Queens (vi)

```
placement([]).
placement([(R,F)|P]):- placement(P), rank(R), file(F).
```

n Queens (vii)

```
ok_place([]).
ok_place([(R,F)|P]):- no_attacks((R,F),P),ok_place(P).


no_attacks(_,[]).
no_attacks((R,F),[(R2,F2)|P]):-
  my_not(attacks((R,F),(R2,F2))), no_attacks((R,F),P).
```


The End