Logic Programming

Logic Programming

- Logic programming is based on formal logic
- Logic programming allows a programmer to state a set of axioms from which theorms can be proved
- Prolog follows logic programming
- Axioms are written in a standard form called Horn Clause
- A Horn Clause contains a head(H) and a set of terms (Bi)

- $H \leftarrow B1, B2, \ldots, Bn$

- To derive a new statement, logic programming follows a method called **resolution**
- Combines existing statements
- Cancels like terms

  C ← A, B

  D ← C

  D ← A, B

- During resolution, free variables can get new values by a proess called **unification**

  flowery(X) ← rainy(X)

  rainy(Rochester)

  ----

  flowery(Rochester)

Prolog

- Prolog interpreter works on a database on Horn Clauses that are assumed to be true
- Each Clause contains constants, variables or structures
- Constant
  - Either an atom or a number
  - Atoms are like symbols in Lisp
  - Numbers are integers or floating point

- Variable
  - Identifiers beginning with uppercase letter
  - instantiated to arbitrary values at run time as a result of unification
  - Scope variable is limited to its clause
  - No declaration are available
- Structure
  - contains an atom called functor and a list of arguments
- rainy(rochester)
- teaches(scott, cs254)
- Arity of structure- No. of arguments

- Arguments can be constants,variables or nested structures
- Structures are also called logical predicates
- Clause can be classified as facts or rules
- Each one ends with a period(.)
- **Fact**
  - Clause without a right hand side
  - rainy(rochester).
- **Rule**
  - Has right hand side
  - snowy(X) :- rainy(X), cold(X).

- Token
  - :- indicates implication symbol
  - . indicates 'end'
- Variables in the head of Clause are universally quantified
  - for all X,X is snowy if X is rainy and X is cold
- Clause with empty left hand side is called a **query or goal**
- Queries donot appear in Prolog
- They are inserted using symbol?-

- rainy(seattle).
- rainy(rochester).
- ?- rainy(C).
- the Prolog interpreter would respond with
- C = seattle ;
- C = rochester ;
- No

- Similarly, given

  rainy(seattle).

  rainy(rochester).

  cold(rochester).

  snowy(X) :- rainy(X), cold(X).
- the query

  ?- snowy(C).
- will yield only one solution –Y/N

Resolution and Unification

- Resolution priciple of Robinson says
  - if C1 and C2 are Horn clauses
  - Head of C1 matches one of the terms in the body of C2 then we can replace the term in C2 with the body of C1

takes(jane_doe, his201).

takes(jane_doe, cs254).

takes(ajit_chandra, art302).

takes(ajit_chandra, cs254).

classmates(X, Y) :- takes(X, Z), takes(Y, Z).

let X be jane_doe and Z be cs254,

(Head of C1 matches one of the terms in the body of C2 then we can replace the term in C2 with the body of C1) - we can replace the first term on the right-hand side of the last clause with the (empty) body of the second clause, yielding the new rule

- classmates(jane_doe, Y) :- takes(jane_doe ,cs254), takes(Y, cs254).
- classmates(jane_doe, Y) :- takes(Y, cs254).
- Y is a classmate of jane_doe if Y takes cs254

Unification

- The pattern-matching process used to associate X with jane_doe and Z with cs254 is known as *unification*

- Variables that are given values as a result of unification are said to be *instantiated*.

Rules

- A constant unifies only with itself.
- Two structures unify if and only if they have the same functor and the same arity
  - corresponding arguments unify recursively.
- A variable unifies with anything.
  - If the other thing has a value, then the variable is instantiated.
  - If the other thing is an uninstantiated variable, then the two variables are associated in such a way that if either is given a value later, that value will be shared by both.

Example

```
?- a = a.
Yes                 % constant unifies with itself

?- a = b.
No                  % but not with another constant

?- foo(a, b) = foo(a, b).
Yes                 % structures are recursively identical
?- X = a.
X = a ;             % variable unifies with constant
No                  % only once
?- foo(a, b) = foo(X, b).
X = a ;             % arguments must unify
No                  % only one possibility
```

Lists

- List is an ordered sequence of elements of arbitrary length
- Elements can be constants,variables,structures or other lists
- Elements are enclosed in square brackets separated by commas

[1,2,3]

[John,Mary]

[a,b,[a,b,c]]

[ ]

- The notation [X|Y] refers to a list whose first element is X and whose tail is Y
- List [1,2,3] could be expressed as

[1|[2,3]]=[1,2|[3]]=[1,2,3|[ ]]

Member Predicate

- member(?Elem,?List)
- True if Elem is a member of list
  - member(X,[X|_]).
  - member(X,[_|T]):-member(X,T).

- We can read the clause the following way, respectively
  - X is a member of a list whose first element is X
  - X is a member of a list whose tail is T if X is a member of T
- If X is not the first element in list, try to find X

recursively by applying 2nd rule saying that X is member of list if it is a member of the tail of the list

Example

- ?-member(red,[red,green,blue]).
- Prolog answer yes
- ?-member(blue[red,green,blue,yellow]).

- Here we go to 2nd rule-recursive rule
- ?-member(blue[green,blue,yellow]).
- ?-member(blue[blue,yellow]).

- Now the first clause does help and the query succeeds

?-member[X,[1,2,3]).
X=1;
X=2;
X=3;
No
?-member([5],[2,5]).
No
[5] is different from 5
?-member([5],[2,[5]])
Yes

append

- append(L1,L2,A)
- Appends L2 to L1 and the result list is in A
- append([ ],A,A)
- Appending a list to an empty list
- append([H|T],A,[H|L]):- append(T,A,L).
- Gets the A by removing the elements of [H|T] in [H|L]

Example

- ?-append([a,b,c],[d,e],L).

  L=[a,b,c,d,e]

- ?-append([X,[d,e],[a,b,c,d,e]).

  X=[a,b,c]

- ?-append([a,b,c],Y,[a,b,c,d,e])

  Y=[d,e]

Arithmetic

- The usual arithmetic operators are available in Prolog
- But they play the role of predicates, not of functions.

- +(2, 3)= 2+3

- This is two-argument structure;not function call

```
?- (2 + 3) = 5.
No
```

- to handle arithmetics,Prolog provides a built in predicate 'is'
- 'is' unifies 1st arguments with arithmetic value of 2nd argument

Example

- ?- is(X, 1+2).

  X = 3

- ?- X is 1+2.

  X = 3 % infix is also ok

- ?- 1+2 is 4-1.

  No % first argument (1+2) is already instantiated

- ?- X is Y.

  ERROR

Search/Execution Order

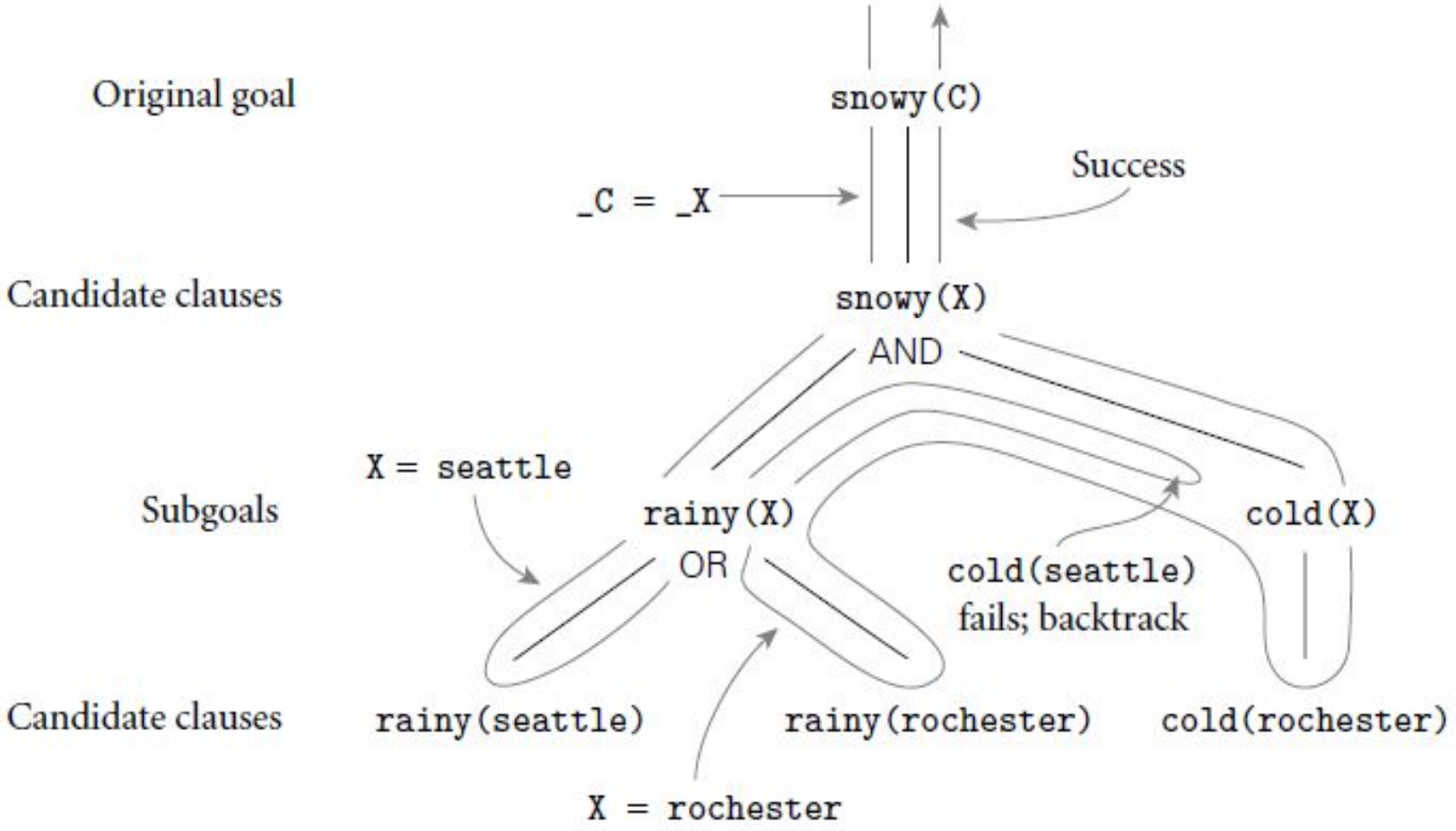
- There are two principal search strategies:
- **Forward Chaining**
  - Start with existing clauses and work forward, attempting to derive the goal
- **Backward Chaining**
  - Start with the goal and work backward, attempting to "unresolve" it into a set of preexisting clauses
  - If the number of existing rules is very large, but the number of facts is small, then forward chaining is more efficient

Backward chaining

- Prolog uses backward chaining interpreter explores tree in depth first from left to right
- Starts with begining of DB searching for a rule r whose head can be unified with top level goal

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

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



Implementation

- Uses single stack implementation
- interpreter pushes a frame very time it pursues a new subgoal G
- If G fails,frame is poped and interprter backtracks
- If G succeeds,control returns to caller
- Later subgoals will be given space above this frame

Recursive Programs

- Programmer must consider the order to ensure if recursive programs terminate
- We have a DB with Directed Acyclic Graph

1. edge(a, b). edge(b, c). edge(c, d).
2. edge(d, e). edge(b, e). edge(d, f).
3. path(X, X).
4. path(X, Y) :- edge(Z, Y), path(X, Z).

- Goal: To determine if there is a path from X to Y

- If we reverse the order of terms

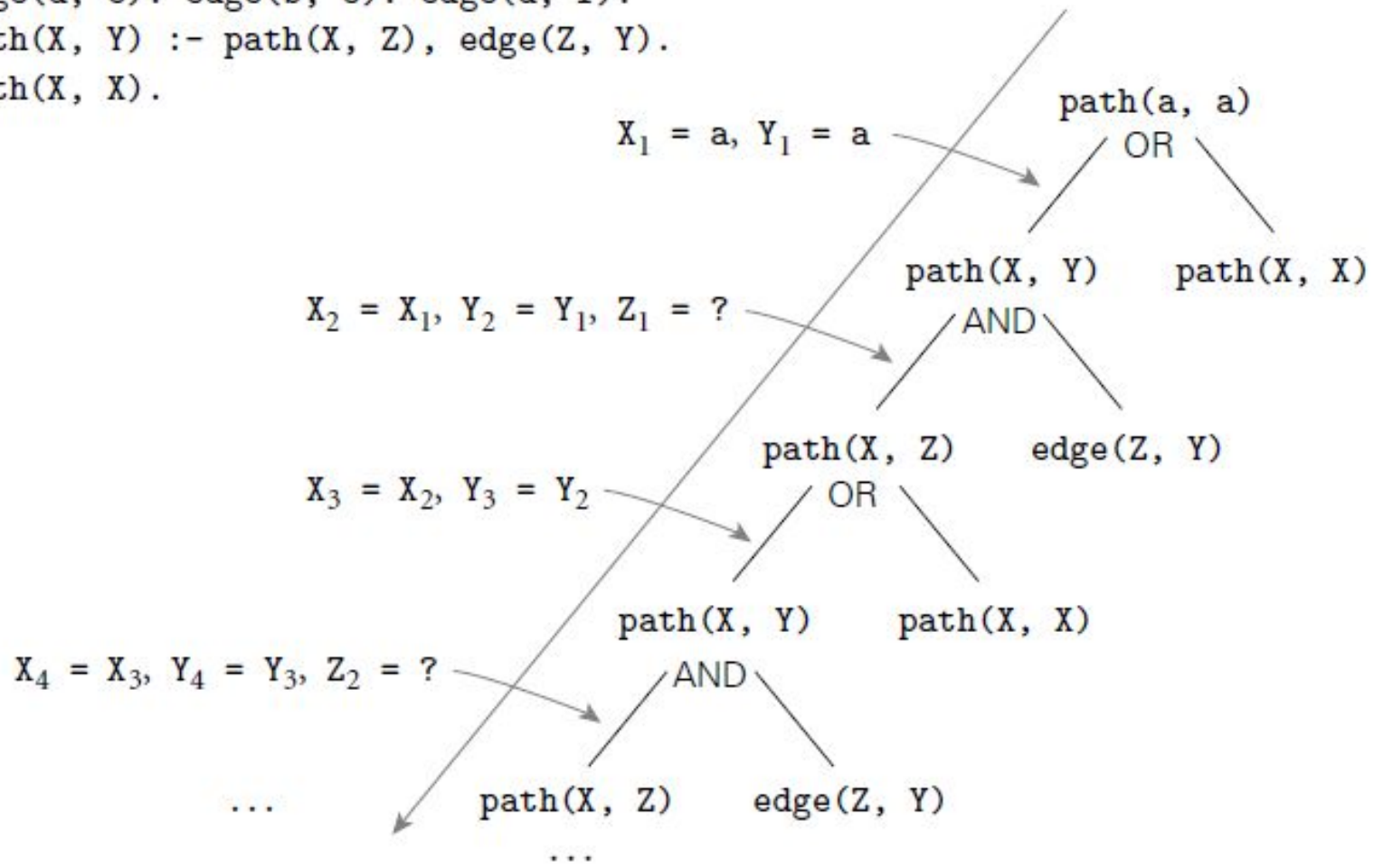
  path(X, Y) :- path(X, Z), edge(Z, Y).

- Search if Z reachable from X before checking edge from Z to Y
- The program would still work,but it would not be that efficient
- But if we reverse the order of last 2 clauses

  path(X, Y) :- path(X, Z), edge(Z, Y).

  path(X, X)
- No answer for interpreter

```
edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).
```



Imperative Control Flow

- Cut
  - Cut is a zero argument predicate
  - Written using exclamation mark(!)
  - Cut is a goal that always succeeds
  - Used to prevent backtracking

Example

```
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).
?-member(a,L).
```

- If 'a' appears n times in L
- ?-memeber(a,L) can succeed n times
- Extra successes leads to wasted computation
- Time can be saved by cutting further searches for 'a' after its 1st occurrence

- This can be accomplished using cut predicate

```
member(X, [X | _ ]) :- !.
member(X, [ _ | T]) :- member(X, T).
```

- If the 1st clause succeeds then we need not execute the 2nd one

- Another way to ensure this is

```
member(X, [X | _ ]).
member(X, [H | T]) :- X \= H, member(X, T).
```

- X \= H implies X not equal to H

- X \= H can be written as \+(X = H).

- \+ can be implemented by a combination of the cut and 2 other built-in predicates - call and fail:

```
\+(P) :- call(P), !, fail.
\+(P).
```

- \+(P) succeeds if P is not provable and fails if P is provable
- call(P) forces an attempt to satisfy P
- fail predicate always fails

\+(P) :- call(P), !, fail.

- Calls P which tries to prove P. If it succeeds, interpreter stops there and fails

- If call(P) fails, not(P) will be invoked making the interpreter return a 'yes'.

- cut can be used to implement if...then ..else as

statement:-condition,!, then_part.

statement:-else_part.

Loops

- 'fail' predicate can be used in conjunction with a generator to implement a loop
- Example
- To partition a list into different combinations
- We use combination of append and fail predicates

```
print_partitions(L) :- append(A, B, L),
write(A), write(' '), write(B), nl,
fail.
```

```
print_partitions(L) :- append(A, B, L),
write(A), write(' '), write(B), nl,
fail.
print_partitions([a,b,c])
```

will give the following o/p

- [][a,b,c]
- [a][b,c]
- [a,b,c][]
- No
- Prolog interpreter fails each time
- It then backtracks and tries with new possibilities
- If we donot want overall predicate to fail , we can add
- print_partitions(-)

Looping with an unbounded generator

```
natural(1).
natural(N) :- natural(M), N is M+1.
```

- We can use this generator in conjunction with a “test-cut” combination to iterate over the first n numbers:

```
my_loop(N) :- natural(I),
write(I), nl,
I = N, !.

?-my_loop(3).
1
2
3
yes
```

I/O FEATURES

- write-print to the current output file
- nl-newline character
- read predicate- used to read terms from current input file
- Individual characters are read and written with get and put
- Input and output can be redirected to different files using see and tell
- consult and reconsult can be used to read database clauses from a file

Database Manipulation

- Clauses in Prolog are simply collections of terms, connected by the built in predicates:- and,
- Both of which can be written in either infix or prefix form

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

$\equiv$

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

Example

- A Program can add clauses to database with built in predicate **assert** and remove using **retract**
- **Example**

?- rainy(X).

X = seattle ;

X = rochester ;

No

?- assert(rainy(syracuse)).

Yes

?- rainy(X).

X = seattle ;

X = rochester ;

X = syracuse ;

No

?- retract(rainy(rochester)).

Yes

```
?- rainy(X).
X = seattle ;
X = syracuse ;
No

?-retract(rainy(rochester)).
yes
?- rainy(X).
X = seattle ;
X = syracuse ;
No
```

Functor predicate

- Individual terms in Prolog can be created, or their contents extracted, using the built-in predicates functor, arg, and =...
- The goal functor(T, F, N) succeeds if and only if T is a term with functor F and arity N:

?- functor(foo(a, b, c), foo, 3).

Yes

?- functor(foo(a, b, c), F, N).

F = foo

N = 3

?- functor(T, foo, 3).

T = foo(_10, _37, _24)

arg(N, T, A)

- The goal arg(N, T, A) succeeds if and only if its first two arguments (N and T) are instantiated, N is a natural number, T is a term, and A is the Nth argument of T:

?- arg(3, foo(a, b, c), A).

A = c

Using functor and arg together

?- functor(T, foo, 3), arg(1, T, a), arg(2, T, b), arg(3, T, c).

T = foo(a, b, c)

- Alternatively, we can use the (infix) =.. predicate, which “equates” a term with a list:

?- T =.. [foo, a, b, c].

T = foo(a, b, c)

?- foo(a, b, c) =.. [F, A1, A2, A3].

F = foo

A1 = a

A2 = b

A3 = c