

CS571: Artificial Intelligence

Pushpak Bhattacharyya
Predicate calculus, Prolog, Circuit
verification

28th August, 2018

Predicate calculus

Introduce through the “Himalayan
Club Example”

Himalayan Club example

- Introduction through an example (*Zohar Manna, 1974*):
 - Problem: A, B and C belong to the Himalayan club. Every member in the club is either a mountain climber or a skier or both. A likes whatever B dislikes and dislikes whatever B likes. A likes rain and snow. No mountain climber likes rain. Every skier likes snow. *Is there a member who is a mountain climber and not a skier?*
- Given knowledge has:
 - Facts
 - Rules

Example contd.

- Let mc denote mountain climber and sk denotes skier. Knowledge representation in the given problem is as follows:

1. $member(A)$
2. $member(B)$
3. $member(C)$
4. $\forall x[member(x) \rightarrow (mc(x) \vee sk(x))]$
5. $\forall x[mc(x) \rightarrow \sim like(x, rain)]$
6. $\forall x[sk(x) \rightarrow like(x, snow)]$
7. $\forall x[like(B, x) \rightarrow \sim like(A, x)]$
8. $\forall x[\sim like(B, x) \rightarrow like(A, x)]$
9. $like(A, rain)$
10. $like(A, snow)$
11. Question: $\exists x[member(x) \wedge mc(x) \wedge \sim sk(x)]$

- We have to infer the 11th expression from the given 10.
- Done through Resolution Refutation.

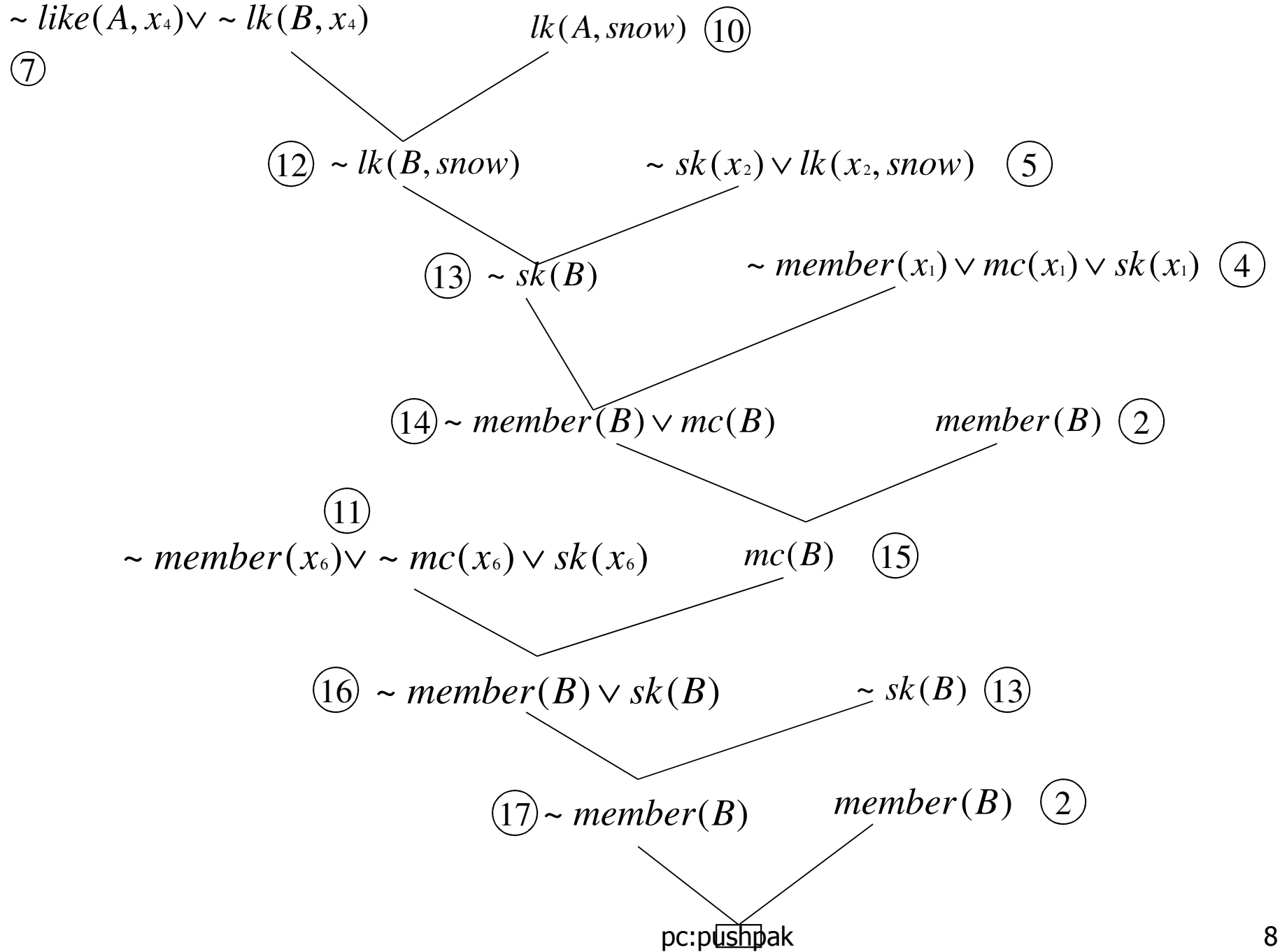
Club example: Inferencing

1. $member(A)$
2. $member(B)$
3. $member(C)$
4. $\forall x[member(x) \rightarrow (mc(x) \vee sk(x))]$
 - Can be written as
 - $\sim member(x) \vee mc(x) \vee sk(x)$
5. $\forall x[sk(x) \rightarrow lk(x, snow)]$
 - $\sim sk(x) \vee lk(x, snow)$
6. $\forall x[mc(x) \rightarrow \sim lk(x, rain)]$
 - $\sim mc(x) \vee \sim lk(x, rain)$
7. $\forall x[like(A, x) \rightarrow \sim lk(B, x)]$
 - $\sim like(A, x) \vee \sim lk(B, x)$

8. $\forall x[\sim lk(A, x) \rightarrow lk(B, x)]$
 – $lk(A, x) \vee lk(B, x)$
9. $lk(A, rain)$
10. $lk(A, snow)$
11. $\exists x[member(x) \wedge mc(x) \wedge \sim sk(x)]$
 – Negate– $\forall x[\sim member(x) \vee \sim mc(x) \vee sk(x)]$

- Now standardize the variables apart which results in the following

1. $member(A)$
2. $member(B)$
3. $member(C)$
4. $\sim member(x_1) \vee mc(x_1) \vee sk(x_1)$
5. $\sim sk(x_2) \vee lk(x_2, snow)$
6. $\sim mc(x_3) \vee \sim lk(x_3, rain)$
7. $\sim like(A, x_4) \vee \sim lk(B, x_4)$
8. $lk(A, x_5) \vee lk(B, x_5)$
9. $lk(A, rain)$
10. $lk(A, snow)$
11. $\sim member(x_6) \vee \sim mc(x_6) \vee sk(x_6)$
pc:pushpak



Well known examples in Predicate Calculus

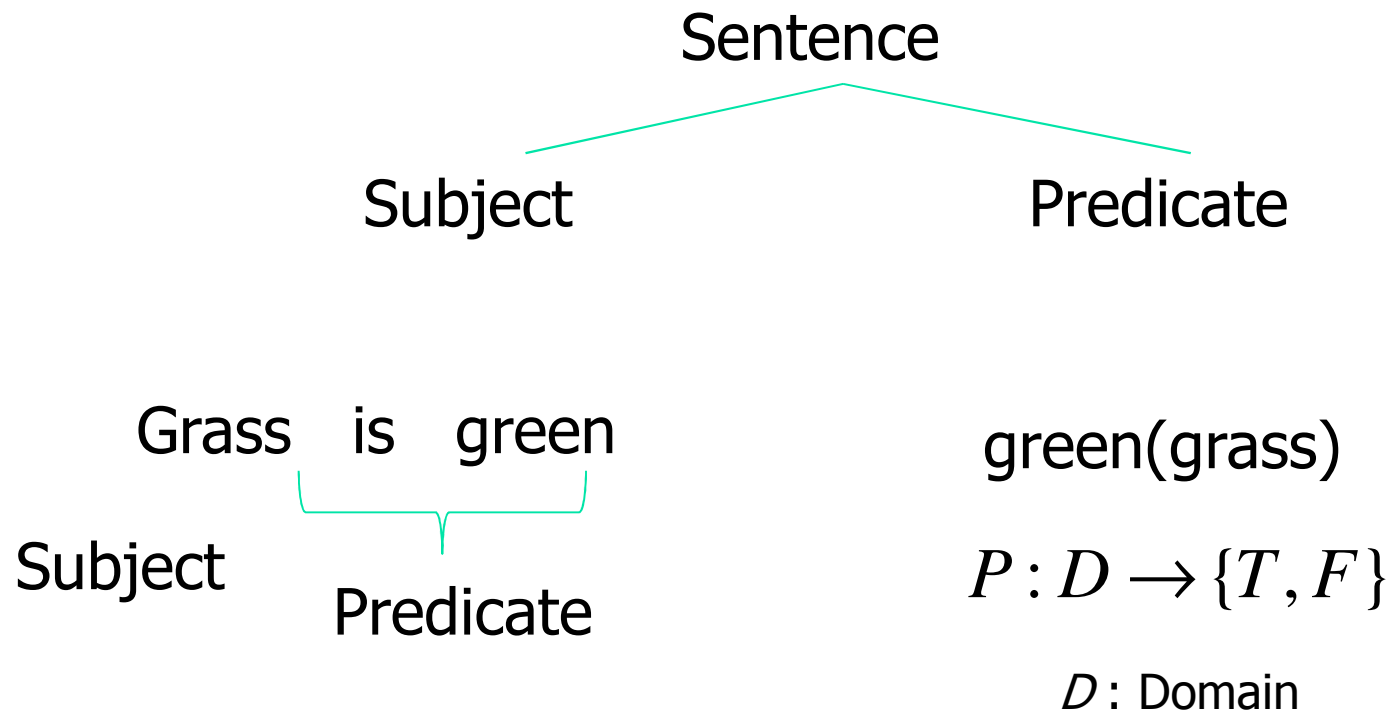
- Man is mortal : rule

$$\forall x[man(x) \rightarrow mortal(x)]$$

- shakespeare is a man
man(shakespeare)
- To infer shakespeare is mortal
mortal(shakespeare)

Predicate Calculus: origin

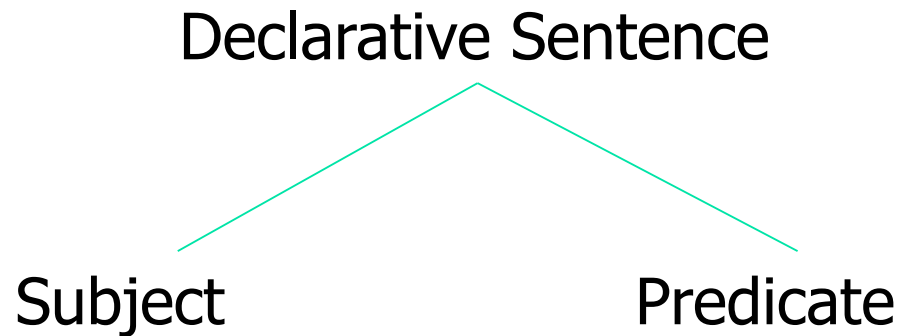
- Predicate calculus originated in language



Predicate Calculus: only for declarative sentences

just like propositional calculus

- Is grass green? (Interrogative)
- Oh, grass is green! (Exclamatory)



- Grass which is supple is green

$$\forall x(\text{grass}(x) \wedge \text{supple}(x) \rightarrow \text{green}(x))$$

Predicate Calculus: more expressive power than propositional calculus

- 2 is even and is divisible by 2: P1
- 4 is even and is divisible by 2: P2
- 6 is even and is divisible by 2: P3

Generalizing, predicate calculus has the power to generalize

$$\forall x((Integer(x) \wedge even(x) \Rightarrow divides(2, x))$$

Predicate Calculus: finer than propositional calculus

1. ■ Finer Granularity (Grass is green, ball is green, leaf is green (green(x)))
2. Succinct description for infinite number of statements which would need ∞ number of properties

3 place predicate

Example: x gives y to z give(x,y,z)

4 place predicate

Example: x gives y to z through w give(x,y,z,w)

Double causative in Hindi giving rise to higher place predicates

- जॉन ने खाना खाया
John ne khana khaya
John <CM> food ate
John ate food
eat(John, food)
- जॉन ने जैक को खाना खिलाया
John ne Jack ko khana khilaya
John <CM> Jack <CM> food fed
John fed Jack
eat(John, Jack, food)
- जॉन ने जैक को जिल के द्वारा खाना खिलाया
John ne Jack ko Jill ke dvara khana khilaya
John <CM> Jack <CM> Jill <CM> food made-to-eat
John fed Jack through Jill
eat(John, Jack, Jill, food)

PC primitive: N-ary Predicate

predicates produce a T,F value for the given function taking propositional things as input

$$P(a_1, \dots, a_n)$$

$$P : D^n \rightarrow \{T, F\}$$

- Arguments of predicates can be variables and constants
- Ground instances : Predicate all whose arguments are constants

N-ary Functions

Functions take in constant as input and produce a variable or constant as output

$$f : D^n \rightarrow D$$

president(India) : Pranab Mukherjee

- Constants & Variables : Zero-order objects
- Predicates & Functions : First-order objects

Prime minister of India is older than the president of India

older(prime_minister(India), president(India))

Operators

$$\wedge \vee \sim \oplus \forall \rightarrow \exists$$

- Universal Quantifier
- Existential Quantifier

All men are mortal

$$\forall x[man(x) \rightarrow mortal(x)]$$

Some men are rich

$$\exists x[man(x) \wedge rich(x)]$$

Tautologies

$$\sim \forall x(p(x)) \rightarrow \exists x(\sim p(x))$$

$$\sim \exists x(p(x)) \rightarrow \forall x(\sim p(x))$$

- 2nd tautology in English:
 - *Not a single man in this village is educated implies all men in this village are uneducated*
- Tautologies are important instruments of logic, but uninteresting statements!

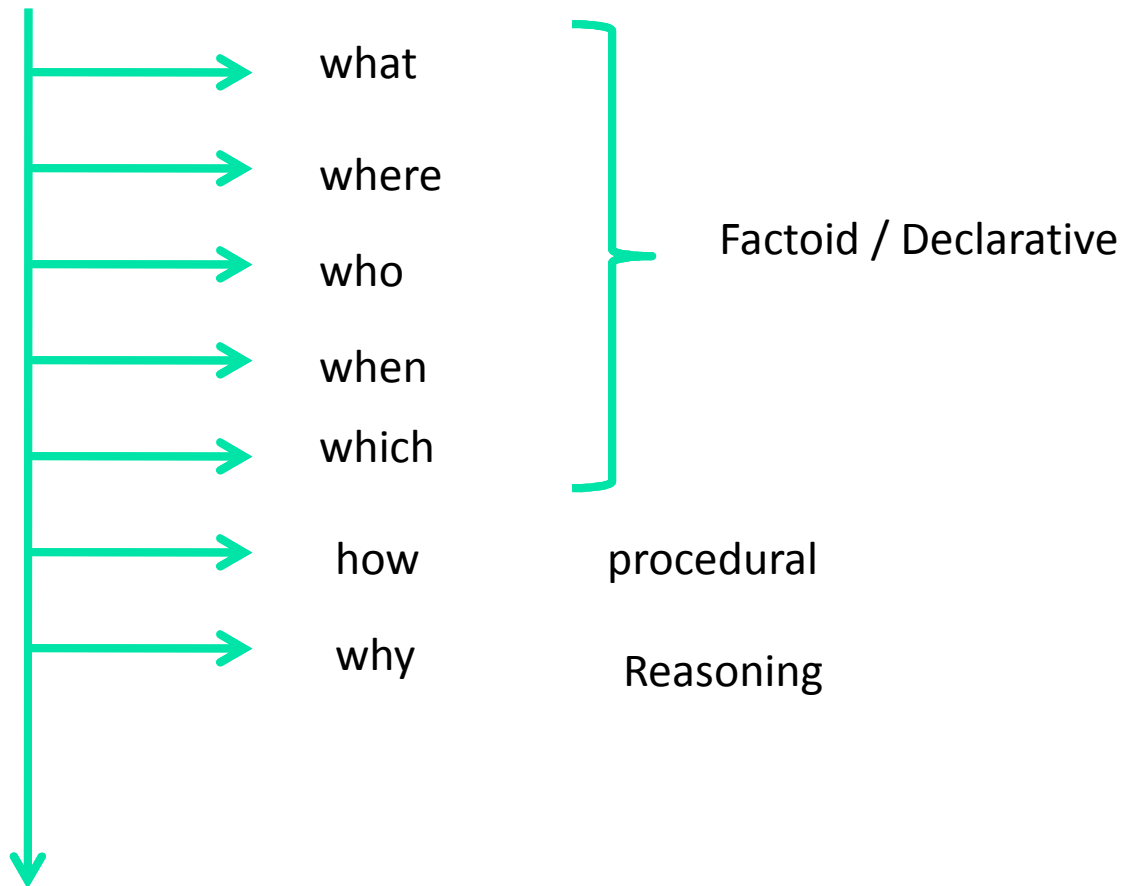
Inferencing: Forward Chaining

- $man(x) \rightarrow mortal(x)$
 - *Dropping the quantifier, implicitly Universal quantification assumed*
 - $man(shakespeare)$
- Goal $mortal(shakespeare)$
 - Found in one step
 - $x = shakespeare$, unification

Backward Chaining

- $man(x) \rightarrow mortal(x)$
- Goal $mortal(shakespeare)$
 - $x = shakespeare$
 - Travel back over and hit the fact asserted
 - $man(shakespeare)$

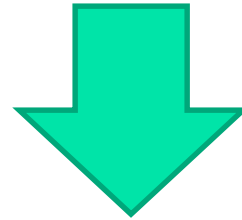
Wh-Questions and Knowledge



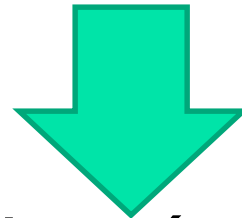
Fixing Predicates

- Natural Sentences

<Subject> <verb> <object>



Verb(subject,object)



predicate(subject)

Examples

- Ram is a boy
 - Boy(Ram)?
 - Is_a(Ram,boy)?
- Ram Plays Football
 - Plays(Ram,football)?
 - Plays_football(Ram)?

Knowledge Representation of Complex Sentence

- *"In every city there is a thief who is beaten by every policeman in the city"*

Knowledge Representation of Complex Sentence

- *"In every city there is a thief who is beaten by every policeman in the city"*

$\forall x[\text{city}(x) \rightarrow \{\exists y((\text{thief}(y) \wedge \text{lives_in}(y, x)) \wedge \forall z(\text{policeman}(z, x) \rightarrow \text{beaten_by}(z, y)))\}]$

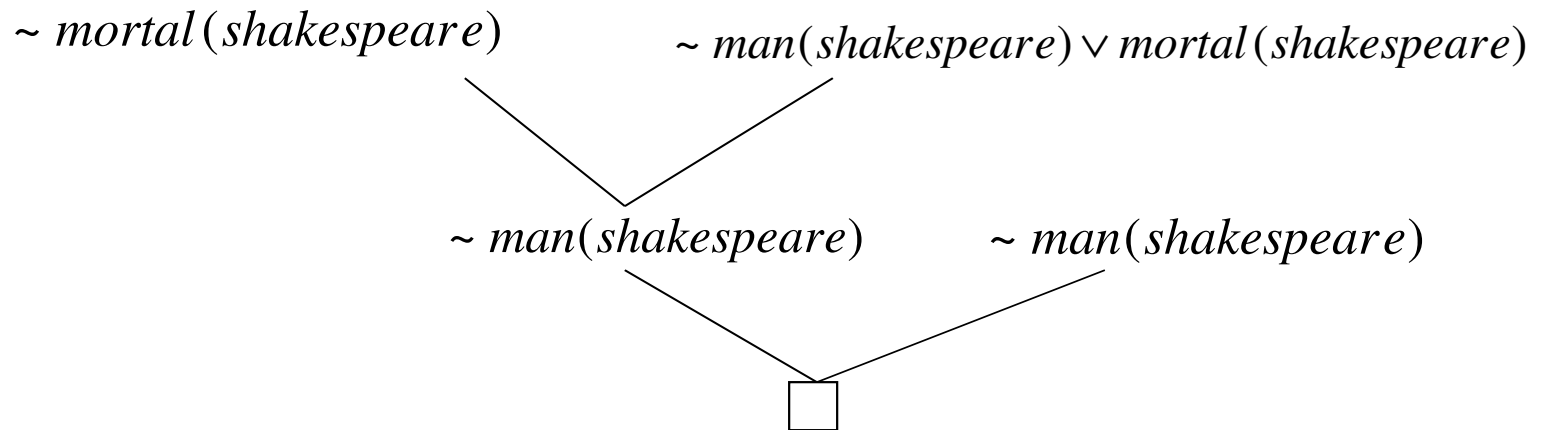
Insight into resolution

Resolution - Refutation

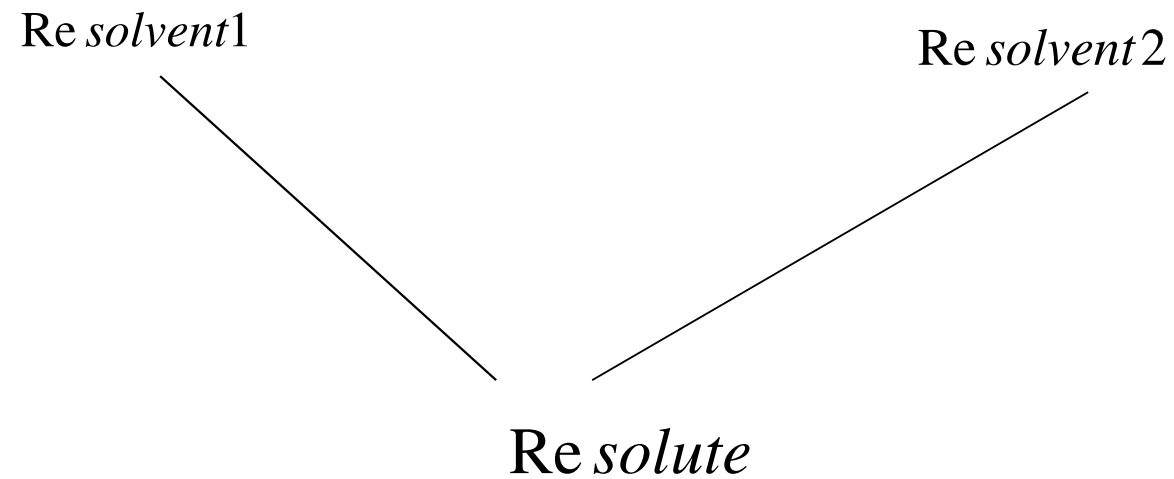
- $man(x) \rightarrow mortal(x)$
 - *Convert to clausal form*
 - $\sim man(shakespeare) \vee mortal(x)$
- **Clauses in the knowledge base**
 - $\sim man(shakespeare) \vee mortal(x)$
 - $man(shakespeare)$
 - $mortal(shakespeare)$

Resolution – Refutation contd

- *Negate the goal*
 - $\sim \text{man}(\text{shakespeare})$
- Get a pair of resolvents



Resolution Tree



Search in resolution

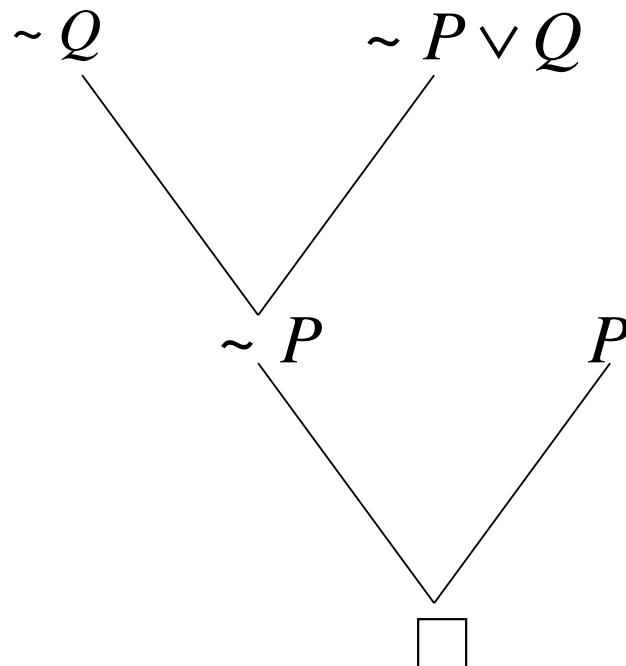
- Heuristics for Resolution Search
 - Goal Supported Strategy
 - Always start with the negated goal
 - Set of support strategy
 - Always one of the resolvents is the most recently produced resolute

Inferencing in Predicate Calculus

- Forward chaining
 - Given P , $P \rightarrow Q$, to infer Q
 - P , match *L.H.S* of
 - Assert Q from *R.H.S*
- Backward chaining
 - Q , Match *R.H.S* of $P \rightarrow Q$
 - assert P
 - Check if P exists
- Resolution – Refutation
 - Negate goal
 - Convert all pieces of knowledge into clausal form (disjunction of literals)
 - See if contradiction indicated by null clause \square can be derived

1. P
2. $P \rightarrow Q$ converted to $\sim P \vee Q$
3. $\sim Q$

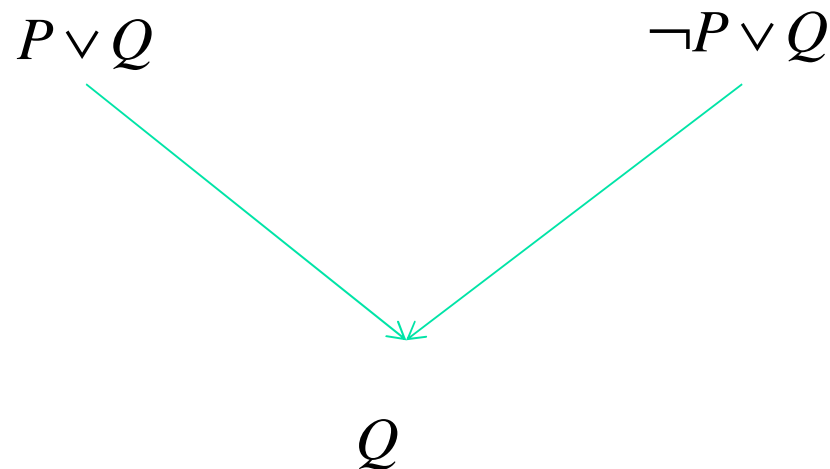
Draw the resolution tree (actually an inverted tree). Every node is a clausal form and branches are intermediate inference steps.



pc:pushpak

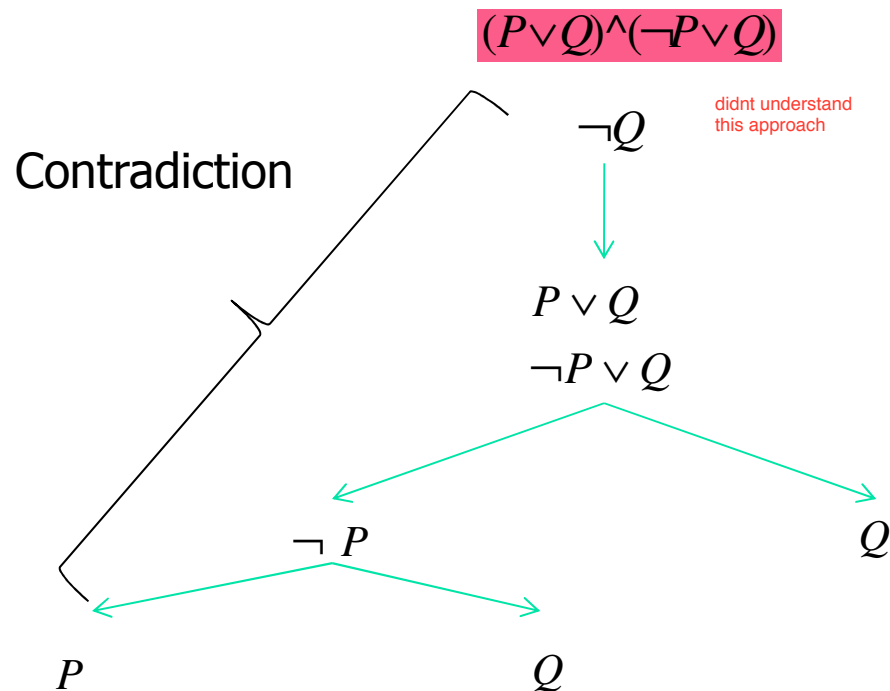
Theoretical basis of Resolution

- Resolution is proof by contradiction
- ***resolvent1 .AND. resolvent2 => resolute*** is a tautology



Tautologiness of Resolution

- Using Semantic Tree



Theoretical basis of Resolution (cont ...)

- Monotone Inference

- Size of Knowledge Base goes on increasing as we proceed with resolution process since intermediate resolvents added to the knowledge base

- Non-monotone Inference

- Size of Knowledge Base does not increase
- Human beings use non-monotone inference

Interpretation in Logic

- Logical expressions or formulae are “FORMS” (placeholders) for whom contents are created through interpretation.

- Example:

$$\exists F[\{F(a) = b\} \wedge \forall x\{P(x) \rightarrow (F(x) = g(x, F(h(x))))\}]$$

- This is a Second Order Predicate Calculus formula.
- Quantification on 'F' which is a function.

Examples

- Interpretation:1

$D=N$ (natural numbers)

$a = 0$ and $b = 1$

$x \in N$

$P(x)$ stands for $x > 0$

$g(m,n)$ stands for $(m \times n)$

$h(x)$ stands for $(x - 1)$

- Above interpretation defines **Factorial**

Examples (contd.)

- Interpretation:2

$D = \{\text{strings}\}$

$a = b = \lambda$

$P(x)$ stands for “x is a non empty string”

$g(m, n)$ stands for “append head of m to n”

$h(x)$ stands for *tail*(x)

- Above interpretation defines “reversing a string”

Other examples

$$\exists P[\forall x \exists y P(x, y) \wedge \forall x \neg P(x, x) \wedge \forall x \forall y \forall z [(P(x, y) \wedge P(y, z)) \Rightarrow P(x, z)]]$$

P is greater than function this is a possible interpretation : 1. For all x there exists a y which is smaller than x and 2. for all x x is not greater than x and for all x y and z if x > y and y > z then x > z

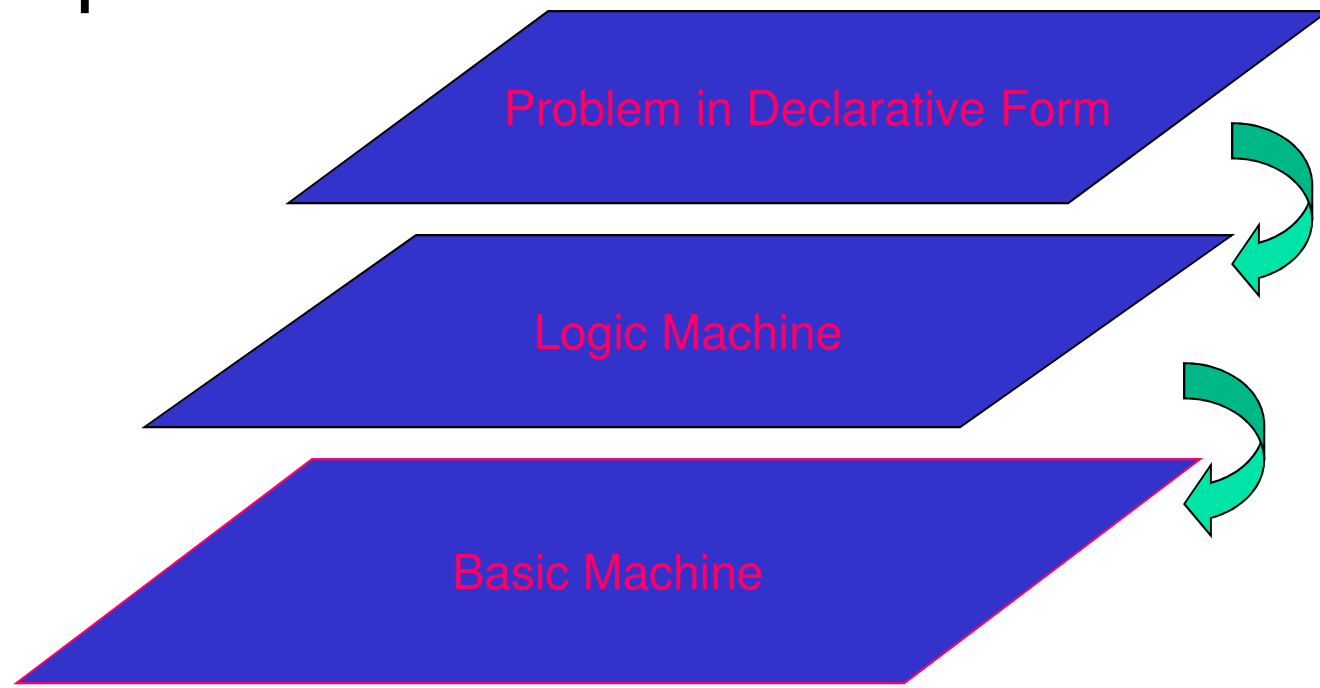
$$\forall x_1 x_2 x_3 [\{P(x_1, x_1) \wedge P(x_2, x_2) \wedge P(x_3, x_3)\} \Rightarrow \\ \{P(x_1, x_2) \vee P(x_1, x_3) \vee P(x_2, x_3)\}]$$

True in all domains of cardinality ≤ 3

Prolog

Introduction

- PROgramming in LOGic
- Emphasis on *what* rather than *how*



A Typical Prolog program

Compute_length ([],0).

Compute_length ([Head|Tail], Length):-

Compute_length (Tail,Tail_length),

Length is Tail_length+1.

High level explanation:

The length of a list is 1 plus the length of the tail of the list, obtained by removing the first element of the list.

This is a declarative description of the computation.

Fundamentals

(absolute basics for writing Prolog Programs)

Facts

- *John likes Mary*
 - *like(john,mary)*
- Names of relationship and objects must begin with a lower-case letter.
- Relationship is written *first* (typically the *predicate* of the sentence).
- *Objects* are written separated by commas and are enclosed by a pair of round brackets.
- The full stop character `.'` must come at the end of a fact.

More facts

Predicate	Interpretation
valuable(gold)	Gold is valuable.
owns(john,gold)	John owns gold.
father(john,mary)	John is the father of Mary
gives (john,book,mary)	John gives the book to Mary

Questions

- *Questions* based on facts
- Answered by *matching*

Two facts *match* if their predicates are same (spelt the same way) and the arguments each are same.

- If matched, prolog answers *yes*, else *no*.
- *No* does not mean falsity.

then what does it mean

Prolog does *theorem proving*

- When a question is asked, prolog tries to match *transitively*.
- When no match is found, answer is *no*.
- This means *not provable* from the given facts.

Not provable basically tells that the given statement is not provable from the given facts

Variables

- Always begin with a capital letter
 - *?- likes (john,X).*
 - *?- likes (john, Something).*
- But *not*
 - *?- likes (john,something)*

Example of usage of variable

Facts:

likes(john,flowers).

likes(john,mary).

likes(paul,mary).

Question:

?- likes(john,X)

Answer:

X=flowers and wait

;

mary

;

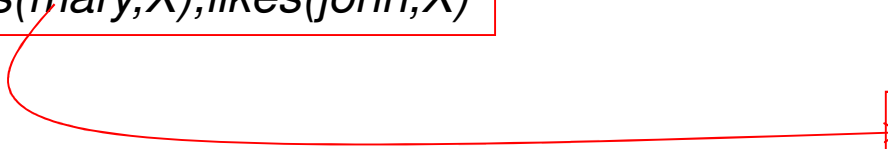
no

Conjunctions

- Use `,' and pronounce it as *and*.
- Example
 - Facts:
 - likes(mary,food).
 - likes(mary,tea).
 - likes(john,tea).
 - likes(john,mary)
 - ?-
 - likes(mary,X),likes(john,X).
 - Meaning *is anything liked by Mary also liked by John?*

Backtracking *(an inherent property of prolog programming)*

likes(mary,X),likes(john,X)



likes(mary,food)
likes(mary,tea)
likes(john,tea)
likes(john,mary)

1. First goal succeeds. *X=food*
2. Satisfy *likes(john,food)*

Backtracking (*continued*)

Returning to a marked place and trying to resatisfy is called ***Backtracking***

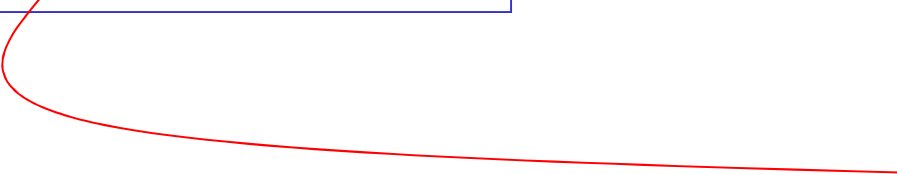
likes(mary,X),likes(john,X)

likes(mary,food)
likes(mary,tea)
likes(john,tea)
likes(john,mary)

1. Second goal fails
2. Return to marked place
and try to resatisfy the first goal

Backtracking (*continued*)

likes(mary,X),likes(john,X)



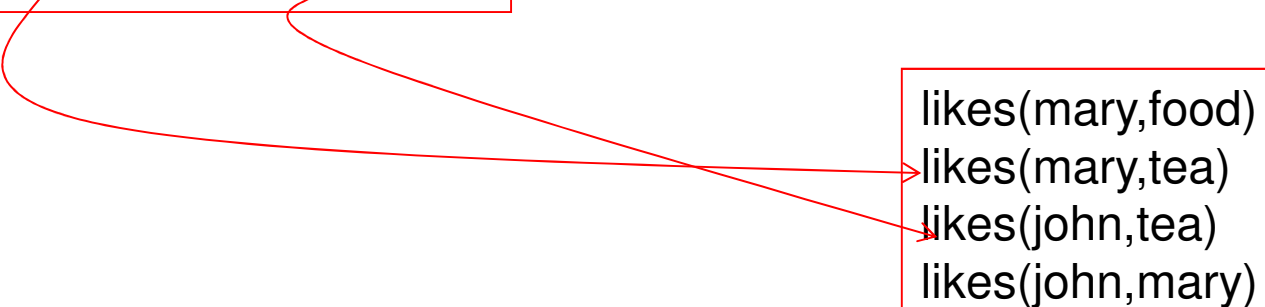
likes(mary,food)
likes(mary,tea)
likes(john,tea)
likes(john,mary)

1. First goal succeeds again, *X=tea*
2. Attempt to satisfy the *likes(john,tea)*

Backtracking *(continued)*

likes(mary,X),likes(john,X)

likes(mary,food)
likes(mary,tea)
likes(john,tea)
likes(john,mary)



The diagram illustrates the backtracking process. A box on the left contains the goal *likes(mary,X),likes(john,X)*. Two red arrows originate from this box: one points to the first fact, *likes(mary,tea)*, in a box on the right, and the other points to the second fact, *likes(john,tea)*. This represents the system finding a match for the first goal and then attempting to match the second goal against the remaining facts.

1. Second goal also succeeds
2. Prolog notifies success and waits for a reply

Rules

- Statements about *objects* and their *relationships*
- Express
 - *If-then conditions*
 - *I use an umbrella if there is a rain*
 - *use(i, umbrella) :- occur(rain).*
 - *Generalizations*
 - *All men are mortal*
 - *mortal(X) :- man(X).*
 - *Definitions*
 - *An animal is a bird if it has feathers*
 - *bird(X) :- animal(X), has_feather(X).*

pc:pushpak

Syntax

- **<head> :- <body>**
- Read **':-'** as **'if'**.
- E.G.
 - *likes(john,X) :- likes(X,cricket).*
 - *"John likes X if X likes cricket".*
 - *i.e., "John likes anyone who likes cricket".*
- Rules always end with **'.'**.

Another Example

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

X is a sister of Y is

add the condition of X being sister of herself

X is a female and

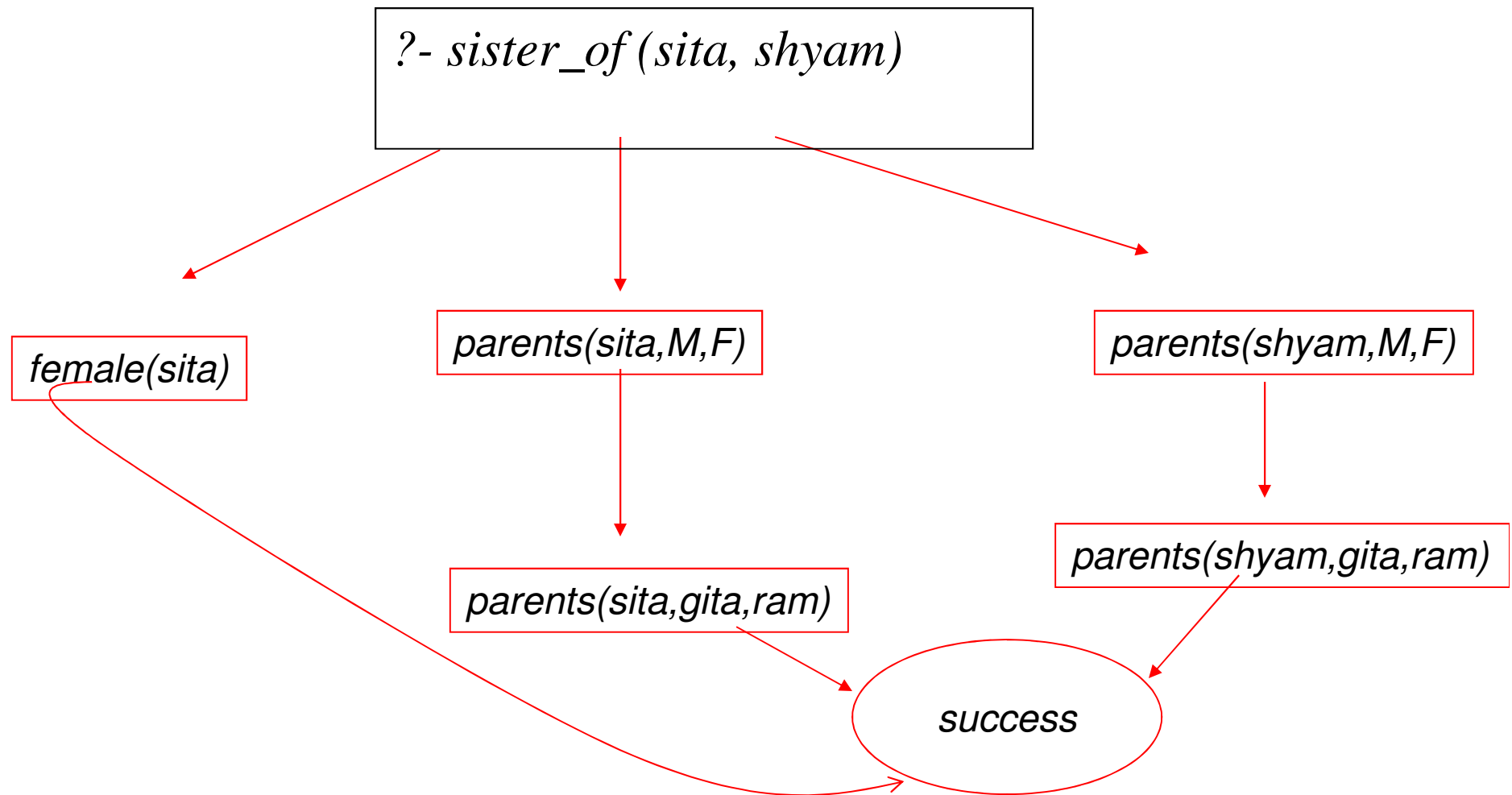
X and Y have same parents

Question Answering in presence of *rules*

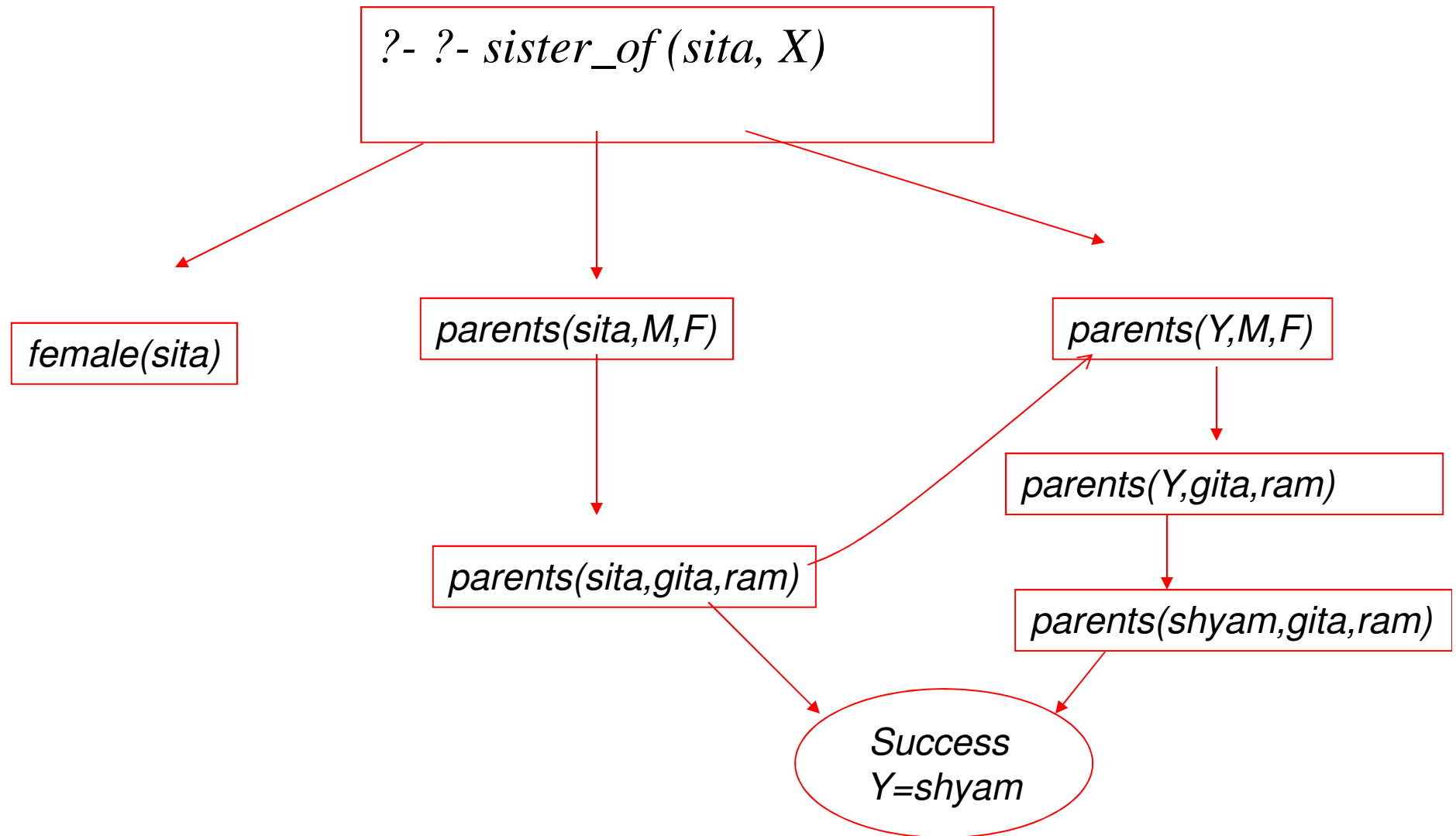
■ Facts

- male (ram).
- male (shyam).
- female (sita).
- female (gita).
- parents (shyam, gita, ram).
- parents (sita, gita, ram).

Question Answering: Y/N type: *is sita the sister of shyam?*



Question Answering: wh-type: *whose sister is sita?*



Rules

- Statements about *objects* and their *relationships*
- Express
 - *If-then conditions*
 - *I use an umbrella if there is a rain*
 - *use(i, umbrella) :- occur(rain).*
 - *Generalizations*
 - *All men are mortal*
 - *mortal(X) :- man(X).*
 - *Definitions*
 - *An animal is a bird if it has feathers*
 - *bird(X) :- animal(X), has_feather(X).*

Make and Break

Fundamental to Prolog

Prolog examples using making and breaking lists

```
%incrementing the elements of a list to produce another list  
incr1([],[]).  
incr1([H1|T1],[H2|T2]) :- H2 is H1+1, incr1(T1,T2).
```

```
%appending two lists; (append(L1,L2,L3) is a built in  
function in Prolog)  
append1([],L,L).  
append1([H|L1],L2,[H|L3]):- append1(L1,L2,L3).
```

```
%reverse of a list (reverse(L1,L2) is a built in function  
reverse1([],[]).  
reverse1([H|T],L):- reverse1(T,L1),append1(L1,[H],L).
```

Remove duplicates

Problem: to remove duplicates from a list

1. `rem_dup([],[]).`
2. `rem_dup([H|T],L) :- member(H,T), !, rem_dup(T,L).`
3. `rem_dup([H|T],[H|L1]) :- rem_dup(T,L1).`

Note: The cut ! in the second clause needed, since after succeeding at `member(H,T)`, the expression no. 3 clause should not be tried even if `rem_dup(T,L)` fails, which prolog will otherwise do.

Member (membership in a list)

`member(X,[X|_]).`

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

Union (lists contain unique elements)

```
union([],Z,Z).
```

```
union([X|Y],Z,W):-  
    member(X,Z),!,union(Y,Z,W).
```

```
union([X|Y],Z,[X|W]):- union(Y,Z,W).
```

Intersection (lists contain unique elements)

```
intersection([],Z,[]).
```

```
intersection([X|Y],Z,[X|W]):-  
    member(X,Z),!,intersection(Y,Z,W).
```

```
intersection([X|Y],Z,W):-  
    intersection(Y,Z,W).
```

XOR

%xor of two lists

xor(L1,L2,L3):-

diff1(L1,L2,X),diff1(L2,L1,Y),append(X,Y,L3).

%diff(P,Q,R) returns true if R is P-Q

diff1([],Q,[]).

diff1([H|T],Q,R):- member(H,Q),!,diff1(T,Q,R).

diff1([H|T1],Q,[H|T2]):- diff1(T1,Q,T2).

Prolog Programs are close to Natural Language

Important Prolog Predicate:

member(e, L) / true if e is an element of list L*

member(e,[e/L1). / e is member of any list which it starts*

*member(e,[_ /L1]):- member(e,L1) /*otherwise e is member of a list if the tail of the list contains e*

Contrast this with:

P.T.O.

Prolog Programs are close to Natural Language, C programs are not

```
For (i=0;i<length(L);i++){  
    if (e==a[i])  
        break(); /*e found in a[]  
}  
If (i<length(L){  
    success(e,a); /*print location where e appears in  
        a[]/*  
else  
    failure();  
}
```

What is *i* doing here? Is it natural to our thinking?

Machine should ascend to the level of man

- A prolog program is an example of reduced man-machine gap, unlike a C program
- That said, a very large number of programs far outnumbering prolog programs gets written in C
- The demand of practicality many times incompatible with the elegance of ideality
- But the ideal should nevertheless be striven for

Prolog Program Flow, BackTracking and Cut

Controlling the program flow

Prolog's computation

- **Depth First Search**
 - Pursues a goal till the end
- **Conditional AND; *falsity* of any goal prevents satisfaction of further clauses.**
- **Conditional OR; *satisfaction* of any goal prevents further clauses being evaluated.**

Control flow (top level)

Given

$g:- a, b, c.$ (1)

$g:- d, e, f; p.$ (2)

If prolog cannot satisfy (1), control will automatically fall through to (2).

Control Flow within a rule

Taking (1),

$g:- a, b, c.$

If a succeeds, prolog will try to satisfy b , succeeding which c will be tried.

For ANDed clauses, control flows forward till the `.', iff the current clause is *true*.

For ORed clauses, control flows forward till the `.', iff the current clause evaluates to *false*.

What happens on failure

- **REDO the immediately preceding goal.**

Fundamental Principle of prolog programming

- **Always place the more general rule AFTER a specific rule.**

CUT

- **Cut tells the system that**

IF YOU HAVE COME THIS FAR

DO NOT BACKTRACK

EVEN IF YOU FAIL SUBSEQUENTLY.

**'CUT' WRITTEN AS '!' ALWAYS
SUCCEEDS.**

Fail

- This predicate always fails.
- *Cut* and *Fail* combination is used to produce negation.
- Since the LHS of the neck cannot contain any operator, $A \rightarrow \sim B$ is implemented as

$$B \text{ :- } A, \text{ !, Fail.}$$

Prolog and Himalayan Club example

- *(Zohar Manna, 1974):*
 - Problem: A, B and C belong to the Himalayan club. Every member in the club is either a mountain climber or a skier or both. A likes whatever B dislikes and dislikes whatever B likes. A likes rain and snow. No mountain climber likes rain. Every skier likes snow. *Is there a member who is a mountain climber and not a skier?*
- Given knowledge has:
 - Facts
 - Rules

A syntactically wrong prolog program!

1. belong(a).
2. belong(b).
3. belong(c).
4. mc(X);sk(X) :- belong(X) /* X is a mountain climber or skier or both if X is a member; operators NOT allowed in the head of a horn clause; hence wrong*/
5. like(X, snow) :- sk(X). /*all skiers like snow*/
6. \+like(X, rain) :- mc(X). /*no mountain climber likes rain; \+ is the not operator; negation by failure; wrong clause*/
7. \+like(a, X) :- like(b,X). /* a dislikes whatever b likes*/
8. like(a, X) :- \+like(b,X). /* a likes whatever b dislikes*/
9. like(a,rain).
10. like(a,snow).
- ?- belong(X),mc(X),\+sk(X).

Correct (?) Prolog Program

belong(a).

belong(b).

belong(c).

belong(X):-\+mc(X),\+sk(X),!,fail.

According to these two statements if X is either a mountaineer or a skier he/she will automatically become a member

belong(X).

like(a,rain).

like(a,snow).

like(a,X) :- \+ like(b,X).

Universal quantification on x over here i.e., and b have to like or dislike everything in the world
there is no neutral stand over here

like(b,X) :- like(a,X),!,fail.

like(b,X).

mc(X):-like(X,rain),!,fail.

mc(X).

sk(X):- \+like(X,snow),!,fail.

sk(X).

g(X):-belong(X),mc(X),\+sk(X),!. /*without this cut, Prolog will look for the next
answer on being given ';' and return 'c' which is wrong*/

Himalayan club problem: working version

```
belong(a).  
belong(b).  
belong(c).
```

```
belong(X):-notmc(X),notsk(X),!, fail. /*contraposition to have horn clause  
belong(X).
```

```
like(a,rain).  
like(a,snow).  
like(a,X) :- dislike(b,X).  
like(b,X) :- like(a,X),!,fail.  
like(b,X).
```

```
mc(X):-like(X,rain),!,fail.  
mc(X).  
notsk(X):- dislike(X,snow). /*contraposition to have horn clause  
notmc(X):- mc(X),!,fail.  
notmc(X).
```

This makes the assumption that if the person doesnt like rain he is a mountain climber

```
dislike(P,Q):- like(P,Q),!,fail.  
dislike(P,Q).
```

```
g(X):-belong(X),mc(X),notsk(X),!.
```

pc:pushpak

Circuit verification

Circuit Verification

- Does the circuit meet the specs?
- Are there faults?
- are they locatable?

Example : 2-bit full adder

C1	X2	X1	Y	C2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

X_1 , X_2 : inputs; C_1 : prev. carry; C_2 : next carry; Y: output

K-Map

		Y			
C1	X2X1	00	01	11	10
	0	0	1	0	1
	1	1	0	1	0

$$\begin{aligned}
 Y &= C1(\overline{X1 \oplus X2}) + \overline{C1}(X1 \oplus X2) \\
 &= (C1 \oplus (X1 \oplus X2))
 \end{aligned}$$

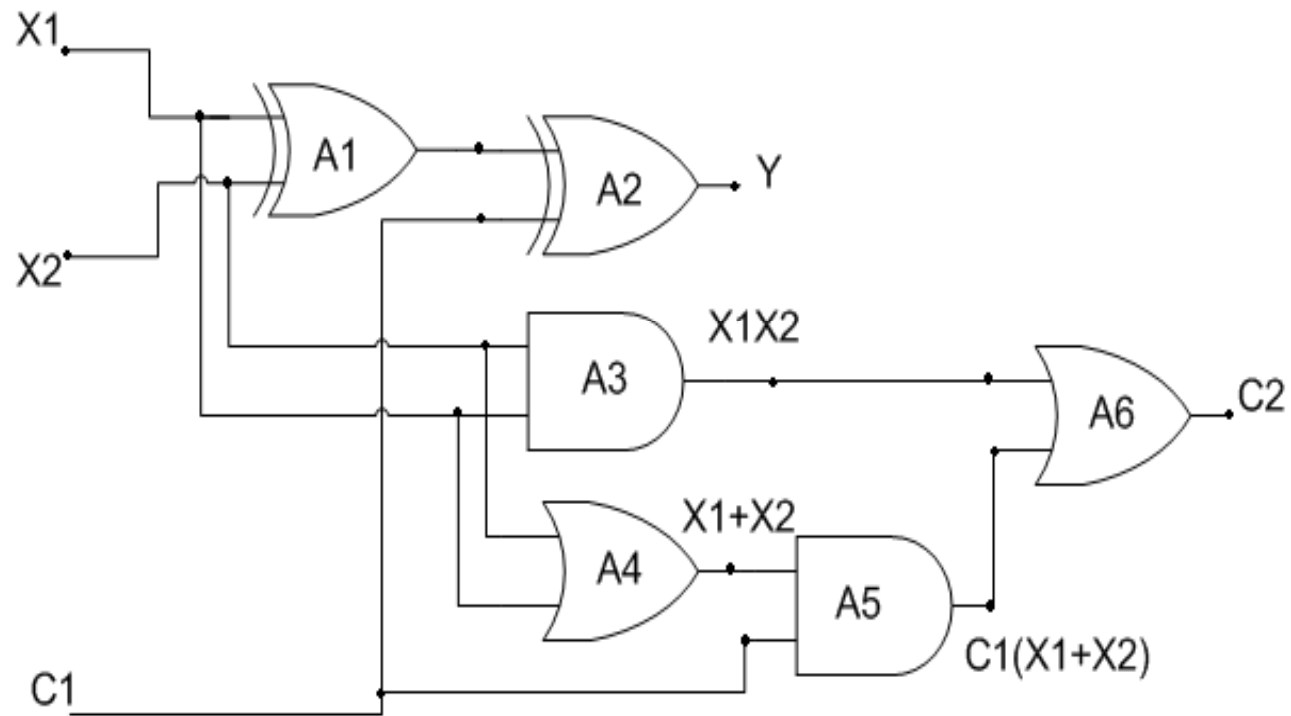
K-Map (contd..)

C_2

$\begin{array}{c} \diagup \\ C_1 \backslash X_2 X_1 \end{array}$		C_2			
		00	01	11	10
0	0	0	0	1	0
1	0	1	1	1	1

$$C_2 = X_2 X_1 + C_1(X_1 + X_2)$$

Circuit



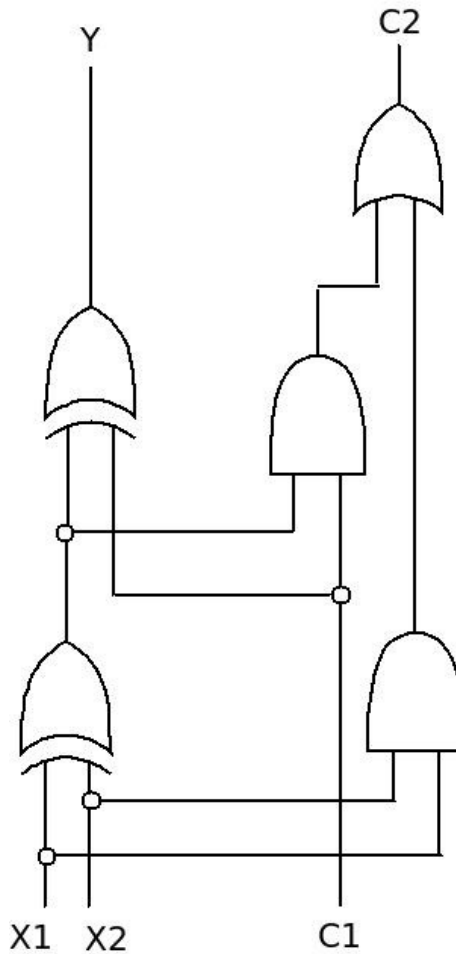
Verification

- First task (most difficult)
 - Building blocks : predicates
 - Circuit observation : Assertion on terminals

Predicates & Functions

Function-1	signal(t)	t is a terminal ; signal takes the value 0 or 1
Function-2	type(x)	x is a circuit element; type(x) takes the value AND, OR, NOT, XOR
Predicate - 3	connected(t1,t2)	t1 is an output terminal and t2 is an input terminal
Function-3	In(n,x)	n th input of ckt element x
Function-4	Out(x)	Output of ckt element x

Alternate Full Adder Circuit



Functions

- $\text{type}(X)$: takes values AND, OR NOT and XOR, where X is a gate.
- $\text{in}(n, X)$: the value of signal at the n^{th} input of gate X .
- $\text{out}(X)$: output of gate X .
- $\text{signal}(t)$: state at terminal $t = 1/0$

Predicates

- $\text{connected}(t1, t2)$: true, if terminal $t1$ and $t2$ are connected

General Properties

- Commutativity:

$$\forall t_1, t_2 [\text{connected}(t_1, t_2) \rightarrow \text{connected}(t_2, t_1)]$$

- By definition of connection:

$$\forall t_1, t_2 [\text{connected}(t_1, t_2) \rightarrow \{ \text{signal}(t_1) = \text{signal}(t_2) \}]$$

Gate properties

1. OR definition:

$$\forall X [\{\text{type}(X) = \text{OR}\} \equiv \\ \{(\text{out}(X) = 1) \equiv \exists y (\text{in}(y, X) = 1)\}]$$

2. AND definition:

$$\forall X [\{\text{type}(X) = \text{AND}\} \equiv \\ \{(\text{out}(X) = 1) \equiv \forall y (\text{in}(y, X) = 1)\}]$$

Gate properties contd...

1. XOR definition:

$$\forall X [\{\text{type}(X) = \text{XOR}\} \equiv \\ \{(\text{out}(X) = 1) \equiv (\text{in}(1, X) \neq \text{in}(2, X))\}]$$

2. NOT definition:

$$\forall X [\{\text{type}(X) = \text{NOT}\} \equiv \\ \{\text{out}(X) \neq \text{in}(1, X)\} \wedge (\text{no_of_input}(X) = 1)]$$

Some necessary functions

- a. no_of_input(x), takes integer values
- b. Count_ls(x), returns *no. of 1s* in the input of X

$$\forall X [\{\text{type}(X) = \text{XOR}\} \equiv$$

$$\{(\text{out}(X) = 1) \equiv \text{odd}((\text{count_ls}(X))\}]$$

Circuit specific properties

- Connectivity:

`connected(x1, in(1,A1))`

`connected(x1, in(2, A1))`

`connected(out(A1), in(1, A2))`

`connected(c1, in(2, A2))`

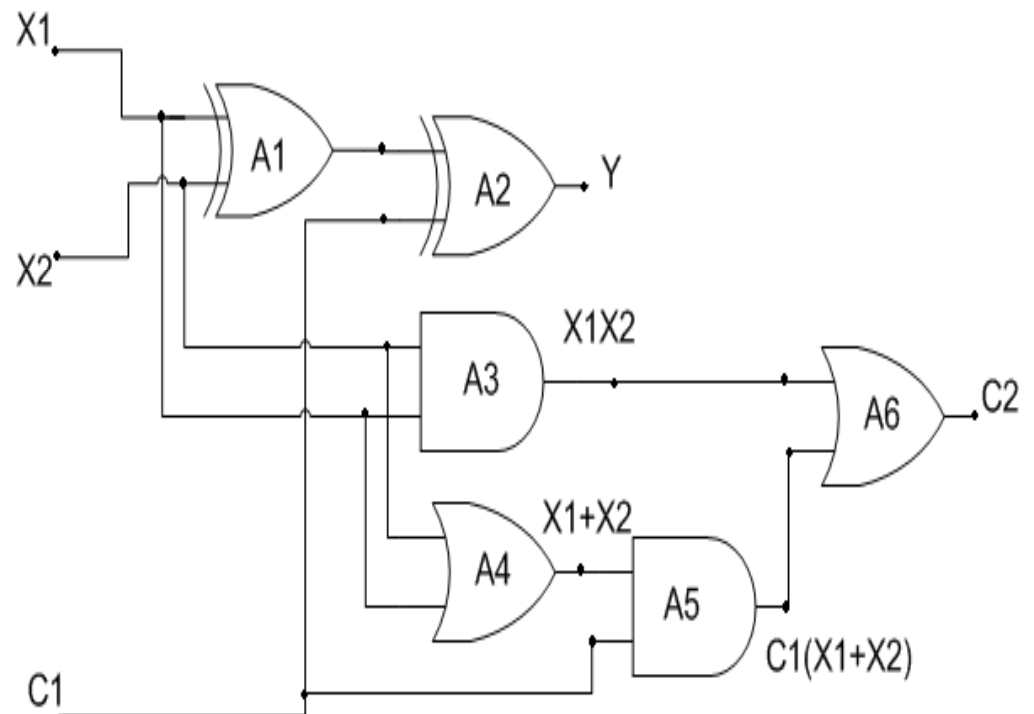
`connected(y, out(A2)) ...`

- Circuit elements:

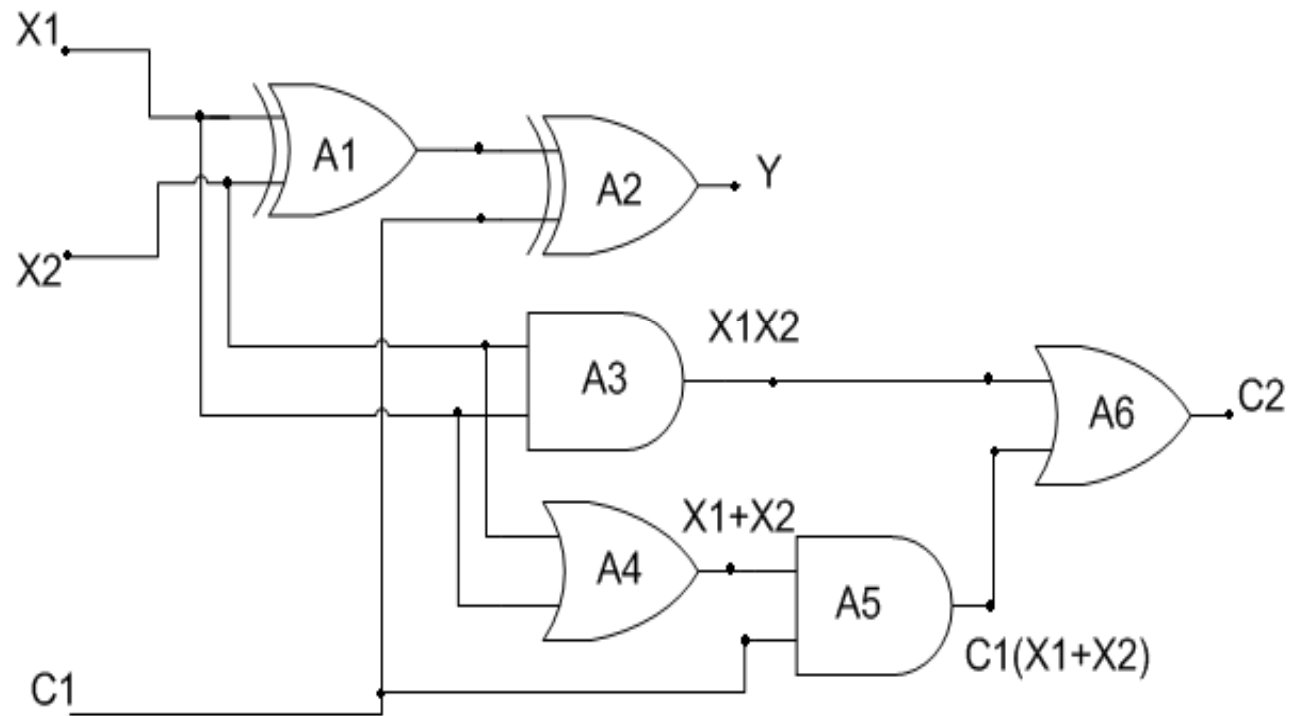
`type(A1) = XOR,`

`type(A2) = XOR,`

`type(A3) = AND ...`



Circuit



Models human reasoning (1/2)

■ Non-numerical

■ Non monotonic Logic

- Negation by failure ("*innocent unless proven guilty*")
- Abduction ($P \rightarrow Q$ AND Q gives P)

■ Modal Logic

- New operators beyond AND, OR, IMPLIES, Quantification etc.

■ Naïve Physics

Abduction Example

- **If**

there is rain (P)

- **Then**

there will be no picnic (Q)

- **Abductive reasoning:**

Observation: There was no picnic(Q)

Conclude : There was rain(P); *in absence of any other evidence*

Alternatives to fuzzy logic model human reasoning (2/2)

■ Numerical

- Fuzzy Logic
- Probability Theory
 - Bayesian Decision Theory
- Possibility Theory
- Uncertainty Factor based on Dempster Shafer Evidence Theory (e.g. *yellow_eyes* → *jaundice*; 0.3)