University of Technology
الجامعة التكنولوجية

Computer Science Department
قسم علوم الحاسوب

Knowledge Representation

تمثيل المعرفة

Assist. Prof. Dr. Suhad Malallah Kadhem
أ.م.د. سهاد مال الله كاظم

cs.uotechnology.edu.iq

Introduction to prolog language

Prolog (**pro**gramming in **log**ic) is one of the most widely used programming languages in artificial intelligence research.

Programming languages are of two kinds:

– **Procedural** (BASIC, Fortran, C++, Pascal, Java).

– **Declarative** (LISP, Prolog, ML).

In **procedural programming**, we tell the computer **how** to solve a problem, but in **declarative programming**, we tell the computer **what** problem we want to solved.

Prolog: is a computer programming language that is used for solving problems involves objects and relationships between objects.

Example:

"John owns the book"

Owns (john,book) relationship(object1,object2)

The relationship has a specific order, johns own the book, but the book dose not owns john, and this relationship and its representation above called fact.

What is Prolog used for?

Prolog good at:

- Grammars and Language processing.
- Knowledge representation and reasoning.
- Unification.
- Pattern matching.
- Planning and Search.

Some of prolog language characteristics:

1. We can solve a particular problem using prolog in less no of lines of code.
2. It's an important tool to develop AI application and ES.
3. Prolog program consist of fact and rule to solve the problem and the

output is all possible answer to the problem.

4. Prolog language is a descriptive language use the inference depend on facts and rules we submit to get all possible answer while in other language the programmer must tell the computer on how to reach the solution by gives the instruction step by step.

Component of computer programming in prolog

Computer programming in prolog consist of:

1. Declaring some facts about object and their relationships.
2. Declaring some rules about objects and their relationships.
3. Asking questions about objects and their relationships.

We can consider prolog as a store house of facts and rules, and it uses the facts and rules to answer questions.

Prolog is a conversational language, which means you and the computer carry out a kind of conversation.

Basic Elements of Prolog

There are only three basic constructs in Prolog: **Facts**, **Rules**, and **Queries**. A collection of facts and rules is called a knowledge base (or a database) Prolog programs simply are knowledge bases, collections of facts and rules which describe some collection of relationships that we find interesting.

• *Some are always true (facts):*

father(john, jim).

• *Some are dependent on others being true (rules):*

parent(Person1, Person2) :- father(Person1, Person2).

• To run a program, we ask questions about the database.

Facts

Is the mechanism for representing knowledge in the program.

Syntax of fact:

1. The name of all relationship and objects must begin with a lower-case letter,

for example likes (john, mary).

2. The relationship is written first, and the objects are written separated by commas, and enclosed by a pair of round brackets.

3. The full stop character '.' Must come at the end of fact.

Example:

Gold is valuable	valuable (gold).
Jane is female	female (jane).
John owns gold	owns (johns, gold).
Johns is the father of Mary	father (john, marry).

The names of objects that are enclosed within the round brackets are called arguments. And the name of relationship called predicates. Relationship has arbitrary number of argument. If we want to define predicate called play, were we men~~tion two players~~ a~~nd a game t~~hey play
with each other, it can be: Play (john, Mary, football).
In prolog the collection of facts is called database.

Rules

Rules are used when you want to say that a fact depends on a group of other facts, and we use the following syntax:

1. One fact represents the head (conclusion).
2. The word if used after the head and represented as ":-'.
3. One or more fact represents the requirement (condition).

The syntax of if statement

If (condition) then (conclusion)

Conclusion: - conditions

Example:

I use the umbrella if there is rain

Conclusion condition

Represent both as fact like:

weather (rain).

use (i,umbrella)

use (i, umberella):-whether (rain).

Example of Rules:

 Person1 is a parent of Person2 **if** Person1 is the father of Person2 **or** Person1 is the mother of Person2.

parent(Person1,Person2):- father(Person1,Person2).

parent(Person1,Person2):- mother(Person1,Person2).

 Person1 is a grandparent of Person2 **if** some Person3 is a parent of Person2 **and** Person1 is a parent of Person3.

grandparent(Person1,Person2):- parent(Person3,Person2),

parent(Person1,Person3).

Questions

Question used to ask about facts and rules.

Question looks like the fact and written under the goal program section, while fact and rule written under clauses section.

Example: for the following fact:

owns (mary , book).

We can ask: does Mary own the book in the following manner:

Goal

owns (mary,book).

When Question is asked in prolog, it will search through the database you typed before; it looks for facts that match the fact in the question. *Two facts matches if their predicates are the same and their corresponding arguments are the same*, if prolog finds a fact that matches the question, prolog will respond with Yes, otherwise the answer is No.

Example questions:

• Who is Jim's father? ?- father(Who, jim).

• Is Jane the mother of Fred? ?- mother(jane, fred).

• Is Jane the mother of Jim? ?- mother(jane, jim).

• Does Jack have a grand parant? ?- grandparent(_, jack).

Variables

If we want to get more interest information about fact or rule, we can use variable to get more than Yes/No answer.

- Variables dose not name a particular object but stand for object that we cannot name.
- Variable name must begin with capital letter.
- Using variable we can get all possible answer about a particular fact or rule.
- Variable can be either bound or not bound.

Variable is bound when there is an object that the variable stands for.
The variable is not bound when what the variable stand for is not yet known.

Example:

Facts

*L*ike (john, mary).

Like (john, flower).

Like (ali, mary).

Question:

Like (john,X).

X= mary

X = flower

like(X, mary)

X=john

X=ali

Like(X, Y)

X=john Y=flower

X=john Y=mary

X=ali Y=mary

Type of questing in the goal

There are three type of question in the goal summarized as follow:

1.Asking with constant: prolog matching and return Yes/No answer.

2.Asking with constant and variable: prolog matching and produce result for the Variable.

3.Asking with variable: prolog produce result.

Example:

Age(a,10).

Age(b,20).

Age(c,30).

Goal:

1.Age(a,X). ans:X=10 *Type2*

2.age(X,20). Ans:X=b *Type2*

3.age(X,Y). ans: X=a Y=10, X=b Y=20, X=c Y=30. *Type3*

4.Age(_,X). ans:X=10 , X=20, X=30. '_' means don't care *Type3*

5.Age(_,_). Ans:Yes *Type1*

H.W:

Convert the following paragraphs into facts or rules:

1.a person may steal something if the person is a thief and he likes the thing and the thing is valuable.

2.Bob likes all kind of game. Football is a game. Anything anyone plays and not killed by is a game.

Data Type

Prolog supports the following data type to define program entries:

1. **integer:** to define numerical value like 1, 20, 0,-3,-50, …ect.
2. **real:** to define the decimal value like 2.4, 3.0, 5,-2.67, …ect.

3. **char:** to define single character, the character can be of type small letter or capital letter or even of type integer under one condition it must be surrounded by single quota. For example, ‘a’,’C’,’1’,’3’,...etc.
4. **string :** to define a sequence of character like “good” i.e define word or statement entries the string must be surrounded by double quota for example “computer”, “134”, “a”. The string can be of any length and type.
5. **symbol:** another type of data type to define single character or sequence of character but it must begin with small letter and don’t surround with single quota or double quota.

Program Structure

Prolog program structure consists of five segments, not all of them must appear in each program. The following segment must be included in each program predicates, clauses, and goal.

1. **Domains:** define global parameter used in the program.

```
Domains
    I= integer
    C= char
    S = string
    R = real
```

2. **Data base:** define internal data base generated by the program

```
Database
    greater (integer)
```

3. **Predicates:** define rule and fact used in the program.

```
Predicates
    mark(symbol,integer).
```

4. **Clauses:** define the body of the program.. For the above predicates the clauses portion may contain mark (a, 20).

5.**Goal:** can be internal or external, internal goal written after clauses portion , external goal supported by the prolog compiler if the program syntax is correct. This portion contains the rule that drive the program execution.

Mathematical and logical operation

a .mathematical operation:

Operation	symbol
Addition	+
Subtraction	-
Multiplication	*
Integer part of division	div
Remainder of division	mod

We can make compound sums using round brackets

for example X = (5+4)*2 then X=18.

B .logical operation

Operation	symbol
Greater	>
Less than	<
Equal	=
Not equal	<>
Greater or equal	>=
Less than or equal	<=

Other mathematical function

Function name	operation
Cos(X)	**Return the cosine of its argument**
Sine(X)	**Return the sine of its argument**

Tan(X)	**Return the tan of its argument**
Exp(X)	**Return exp raised to the value to which X is bound**
Ln(X)	**Return the natural logarithm of X (base e)**
Log(X)	**Return the base 10 logarithm of log 10^x**
Sqrt(X)	**Return the positive square of X**
Round(X)	**Return the rounded value of X. Rounds X up or down to the nearest integer**
Trunc(X)	**Truncates X to the right of the decimal point**
Abs(X)	**Return the absolute value of X**

Tests within clauses

These operators can be used within the body of a clause to manipulate values:

sum(X,Y,Sum):- Sum = X+Y.

Goal: sum(3,5,S)

Output: S=8

sum(X,Y):-Sum=X+Y, write(Sum).

Goal: sum(3,5)

Output: 8

We can write the rule without arguments for example:

Sum:-readint(X),readint(Y),Sum=X+Y, write(Sum).

Goal: sum.

Output: 8

Also, these operators can be used to distinguish between clauses of a predicate definition:

bigger(N,M):- N < M, write('The bigger number is '), write(M).

bigger(N,M):- N > M, write('The bigger number is '), write(N).

bigger(N,M):- N = M, write('Numbers are the same').

Goal: bigger(6,7)

Output: The bigger number is 7

Example1: Write a prolog program to check if the given number is positive or negative.

Basic rule to check the number

If X>=0 then X is positive

Else X is negative

Domains
I= integer
Predicates
pos_neg(i)
Clauses
pos_neg(X):-X>=0, write("positive number"),nl.
pos_neg(_):-write("negative number"),nl.
Goal
pos_neg(4).
Output:
positive number
Note: nl mean new line.
Example2: write a prolog program to check if a given number is odd or even.
Basic rule to check number
If X mod 2=0 then X is even number
Else X is odd number
Predicates
odd_even(integer)
Clauses
odd_even(X):-X mod 2= 0, write ("even number"), nl.
odd_even(X):- X mod 2 <> 0, write ("odd number"), nl.
Goal
odd_even(5).
Output:
odd number

Example3: Write a prolog program to combine both rules in example1 and example2.

```
Domains
I= integer
Predicates
pos_neg(i)
odd_even(i)
oe_pn(i)
Clauses
oe_pn(X):-pos_neg(X),odd_even(X).
odd_even(X):-X mod 2= 0, write(" even number"),nl.
odd_even(X):- write("odd number"),nl.
pos_neg(X):-X>=0, write("positive number"),nl.
pos_neg(_ ):-write("negative number"),nl.
Goal
```

oe_pn(3).

Output:

odd number

positive number

Note: the rule of same type must be gathering with each other.

Example4 : Write a prolog program to describe the behavior of the logical And gate.

Truth table of And gate

X	Y	Z
0	0	0
1	0	0
0	1	0
1	1	1

Domains

I= integer

Predicates

and1(I, I , I)

Clauses

and1(0,0,0).

and1(0,1,0).

and1(1,0,0).

and1(1,1,1).

Goal

and1 (0,1,Z).

Output:

Z =0

Sol 2:

From the truth table we can infer the following rule:

If X= Y then

Z= X

Else

Z =0

Domains

I= integer

Predicates

and1 (I ,I, I)

Clauses

and1 (X,Y,Z):- X=Y, Z=X.

and1(X,Y,Z):- X<> Y, Z=0.

Goal

and1(0,0,Z).

Output

Z=0

H.W

1. Write prolog program that read character and check if it's a capital letter, small letter, digit or special character.

2. Modify prolog program in example 3 such that the value of X is read inside the program.

3. Write prolog program that describe the operation of logical Or gate.

Read and write function

Read function:

readint(Var) : read integer variable.

readchar(Var) : read character variable.

readreal(Var) : read read (decimal) variable.

readln(Var) : read string.

readterm(data type,Var): read any specified data type.

Write function

Write(Var) : write variable of any type.

Example 1: write prolog program to read integer value and print it.

```
Domains
    I = integer
Predicates
   print.
Clauses
      Print:- write ("please read integer number"), readint(X),
              write("you read",X).
Goal
   Print.
```

Output:

Please read integer number 4

You read 4

Example2: write a prolog program that take two integers as input and print the greater one.

```
Domains
    i = integer
Predicates
   greater ( i,i)
Clauses
   greater(X,Y):- X>Y,write("the greater is",X).
   Greater(X,Y):- Y>X, write (" the greater is ",Y).
Goal
```

Greater(4,3).

Output:
The greater is 4

H.W:
1. write a prolog program that read any phrase then print it.
2.write a prolog program that read an integer number then print it after multiplying it by any other integer like 5.

Conjunctions

1. Conjunctions
    1. and ‘,’.
    2. or ‘;’.

Used to combine facts in the rule , or to combine fact in the goal to answer questions about more complicated relationship.

Example:

Facts

Like (mary,food).

Like(mary,wine).

Like(john,mary).

Goal

Like(mary,john),like(john,mary).

We can ask does mary like john and does john like mary?

Now, how would prolog answer this complicated question?

Prolog answers the question by attempting to satisfy the first goal. if the first goal is in the database, then prolog will mark the place in the database, and attempt to satisfy the second goal. If the second goal is satisfied, then prolog marks that goal‘s place in the database, and we have a solution that satisfy both goals. It is important to remember that each goal keeps its own place marker. If, however, the second goals are not satisfied, then prolog will attempt to re-satisfy the previous goal.

Prolog searches the database in case it has to re-satisfy the goal at a later time. But when a goal needs to be re-satisfied, prolog will begin the search database completely for each goal. If a fact in the database happens to match , satisfying the goal, then prolog will mark the place in the database in case it has to re-satisfy the goal at the later time. But when a goal needs to be re-satisfied, prolog will begin the search from the goal's own place marker, rather than from the start of database and this behavior called "backtracking".

Example: about backtracking

Like(mary,food).

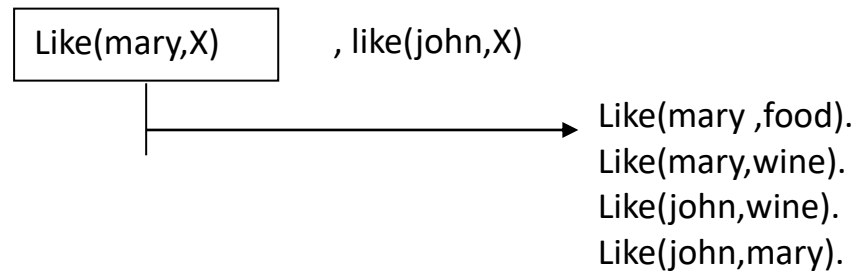
Like(mary,wine).

Like(john,wine).

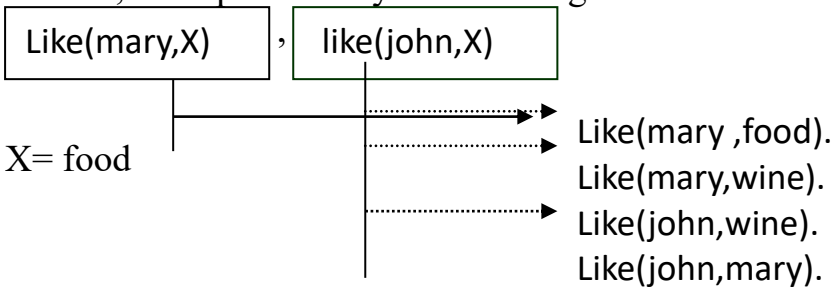
Like(john,mary).

Goal

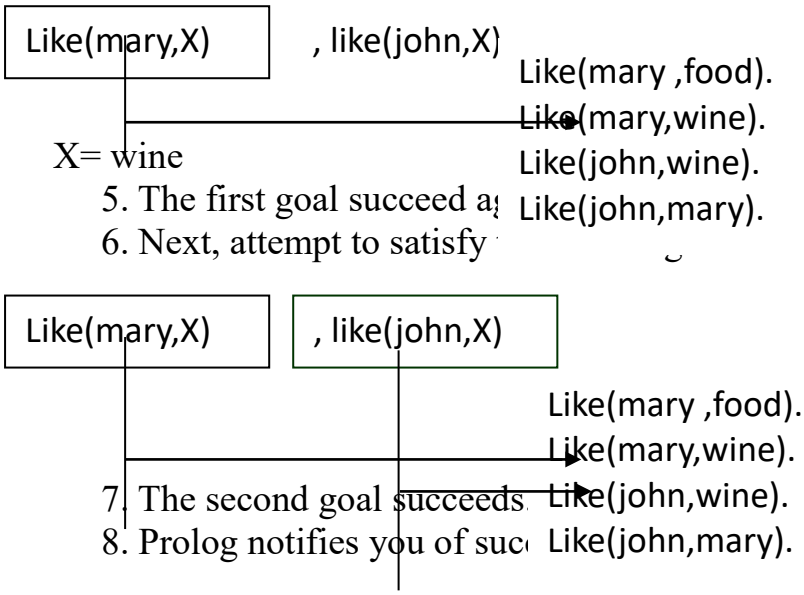
Like(mary,X),like(john,X).



1. The first goal succeed, bound X to food.
2. Next, attempt to satisfy the second goal.



3. The second goal fails.
4. Next, backtrack: forget previous value of X and attempt to resatisfy the first goal.



H.W

Trace the following goal to find the value of X,Y,W,Z.

```
Mark(a,10).
Mark(b,20).
Mark(c,30).
```

Goal

```
Mark(X,Y),Mark(W,Z).
```

1. Repetition

In prolog there is a constant formula to generate repetition; this technique can generate repetition for some operation until the stopping condition become true.

Example: Prolog program read and write a number of characters continue until the input character equal to '#'.

Predicates

```
repeat.
typewriter.
```

Clauses

```
repeat.
repeat:-repeat.
typewriter:-repeat,readchar(C),write(C ),nl,C='#',!.
```

2.Recursion

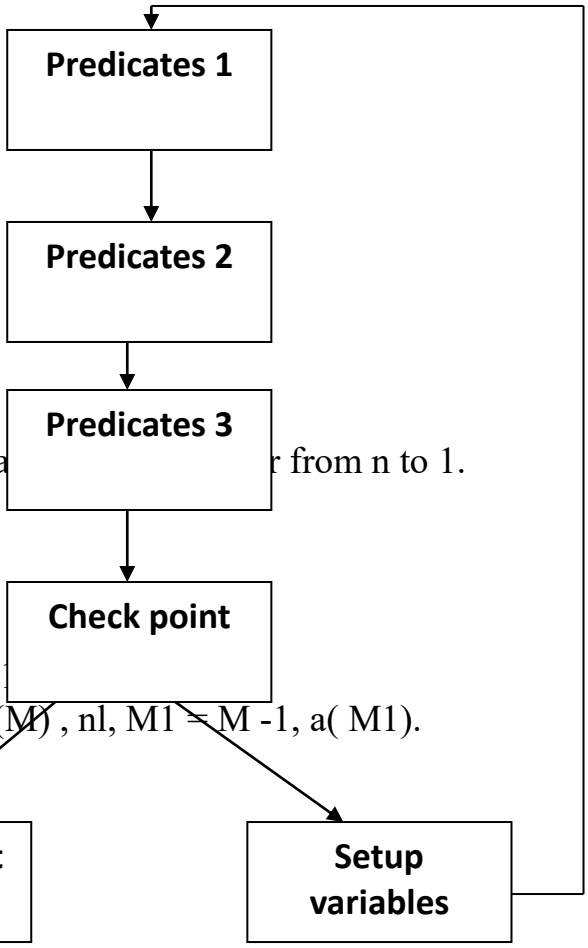
In addition to have rules that use other rules as part of their requirements, we can have rules that use themselves as part of their requirements.

This kind of rule called "recursive "because the relationship in the conclusion appears again in the body of the rule, where the requirements are specified. A recursive rule is a way of generating a chain of relationship for a recursive rule to be effective. However, there must be some place in the chain of relationship where the recursion stops. This stopping condition must be answerable in the database like any other rule.

2.1 Tail Recursion

We place the predicate that cause the recursion in the tail of the rule as shown below:

Head :- p1,p2,p3, head.



Example1: progra r from n to 1.

Predicates
a (integer)
Clauses
a(1) :- write (
a(M):- write (M) , nl, M1 = M -1, a(M1).
Goal
a(4).

O
4
3
2
1
Yes

Example 2: program to find factorial.
5! = 5*4*3*2*1

```
Predicates
    fact ( integer, integer, integer)
Clauses
   fact(1, R, R):-!.
   fact(X,F1,R):- F2=F1*X , X1=X-1, fact(X1,F2,R).
Goal
fact (5,1,R).
```

Output:
F = 120.

Example 3: program to find power .
$3^4 = 3*3*3*3$

```
Domains
   I= integer
Predicates
  power ( I,I,I, I ).
Clauses
power (_,0,R,R):-!.
  power ( X,Y,R1,R):-  R2= R1*X, Y1 =Y-1, power(X,Y1,R2,R).

Goal
power(3,2,1,R).
```

Output R= 9

2.2 Non –Tail Recursion (Stack Recursion)

This type of recursion us the stack to hold the value of the variables till the recursion is complete. The statement is self – repeated as many times as the number of items in the stack.. Below a simple comparison between tail and non-tail recursion.

Tail recursion	Non-tail recursion
1. Call for rule place in the end of the rule. 2. It is not fast as much as stack recursion. 3. Use more variable than stack recursion.	1. Call for the rule place in the middle in the rule. 2. Stack recursion is fast to implement. 3. Few parameters are used.

Example 4: factorial program using non-tail recursion.

Predicates

fact(integer,integer).

Clauses

fact(1,1).

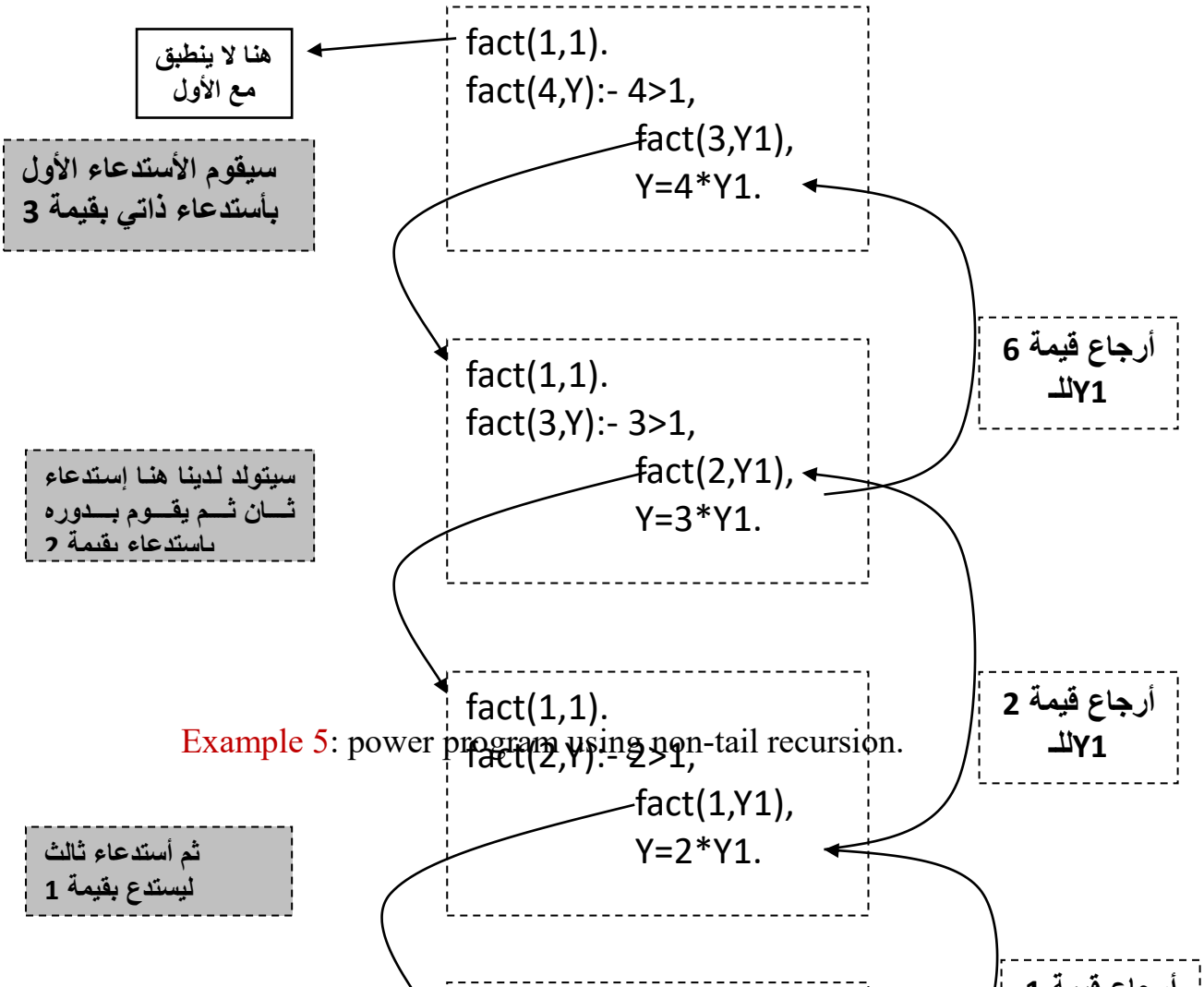
fact(X,R):- X>1,X1=X-1,fact(X1,R1),R=X*R1.

Goal

fact (4,R).

Output:

Y =24.



Example 5: power program using non-tail recursion.

```
Predicates
  power (integer, integer, integer)
 Clauses
   power (_,0,1):-!.
   power ( X,Y,R) :- Y> 0, Y1=Y -1, power (X,Y1,R1),R= X*R1.
Goal
   power (3,2,R).
Output
    Z = 9.
```

H.W

1. *Write prolog program to find the sum of 10 integer elements using tail and non tail recursion.*
2. *Write prolog program to find the maximum value between 10 elements.*
3. *Write prolog program to find the minimum value between 10 elements.*
4. *Find the sum S = 1+2 + 3 ….+N*

Pattern Matching

Prolog uses *unification* to match variables to values. An expression that contains variables like *X+Y*Z* describes a pattern where there are three blank spaces to fill in named X, Y, and Z. The expression *1+2*3* has the same structure (pattern) but no variables. If we input this query X+Y*Z=1+2*3, then Prolog will respond that X=1, Y=2, and Z=3. The pattern matching is very powerful because you can match variables to expressions like this X+Y=1+2*3 and get X=1 and Y=2*3. You can also match variable to variables: X+1+Y=Y+Z+2. This sets X=Y=2 and Z=1.

Example 1:

```
a(1).
a(2).
a(4).
b(2).
b(3).
d(X,Y):-a(X),b(Y), X=4.
```

Goal: d(X,Y)

X=1, Y=2, 1=4 Fail

X=1, Y=3, 1=4 Fail

X=2, Y=2, 2=4 Fail

X=2, Y=3, 2=4 Fail

X=4, Y=2, 4=4 True

X=4, Y=3, 4=4 True

Output: 2 solutions X=4, Y=2 and X=4 Y=3.

Goal: d(A,2)

X=1, Y=2, 1=4 Fail

X=2, Y=2, 2=4 Fail

X=4, Y=2, 4=4 True

Output: 1solution A=4.

Goal: d(2,2)

Output: no solution.

Backtracking

Backtracking is a systematic method to iterate through all the possible states of a search space.

Backtracking can easily be used to iterate through all subsets or permutations of a set. Backtracking ensures correctness by enumerating all possibilities.

Cut

Represented as "**!**" is a built in function always True , used to stop backtracking and can be placed anywhere in the rule, we list the cases where "!" can be inserted in the rule:

1 .R:-f1, f2,!. "f1, f2 will be deterministic to one solution.

2. R:-f1,!,f2. " f1 will be deterministic to one solution while f2 to all .

3. R:- !,f1,f2. "R will be deterministic to one solution.

Example1 : program without using cut.

```
Domains
    I= integer
Predicates
   no( I )
Clauses
  no (5).
  no (7).
  no (10).
Goal
   no (X).
```

Output:

X=5
X=7
X=10

Example 2: program using cut.

```
Domains
    I= integer
Predicates
    no( I )
Clauses
    no (5):-!.
     no (7).
    no (10).
Goal
   no (X).
```

Output:

X=5.

Example3: program without using cut.

```
Domains
     I =integer
     S = symbol
Predicates
    a (I )
```

```
    b (s )
    c ( I, s )
Clauses
    a(10).
    a(20)
    b(a)
    b(c)
     c (X,Y):- a(X), b(Y).
Goal
   c(X,Y).
```

Output:

```
     X= 10    Y=a
     X=10     Y=c

     X=20     Y=a
     X=20     Y=c
```

Example 4: using cut at the end of the rule.

```
Domains
     I =integer
     S = symbol
Predicates
    a(I )
    b (s )
   c (I, s )
Clauses
    a(10).
     a(20)
     b(a)
     b(c)
      c (X, Y):- a (X), b (Y),!.
Goal
    c(X,Y).
```

Output:

```
    X= 10    Y=a
```

Example 5: using cut at the middle of the rule.

```
Domains
    I =integer
    S = symbol
```

Predicates
a(I)
b (s)
c (I, s)
Clauses
a(10).
a(20)
b(a)
b(c)
c (X,Y):- a(X),!, b (Y).

Goal
c(X,Y).

Output:
X=10 Y=a
Y=c

Example 6:

a(1).

a(2).

a(4).

b(2).

b(3).

d(X,Y):-a(X),b(Y), X=4,!.

Goal: d(X,Y).

X=1, Y=2, 1=4 Fail

X=1, Y=3, 1=4 Fail

X=2, Y=2, 2=4 Fail

X=2, Y=3, 2=4 Fail

X=4, Y=2, 4=4 True

Output: 1 solutions X=4, Y=2.

Example 7:

a(1).

a(2).

a(4).

b(2).

b(3).

d(X,Y):-a(X),**!**,b(Y), X=4.

Goal: d(X,Y)

X=1, Y=2, 1=4 Fail

X=1, Y=3, 1=4 Fail

Output: no solutions found.

Fail

The **fail** predicate is provided by Prolog. When it is called, it causes the failure of the rule. And this will be forever; nothing can change the statement of this predicate.

Built in function written as word “fail” used to enforce backtracking, place always in the end of rule, produce false and can be used with internal goal to produce all possible solution.

Example :

```
Predicates
      Student ( symbol , integer)
       Printout.
Clauses
     Student (aymen,95).
     Student(zainab,44).
     Student(ahmed,60).

     Printout:-student(N,M),write(N,”   “,M),nl,fail.

Goal
   Printout.
```

Output:

```
      aymen   95
```

zainab 44
ahmed 60
No

Example 7:

```
Predicates
      Student ( symbol , integer)
       Printout.
Clauses
     Student (aymen,95).
      Student(zainab,44).
      Student(ahmed,60).

      Printout:-student(N,M),write(N,"  ",M),nl,fail.
       Printout.

Goal
   Printout.
```

Output:

aymen 95
zainab 44
ahmed 60
Yes

Negation

Exceptions and return false in specific situation. Can be implemented using:

1. Cut-fail.
2. Not.

1. Cut-fail

Example 8:

Ahmed likes swimming and he want to visit all middle east seas accept the dead sea. Write prolog program to describe this situation.

A: using fail.

Predicates
Visit (symbol)
Middle_east (symbol)

Clauses
Visit (Sea) :- middle_east (Sea).
Middle_east (deadsea):- fail.
Middle_east(redsea).
Middle_east(arabsea).

Goal
1. Visit (deadsea)
2. Visit (W).

Output:
1. No

2. W= redsea
W=arabsea

B: using cut- fail

Predicates
Visit (symbol)
Clauses
Visit (Sea) :- Sea=deadsea,!,fail.
Visit (X):-middle_east(X).
Middle_east(redsea).
Middle_east(arabsea).

Example 9: ban like all animals but snake, write prolog program for this case.

Predicates
Like(symbol, symbol)
Animal(symbol)

Clauses
Like(ban ,X):- animal(X),X=snake,!,fail.
Like(ban,X):- animal(X).
Animal(cat).
Animal(bird).
Animal(dog).

2. using not

For example 8: we can write it using not as follow.

```
Predicates
   Visit ( sym bol)
   Middle_east(symbol).
Clauses
      Visit ( X):- middle_east(X),not (X = deadsea).
      Middle_east( redsea).
     Middle_east(arabsea).
```

H.w:

1. Trace the following clauses and find the output:

a. clauses

```
reading:- readchar(Ch),writ(Ch),Ch='#'.
Reading.
```

b.clauses

```
Go.
Go:-go.
Reading:- go,readchar(Ch),write(Ch),Ch='#,!.
```

3. Use negation to define the different relation: diff(X,Y) which is true when X and Y are different numbers.

Example

```
predicates
a(integer)
begin
clauses
a(1).
a(2).
a(3).
a(4).
begin:-a(X),write(X),fail.
```

Goal: begin

Output: 1234 NO

clauses

a(1).

a(2).

a(3).

a(4).

begin:-a(X),write(X),fail.

begin.

Goal: begin

Output: 1234 yes

Complete Prolog Programs

domains

i=integer

predicates

counter(i)

clauses

/* counter from 1-10*/

counter(10):-!.

counter(X):-write(X), X1=X+1, counter(X1).

Goal: counter(1)

Output: 1 2 3 4 5 6 7 8 9

/*summation of 10 numbers*/

predicates

sum(integer,integer,integer)

clauses

sum(10,R,R):-!.

sum(X,R1,R):-R2=R1+X,X1=X+1,sum(X1,R2,R).

Goal

sum(1,0,R).

Output: ?

```
/* summation of 10 given integer numbers*/
domains
int=integer
predicates
sum_int(int,int,int)
clauses
sum_int(10,R,R):-!.
sum_int(X,R1,R):-readint(Z),Z>0, X1=X+1,R2=R1+Z,sum_int(X1,R2,R).
sum_int(X,R1,R):-X1=X+1,sum_int(X1,R1,R),!.
Goal
 sum_int(1,0,R).
```

Output: ?

```
/* factorial program*/
predicates
fact(integer,integer,imteger)
clauses
fact(0,_,1):-!.
fact(1,R,R):-!.
fact(X,R1,R):-R2=X*R1,X1=X-1,fact(X1,R2,R).
Goal
 fact(3,1,R).
```

Output: R=6

```
/*power program*/
predicates
power(integer,integer,integer,integer)
clauses
power(_,0,R,R):-!.
power(X,Y,R1,R):-R2=R1*X,Y1=Y-1,power(X,Y1,R2,R).
```

Goal

power(5,2,1,R).

Output: R=25

Lists in Prolog

Lists are ordered sequences of elements that can have any length. Lists can be represented as a special kind of tree. A list is either empty, or it is a structure that has two components: the head **H** and tail **T**. List notation consists of the elements of the list separated by commas, and the whole list is enclosed in square brackets.

Lists correspond roughly to array in other languages but unlike array, a list does not require you to how big it will be before use it.

syntax of list

List always defined in the domains section of the program as follow:

Domains

list = integer*

- '*' refer to list object which can be of length zero or un defined.
- The type of element list can be of any standard defined data type like integer, char … ect or user defined data type explained later.
- List element surrounded with square brackets and separated by comma as follow: l = [1, 2, 3, 4].
- List consist of two parts head and tail , the head represent the first element in the list and the tail represent the remainder (i.e head is an element but tail is a list) . for the following list :

L = [1,2,3]

H = 1 T =[2,3]

H =2 T =[3]

H =3 T=[ ]

[ ] refer to empty list.

List can be written as [H|T] in the program, if the list is non empty then this statement decompose the list into Head and tail otherwise (if the list is empty) this statement add element to the list.

For example:

- [a] and [a,b,c], where a, b and c are symbols type.

- [1], [2,3,4] these are a lists of integer.
- [] is the atom representing the empty list.
- Lists can contain other lists.

Split a list into its head and tail using the operation [X|Y].

Examples about Lists

1. p([1,2,3]).

p([the,cat,sat,on,the,hat]).

Goal: p([X|Y]).

Output:

X = 1 Y = [2,3] ;

X = the Y = [cat,sat,on,the,hat].

2. p([a]).

Goal: p([H | T]).

Output:

H = a, T = [].

3. p([a, b, c, d]).

Goal: p([X, Y | T]).

Output:

X = a, Y = b, T = [c, d].

4. P([[a, b, c], [d, e]]).

Goal: p([H|T])

Output:

H = [a, b, c], T = [[d, e]].

List and Recursion

As maintained previous list consist of many element, therefore to manipulate each element in the list we need recursive call to the list until it become empty.

List Membership

• Member is possibly the most used user-defined predicate (i.e. you have to define it every time you want to use it!).

• It checks to see if a term is an element of a list.

– it returns **yes** if it is.

– and **fails** if it isn't.

domains

ilist=integer*

predicates

member (integer,ilist)

clauses

member(X,[X|_]).

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

goal

member(3,[1,2,3,4]).

trace

member(3,[1,2,3,4]).

member(3,[2,3,4]).

member(3,[3,4]). yes

 It 1st checks if the Head of the list unifies with the first argument.

If yes then succeed.

If no then fail first clause.

 The 2nd clause ignores the head of the list (which we know doesn't match) and recourses on the Tail.

Goal: member(a, [b, c, a]).

Output: Yes

Goal: member(a, [c, d]).

Output: No.

Print the contents of a list.

domains

ilist=integer*

predicates

print(ilist)

clauses

print([]).

print([X|T]):-write(X),print(T).

Goal: print([3,4,5])

Output: 3 4 5 yes

Program to find sum of integer list.

```
Domains
 I= integer
L=i*
Predicates
  Sum ( L ,I, I)
Clauses
Sum ([],R,R):-!.
Sum([H|T],R1,R):-R2=R1+H,Sum(T,R2,R).
Goal
Sum ([ 1,4,6,9],0,R).

Output
R = 20   yes
```

Prolog program to spilt list into positive list and negative list.

```
Domains
L= integer*
Predicates
Spilt ( L,L,L)
```

Clauses
Spilt ([],[],[]):-!.
Spilt ([H|T],[H|T1],L2):- H>= 0,!,spilt(T,T1,L2).
Spilt ([H|T],L1,[H|T2]) :- spilt(T,L1,T2).
Goal
Spilt ([-1,4,-9,8,0],L1,L2).
Output:
L1 = [4,8,0]
L2 = [-1,-9]

Find the maximum value of the list

domains

ilist=integer*

predicates

list(ilist,integer)

clauses

list([H],H).

list([H1,H2|T],H):-H1>H2,list([H1|T],H).

list([_,H2|T],H):-list([H2|T],H).

Goal: list([3,9,4,5],M)

Output: M=9 yes

[3,9,4,5]

[9,4,5]

[9,5]

[9]

Append two lists

domains

ilist=integer*

predicates

app(ilist,ilist,ilist)

clauses

```
app([],L,L).
app([H|T],L1,[H|T1]) :-app(T,L1,T1).
```

Goal

```
app([3,4,5],[6,7,8],L).
```

Output: L=[3,4,5,6,7,8] yes

Write the contents of list inside at the given list

```
domains
ilist=integer*
lists=ilist*
predicates
list(lists)
clauses
list([]).
list([[]]).
list([[H|T]]):-write(H),list([T]).
list([H|T]):-list([H]),list(T).
```

Goal

```
list([[3,4,5],[6,7,8]]).
```

Output: 3 4 5 6 7 8 yes

Reveres the contents of the given list.

```
domains
slist=symbol*
predicates
rev(slist,slist)
app(slist,slist,slist)
clauses
app([],L,L):-!.
app([H|T],L1,[H|T1]):-app(T,L1,T1).
```

```
rev([X],[X]):-!.
rev([H|T],L):-rev(T,L1),app(L1,[H],L).
```

Goal

```
rev([a,b,c,d],R).
```

Output: R=[d,c,b,a]

Delete a particular element from a list

```
domains
i= integer
l= i*
Predicates
Delall (i,l,l)
Clauses
Delall(_,[ ],[ ]):-!.
Delall(X,[X|T],T) :-!.
Delall(X,[ H|T],[H|T1]):-Delall(X,T,T1).
Goal
Delall(2,[1,3,2,3,2,1],L).
```

trace

```
Delall(2,[1,3,2,3,2,1],L).
Delall(2,[1,3,2,3,2,1],[1|T1])
Delall(2,[3,2,3,2,1],T1)
Delall(2,[3,2,3,2,1],[3|T1])
Delall(2,[2,3,2,1],T1)
Delall(2,[2,3,2,1],[3,2,1])
```

Output L=[1,3,3,2,1]

```
domains
i= integer
l= i*
Predicates
Delall (i,l,l)
Clauses
Delall (_,[ ],[ ]):-!.
Delall (X,[X|T],T1) :- Delall (X,T,T1).
Delall (X,[ H|T],[H|T1]):- Delall (X,T,T1).
Goal
Delall(2,[1,3,2,3,2,1],L).
Delall(2,[1,3,2,3,2,1],[1|T1])
Delall(2,[3,2,3,2,1],T1)
Delall(2,[3,2,3,2,1],[3|T1])
Delall(2,[2,3,2,1],T1)
Delall(2,[3,2,1],T1)
Delall(2,[3,2,1],[3|T1])
Delall(2,[2,1],T1)
Delall(2,[1],T1)
Delall(2,[1],[1|T1])
Delall(2,[],T1)
Delall(_,[],[])
Output X=[1,3,3,1]
```

Delete an element at a particular location from a list

```
domains
i=integer
l= i*
predicates
```

Delatloc (i,l,l)

Clauses

Delatloc (1, [_|T],T):-!.

Delatloc (N,[H|T],[H|T1]):-N1=N-1, Delatloc (N1,T,T1).

Goal

Delatloc(2,[2,4,6,8],X).

trace

Delatloc(2,[2,4,6,8],X).

Delatloc(2,[2,4,6,8],[2|T1])

N1=1

Delatloc(1,[4,6,8],T1)

T1=[6,8]

Output X=[2,6,8]

Collect in a list all the elements that found in list1 and not found in list2

domains

l=integer*

predicates

dif(l,l,l)

member(integer,l)

clauses

Dif ([],_,[]).

Dif ([H|T1],L,[H|T2]):-not(member (H,L)),!, Dif(T1,L,T2).

Dif([_|T1],L1,L2):- Dif(T1,L1,L2).

Member(H,[H|_]):-!.

Member(H,[_|T]):-member(H,T).

Goal

Dif([2,4,6,8],[3,4,8,5],X)

Output X=[2,6]

Tail and non tail Recursive Programs.

Non-tail Summation of 10 integer number:

predicates

sum_nontail(integer,integer)

clauses

sum-nontail(11,0).

sum-nontail(X,S):-X1=X+1, sum-nontail(X1,S1),S=S1+X.

Trace the above program with the goal ? sum-nontail(1,S)

sum-nontail(1,S):-X1=2, sum-nontail(2,S1),S=S1+1.

sum-nontail(2,S):-X1=3, sum-nontail(3,S1),S=S1+2.

sum-nontail(3,S):-X1=4, sum-nontail(4,S1),S=S1+3.

sum-nontail(4,S):-X1=4, sum-nontail(5,S1),S=S1+4.

sum-nontail(5,S):-X1=4, sum-nontail(6,S1),S=S1+5.

sum-nontail(6,S):-X1=4, sum-nontail(7,S1),S=S1+6.

sum-nontail(7,S):-X1=4, sum-nontail(8,S1),S=S1+7.

sum-nontail(8,S):-X1=4, sum-nontail(9,S1),S=S1+8.

sum-nontail(9,S):-X1=4, sum-nontail(10,S1),S=S1+9.

sum-nontail(10,S):-X1=4, sum-nontail(11,S1),S=S1+10.

sum-nontail(11,0).

Then ? sum-nontail(1,S)

S=55.

Non-tail Factorial program:

predicates

fact(integer,integer)

clauses

fact(0,1).

fact(1,1).

fact(X,Y):-X1=X-1,fact(X1,Y1),Y=X*Y1.

Non-tail Power program.

predicates

power(integer,integer,integer)

clauses

power(_,0,1).

power(X,Y,Z):-Y1=Y-1,power(X,Y1,Z1),Z=Z1+X.

H.W

1. Write prolog program to find the union of two lists.
2. Write prolog program to find the intersection between two lists.
3. Write prolog program to find the difference between two lists.
4. Write prolog program that check the equality between two lists.
5. Write prolog program to find the last element in a list.
6. Write prolog program to find the union of two lists.
7. Write prolog program to find the length of a list.
8. Write prolog program to find the index of specified element in a list.
9. Write prolog program to get the element at nth index lists.
10. Write prolog program that replace specified element in a list with value 0.
11. Write prolog program that delete a specified element in a lists.
12. Write prolog program that take two lists as input and produce a third list as output, this list is the sum of the two lists.
13. Write prolog program that multiply each element in the list by 5.
14. Write prolog program that sort a list descending.
15. Write prolog program that convert any given decimal number to its binary representation and store it in a list.

Standard String Predicates

Prolog provides several standard predicates for powerful and efficient string manipulations. In this section, we summarize the standard predicates available for string manipulating and type conversion.

1. **str_len** (String,Length) (string,integer) (i,o): Determines the length of String. Succeeds if the length could be matched with Len.

str_len(“prolog”,X)

X=6.

Str_len("ab",3) no
Str_len(X,3) X="---"

2. **str_char** (string,char) (i,o) (o,i): Converts a string ***(of one char)*** into a character or vice versa. If string is bound, it must have length 1 for the predicate to succeed.

str_char(“A”,X)

X=’A’.

3. **str_int** (string,integer) (i,o) (o,i): Converts a string of one character to ASCII code or vice versa. If string is bound, it must contain a single number for the predicate to succeed.

str_int(“A”,X)

X=65 .

Str_int(X,65)

X=”A”

Str_int(“33”,X)

X=33

Str_int(X,33)

X=”33”

4. str_real (string,real) (i,o) (o,i): Converts the string (of real) to real and the opposit

Str_real("0.5",X)
X=0.5
Str_real(X,0.5)
X="0.5"

Str_real(“5”,X)
X=5.0

5. **char_int** (char,integer) (i,o) (o,i): Converts a character to ASCII code or vice versa.

char_int(‘A’,X)

X=65.

6. **isname** (string) (i): Tests whether a string would match a prolog symbol in other words ***test if the content of the string is name or not***.

isname(“s2”) return YES.

isname(“4r”) return NO.

Isname("abc") yes
Isname("123"). No

7. **frontchar** (String,FrontChar,RestString) (string,char,string) (i,o,o) (o,i,i): Extracts the first character from a string, the remainder is matched with RestString.

frontchar (“prolog”,C,R)

C=’p’, R=”rolog”.

8. **fronttoken** (String,Token,Rest) (string,string,string) (I,o,o)(i,o,i) (i,i,o) (o,i,i): Skips all white space characters (blanks,tabs) and separates from the resulting string the first valid token. The remainder is matched with RestString. A valid token is either a variable or name 'A'...'Z','a'...'z',' ','0'...'9', a number '0'...'9' or a single character. It fails if String was
empty or contained only whitespace.

fronttoken (“complete prolog program”,T,R)

T=”complete”, R=”prolog program”.

9. **frontstr** (StrLen,String,FrontStr,RestStr) (i,i,o,o): Extracts the first n characters from a string. This establishes a relation between String, Count, FronStr, and RestString, thus that String = FrontStr+RestString and str_len(FronStr,Count) is true. The String and Count arguments must be initialized.

frontstr(3,”cdab 2000”,T,R)

T=”cda”, R=”b 2000”.

10. **concat** (Str1,Str2,ResStr) (string,string,string) (i,i,o): Merges to strings to one by appending the second argument to the first.

concat (“prolog”,”2011”,R)

R=”prolog2011”.

11. **Upper_lower*(string,string)***

Convert the string in upper case(in capital letter) to the lower case (small letter) and the opposite.

upper_lower(capital_letter,small_letter)

upper_lower("ABC",X)

X="abc"

upper_lower("Abc",X)

X="abc"

upper_lower(X,"abc")

X="ABC"

Examples:

Program that read two strings and concat them in one string as upper case.

```
predicates
start(string)
clauses
start(X):-readln(S),readln(S1),concat(S,S1,S2),upper_lower(X,S2).
Goal
Start(X).
```

Output:

Ahmed

Ali

X=AHMEDALI yes

program that read string of one character then print the integer value of this char.

```
predicates
start
clauses
start:-readln(S),str_int(S,X), write(X).
goal
start.
```

Output:

a

97 yes

Program that take a string of words and print each word in a line as upper case.

```
predicates
start(string).
Clauses
start("").
start(S):-fronttoken(S,S3,S2), upper_lower(S1,S3), write(S1),  nl,
          start(S2).
Goal
Start("ali is a good boy").
Trace
Start("ali is a good boy").
fronttoken(("ali is a good boy",S3,S2)
S3=”ali”
S2=”is a good boy”
upper_lower(S1,”ali”)
S1=”ALI”
Start(”is a good boy”)
fronttoken(("is a good boy",S3,S2)
S3=”is”
S2=”a good boy”
upper_lower(S1,”is”)
S1=”IS”
Start(“a good boy”)
.
.
.
Sart(“”)
```

Output:

```
ALI
IS
A
GOOD
BOY
yes
```

Program that take a string and convert each character it contains to its corresponding integer value.

```
Predicates
start(string).
clauses
start(S):-frontchar(S,S3,S2), char_int(S3,I), write(I), nl , start(S2).
```

```
start("").
Goal
Start("abc").
```

Output:

```
97
98
99
Yes
```

Program that return the number of names in a specific string.

```
predicates
start(string,intger,integer).
clauses
start(S,X1,X):-
fronttoken(S,S1,S2),isname(S1),!,X2=X1+1,start(S2,X2,X).
start(S,X1,X):-fronttoken(S,_,S2),start(S2,X1,X).
start("",X,X).
```

goal

start("ali has 2 cars",0,X), write("the no. is ",X).

Output:

The no. of names is 3

Yes

Program that split a specific string to small string with length 3 char.

```
predicates
start(string).
clauses
start("").
start(S):-str_len(S,I),  I MOD 3=0,!, frontstr(3,S,S1,S2), write(S1),
          nl,start(S2).
start(S):-concat(S," ",S1),start(S1).
Goal
Start("abcdefg").
```

Output:

```
abc
def
g
yes
```

Convert the string of words into a list of words:

domains

slist=string*

predicates

split(string,slist)

clauses

split("",[]).

split(S,[H|T]):-fronttoken(S,H,R),split(R,T).

goal

split("list of words",L).

fronttoken("list of words",H,R)

H=”list”

R=”of words”

Split(“of words”,T)

fronttoken("of words",H,R)

H=”of”

R=”words”

Split(“words”,T)

fronttoken("words",H,R)

H=”words”

R=””

Split(“”,T).

output: L=[List","of","words"]

Count the number of words in the given string.

Predicates

counts(string,integer)

clauses

counts("",0).

```
counts(S,C):-fronttoken(S,_,R), counts(R,C1),C=C1+1.
```

Goal

```
counts("number of words",N)
fronttoken("number of words",_,R),
R=”of words”
counts(“of words”,C1),                    stack C=C1+1
fronttoken("of words",_,R),
R=”words”
counts(“words”,C1),                       stack C=C1+1
fronttoken("words",_,R),
R=””
counts(“”,C1),                            stack C=C1+1
C1=0
C=C1+1
C=1
C=1+1
C=2+1
```

output N=3

Count the number of characters in the given string.

Predicates

```
counts(string,integer)
```

clauses

```
counts("",0).
counts(S,C):-frontchar(S,_,R), counts(R,C1),C=C1+1.
```

Goal

counts("no of char",N)

output

N=10

Reverse the given string.

rev("","").

rev(Str,R_Str):-frontstr(1,Str,C,RemStr), rev(RemStr,L1),concat(L1,C,R_Str).

Goal

Rev("abc",R).

frontstr(1,"abc",C,RemStr),

C="a"

RemStr="bc"

Rev("bc",L1) stack concat(L1,"a",R_Str)

frontstr(1,"bc",C,RemStr),

C="b"

RemStr="c"

Rev("c",L1) stack concat(L1,"b",R_Str)

frontstr(1,"c",C,RemStr),

C="c"

RemStr=""

Rev("",L1) stack concat(L1,"c",R_Str)

L1=""

concat("","c",R_Str)

R_Str="c"

concat("c","b",R_Str)

R_Str="cb"

concat("cb","a",R_Str)

R_Str="cba"

Convert a list of words to a string

con([],"").

```
con([H|T],Str):-con(T,Str1),concat(H,Str1,Str).
```

Count the number of words that contain “tion” in the given list.

```
count(S):-frontstr(4,S,X,_),X="tion".
count(S):-frontchar(S,_,R),count(R).
set([],0).
set([H|T],L):-count(H),set(T,L1),L=L1+1,!.
set([_|T],L):-set(T,L).
```

H.W

1- *Write a prolog program that do the following: convert the string such as "abcdef" to 65 66 67 68 69 70.*

2-*Program tofind the number of tokens and the number of character in a specific string such as: "ab c def" the output is tokens and 6 character.*

Examples

1. Count the number of words that contain “tion” in a given list.

```
Domains
slist=string*
Predicates
count(string)
set(slist,integer)
clauses
count(S):-frontstr(4,S,X,_),X="tion",!.
count(S):-frontchar(S,_,R),count(R).
set([],0).
set([H|T],L):-count(H),set(T,L1),L=L1+1,!.
set([_|T],L):-set(T,L).
goal
set(["ab","tion"],X).
```

trace

set(["ab","tion"],X).

count("ab")

frontstr(4,"ab",X,_) --- fail

frontchar("ab",_,R),

count("b")

frontstr(4,"b",X,_) --- fail

frontchar("b",_,R)

count("")

frontstr(4,"",X,_) ---- fail

frontchar("",_,X) ---- fail

count("ab") ---- fail

set(["tion"],L)

count("tion")

frontstr(4,"tion",X,_),

X="tion" ---- true

Count("tion") ---- true

Set([],L1) --- true

L1=0

L=0+1

L=1

Set(["ab","tion"],X) ---- true

X=1.

2. Count the number of characters in a given string using external database.

database

single sum(integer)

predicates

count(string)

clauses

```
count(""):-sum(X),write(X),!.
count(S):-frontchar(S,_,S1),sum(X),X1=X+1,assert(sum(X1)),count(S1).
```

Goal

```
assert(sum(0)),
count("yes").
```

trace

```
count("yes")
 frontchar("yes",_,S1) --- true
S1="es"
sum(X)
X=0
X1=1
sssert(sum(1))
count("es")
frontchar("es",_,S1) --- true
S1="s"
sum(X)
X=1
X1=2
assert(sum(2))
count("s")
frontchar("s",_,S1) --- true
S1=""
sum(X)
X=2
X1=3
assert(sum(3))
count("")
```

sum(X)

X=3

write(3) ---- output

count("yes") --- true

3. Find how many times an element occurs in a list using non tail recursion.

domains

ilist=integer*

predicates

count(integer,ilist, integer)

clauses

count(_,[],0):-!.

count(H,[H|T],S):-count(H,T,S1), S=S1+1,!.

count(X,[_|T],S):-count(X,T,S).

Goal

count(3,[2,3,4,3],X).

trace

count(3,[2,3,4,3],X)

count(3,[3,4,3],S)

count(3,[4,3],S1)

count(3,[3],S)

count(3,[],S1)---- true

S1=0

S=S1+1

S=0+1

S=S1+1

S=1+1

count(3,[2,3,4,3],X) ---- true

X=2.

4. Find the average of the even numbers in a given list using non tail recursion.

```
domains
ilist=integer*
predicates
sum_count(ilist, integer,integer)
average(ilist)
clauses
average(L):-sum_count(L,S,N), A=S/N,
                write("The average is ",A).
sum_count([],0,0):-!.
sum_count([H|T],S,N):-H mod 2 = 0, sum_count(T,S1,N1),
                                S=S1+H, N=N1+1,!.
sum_count([_|T],S,N):-sum_count(T,S,N).
Goal
average([2,4,3,6,7]).
```

trace

```
average([2,4,3,6,7])
sum_count([2,4,3,6,7],S,N)
2 mod 2=0     true
sum_count([4,3,6,7],S1,N1)
4 mod 2 =0    true
sum_count([3,6,7],S1,N1)
3 mod 2 =0   fail
sum_count([6,7],S,N)
6 mod 2 =0     true
sum_count([7],S1,N1)
7 mod 2 =0      fail
sum_count([],S,N)
S=0, N=0
```

S=0+2

N=0+1

S=2+4

N=1+1

S=6+6

N=2+1

A=12/3

Output

The average is 4

5. Create a list that contains the result of adding pairs of elements in a given list, such as if your input list is [2, 5, -5, 2, 1, 4,9], then the output list will be [7,-3,5,9].

```
domains
ilist=integer*
predicates
add(ilist,ilist)
clauses
add([],[]):-!.
add([H],[H]):-!.
add([H1,H2|T],[H3|T1]):-H3=H1+H2,add(T,T1).
Goal
add([2, 5, -5, 2, 1, 4,9],L).
```

Trace

add([2,5,-5,2,1,4,9],L)

H3=2+5

add([-5,2,1,4,9],T1)

H3=-5+2

add([1,4,9],T1)

H3=1+4

add([9],T1)

T1=[9]

T1=[5,9]

T1=[-3,5,9]

L=[7,-3,5,9]

6. Reverse the contains of a given list.

domains

slist=symbol*

predicates

app(slist, slist,slist)

rev(slist)

clauses

app([],X,X).

app([H|T1],X,[H|T]):-app(T1,X,T).

rev([X],[X]).

rev([H|T],L):-rev(T,L1),app(L1,[H],L).

Goal: rev([a,b,c],R)

Trace

rev([a,b,c],R)

rev([b,c],L1)

rev([c],L1)

L1=[c]

app([c],[b],L)

app([],[b],T)

T=[b]

L=[c,b]

app([c,b],[a],L)

app([b],[a],T)

app([],[a],T)

T=[a]

L=[c,b,a]

7. Write a prolog program that can split a string to two strings according to a given number (without using the library function frontstr), such as if your input is the string "hello" and the number is 3 then the output will be the two strings "hel" and "lo".

predicates

split(string,integer,string,string)

clauses

```
split(S,0,"",S):-!.
Split(S,N,S1,S2):-frontchar(S,Ch,S3), N1=N-1,
                    split(S3,N1,S4,S2),
                      str_char(SS,Ch),concat(SS,S4,S1).
```

Goal

split("hello",3,S1,S2)

trace

split("hello",3,S1,S2)

frontchar("hello",Ch,S3)

Ch='h'

S3="ello"

N1=2

Split("ello",2,S4,S2)

frontchar("ello",Ch,S3)

Ch='e'

S3="llo"

N1=1

Split("llo",1,S4,S2)

frontchar("llo",Ch,S3)

```
Ch='l'
S3="lo"
N1=0
Split("lo",0,S4,S2)
S4=""
S2="lo"
Str_char(SS,'l')
SS="l"
Concat("l","",S1)
S1="l"
Str_char(SS,'e')
SS="e"
Concat("e","l",S1)
S1="el"
Str_char(SS,'h')
SS="h"
Concat("h","el",S1)
S1="hel"
```

Introduction to Artificial Intelligence

INTRODUCTION

The term AI was first introduced by john McCarthy in 1956, since then, several areas of applications and researches have been developed, (expert systems, knowledge based system,…).

What is Intelligence?

Intelligence is the ability to learn about, to learn from, to understand about, and interact with one's environment.

What is Artificial Intelligence (AI)?

AI:- Is simply a way of making a computer think.
AI: it is the part of the computer science concerned in designing intelligent computers, that is, computers that exhibits the characteristics we associate with intelligence in human behavior (understanding language, learning, reasoning, solving problems, …).
AI: is the study of how to make a computer do thing at which, at the moment people do them better.
AI: is a field of study that it's goal is to make the computer perform tasks that require intelligence when performed by humans.

AI applications

AI programs fall into three basic categories:

- Natural language processing systems (like machine translation).
- Perception systems (vision, speech, touch).
- Expert systems (like chemical analysis, medical diagnoses).

What are the goals of AI research?

The central problems (or goals) of AI research include reasoning, knowledge, planning, learning, natural language processing (communication), perception and the ability to move and manipulate objects.

Fundamental issues of AI involves

- Knowledge representation
- Search strategy
- Perception and inference.

What are Knowledge Representation Schemes ?

In Al, there are four basic categories of representational schemes: logical, procedural, network and structured representation schemes.

1) **Logical representation** uses expressions in formal logic to represent its knowledge base. Predicate Calculus is the most widely used representation scheme.
2) **Procedural representation** represents knowledge as a set of instructions for solving a problem. These are usually if-then rules we use in rule-based systems.

3) **Network representation** captures knowledge as a graph in which the nodes represent objects or concepts in the problem domain and the arcs - represent relations or associations between them.
4) **Structured representation** extends network representation schemes by allowing each node to have complex data structures named slots with attached values.

1) The Prepositional and Predicates Calculus:

1.1 The Prepositional Calculus:

The propositional calculus and predicate calculus are first of all languages. Using their words, phrases, and sentences, we can represent and reason about properties and relationships in the world. The first step in describing a language is to produce the pieces that make it up: its set of symbols.

Propositional Calculus Symbols

- The symbols of propositional calculus are: {P, Q, R, S, …}
- Truth symbols: **{True, false}**
- Connectives: **{∧ ,∨ , ¬ , → , ≡}**

Propositional symbols denote ***propositions***, or statements about the world that may be either true or false, Propositions are denoted by uppercase letters near the end of the English alphabet Sentences, **For example:**

P: It is sunny today.

Q: The sun shines on the window.

R: The blinds are brought down.

(P→Q): If it is sunny today, then the sun shines on the window

(Q→R): If the sun shines on the window, the blinds are brought down.

(¬R): The blinds are not brought down.

Propositional Calculus Sentence

- Every propositional symbol and truth symbol is a sentence.

  For example: **true, P, Q,** and **R** are sentences**.**
- The ***negation*** of a sentence is a sentence.

  For example: **¬P** and **¬false** are sentences.

- The ***conjunction***, **AND**, of two sentences is a sentence.

  For example: **P** $\wedge$ $\neg$**P** is a sentence.
- The ***disjunction***, **OR** of two sentences is a sentence.

  For example: **P** $\vee$ $\neg$**P** is a sentence.
- The ***implication*** of one sentence from another is a sentence.

  For example: **P** $\rightarrow$ **Q** is a sentence.
- The ***equivalence*** of two sentences is a sentence.

  For example: **P** $\vee$ **Q** $\equiv$ **R** is a sentence.
- Legal sentences are also called ***well-formed formulas*** or ***WFFs***.

In expressions of the form **P** $\wedge$ **Q, P** and **Q** are called the ***conjuncts***. In **P** $\vee$ **Q**, **P** and **Q** are referred to as ***disjuncts***. In an implication, **P** $\rightarrow$ **Q, P** is the ***premise*** and ***Q***, the ***conclusion*** or ***consequent.***

In propositional calculus sentences, the symbols **( )** and **[ ]** are used to group symbols into sub-expressions and so to control their order of evaluation and meaning.

For Example: (P $\vee$ **Q)** $\equiv$ **R** is quite different from **P** $\vee$ **(Q** $\equiv$ **R)** as can be demonstrated using truth tables. An expression is a sentence, or well-formed formula, of the propositional calculus if and only if it can be formed of legal symbols through some sequence of these rules.

For Example: **((P** $\wedge$ **Q)** $\rightarrow$ **R)** $\equiv$ $\neg$**P** $\vee$ $\neg$**Q** $\vee$ **R** is a well-formed sentence in the propositional calculus because:

P, Q, and **R** are propositions and thus sentences.

P $\wedge$ **Q**, the conjunction of two sentences, is a sentence.

(P $\wedge$ **Q)** $\rightarrow$ **R**, the implication of a sentence for another, is a sentence.

$\neg$**P** and $\neg$**Q**, the negations of sentences, are sentences.

$\neg$**P** $\vee$ $\neg$**Q** the disjunction of two sentences, is a sentence.

$\neg$**P** $\vee$ $\neg$**Q** $\vee$ **R**, the disjunction of two sentences, is a sentence.

((P ∧ Q) → R) ≡ ¬P ∨ ¬Q ∨ R, the equivalence of two sentences, is a sentence.

This is our original sentence, which has been constructed through a series of applications legal rules and is therefore "***well formed***".

Example: Convert the following engilsh sentences to propositional calculus sentences:

- It is hot.
- It is not hot.
- If it is raining, then will not go to mountain.
- The food is good and the service is good.
- If the food is good and the service is good then the restaurant is good.

Answer:

- It is hot
  p
- It is not hot
  ¬p
- If it is raining, then will not go to mountain
  p → ¬q
- The food is good and the service is good
  x ∧ y
- If The food is good and the service is good then the restaurant is good
  x ∧ y → z

1.2 The Predicate Calculus (Also known as First-Order Logic):

In prepositional calculus, each atomic symbol (P, Q, etc.) denotes a proposition of some complexity. There is no way to access the components of an individual assertion. In other words, the prepositional calculus has its limitations that you cannot deal properly with general statements because it represents each statement by using some symbols jointed with connectivity tools. To solve the

limitations in the prepositional calculus, you need to analyze propositions into predicates and arguments, and deal explicitly with quantification. Predicate calculus provides formalism for performing this analysis of prepositions and additional methods for reasoning with quantified expressions. For example, instead of letting a single prepositional symbol, **P**, denotes the entire sentence "**it rained on Tuesday**," we can create a predicate weather that describes a relationship between a date and the weather:

weather (rain, Tuesday)

through inference rules we can manipulate predicate calculus expression accessing their individual components and inferring new sentences. Predicate calculus also allows expressions to contain variables. Variables let us create general assertions about classes of entities. For example, we could state that for all values, of X, where X is a day of the week, the statement:

weather (rain, X) is true ;

I,e., it rains it rains every day. As with propositional calculus, we will first define the syntax of the language and then discuss its semantics.

Example: Convert the following engilsh sentences to predicate calculus sentences:

1. If it is raining, tom will not go to mountain

2. If it doesn't rain tomorrow, Tom will go to the mountains.

3. All basketball players are tall.
4. Some people like anchovies.
5. John like anyone who likes books.
6. Nobody likes taxes.
7. There is a person who writes computer class.
8. All dogs are animals.
9. All cats and dogs are animals.
10. John did not study but he is lucky.
11. There are no two adjacent countries have the same color.
12. All blocks supported by blocks that have been moved have also been moved. Note: you can use the following predicates:

• block(X) means X is a block
• supports(X, Y) means X supports Y
• moved(X) means X has been moved

Answer:

1. weather (rain) → ¬ go(tom,mountain)
2. ¬ weather (rain, tomorrow) →go(tom, mountains).
3. ∀ X (basketball _ player(X) → tall (X))
4. ∃ X (person(X) ∧ likes(X, anchovies)).
5. ∃ X like(X,book) →like(john,X)

6. ¬ ∃ X likes(X,taxes).
7. ∃X write(X,computer_class)
8. ∀X dogs(X) →animals(X)
9. ∀X∀Y cats(X)∧dogs(Y) →animals(X)∧animals(Y).
10. ¬ study(john)∧ lucky(john)
11. ∀X∀Y (county(X) ∧ county(Y) ∧ adjacent(X,Y)) → ¬ (color(X) ≡ color(Y)). Or we say: ∀X∀Y ¬ county(X) ∨ ¬county(Y) ∨ ¬adjacent(X,Y) ∨ ¬ (color(X) ≡ color(Y)).
12. ∀X∀Y block(X) ∧ block(Y) ∧ supports(X,Y) ∧ moved(X) → moved(Y)

2. Resolution:

Resolution is a technique for proving theorems in the predicate calculus using the resolution by refutation algorithm. The resolution refutation proof procedure answers a query or deduces a new result by reducing the set of clauses to a contradiction.

The Resolution by Refutation Algorithm includes the following steps:-

a) Convert the statements to **predicate calculus** (predicate logic).
b) Convert the statements from **predicate calculus** to **clause forms**.
c) Add the negation of what is to be proved to the clause forms.
d) Resolve the clauses to producing new clauses and producing a contradiction by generating the empty clause.

Clause Forms

The statements that produced from **predicate calculus** method are nested and very complex to understand, so this will lead to more complexity in resolution stage , therefore the following algorithm is used to convert the **predicate calculus** to **clause forms**:-

1. Eliminate all (→) by replacing each instance of the form **(P → Q)** by expression **(¬P∨Q)**
2. Reduce the scope of negation.

   ¬(¬a) ≡a

   ¬(∀X) b(X)≡∃X ¬ b(X)

   ¬(∃X) b(X)≡∀X ¬b(X)

   ¬(a∧b)≡¬a∨¬b

   ¬(a∨b)≡¬a∧¬b

3. Standardize variables: rename all variables so that each quantifier has its own unique variable name. *For example*,

   $\forall X\ a(X) \vee \forall X\ b(X) \equiv \forall X\ a(X) \vee \forall Y\ b(Y)$

4. Move all quantifiers to the left without changing their order. *For example*,

   $\forall X\ a(X) \vee \forall Y\ b(Y)$

   $\forall X\ \forall Y\ a(X) \vee b(Y)$

5. Eliminate existential quantification by using the equivalent function. *For example*,

   $\forall X \quad \exists Y\ (\text{mother}(X,Y)) \equiv \forall X\ (\text{mother}(X,m(X)))$

   $\forall X\ \forall Y\ \exists Z\ (p(X,Y,Z) \equiv \forall X\ \forall Y\ (p(X,Y,f(X,Y))$

6. Remove universal quantification symbols. *For example*,

   $\forall X\ \forall Y\ (p(X,Y, f(X,Y))) \equiv p(X,Y, f(X,Y))$

7. Use the associative and distributive properties to get a conjunction of disjunctions called **conjunctive normal form**. *For example*,

   $a \vee (b \vee c) \equiv (a \vee b) \vee c$

   $a \wedge (b \wedge c) \equiv (a \wedge b) \wedge c$

   $a \vee (b \wedge c) \equiv (a \vee b) \wedge (a \vee c)$

   $a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c)$

8. Split each conjunct into a separate **clause**. *For example*,

   $(\neg a(X) \vee \neg\ b(X) \vee e(W)) \wedge (\neg b(X) \vee \neg d(X,f(X)) \vee e(W))$

   $\neg a(X) \vee \neg\ b(X) \vee e(W)$

   $\neg b(X) \vee \neg d(X,f(X)) \vee e(W)$

9. Standardize variables apart again so that each clause contains variable names that do not occur in any other clause. *For example*,

   $(\neg a(X) \vee \neg\ b(X) \vee e(W)) \wedge (\neg b(X) \vee \neg d(X,f(X)) \vee e(W))$

¬a(X)∨¬ b(X)∨e(W)

¬b(Y)∨¬d(X,f(X))∨e(V)

Example: Use the Resolution Algorithm for proving that John is happy with regard the following story:

Everyone passing his AI exam and winning the lottery is happy. But everyone who studies or lucky can pass all his exams, John did not study but he is lucky. Everyone who is lucky wins the lottery. Prove that John is happy.

Solution:

a) Convert all statement to predicate calculus.

∀X pass(X,ai_exam)∧win(X,lottery)→happy(X)

∀Y∀E study(Y)∨lucky(E)→pass(Y,E)

¬study(john)∧lucky(john)

∀Z lucky(Z)→win(Z,lottery)

happy(john)?

b) Convert the statements from predicate calculus to clause forms:

1.

∀X ¬(pass(X,ai_exam)∧win(X,lottery))∨ happy(X)

∀Y∀E¬(study(Y)∨lucky(Y))∨ pass(Y,E)

¬study(john)∧lucky(john)

∀Z ¬(lucky(Z))∨ win(Z,lottery)

happy(john)?

2.

∀X (¬pass(X,ai_exam)∨¬win(X,lottery))∨ happy(X)

∀Y∀E(¬study(Y)∧¬lucky(Y))∨ pass(Y,E)

¬study(john)∧lucky(john)

∀Z ¬lucky(Z)∨win(Z,lottery)

happy(john)?

3. Nothing to do here.

4. Nothing to do here.

5. Nothing to do here.

6.

(¬pass(X,ai_exam)∨¬win(X,lottery))∨ happy(X)

(¬study(Y)∧¬lucky(Y))∨ pass(Y,E)

¬study(john)∧lucky(john)

¬lucky(Z)∨win(Z,lottery)

happy(john)?

7.

¬pass(X,ai_exam)∨¬win(X,lottery)∨ happy(X)

(¬study(Y)∧¬lucky(Y))∨ pass(Y,E) ≡ (a∧b)∨c ≡ c∨(a∧b)

The second statement become: pass(Y,E)∨¬study(Y)∧pass(Y,E)∨¬lucky(Y)

¬study(john)∧lucky(john)

¬lucky(Z)∨win(Z,lottery)

happy(john)?

8.

¬pass(X,ai_exam)∨¬win(X,lottery)∨ happy(X)

pass(Y,E)∨¬study(Y)

pass(Y,E)∨¬lucky(Y)

¬study(john)

lucky(john)

¬lucky(Z)∨win(Z,lottery)

happy(john)?

9.

¬pass(X,ai_exam)∨¬win(X,lottery)∨ happy(X)

pass(Y,E)∨¬study(Y)

pass(M,G)∨¬lucky(M)

¬study(john)

lucky(john)

¬lucky(Z)∨win(Z,lottery)

happy(john)?

c) Add the negation of what is to be proved to the clause forms.

¬happy(john).

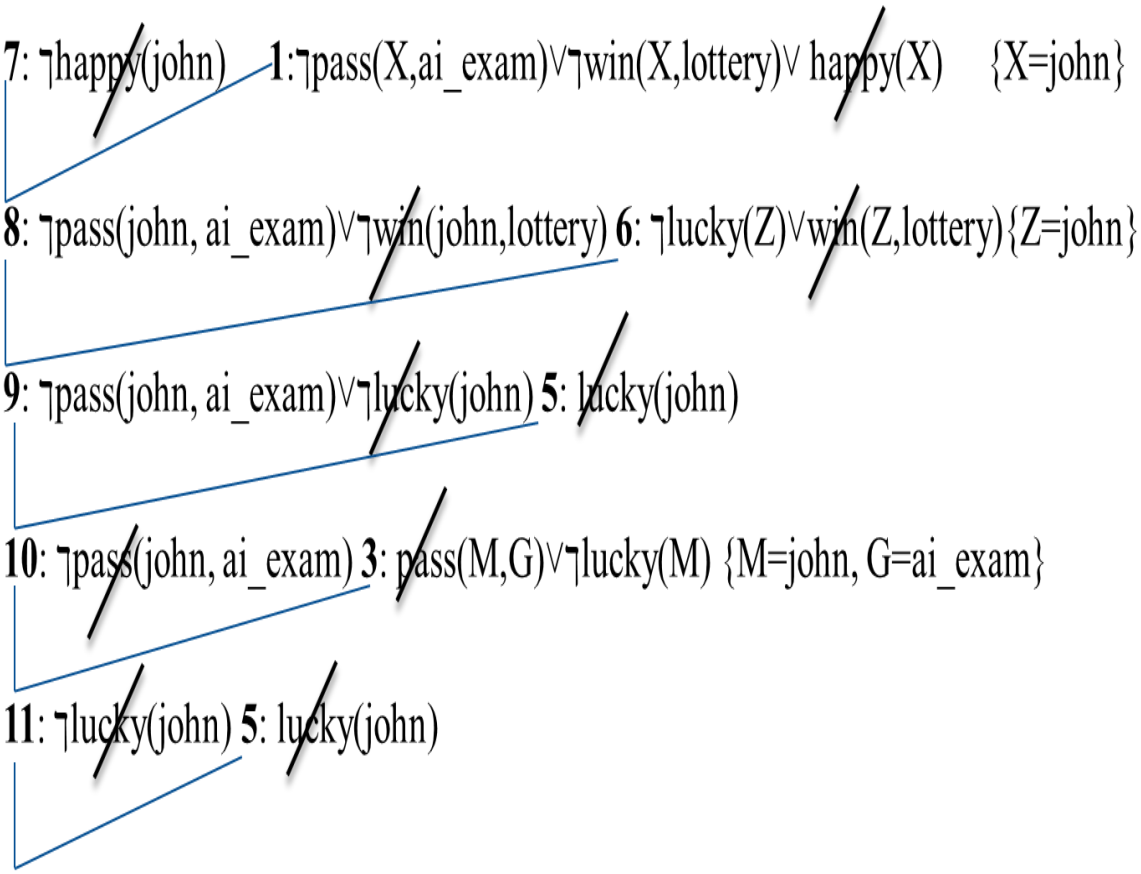
d) Resolve the clauses to producing new clauses and producing a contradiction by generating the empty clause.

There are two ways to do this, the first is *backward resolution* and the second is *forward resolution*.

d_1) Backward Resolution

The proving for ***happy(john)*** using **Backward Resolution** is shown as follows:

1. ¬pass(X,ai_exam)∨¬win(X,lottery)∨happy(X)
2. pass(Y,E)∨¬study(Y)
3. pass(M,G)∨¬lucky(M)
4. ¬study(john)
5. lucky(john).
6.¬lucky(Z)∨win(Z,lottery).
7.¬happy(john).



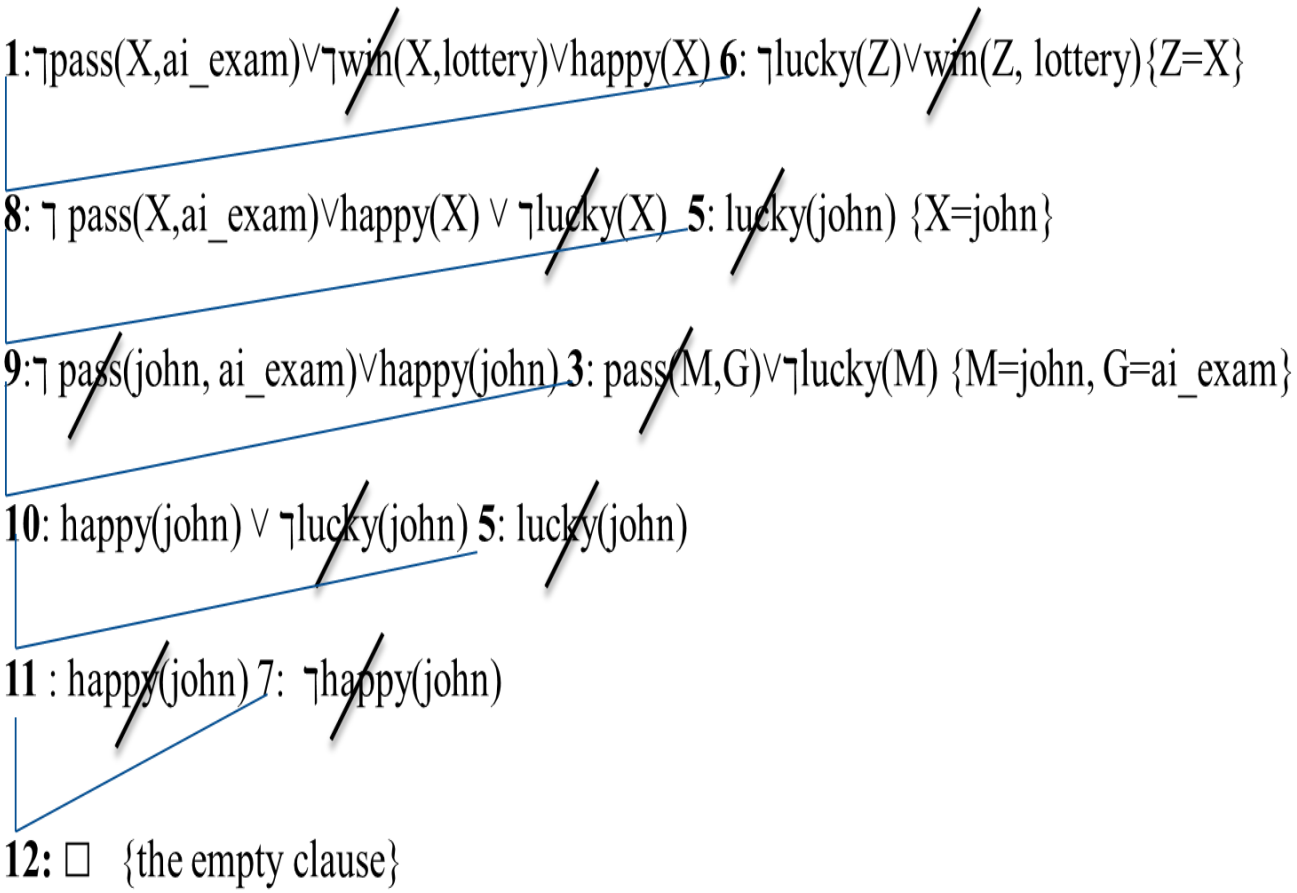
12: □ {the empty clause}

∴ John is happy

d_2) Forward Resolution

The proving for ***happy(john)*** using **Backward Resolution** is shown as follows:

1. ¬pass(X,ai_exam)∨¬win(X,lottery)∨happy(X)
2. pass(Y,E)∨¬study(Y)
3. pass(M,G)∨¬lucky(M)
4. ¬study(john)
5. lucky(john).
6.¬lucky(Z)∨win(Z,lottery).
7.¬happy(john).



∴ John is happy

Homework

Everyone has a parent. The parent of a parent is a grandparent. Prove that Ali has a grandparent using Backward Resolution.

Semantic Nets

The term semantic nets is used to describe a knowledge representation method based on a network structure. Semantic nets were originally developed for use as psychological models of human memory but are now a standard representation method for AI and expert systems.

— It is consist of a set of nodes and arcs , each node is represented as a rectangle to describe the objects, the concepts and the events. The arcs are used to connect the nodes and they divided to three parts:-

— Is a: for objects & types

— Is : To define the object or describe it

— Has a

— can To describe the properties of objects or the actions that the object can do

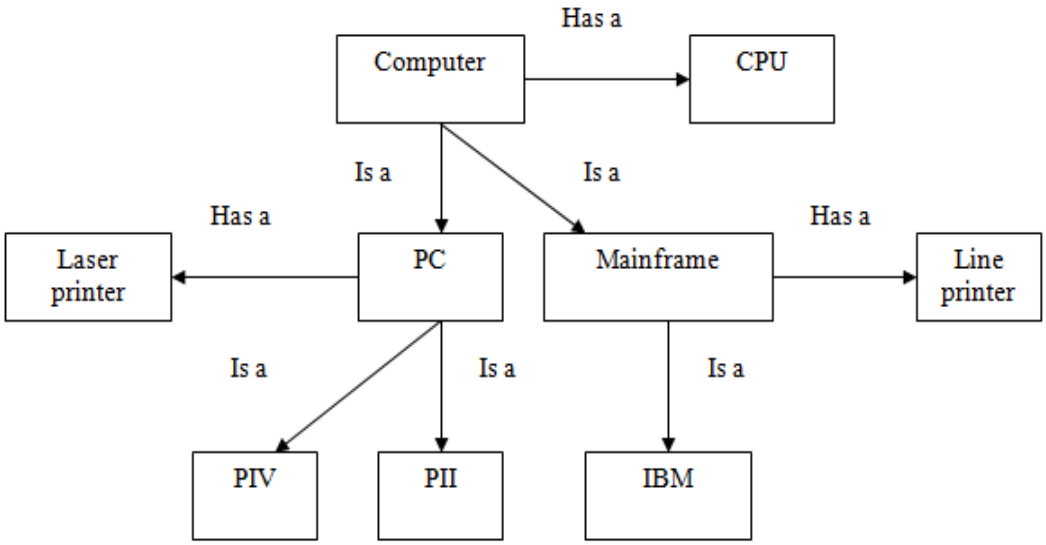
—

To represent the actions, events and objects

To represent the relation among objects

Example1: Computer has many part like a CPU and the computer divided into two type, the first one is the mainframe and the second is the personal computer ,Mainframe has line printer with large sheet but the personal computer has laser printer , IBM as example to the mainframe and PIII and PIV as example to the

personal computer.



— **Example2:** Create the semantic network for the following facts (Note: You must append new indirect facts if they exist):

— A trout is a fish.

— A fish has gills.

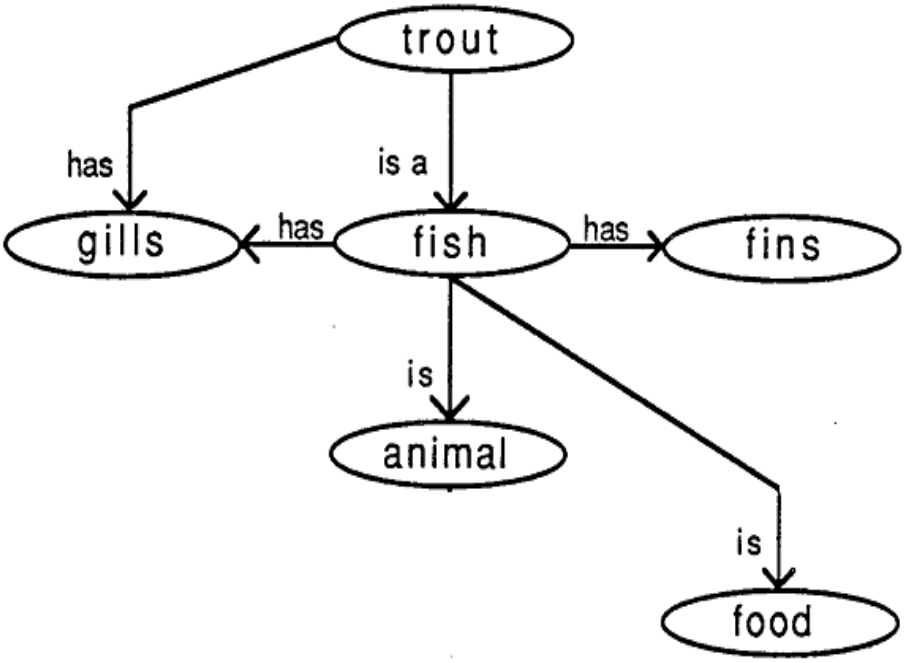
— A fish has fins.

— Fish is food.

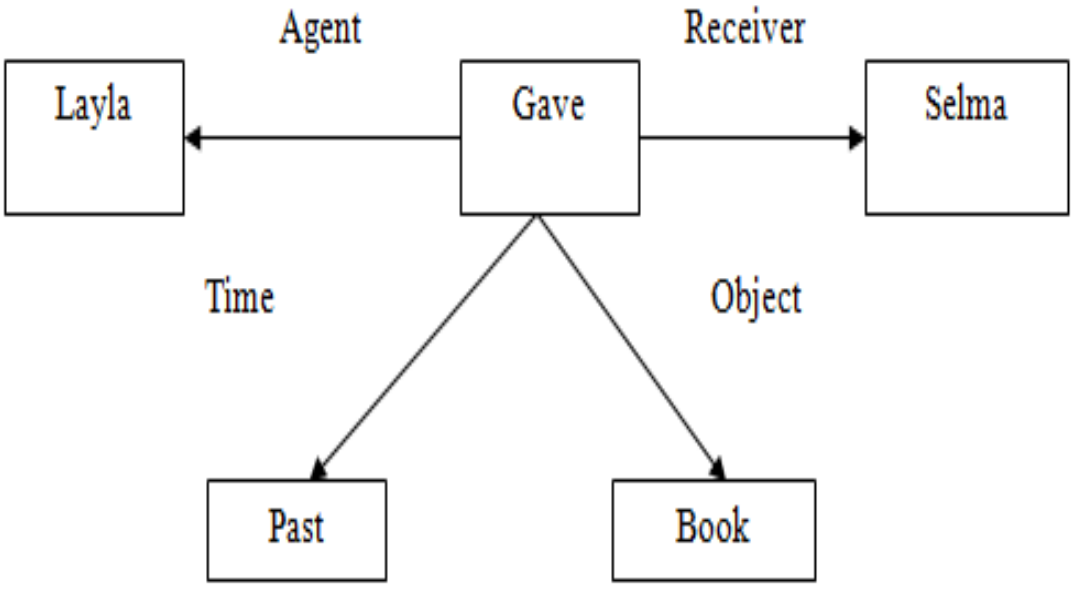
— Fish is animal.

— Solution:

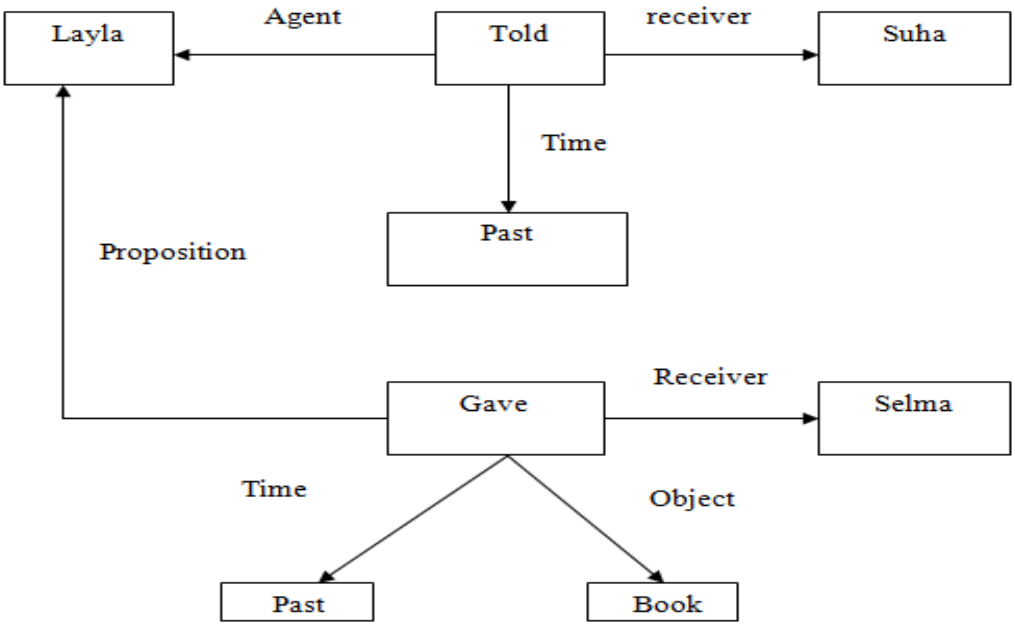
— There is a fact must be added that is “A trout has gills” because all the fishes have gills. The semantic network is shown below:



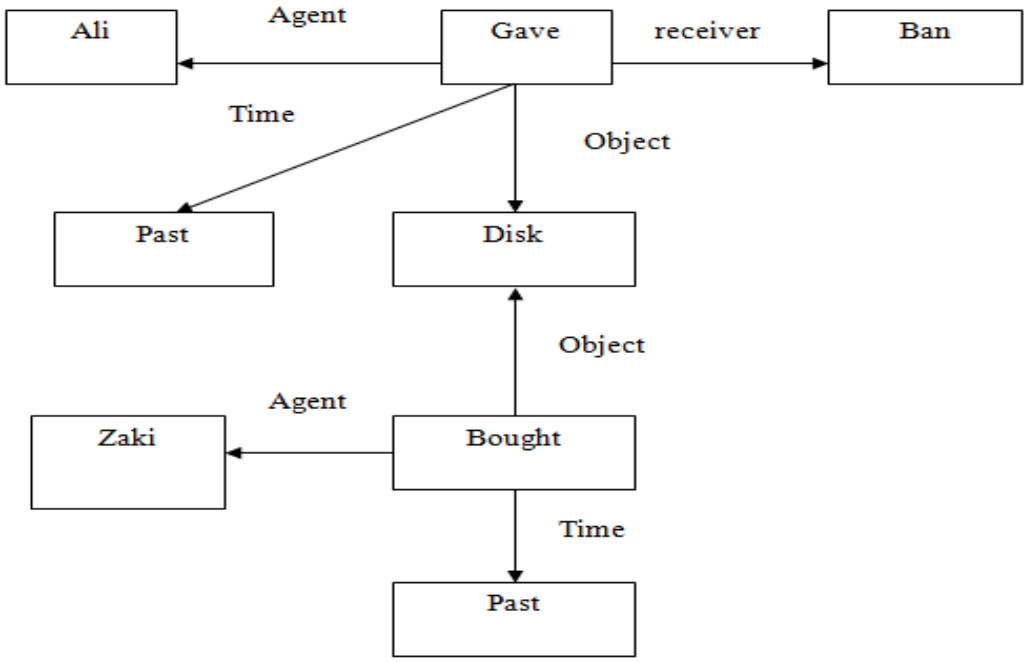
Example 2: Layla gave Selma a book



Example 3: Layla told Suha that she gave Selma a book



Example 4: Ali gave Ban a disk which is Zaki bought



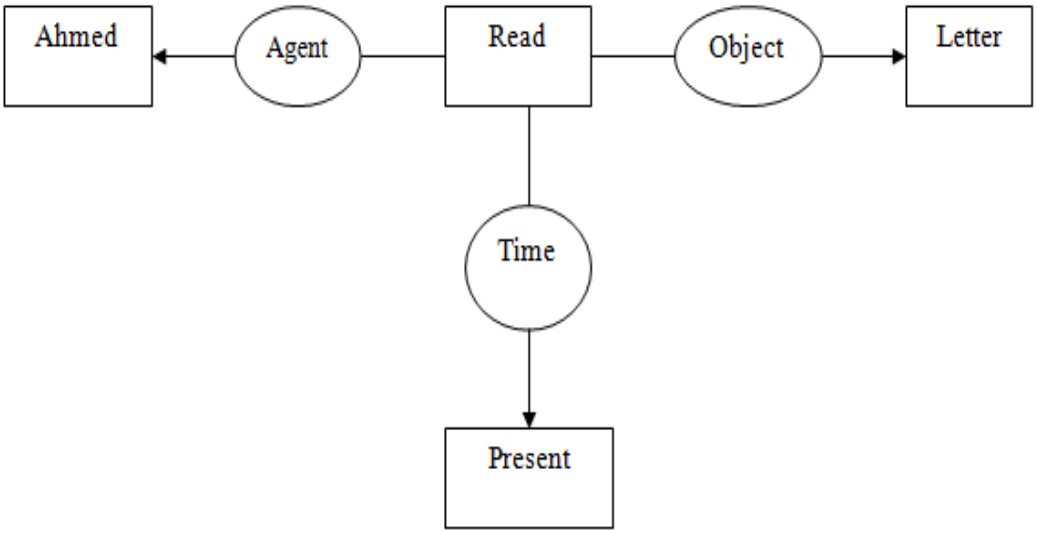
2) The Conceptual Graph

— Conceptual Graphs is a logical formalism that includes classes, relations, individuals and quantifiers. This formalism is based on semantic networks, but it has direct translation to the language of first order

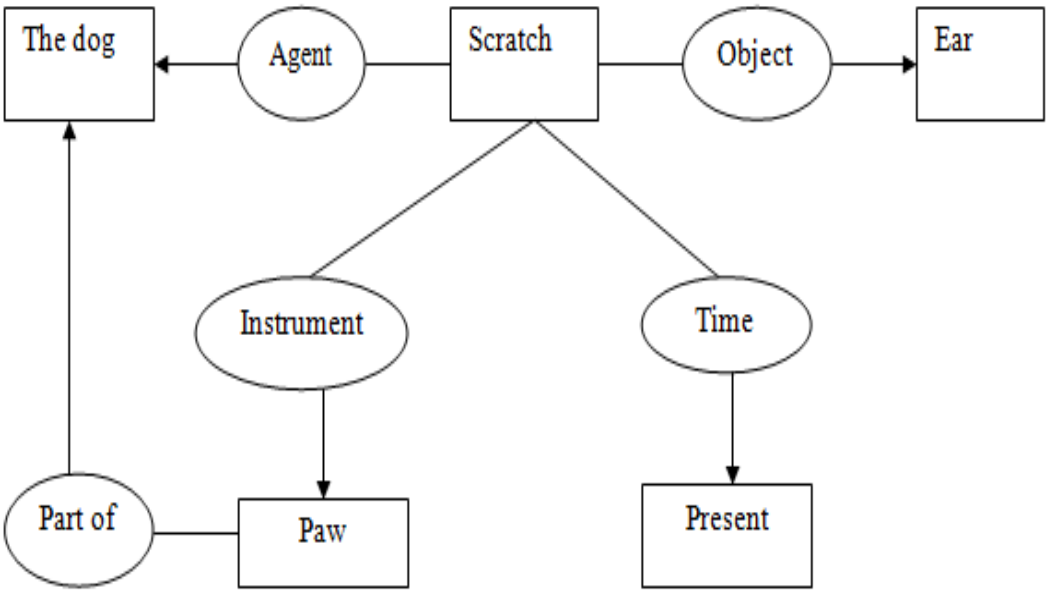
predicate logic, from which it takes its semantics. The main feature is standardized graphical representation that like in the case of semantic networks allows human to get quick overview of what the graph means. Conceptual graph is a bipartite orientated graph where instances of concepts are displayed as *rectangle* and conceptual relations are displayed as *ellipse*. Oriented edges then link these vertices and denote the existence and orientation of relation. A relation can have more than one edges, in which case edges are numbered.

It is similar to semantic net with two parts:
Is used to describe the nouns, the adjectives , the verbs(actions) and the objects.
Is used to represent the relations among objects

Example 1: Ahmed read a letter yesterday



Example 2:- The dog Scratch it ear with is paw



3) Frame:

— Consideration of the use of cases suggests how we can tighten up on the semantic net notation to give something which is more consistent, known as the frame notation. In the place of an arbitrary number of arcs leading from a node there are a fixed number of slots representing attributes of an object.

— Every object is a member or instance of a class, which it may be thought of as linking to with an is_a link as we saw before. The class indicates the number of slots that an object has, and the name of each slot. In the case of a giving object, for instance, the class of giving objects will indicate that it has at least three slots: the donor, the recipient and the gift. There may be further slots indicated as necessary in the class, such as ones to give the time and location of the action. The time slot may be considered a formalization of the tense of the verb in a sentence.

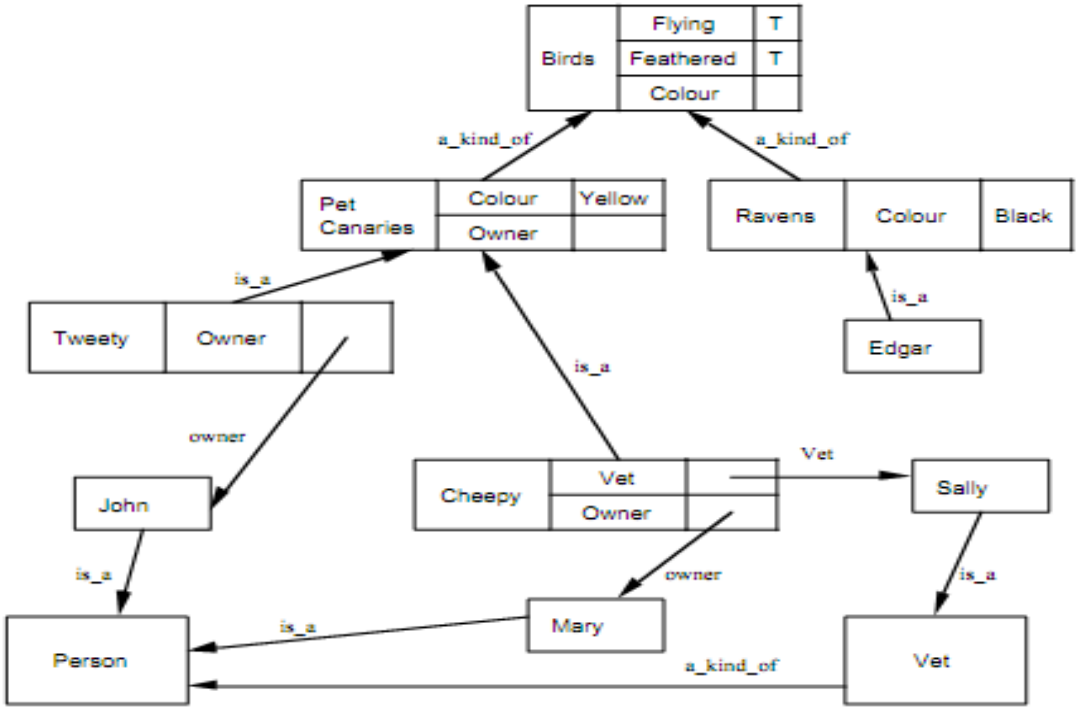
— In our example we have a general class of birds, and all birds have attributes flying, feathered and color. The attributes flying and feathered are Boolean values and are fixed to true at this level, which means that for all birds the attribute flying is true and the attribute feathered is true. The attribute color, though defined at this level is not filled, which means that though all birds have a color, their color varies.

— Two subclasses of birds, pet canaries and ravens are defined. Both have the color slot filled in, pet canaries with yellow, ravens with black. The class pet canaries has an additional slot, owner, meaning that all pet canaries have an owner, though it is not filled at this level since it is obviously not the case that all pet canaries have the same owner.

— We can therefore say that any instance of the class pet_canary has attributes color yellow, feathered true, flying true, and owner any instance of class raven has color black, feathered true, flying true, but no attribute owner.

— The two instances of pet canary shown, Tweety and Cheepy have owners John and Mary who are separate instances of the class person, for simplicity no attributes have been given for class person. , the last of these varying among instances.

— The instance of pet canary Cheepy has an attribute which is restricted to itself, vet (since not all pet canaries have their own vet), which is a link to another person instance, but in this case we have subclass of person, vet. The frame diagram for this is:



— We can define a general set of rules for making inferences on this sort of frame system. We can say that an object is an instance of a class if it is a member of that class, or if it is a member of a class which is a subclass of that class. A class is a subclass of another class if it is a kind of that class, or if it is a kind of some other class which is a subclass of that class. An object has a particular attribute if it has that attribute itself, or if it is an instance of a class that has that attribute. In Prolog:

— *aninstance(Obj,Class) :– is_a(Obj,Class).*

— *aninstance(Obj,Class) :– is_a(Obj,Class1), subclass(Class1,Class).*

— *subclass(Class1,Class2) :– a_kind_of(Class1,Class2).*

— *subclass(Class1,Class2) :– a_kind_of(Class1,Class3), subclass(Class3,Class2).*

— We can then say that an object has a property with a particular value if the object itself has an attribute slot with that value, or it is an instance of a class which has an attribute slot with that value, in Prolog:

— *value(Obj,Property,Value) :– attribute(Obj,Property,Value).*

— *value(Obj,Property,Value):–*

— *aninstance(Obj,Class), attribute(Class,Property,Value).*

— The diagram above is represented by the Prolog facts *attribute(birds,flying,true).*

— *attribute(birds,feathered,true).*

— *attribute(pet_canaries,colour,yellow).*

— *attribute(ravens,colour,black).*

— *attribute(tweety,owner,john).*

— *attribute(cheepy,owner,mary).*

— *attribute(cheepy,vet,sally).*

— *a_kind_of(pet_canaries,birds).*

— *a_kind_of(ravens,birds).*

— *a_kind_of(vet,person).*

— *is_a(edgar,ravens).*

— *is_a(tweety,pet_canaries).*

— *is_a(cheepy,pet_canaries).*

— *is_a(sally,vet).*

— *is_a(john,person).*

— *is_a(mary,person).*

— Note in particular how we have used reification leading to a representation of classes like birds, pet_canaries and so on by object constants, rather than by predicates as would be the case if we represented this situation in straightforward predicate logic. The term superclass may also be used, with X being a superclass of Y whenever Y is a subclass of X.

— Using the Prolog representation, we can ask various queries about the situation represented by the frame system, for example if we made the Prolog query:

— *Goal: value(tweety,colour,V).*

— we would get the response:

— *V = yellow*

— while

— *Goal: value(john,feathered,V).*

— gives the response:

— *no*

—

— This indicating that feathered is not an attribute of John. Note that the no indicates that this is something which is not recorded in the system. If we wanted to actually store the information that persons are not feathered we would have to add:

— *attribute(person,feathered,true).*

— then the response would have been:

— *V = false*

— The only thing that has not been captured in this Prolog representation is the way that an attribute can be defined at one level and filled in lower down, like the colour attribute of birds.

<u>Example:</u> Convert the following statements to frame representation.

— ☒ Birds and fish are animals. Birds are moving by fly and they lay eggs. They have wings and their active at day light. kiwi and alberto are birds. The color of kiwi is black and white and the color of alberto is brown. Fish moves by swimming and has a skin.