Introduction to Artificial Intelligent

1.1 Artificial Intelligent (AI)

Is the study of how to make computers do things which at the moment, people do better . or in specific definition AI is a branch of computer science concerned with the study & creation of computer systems that exhibit some form of intelligence: systems that learn new concepts & tasks ,system that can reason & draw useful conclusion about the world around us, system that can understand a natural language or perceive & comprehend a visual sense ,& system perform other types of fact that require human types of intelligence or in other words we say

AI is the study of mental faculties through the use of computational models

(CHARNAIK & MCDERMORTT,1985)

The use of the term "natural faculties" may make the field sound like of psychology. AI is concerned with working programs whereas psychologists feel more importantly, while AI is concerned with the general behavior that goes with intelligence, it's not committed to any particular way of producing the result.

Dictionary define **intelligence as the ability to acquire, understand & apply knowledge or ability to acquire or the ability to exercise thought & reason**

1.2 Commercial Products of AI

- Robotics device.
- Vision systems that recognize shapes & objects.

- Expert systems that perform difficult task as well as or better than human expert counterpart.
- Intelligent instructions system that help pace student's learning & monitor the student's progress.
- Intelligent editors that assets users in building special knowledge bases.
- System which can learn to improve this performance.

1.3 Importance of AI

- AI may well be one of the most important development of the world. It will affect government & private companies interested in the development of computer products, robotics & related field.
- Japanese realized that many of their goals to produce systems that can converse in a natural language, understand speech & visual sense, learn & refine their knowledge , make decisions, & exhibit other traits can be achieved.
- British initiated a plan called the Alvey project with a reasonable budget.
- France, Canada, Russsia, Italy, Australia, & Singapore have committed to some extent to funded research & development.
- 1983, developed VLSI to use in AI technologies.
- MCC(Microelectronics & computer technology corporation) & is headquartered in Austin, Texas.
- Second DARPA(Defense Advanced Research Projects Agency) has increased its funding for research in AI & supported in three significant programs:-

1. development of an autonomous land vehicle(ALV) a driverless military vehicle.

2. the development of a pilot's associate(an expert system which provides assistance to fighter pilot)
3. the strategic computing program(an AI based military super computer project).

1.4 Goals of AI

the goal of AI is to develop working computer system that truly capable of performing tasks that require high levels of intelligence. The programs are not necessarily meant to imitate human sense & thought processes.
Indeed, in performing some tasks differently, they may actually exceed human capabilities. The important point is that the systems all be capable of performing intelligent tasks effectively & efficient.

1.5 AI Technique

The three important AI techniques are:-

1. search: provides a way of solving programs for which no more direct approach is available as well as a frame work into which any direct techniques that are available can be embedded.
2. use of knowledge:- provides a way of solving complex problems by exploiting the structures of the objects that are involved.
3. abstraction: provides a way of separating important features & variations from the many unimportant ones that would otherwise overwhelm any process.

- the techniques of AI must often require that the problem defined in some specific way, for ex. Breaking a complex decision into a series of simpler sub problems that lead to the final solution.
- The presentation of a problem in a simple, easily process able form then aids in the development of a solution.

- In other words, we say, define the data structure for the problem domain , develop an algorithm for it & solved it with a heuristic evaluation function.
- **Algorithm is a specific set of operations, procedures & decisions which guarantees to yield correct results.(Gloriose & Osorio,1990)**
- Where heuristic is a rule of thumb ,trick, strategy, simplification or any other method that aids the solution of complex problems.
- One of the difference between a heuristic & an algorithm is that while a heuristic generally aids in finding the solution, it does not guarantee an optimal solutions or no a solution at all. However, with an algorithm, one can be sure of finding the correct results.

Introduction to Program in Logic

- imperative language such as c++, Java, pascal, a program is a specification of a sequence of instructions to be executed one after the other by a target machine to solve the problem. The description of the problem is incorporated implicitly in this specification & usually it's not possible to clearly distinguish between the descriptions of the problem & the method used for its solution.
- In logic programming, the description of the problem & the method used for its solution it explicitly separated from each other. This separation has been expressed by R.A.Kowalski in the following equation.

Algorithm = logic + control

- logic in this equation indicate the description component of the algorithm, that is, the description of the problem
- control indicate the component that tries to find a solution taking the description of the problem as a point of departure, the logic component defines what the algorithm is supposed to do , the control indicate how it should be done.
- A specified problem is described in terms of relevant objects & relations between objects , which are then represent in clausal form of logic, a restricted form of first order predicate logic.
- The control component employs logical deduction or reasoning for deriving new facts from the logic program.

Programming in Prolog

- it was designed by A. Colmerauer & P.Roussel at the university of Marseille, influenced by the ideas of R.A.Kowalski in 1970.

- Its simple
- Is a high level logic programming(PROgramming LOGique).
- To interact with the prolog system directly by typing commands directly into the terminal.
- Best for pattern matching & searching.
- Excellent for language processing, rule based expert system planning & other AI application.
- Use depth first search & backtracking to search for solution automatically.
- Best written in a little chunks(modular code): indeed this is assumed in its syntax.

Algorithem = logic + control

What how

Horn clause resolution

Prolog DB prolog Interpreter

The relationship between prolog & logic programming

- A program consist of
  1. facts
  2. rules
  3. questions or query

Prolog Basics

Main Concept

- **Fact:** assert some property of an object or states some relation between two or more objects. Is made up of a predicate which states the relation or property of a number of arguments (which are the objects).
- **Ex.**

Alan Likes coffee.	In English
likes(alan,coffee).	in prolog with two arguments
listing.	in prolog with zero argument

likes(alan, coffee).

Predicate arguments

Ex2.

Mary is female	in English
female(mary)	in prolog

DB is a collection of facts in prolog.

Predicate: is something that asserts a fact about one or more entities.

- **Note** : predicate should always be the verb as it can then apply to many subjects & objects. The subject should always the first argument , while object the second argument.
- Questions:- if a question is asked by the user, the prolog interpreter looks at the DB of facts that's if the information is enough to answer
- Ex

Does Alan like coffee?

Who drinks tea?

1. first, prolog finds a fact that matches the predicate in the question.
2. if this match succeeds , prolog then matches the first argument to the predicate
3. if this match succeeds, prolog matches the second argument & so on for the rest argument
4. if the match fails at any point, prolog looks for the next assertion that the predicate matches & tries again to match the arguments
5. if the predicate, & all argument are successfully matched, the process stops the interpreter printed yes, otherwise the goal not satisfied.

Variables

***What does Alan like?**

What: begins with a capital letter, this is indicate it's a variable. Its placeholder that can take on any value through instantiation.

alan coffee tea….

These do not change, they always represent the same object & it is called **a constant**.

Note:-

- **Term** is used to refer to any data object in prolog, there are four types of terms: atoms, variables, constant & compound terms.

Atoms & numbers sometimes are grouped together and called atomic terms.

- A constant, variable is a term

- **Constants** can be (atoms, integers, real numbers)
- **Atoms** are usually string made of lower &uppercase letter,digits, & the underscores, **starting with lower case letter.**

elephant, abXY, x_123, my_College

Also any series of character enclosed with single qout are also atom.
Also special char like + - : */<> … are also atom.

The declarative semantics Information (facts, rules, and queries) is represented in PROLOG using the formalism of **Horn clause logic**. A Horn clause takes the following form:

$B \leftarrow A1,\ldots,An$

where B, $A_1,\ldots A_n$, $n \geq 0$, are atomic formulas. Instead of the (reverse) implication symbol,

in PROLOG usually the symbol :- is used, and clauses are terminated by a dot. An atomic formula is an expression of the following form:

$P(t_1,\ldots, t_m)$

Formal	name	In prolog	name
$A\leftarrow$	Unit clause	A.	fact
$\leftarrow B_1,\ldots,B_n$	Goal clause	?-$B_1,\ldots,B_n$.	query
$A_1 \leftarrow B_1,\ldots,B_n$	Clause	A:-$B_1,\ldots,B_n$.	rule

Table 1: Horn clauses and PROLOG

where P is a predicate having m arguments, $m \geq 0$, and $t_1,\ldots, t_m$ are terms. A term is either a constant, a variable, or a function of terms. In PROLOG two types of constants are distinguished: numeric constants, called numbers, and symbolic constants, called atoms.

(Note that the word atom is used here in a meaning differing from that of atomic formula, thus deviating from the standard terminology of

predicate logic.) Because of the syntactic similarity of predicates and functions, both are called functors in PROLOG. The terms of a
functor are called its arguments. The arguments of a functors are enclosed in parentheses, and separated by commas.

Seen in the light of the discussion from the previous section, the predicate P in the atomic formula $P(t_1,\ldots, t_m)$ is interpreted as the name of the relationship that holds between the objects t1,…, tm which occur as the arguments of P. So, in a Horn clause B :- $A_1,\ldots,A_n$,
the atomic formulas B, $A_1,\ldots,A_n$, denote relations between objects. A Horn clause now is interpreted as stating:

`B (is true) if A_1 and A_2 and … and A_n (are true)'

$A_1,\ldots,A_n$ are called the conditions of the clause, and B its conclusion. The commas between the conditions are interpreted as the logical ^, and the :- symbol as the (reverse) logical implication .

If n = 0, that is, if conditions A_i are lacking in the clause, then there are no conditions for the conclusion to be satisfied, and the clause is said to be a fact. In case the clause is a fact, the :- sign is replaced by a dot.

Both terminology and notation in PROLOG differ slightly from those employed in logic programming. Table 1 summarizes the differences and similarities.

Syntax of the prolog program

- All predicate start with a lower case letter.
- All variable start with upper case letter.
- The format of each fact or assertion is:-
  - A predicate followed by any number of arguments.
  - The argument are separated by comma and round by closed brackets.
  - There is no space between predicate & open bracket.
  - A full stop followed closed bracket.
- the predicate can be string like son_of, drinks, likes, & so on.
- The argument can be constant, variable, even other assertion.

Rules

- Prolog rules have a left side & right side ,& the symbol :- is between.
- Left side is a single predicate expression.
- Right side is a query, with possibly multiple predicate expression combined with the comma "and" ,semicolon "or, & "not" symbol.

Ex:- write the prolog program for the rule down.

```
gray_ship(x):-part_of(navy,X),
          color(X,gray).
```

Yes if right side succeeds

No if right side fails.

X is local var. , it value will be thrown a way when the rule is done.

Ex2:- write a prolog program for the rule below.

```
color_object(X,C):-part_of(X,Y),
                   color(Y,C).
```

X & C parameter var.s values for X & C will be returned.

Rules in Natural Language

Ex:-

If a vehicle floats on water, then it's a ship.

```
ship(X):-vehicle(X, floats(X,water).
```

If a vehicle floats on water , it's ship.

Define a ship as anything that floats on water.

Assume as a ship anything that floats on water.

A ship is any vehicle that floats on water ,ships are water-floating vehicle.

Something that is a vehicle and floats on water is a ship.

"postponed" binding of variables.

?-color_object(ship_1,C).

color_object(ship_1,green).

Means c=green

Binding is only done when truly necessary to answer a query.

Ex3:- write a prolog program to find the sister relation.

Domain

```
        X,Y,F=string.
Predicates
        femal(X).
        parents(X,X).
Clauses
        female(nada).
        female(suha).
        female(muna).
        parents(ahmed,nada).
        parents(sami,muna).
        parents(ahmed,suha).
sister_of(X,Y):-female(X),
                parents(M,X),
                parents(M,Y).
```

Ex4:-

X is a bird if X is an animal, and X has feathers.

```
Domains
X=string.
Predicates
        animal(X).
        has_feather(X).
Clauses
        animal(duck).
        animal(hen).
        animal(sparrow).
        has_feather(duck).
        has_feather(hen).
        has_feather(sparrow).
```

bird(X):-animal(X),has_feather(X).

ex

A man is happy if he is rich and he is famous.

happy(Person):- man(Person),
rich(Person),
famous(Person).

Ex

Someone is happy if they are healthy or

Someone is happy if they are wealthy or

Someone is happy if they are wise.

happy(Person):- healthy(Person).

happy(Person):-wealthy(Person).

happy(Person):-wise(Person).

Recursive Definition

When defining some thing , we can use the same thing that has not yet been completely defined.

- Prolog uses recursive definition very easily.
- Recursive programming is one of the fundamental principles of programming in prolog.

?-predecessor(pam,X)

X=bob;

X=ann;

X=pat;

X=jim;

predecessor(X,Z):-parent(X,Y1),parent(Y1,Y2),parent(Y2,Z). indirect

For all X and Z

1. X is a parent of y and
2. Y is predecessor of z

predecessor(X,Z):-parent(X,Y),
predecessor(Y,Z).

general rules for matching two terms S & T

1. if S & T are constant then S & T match only if they are the same object.
2. if S a var. & T is anything, then they match, and s is instantiated to T. conversely, if T is a var. then T is instantiated to s.
3. if s and T are structures they match only if

a. S & T have the same principle functor , and
b. All their corresponding components match.

The resulting instantiation is determined by the matching of the components.

Ex:-

?-date(D,M,1983)=date(D1,may,Y1),
date(D,M,1983)=date(15,M,Y).

First goal:

D=D1

M=may

Y1=1983

Second goal

D=15

D1=15

M=may

Y1=1983

Y=1983

Ex2:-

parent(tom,bob).
parent(pam,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).

Clauses of facts

?-parent(bob,pat).

Yes

?-parent(tom,john).

No

?-parent(bob,jim).

Yes

?-parent(X,bob).

X=tom;

X=pam

Ex. Write prolog program to satisfy this sentence.

You talk about someone if you know them or you know someone who talks about them

Solution

```
Domains
A,B,P,R,Q=string
Predicates
        talks_about(A,A)
        knows(A,A)
Clauses
knows(bill,jane).
knows(jane,pat).
knows(jane,fred).
knows(fred,bill).
talks_about(A,B):-
                knows(A,B).
talks_about(P,R):-
                knows(P,Q),
                talks_about(Q,R).
```

H.W

write a program for the following sentence

some body has flu if he infected with flu or he kisses person who has flu.

```
has_flu(X):-infected(X).
has_flu(X):-kisses(X,Y),has_flu(Y).
```

Built in Predicates or System Predicate

A predicate is a collection of clauses with the same predicate name & the number f arguments, the number of argument of a predicate is the arity of it.

- The partition of the program below represents knowledge about books , their publisher & the shops that stock these publishers.

```
stocks(james,).
askbook:-
        write('what book would you like to buy'),nl,
        read(book),nl,
        canbuy(Book,Shop),
        write('you can buy'),
        write(Book),
        Write('at'),
        Write(Shop),
        Write('.').
canbuy(Book,Shop):-
                book(Book,Publisher),
                stocks(Shop,Publisher),
                open(Shop).
askbook:-
        write(' I don't know where you can buy that book, sorry.').
        go:-
        askbook, nl,
        write('would you like another book?'), nl,
        read(Reply),nl,
check(Reply),nl.
check(Reply):_
```

```
        Reply = yes,
        go.
check(Reply):-
        write('I hope I Was of some help to you have a nice day').
```

Arithmetic operator

Arithmetic operator are another type of system predicat which enables us to do arithmetic in prolog

+ - * /

There is also a system that the results of applying these operators called is/2

All arithmetic operator can be used as infix (between argument) or prefix (like predicate).

Ex.

```
?-x is 3+7.
X=10
?-B is +(2,99).
B=101
?-4=3+1
No
?- 4 is 3+1
Yes
?- s is H + 2
Error: un instantiated var. in arithmetic expression
No
```

Not:- is/2 means evaluate to= means will unify with.

```
?- A is 3+7.
A=10
?- B is +(2,5).
```

B=7

?-Bis 3+3.

No

?-3 is 2 +1.

Yes

?- 4 is 4.

Yes

?- 3 is 2 +1.

Yes

?-2+1 is 3.

No

?- D is H+2.

Error

?- A is 3+3, B is A+2.

A=6

B=8

?- T is +(-(3,1),*(6,4)).

T=26

Yes

?- B is <(+(-(3,1),*(6,4)),+(*(7,8),-(8,10))).

(3-1)+(6*4) < (7*8)+(8-10)

2 +24 < 56+(-2)

26 < 54

Yes

Backtracking

cut

Prolog provides a predicate that perform this function. It's called cut which represented by ! (exclamation point). The cut effectively tells prolog to freeze the decision mode in this predicate. That is, if required to backtrack, it will automatically fail without trying other alternatives.

Ex.

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

o/p

1 solution X=4 , Y=2

ex2:-

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

Goal:- d(X,Y)

x=1 ,y=2 ,2=4 fail

x=1 ,y=3 ,1=4 fail

o/p no solution found

Fail

the fail predicate is provided when it's called, it causes the failure of the rule and this will be forever, nothing can change the statement of this predicate.

Ex.

x(1).

x(2).

x(3).

loop:-x(A), write(A),fail.

Goal:-loop

Output:1 2 3 No

Note:-

To report back answer, we can put an un instantiated var in the query, then instantiated the answer to that var when the query succeeds. Lastly, pass the var all the way back to the query.

Ex.

bigger_than(X,Y,Big):- X⟩Y,Big=X.

bigger_than(X,Y,Big):-X≤Y,Big=Y.

Goal:-bigger_than(5,2,Big),bigger_than(Big,4,Newbig).

Output Newbig =5

Built in mathematical function

Prolog has a full rang of built in mathematical function & predicates that operate on integer & real values.

Function	**Description**
X mod Y	return the remainder (modules) of x divided by Y
X div Y	return the quotient of X divided by Y
abs(X)	return the absolute value of X
cos(x)	return the cosine value of angle X in radians
sin(x)	return the sin value of X
tan(X)	return the tan value of X
arctan(X)	return the arc tan of X
exp(X)	return the exponential of x
ln(X)	logarithm of X with base e
log(X)	logarithm of X base 10
sqrt(X)	square root of X
random(X)	randomize of X, $0 \leq X \langle 1$
random(X,Y)	randomize y less than X, $0 \leq Y \langle X$
round(X)	return the rounded value of X
trunk(X)	truncate X

Arithmetic Comparison Operators

The arithmetic comparison operators are :<=<>>==:=/=

X<Y

True if X is less than Y.

X=<Y

True if X is less than or equal to Y.

X>Y

True if X is greater than Y.

X>=

True if X is greater than or equal to Y.

X=:=Y

True if X is equal to Y.

X=/=Y

True if the values of X and Y are not equal

Unlike unification theses operators cannot be used to give values to a variable. The can only be evaluated when every term on each side have been instantiated.

List

Lists are powerful data structures for holding and manipulating groups of things. In Prolog, a list is simply a collection of terms. The terms can be any Prolog data types,including structures and other lists.

list is denoted by square brackets with the terms separated by commas.

This gives us an alternative way of representing the locations of things. Rather than having separate location predicates for each thing, we can have one location predicate per container, with a list of things in the container.

The empty list is represented by a set of empty brackets []. This is equivalent to the nil in other programming language.

For our example in this section, it can describe the lack of things in a place :

```
list_where([], cave).
```

The Unification works on lists .

```
?- [_,_,X] = [lesson, work, sleeping].
X = sleeping
```

At last, the special notation for list structures.

```
[X | Y]
```

This structure is unified with a list, X is bound to the first element of the list, called the head. Y is bound to the list of remaining elements, called the tail.

Note that the tail is considered as a list for Prolog and the empty list does not unify with the standard list syntax because it has no head. Here is an example :

```
?- [X|Y] = [a, b, c, d, e].
X = a
Y = [b, c, d, e]

?- [X|Y] = [].
no
```

The empty list does not unify with the standard list syntax because it has no head.

```
?- [X|Y] = [].
no
```

This failure is important, because it is often used to test for the boundary condition in a recursive routine. That is, as long as there are elements in the list, a unification with the [X|Y] pattern will succeed. When there are no elements in the list, that unification fails, indicating that the boundary condition applies. We can specify more than just the first element before the bar (|). In fact, the only rule is that what follows it should be a list.

How to manipulate list

For lists to be useful, there must be easy way to access, add, and delete list elements. Moreover, we should not have to concern ourselves about the number of list items, or their order.

In Prolog features enable to accomplish this easy access. One is a special notation that allows reference to the first element of a list and the list of remaining elements, and the other is recursion.

The first one we will look at is member. As with most recursive predicates, we will start with the boundary condition, or the simple case. An element is a member of a list if it is the head of the list.

```
member(T,[T|Q]).
```

This clause also illustrates how a fact with variable arguments acts as a rule. The second clause of member is the recursive rule. It says an element is a member of a list if it is a member of the tail of the list.

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

As with many Prolog predicates, member can be used in multiple ways. If the first argument is a variable, member will, on backtracking, generate all of the terms in a given list.

```
?- membre(X, [baghdad, babylon, arbil]).
X = baghdad;
X = babylon;
X = arbil;
```

Another very useful list predicate builds lists from other lists or alternatively splits lists into separate pieces. This predicate is usually called append. In this predicate the second argument is appended to the first argument to yield the third argument. For example

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

It is a little more difficult to follow, since the basic strategy of working from the head of the list does not fit nicely with the problem of adding something to the end of a list. append solves this problem by reducing the first list recursively. The boundary condition states that if a list X is appended to the empty list, the resulting list is also X.

append([],X,X).
The recursive condition states that if list X is appended to list [T|Q1], then the head of the new list is also H, and the tail of the new list is the result of appending X to the tail of the first list.

```
append([T|Q1],X,[T|Q2]) :- append(Q1,X,Q2).
```

If we want to print the elements of list you can use the following rule.

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

Ex.

A list of integer can be shown as a bar graph for ex.

```
?-bars[3,4,6,5].
***
****
******
*****
bars([]).
bars([N|L]):-
        stars(N),nl,
        bars(L).
stars(N):-
        X>0,
        write(*),
        N1 is N-1,
        stars(N1).
stars(N):-  N<=0,!.
```

Ex2. Add an element to a list without duplicate

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

Ex3. Find the length of list

```
length([ ],0).
length([-|Tail],N):-
        length(Tail, N1),
        N is 1+N1.
```

Ex4.

Delete an item.

```
del(X, [X|Tail], Tail).
del(X,[Y|Tail], [Y|Tail1]):-
                del(X,Tail,Tail1).
```

Note1

List	**head**	**tail**
[a]	a	[ ]
[a,b,c,d]	a	[b,c,d]
[ ]	fails	fails
[[the,cat],sat]	[the,cat]	[sat]
[the,cat]	the	[cat]
[the,[cat,sat]]	the	[[cat,sat]]
[the,[cat,sat],down]	the	[[cat,sat],down]

Note 2

List1	**list2**	**result**
[X,Y/Z]	[mary,likes,tea]	X=mary, Y=likes, Z=[tea]
[[the,Y],/Z]	[[X,hare],[is,here]]	X=the,Y=hare,Z=[[is,here]]
[vale,horse]	[hores,X]	fails
[X,Y/Z,W]		incorrect syntax

4.1 Lists in Prolog

Lists is a simple data structure widely used in non –numeric programming . is a sequence of any number of items.

- A list is either empty, or it is a structure that has two components: the head **H** and tail **T**. tail itself has to be a list.
- List notation consists of the elements of the list separated by commas, and the whole list is enclosed in square brackets.
- List are handled in prolog as a special case of binary trees.
- Lists are written as

[Item1,Item2,…]

Or

[Head| Tail]

Or
[Item1,Item2,…,|Others]

Some operation on lists

List can be used to represent sets, there is a difference :the order of elements in a set does not matter the order of items in a list does, also , the same object can occur repeatedly in a list . still, the most common operations on the lists are similar to those on sets.

- Checking whether some object is in an element of a list ,which corresponds to checking for the set membership
- Concatenation of two list, obtained a third list, which may be correspond to the union of sets
- Adding a new object to a list or deleting some object from it.

Membership

membership(X,L)
Where X is an object & L is a list. The goal member(X,L) is true if X occurs in L.

X is a member of L if either

1. X is the Head of L or
2. X is a element of the tail of L.

This can be written as

```
member(X,[X| Tail]).
member(X,[Head| Tail]:-
    Member(X, Tail).
```

Concatenation

Conc(L1,L2,L3)

L1,L2 are two lists, & L3 is their concatenation . for ex.

conc([a,b],[c,d],[a,b,c,d])
Is true , but
conc([a,b],[c,d],[a,b,a,c,d]) is false.
In addition of conc, we will have again two cases depending on the first argument, L1

1. if the first argument is empty list then the second & third argument must be the same list (call it L);

```
conc([ ],L,L).
```

if the first argument of conc is non-empty list then it has a head & a tail & must look like this:

```
conc([ ],L,L).
conc([X|L1],L2,[X|L3]):-
     conc(L1,L2,L3).
```

Ex.
Conc([a,b,c],[1,2,3],L).
L=[a,b,c,1,2,3]

Concatenation of lists

Ex.
?-conc(L1, L2, [a,b,c]).
L1=[]
L2=[a,b,c];

L1=[a]
L2=[b,c];

L1=[a,b]
L2=[c];

L1=[a,b,c]
L2=[];

No

?-conc(Before,[may|After],
[jan, feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec]).
Before=[jan,feb,mar,apr]
After=[jun,jul,aug,sep,oct,nov,dec].

Ex.
?-conc(_,[Month1,may,Month2|_],
[jan,feb,mar,…..,dec]).
Month1=apr.
Month2=jun.

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].

Ex.
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]].

Ex.
p([a]).
Goal: p([H | T]).
Output:
H = a, T = [].

Ex.
p([a, b, c, d]).
Goal: p([X, Y | T]).
Output:
X = a, Y = b, T = [c, d].

Ex

Print the contents of the list.

```
print([]).
print([(X,Y)|T]):-write(X,Y),print(T).
```
Goal: print([3,4,5])
Output: 3 4 5

ex.

A list of integer can be shown as a bar graph for ex.

```
?-bars[3,4,6,5].
***
****
******
*****
bars([]).
bars([N|L]):-
        stars(N),nl,
        bars(L).
stars(N):-
        X>0,
        write(*),
        N1 is N-1,
        stars(N1).
stars(N):-  N<=0,!.
```

Ex2. Add an element to a list without duplicate

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

Ex. Find the length of list

```
length([ ],0).
length([-|Tail],N):-
        length(Tail, N1),
        N is 1+N1.
```

Ex.

Delete an item.

```
del(X, [X|Tail], Tail).
del(X,[Y|Tail], [Y|Tail1]):-
                del(X,Tail,Tail1).
```

Various kinds of equality & comparison

1. use =

ex. X=Y

2. X is E

Is true if X match the value of arithmetic expression E

3. E1=:=E2 is true if the value of arithmetic expression E1 & E2 are equal
4. E1=\=E2 is true if not equal
5. T1==T2 is true if terms T1 & T2 are identical(they exactly the same structure & all the corresponding component the same) in particular ,the name of the variable also have to be the same.
6. T1\==T2 is true if not identical

Ex.

f(a,b)==f(a,b).

Yes

Ex. f(a,X)==f(a,Y).

No

Ex. f(a,b)==f(a,Y).

X\==Y.

Yes

Ex. t(X,f(a,Y))==t(X,f(a,Y))

yes

- A db can be naturally represented as a set of facts for ex.a clause in fig below show how the information about each family can be structured.
- Each family has three component ,husband,wife,child . as the no. of child is vary from family to other then it represented by a list. Each person represented by four components.

Ex.

```
family(
        person(tom,fox,date(7,may,1960),works(bbc,15200)),
        person(ann,fox,date(9,may,1981),unemployed),
        [person(pat,fox,date(5,may,1983),unemployed),
         person(jim,fox,date(5,may,1983),unemployed)]).
```

Ex.

```
family(person(_,Armstrong,_,_),_,_).
```

Names all Armstrong families

_ underscores denote different anonymous variable we don't care about their value

Ex.

family(_,_,[_,_,_])

Family with three childs

Ex.

To find all married women that at least 3 child

?-family(_,person(Name,Surname,_,_),[_,_,_|_]).

- We can provide a set of procedures that can be serve as a utility to make interaction with the db

```
husband(X):-
                family(X,_,_).

wife(X):-family(_,X,_).

child(X):- family(_,_,X).

exists(Person):-
                husband(Person)
                ;
                wife(Person)
                ;
                child(X).
dateofbirth(person(_,_,Date,_),Date).
salary(person(_,_,_, works(_,S),S).
salary(person,(_,_,_,  unemployed),0).
```

- Find the names of all people in the db.

?- exists(person(Name, Surname, _, _)).

- Find all child born in 2000.
- ?- child(X),dateofbirth(X, date(_,_,2000).
- Find people born before 1960 whose salary is less than 8000.

?- exists(Person),
dateofbirth(Person, date(_,_, Year)),
year < 1960,
salary(Person, Salary),
Salary < 8000.