PARADIGMS AND COMPUTER PROGRAMMING FUNDAMENTALS (PCPF)

ITC305

2022-23

Subject In-charge

Ms. Aaysha Shaikh

Professor Dept. of Information Technology

SFIT Room No. 322

email: aayshashaikh@sfit.ac.in

Module 4

Logic Programming with PROLOG -

Resolution and Unification, Lists, Arithmetic execution order, imperative control flow, database manipulation, PROLOG facilities and deficiencies.

Self-Learning Topic: Identification of different application domains for use of Prolog and Logic programming

- Prolog or PROgramming in LOGics is a logical and declarative programming language. • It is one major example of the fourth generation language that supports the declarative programming paradigm.
- This is particularly suitable for programs that involve symbolic or non-numeric computation.

- Some logic programming languages like Datalog or ASP (Answer Set Programming) are known as purely declarative languages.
- However, Prolog, have declarative and imperative properties. This may also include procedural statements like “To solve the problem H, perform B1, B2 and B3”.

Prolog

• Logic Programming is one of the Computer Programming Paradigm, in which the program statements express the facts and rules about different problems within a system of formal logic. • Here, the rules are written in the form of logical clauses, where head and body are present. • For example, H is head and B1, B2, B3 are the elements of the body. Now if we state that “H is true, when B1, B2, B3 all are true”, this is a rule.

- On the other hand, facts are like the rules, but without any body. So, an example of fact is “H is true”.

Some logic programming languages

- Some logic programming languages

are given below – • ALF (algebraic

logic functional programming
language). • ASP (Answer Set
Programming)

- CycL
- Datalog
- FuzzyCLIPS
- Janus
- Parlog
- Prolog
- Prolog++
- ROOP

Logic and Functional Programming

- for the Logic Programming, we will provide knowledge base. Using this knowledge base, the machine can find answers to the given questions, which is totally different from functional programming.
- In functional programming, we must mention how one problem can be solved, but in logic programming we must specify for which problem we want the solution. Then the logic programming automatically finds a suitable solution that will help us solve that specific problem.

What is Prolog?

- This is particularly suitable for programs that involve symbolic or non-numeric computation. This is the main reason to use Prolog as the programming language in Artificial Intelligence, where symbol manipulation and inference manipulation are the fundamental tasks. Prolog language basically has three different elements

- Facts – The fact is predicate that is true, for example, if we say, “Tom is the son of Jack”, then this is a fact.
- Rules – Rules are extinctions of facts that contain conditional clauses. To satisfy a rule these conditions should be met. For example, if we define a rule as –

Eg:grandfather(X, Y) :- father(X, Z), parent(Z, Y)

This implies that for X to be the grandfather of Y, Z should be a parent of Y and X should be father of Z.

- Questions – And to run a prolog program, we need some questions, and those questions can be answered by the given facts and rules.

Some Applications of Prolog

- Prolog is used in various domains. It plays a vital role in automation system. Following are some other important fields where Prolog is used –
- Intelligent Database Retrieval
- Natural Language Understanding
- Specification Language
- Machine Learning
- Robot Planning
- Automation System
- Problem Solving

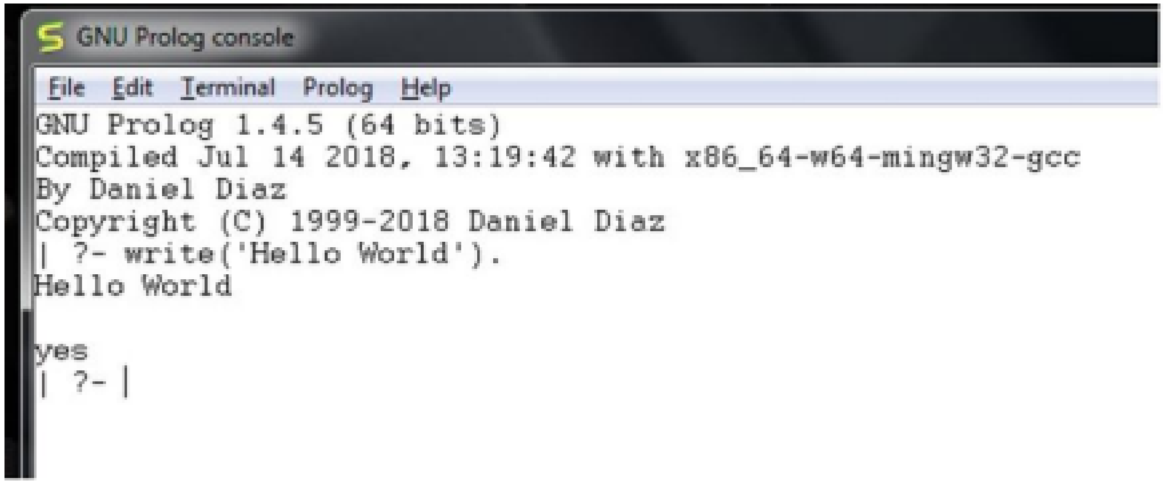
Module-4:Prolog

- **How to download**
  - https://www.swi-prolog.org/download/stable/bin/swipl-8.4.3-1.x64.exe.envelope

Hello World Program

we can write hello world program directly from the console. To do so, we have to write the command as follows –

- write('Hello World’).
- Note – After each line, you have to use one period (.) symbol to show that the line has ended.



```
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 13:19:42 with x86_64-w64-mingw32-gcc
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
| ?- write('Hello World').
Hello World

yes
| ?- |
```

Knowledge Base

- Knowledge Base – This is one of the fundamental parts of Logic Programming. We will see in detail about the Knowledge Base, and how it helps in logic programming
  .

Facts

- We can define fact as an explicit relationship between objects, and properties these objects might have. So facts are unconditionally true in nature. Suppose we have some facts as given below –
- Tom is a cat
- Kunal loves to eat Pasta
- Hair is black
- Kunal loves to play games

• Pratyusha is lazy.

Facts

Example

Following are some guidelines to write facts –

• Names of properties/relationships begin with lower case letters.

• The relationship name appears as the first term.

• Objects appear as comma-separated arguments within parentheses.

• A period "." must end a fact.

• Objects also begin with lower case letters. They also can begin with digits (like 1234), and can be strings of characters enclosed in quotes

- e.g. color(penink, ‘red’).
- phoneno(agnibha, 1122334455). is also called a predicate or clause.

Example of Facts

- Example

cat(tom).

loves_to_eat(kunal,pasta).

```
of_color(hair,black).
loves_to_play_games(nawaz).
lazy(pratyusha).
```

Rules

- We can define rule as an implicit relationship between objects.

facts are conditionally true. • So, when one associated condition is true, then the predicate is also true. Suppose we have some rules as given below –

- Lili is happy if she dances.
- Tom is hungry if he is searching for food.
- Jack and Bili are friends if both of them love to play cricket.
- will go to play if school is closed, and he is free.
- Here the symbol (:-) will be pronounced as “If”, or “is implied by”. This is also known as neck symbol, the LHS of this symbol is called the Head, and right hand side is called Body. Here we can use comma (,) which is known as conjunction, and we can also use semicolon, that is known as disjunction.

Syntax

- Suppose a clause is like :
  - P :- Q;R.

  This can also be written as

  P :- Q.

  P :- R.

. If one clause is like : P :- Q,R;S,T,U.

. Is understood as

```
P :- (Q,R);(S,T,U).
```

Or can also be written as:

```
P :- Q,R.
P :- S,T,U.
```

Rules

Example

```
happy(lili) :- dances(lili).
 hungry(tom) :- search_for_food(tom).
 friends(jack, bili) :- lovesCricket(jack), lovesCricket(bili).
```

```
goToPlay(ryan) :- isClosed(school), free(ryan).
```

Queries

• Queries are some questions on the relationships between objects and object properties. So question can be anything, as given below –

Is tom a cat?

Does Kunal love to eat pasta?

Is Lili happy?

Will Ryan go to play?

Knowledge Base 1

Suppose we have some knowledge, that Priya, Tiyasha, and Jaya are three girls, among them, Priya can cook. Let's try to write these facts in a more generic way as shown below –

```
girl(priya).
girl(tiyasha).
 girl(jaya).
can_cook(priya).
```

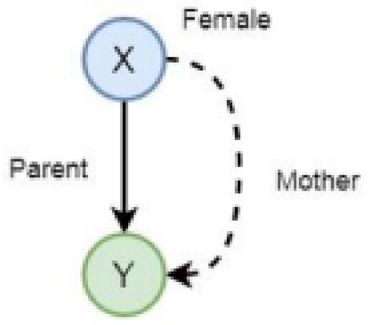
Note – Here we have written the name in lowercase letters, because in Prolog, a string starting with uppercase letter indicates a variable.

Prolog - Relations

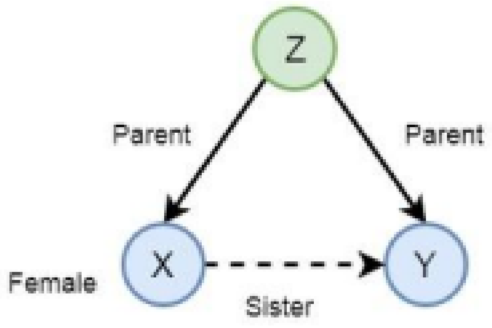
- We can define a brother relationship as follows –
- Two person are brothers, if,
- They both are male.
- They have the same parent.
- Now consider we have the below phrases –
- parent(sudip, piyus).
- parent(sudip, raj).
- male(piyus).
- male(raj).
- A and B are brothers if –
- A and B, both are male
- They have same Parent
- A and B are not same

- brother(X,Y) :- parent(Z,X), parent(Z,Y),male(X), male(Y)

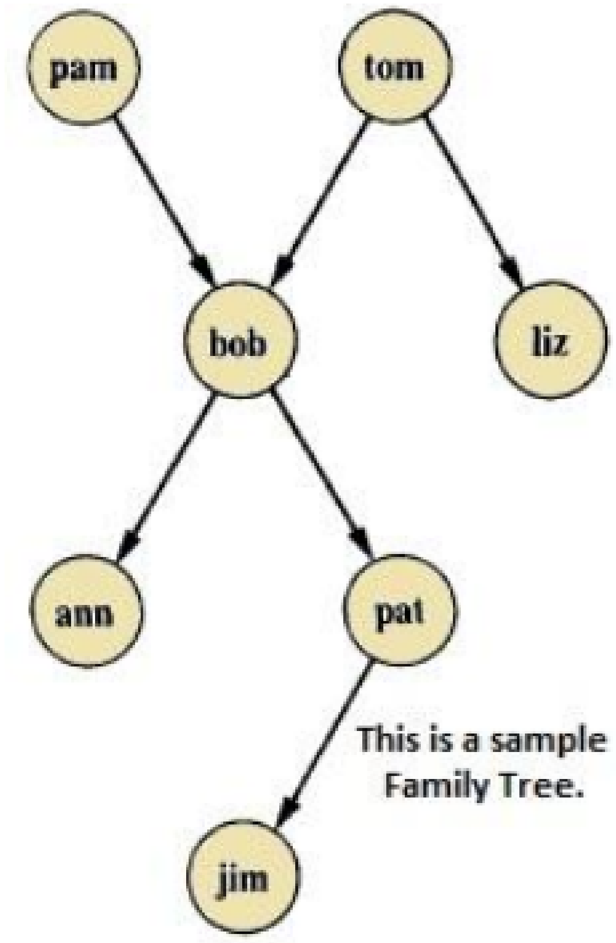
Family Relationship in Prolog



Mother Relationship

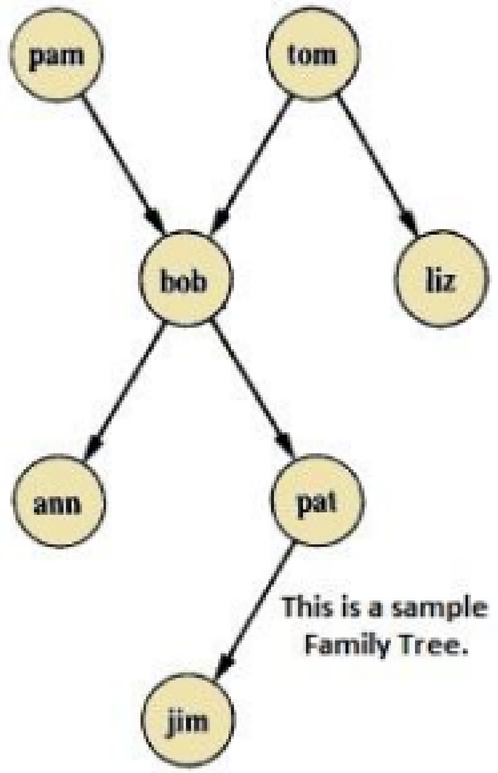


Sister Relationship

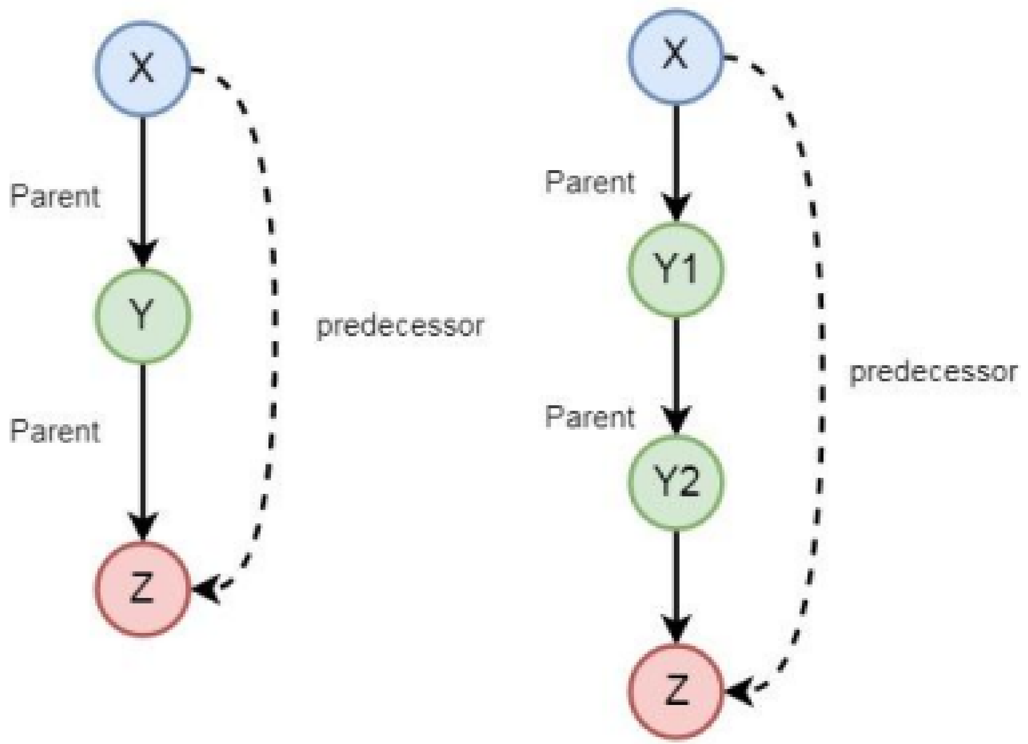


Now if we want to make mother and sister relationship, then we can write as given below –

Recursion in Family Relationship



This is a sample Family Tree.



Prolog - Data Objects

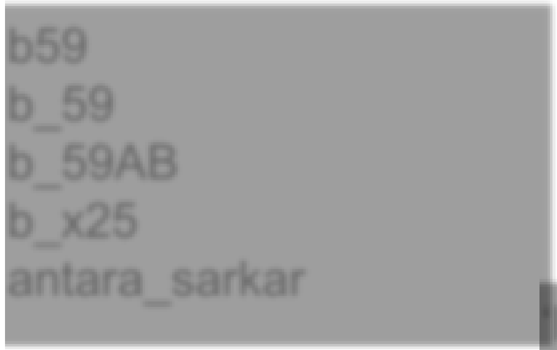
- Below are some examples of different kinds of data objects –
- Atoms – tom, pat, x100, x_45

- Numbers – 100, 1235, 2000.45
- Variables – X, Y, Xval, _X
- Structures – day(9, jun, 2017), point(10, 25)
-

Prolog - Data Objects

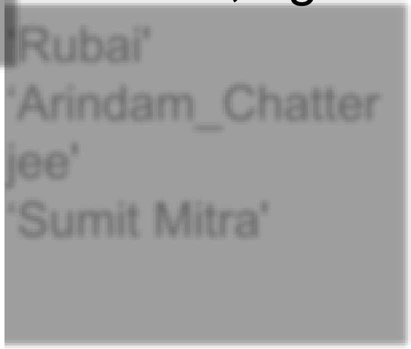
- Atoms – tom, pat, x100, x_45
- Atoms are one variation of constants. They can be any names or objects. There are few rules that should be followed when we are trying to use Atoms as given below –

STRINGS OF LETTERS, DIGITS AND THE UNDERSCORE CHARACTER, '_', STARTING WITH A LOWER-CASE LETTER.



b59
b_59
b_59AB
b_x25
antara_sarkar

Strings of characters enclosed in single quotes. including spaces and upper case letters,e.g.



'Rubai'
'Arindam_Chatter jee'
'Sumit Mitra'

Strings of special characters

characters from a list that includes the following+-*/ ><=&#@:

Any sequence of one or more special

```
<--->
=======>
...
.:.
::=
```

Prolog - Data Objects

- Numbers
- NUMBERS All versions of Prolog allow the use of integers (whole numbers).
- They are written as any sequence of numerals from 0 to 9 Optionally preceded by a + or – sign Ex: 623, -47 ,+5 , 025
-

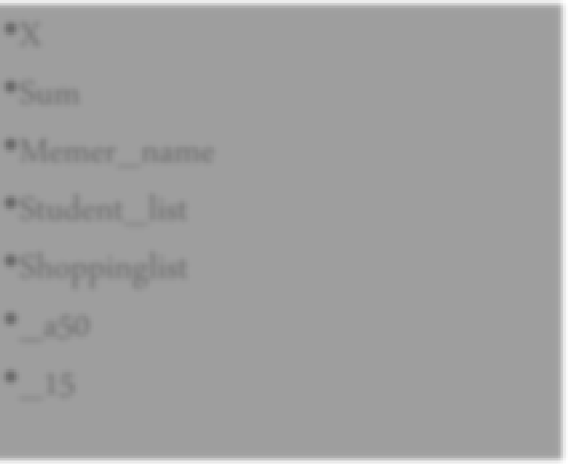
Most versions of Prolog also allow the use of numbers with decimal points.

- They are written in the same way as integers, but contain a single decimal point, anywhere except before an optional + or – sign
- Ex: 6.43 , -.245 , +256.

Prolog - Data Objects

- Variables – X, Y, Xval, _X
- In a query a variable is a name used to stand for a term that is to be determined

- The name of a variable is denoted by any sequence of one or more letters (upper case), numerals and underscores, beginning with an upper case letter or underscore, e.g.



•X
•Sum
•Memer_name
•Student_list
•Shoppinglist
•_a50
•_15

•X
•Sum
•Memer_ name
•Student_ list
•Shoppin glist
•_a50
•_15

Prolog - Data Objects

```
| ?- [var_anonymous]. compiling D:/TP
Prolog/Sample_Codes/var_anonymous.pl for byte code... D:/TP
Prolog/Sample_Codes/var_anonymous.pl compiled, 3 lines
read - 536 bytes
```

- Anonymous Variables in Prolog
- Anonymous variables have no names. The anonymous variables in prolog is written by a single

```
written, 16 ms yes
| ?- hates(X,tom).
```

underscore character '_'. And one important thing is that each individual anonymous variable is X = jim ? ;

treated as different. They are not same.

```
X = peter
yes
```

```
Knowledge Base
(var_anonymous.pl)
hates(jim,tom).
hates(pat,bob).
hates(dog,fox).
hates(peter,tom).
| ?- hates(_,tom).  true
?;
```

```
(16 ms) yes

| ?- hates(_,pat). no

| ?- hates(_,fox). true ?
;
no
| ?-
```

Prolog unification

• When programming in Prolog, we spend a lot of time thinking about how variables and rules "match" or "are assigned.“ • There are two aspects to this.

• The first, "unification," regards how terms are matched and variables assigned to make terms match.

• The second, "resolution,“, resolution is only used if rules are involved

- **Terms**
- Prolog has three kinds of **terms**:

  Constants like 42 (numbers) and franklin (atoms, i.e., lower-case words).
  Variables like X and Person (words that start with upper-case).
  Complex terms like parent(franklin, bo)
- Two terms **unify** if they can be matched. Two terms can be matched if:
- they are the same term (obviously), or
- they contain variables that can be unified so that the two terms without variables are the same. 27

Prolog unification

product(shoes,700) **product**

(hat, 200) product(jacket,550)

Example 1:Unification ?-

product(hat,price) •

Knowledge base

Rule 2:

Rule1:

1.If term1 and term2 are **constants**, then term1 and term2 unify if and only if they are the same atom, or the same number.

apple = orange

apple =apple

2.If term1 is a **variable** and term2 is any type of term, then term1 and term2 unify, and term1 is instantiated to term2. (And vice versa.) (If they are both variables, they're both instantiated to each other, and we say that they share values.)

X = apple Apple = X

X = Y _G1 X = _G1 X

Y = _G1 Y3

Prolog unification

Rule 3:

1.If term1 and term2 are **complex terms**, they unify if and only

if: a. They have the same **functor** and **arity**.
The functor is the "function" name (this functor is foo: foo(X, bar)).
The arity is the number of arguments for the functor (the arity for foo(X, bar) is 2).
b. All of their corresponding arguments unify. **Recursion!** c. The variable instantiations are compatible (i.e., the same variable is not given two different unifications/values).

person(robert) = person (robert).

drink(coke) = drink (Y).

order(drink(coke) , food (hotdog)) = order (X , food (Y)).

Prolog unification

- **Practice**

- **Do these two terms unify?**

  ?- mia = mia. Yes, from rule 1.

- **Do these two terms unify?**

  ?- mia = X. Yes, from rule 2.

- **Do these two terms unify?**

  ?- X = Y. Yes, from rule 2.

- **Do these two terms unify?**

  ?- k(s(g), Y) = k(X, t(k)).

  Yes, from rule 3 and, in the recursion, from rule 2 in two cases (X set to s(g) and Y set to t(k)).

- **Do these two terms unify?** ?- k(s(g), Y) = k(s(g, X), Y).

  No, because rule 3 fails in the recursion (in which rule 3 is invoked again, and the arity of s(g) does not match s(g, X)).

Some more example- Unification

10/3/2022 Ms.

Aaysha Shaikh 31

unification

- List Unification.

Prolog

  - [X, Y, Z] = [1, 2, 3]
  - [H | T] = [mon, tue, wed]
  - [A, B | T] = [mon, tue, wed, thur, fri].
  - [mon, tue, wed] =[mon | [tue, wed]]

RESOLUTION

- In Prolog the programs are written in special form known as the causal form
- Formally a clause is defined as an expression of the form •

(A1A2....An)_(B1B2...Bn)

• Where each of the As and Bs stand for proposition • A clause is used to express a logical implication of the following sort-

• -If all of Bs are true then one of the As must be true as well

• Resolution in Prolog is basically the inference mechanism

RESOLUTION

• Example:

```
loves(vincent, mia).
loves(marcellus, mia).
jealous(A, B) :- loves(A, C), loves(B, C).
```

How does Prolog find the proof for jealous(X, Y), and what values do the variables take? The proof tree is shown below. Note that there different ways to get a proof, there are four different sets of variable assignments.

Comparison Operators

- Comparison operators are used to compare two equations or states. Following are different comparison operators –

Operator	Meaning
X > Y	X is greater than Y
X < Y	X is less than Y
X >= Y	X is greater than or equal to Y
X =< Y	X is less than or equal to Y
X =:= Y	the X and Y values are equal
X =\= Y	the X and Y values are not equal

- Example

Arithmetic Operators in Prolog

Arithmetic operators are used to perform arithmetic operations. There are few different types of arithmetic operators as follows

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
//	Integer Division
mod	Modulus

- Program

Note – The nl is used to create new line.

Prolog - Lists

Representation of Lists

- A list can be either empty or non-empty. In the first case, the list is simply written as a Prolog atom, []. In the second case, the list consists of two things as given below – • The first item, called the head of the list;
- The remaining part of the list, called the tail.
- So the following list representations are also valid –
- [a, b, c] = [x | [b, c]]
- [a, b, c] = [a, b | [c]]
- [a, b, c] = [a, b, c | [ ]]

Basic Operations on Lists

Membership Operation

• To design this predicate, we can follow these observations.
X is a member of L if either – • X is head of L, or

• X is a member of the tail of L

```
list_member(X,[X|_]).
list_member(X,[_|TAIL]) :- list_member(X,TAIL).
```

Length Calculation

• If list is empty, then length is 0.

• If the list is not empty, then L = [Head|Tail], then its length is 1 + length of Tail.

```
list_length([],0).
list_length([_|TAIL],N) :- list_length(TAIL,N1),
N is N1 + 1.
```

Operations on Lists

Concatenation

- If the first list is empty, and second list is L, then the resultant list will be L.
- If the first list is not empty, then write this as [Head|Tail], concatenate Tail with L2 recursively, and store into new list in the form, [Head|New List].

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

list_concat([X1|L1],L2,[X1|L3]) :- list_concat(L1,L2,L3).
```

Delete from List

• If X is the only element, then after deleting it, it will return empty list. • If X is head of L, the resultant list will be the Tail part. • If X is present in the Tail part, then delete from there recursively.

```
list_delete(X, [X], []).
list_delete(X,[X|L1], L1).
list_delete(X, [Y|L2], [Y|L1]) :- list_delete(X,L2,L1).
```

Operations on Lists

- reverse([1,2,3],A).
- append([a,b],[1,2,3],L).
- Write([1,2,3])

- member(x, [x,y,z]).
- length([a,b],M).

Example:

- Miscellaneous Operations on Lists

Misc Operations	Definition
Even and Odd Length Finding	Verifies whether the list has odd number or even number of elements.
Divide	Divides a list into two lists, and these lists are of approximately same length.

Max	Retrieves the element with maximum value from the given list.
Sum	Returns the sum of elements of the given list.
Merge Sort	Arranges the elements of a given list in order (using Merge Sort algorithm).

Repositioning operations of list items

Repositionin g Operations	Definition
Permutation	This operation will change the list item positions and generate all possible outcomes.
Reverse Items	This operation arranges the items of a list in reverse order.
Shift Items	This operation will shift one element of a list to the left rotationally.

Order Items	This operation verifies whether the given list is ordered or not.

Basic Repetition Loop: map

- Some important functions or loops that iterate over lists:
- **map**, **filter**, **zip**

- **map**

maplist(:Goal, ?List1).

maplist(:Goal, ?List1, ?List2).

- True if Goal is successfully applied on all matching elements of the list.

- Examples:

?- length(L, 5)

L = [_, _, _, _, _].

?- length(L, 5), maplist(**=(0)**, L). L = [0, 0, 0, 0, 0].

?- length(L, 5), maplist(**=(*)**, L). L =

[*, *, *, *, *].

?- maplist(**plus(3)**, [1, 4], [Y, Z]). Y =
4,
Z = 7.

?- maplist(**plus(3)**, [1, X], [Y, 2]). X =
-1,
Y = 4

Basic Repetition Loop: filter

- Some important functions or loops that iterate over lists: • map, filter, zip

- **filter**
- **Include** and **exclude** predicates are used for filtering the list

include(:Goal, +List1, ?List2).

- Filter elements, for which Goal succeeds.
  - True if List2 contains those elements Xi of List1 for which call(Goal, Xi) succeeds.
- **Knowledge base**

Basic Repetition Loop: filter

• Some important functions or loops that iterate over lists: • map, filter, zip

• **filter**

• **Include** and **exclude** predicates are used for filtering the list

exclude(:Goal, +List1, ?List2).

• Filter elements, for which Goal fails.

```
is_odd(X) :- 0 =\= X mod 2.
include(is_odd, List, Odd).

filterOdd(In, Out) :-include(is_odd, In,

Out).
```

• **Queries**

```
?- include(is_odd, [1,2,3,4,5,6,7,8,9],
   Odd).  Odd = [1, 3, 5, 7, 9]

?- filterOdd([1,2,3,4,5,6,7,4,9], Out). Out =
   [1, 3, 5, 7, 9].
```

- True if List2 contains those elements Xi of List1 for which call(Goal, Xi) fails.
- **Knowledge base**

```
are_identical(X, Y) :- X == Y.
exclude(are_identical(4),
[1,2,3,4,5,6,7,4,9], Out).
filterList(A, In, Out) :-
exclude(are_identical(A), In,
Out).
```

- **Queries**

```
?- exclude(are_identical(4),
[1,2,3,4,5,6,4,7,4,9], Out).
Out = [1,2,3,5,6,7,9]

?- filterList(4, [1,2,3,4,5,6,4,7,4,9],
   Out)  Out = [1,2,3,5,6,7,9]
```

Basic Repetition Loop: zip

- Some important functions or loops that iterate over lists:
- map, filter, zip

- **zip**
- The zip() function returns a zip

object, which combines individual elements of two iterables into tuples.

- **Knowledge base**

- zip([], [], []).
- zip([X|Xs], [Y|Ys], [X,Y|Zs]) :-

zip(Xs,Ys, Zs).

- **Queries**

?- zip([a, b, c], [x, y, z], X). X = [a, x, b, y, c, z].

Imperative Control Flow

- The control flow is the order in which the computer executes statements in a script.

- Three basic types of control structures:
- Sequential: default mode. ...
- Repetition: used for looping, i.e. repeating a piece of code multiple times in a row.
- Selection: used for decisions, branching -- choosing between 2 or more alternative paths. ...

Loops in Prolog

- The looping facility is contained in most of the programming languages. **Looping** is used to enable a set of instructions to be repeatedly executed either a fixed number of times or until a given condition met.
- **Prolog has no looping facility**, but we can obtain a similar effect.

- Using this effect, we can evaluate a sequence of goals repeatedly. In various ways, this can be done like built-in predicates, recursion, backtracking, or a combination of these.

Example 1:

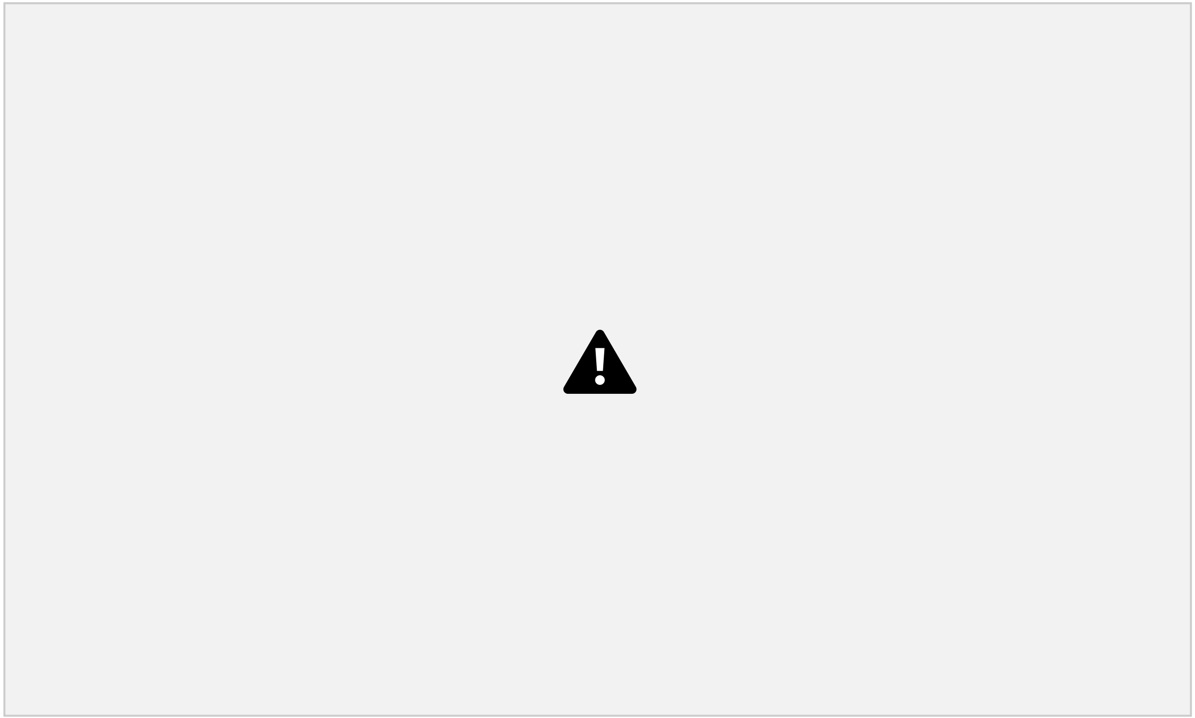
```
1.loop(0).
2.loop(N): N>0, write('value of
N is: '), write(N), nl.
3.S is N-1, loop(S).
```

```
?- loop(4).
value of N is: 4  value of N is: 3
value of N is: 2  value of N is: 1
yes
```

Prolog - Loop & Decision Making

- Loops

```
count_to_10(10) :- write(10),nl.
count_to_10(X) :-
 write(X),nl,
 Y is X + 1,
count_to_10(Y).
```



• OUTPUT:

10/3/2022

Ms. Aaysha Shaikh 51

Now create a loop that takes lowest and highest values. So, we can use the between() to simulate

```
count_down(L, H)
:-
```

```
 between(L, H, Y),
 Z is H - Y,
 write(Z), nl.
count_up(L, H) :-
 between(L, H, Y),
 Z is L + Y,
 write(Z), nl.
```

Applying recursion

```
1.* sum of integers from 1 to N inclusive */
2.sumto(1, 1).
```

3.sumto(N, M) :- N>1, N1 is N-1, sumto(N1, M1), M is M1+N.

•100 plus the sum of the first 99 integers is the sum of the first 100 integers. •99 plus the sum of the first 98 integers is the sum of the first 99 integers. •98 plus the sum of the first 97 integers is the sum of the first 98 integers.
•………………………………………………………………………… •3 plus the sum of the first 2 integers is the sum of the first 3 integers. •2 plus the sum of the first 1 integer is the sum of the first 2 integers. •1 is the sum of the first one integer.

OUTPUT:

```
?- sumto(100, N).
N= 5050
?- sumto(1, 1).
yes
```

Ex5 Recursion: Loop till condition satisfied

- Recursion Example5
- **<u>Knowledge base</u>**

```
go :- loop(start). loop(end).
loop(A) :- A\=end, write('The value is:'), read(Word), write('Input value is: '),
    write(Word), nl, loop(Word).
```

• **<u>Queries</u>**

```
?- go.
The value is:
Input value is a The value is:
Input value is: hi The value is:
Input value is end
    yes
```

Ex5 Repetition: Use of 'repeat’ Predicate

• In Prolog, the easiest way to provide the type of looping is not always recursion. Another method to provide the looping is built-in predicate repeat. • The following program prompts the user repeatedly to enter a term until the user enters either yes or no. This program is an alternative to the previous program. • **Knowledge base**

```
get_answer(Answer) :- write('Enter answer to question'), nl, repeat, write('yes or no
answer: '), read(Answer), valid(Answer),
write('The correct answer is '), write(Answer), nl.
valid(yes). valid(no).
```

• **Queries**

```
?- get_answer(X).
Enter answer to question Yes or no answer: unsure. Yes or no answer: possibly. Yes or
    no answer: yes.
The correct answer is yes X = yes
```

Control Flow By Selection: The cut

• In Prolog, kind of decision making is done using the cut: “!” operator. • This is written as “!”, but read as “cut”.

• It takes no arguments.

• It always succeeds.

• Once it succeeds, backtracking past the cut is not allowed, for the current goal, means, use of another fact or rule for the current goal is not allowed. • Cut can be used as the rough equivalent of a C++ break

• Cut can do something like if-then-else. Consider the C++ below.

```
void test_big(int n)
{
if (n > 6)
cout << n << " IS A BIG NUMBER!" << endl;
else
cout << n << " is not a big number." << endl;
}
```

Backtracking

• Backtracking is a procedure, in which prolog searches the truth value of different predicates by checking whether they are correct or not.

• The backtracking term is quite common in algorithm designing, and in different programming environments.

• In Prolog, until it reaches proper destination, it tries to backtrack. •
When the destination is found, it stops.

• During backtracking there may be multiple answers, we can press semicolon (;) to get next answers one by one, that helps to backtrack. Otherwise when we get one result, it will stop.

• Ex: Suppose A to G are some rules and facts. We start from A and want to reach G. The proper path will be A-C-G,

Backtracking Ex

- Consider two people X and Y can pay each other, but the condition is that a boy can pay to a girl
- **<u>Knowledge Base:</u>**
- boy(tom).
- boy(bob).
- girl(alice).
- girl(lili).
- pay(X,Y) :- boy(X), girl(Y).

Cut in Backtracking

• Sometimes we write the same predicates more than once when our program demands, in such cases uncontrolled backtracking may prove inefficient. • To resolve this, we will use the Cut in Prolog.

• Ex: Consider we have three mutually exclusive rules and any given time only one of them ill be true.

• **Knowledge Base:**

f (X,0) :- X < 3.

f (X,2) :- 3 =< X, X <• **Queries**:

6. f (X,4) :- 6 =< X. ?- f(1,Y), 2<Y.

% Rule 1 % Rule 2 %< 6, !. f (X,4) :- 6 =< X.

Rule 3

f (X,0) :- X < 3, !.

f (X,2) :- 3 =< X, X

% Rule 1 % Rule 2

% Rule 3

- There are two subgoals to be satisfied. As per first fact X=1 so Y will be set to 0, but now second clause or query fails since Y is not >2.
- So Prolog backtracks and goes rule 2, here since X is not between 3 and 6, first query clause itself fails. Prolog backtracks and checks third rule, the goal fails here too. To avoid such backtracking cut is used.

Example-2 of Cut

- **Ex: Knowledge Base:**
- animal(dog).
- animal(cat).
- animal(elephant).
- animal(tiger).
- animal(cobra).

• animal(python).

• snake(cobra).

• snake(python).

• likes(mary, X) :- snake(X).

• likes(mary, X) :- animal(X).

• **Queries:**

?- likes(mary, X).

Use semicolon to see different X values

• **Ex: Knowledge Base: (Using cut)** • animal(dog).

• animal(cat).

• animal(elephant).

• animal(tiger).

• animal(cobra).

• animal(python).

• snake(cobra).

• snake(python).

• likes(mary, X) :- snake(X), !. • likes(mary, X) :- animal(X).

• **Queries:**

?- likes(mary, X).

Use semicolon to see different X

va

lu

es

Cut with Failure Ex-1

- Cut is also used to specify exceptions to general rules, check the output •

Ex: Knowledge Base:

- animal(dog).
- animal(cat).
- animal(elephant).
- animal(tiger).
- animal(cobra).
- animal(python).
- snake(cobra).
- snake(python).
- likes(mary, X) :- snake(X), !, **Fail**.
- likes(mary, X) :- animal(X).

- **Queries:**
- ?- likes(mary, X).

• Use semicolon to see different X values

Cut with Failure Ex2

• we will specify another use of 'cut'. It is used to specify exceptions to general rules. In the following example, we have names of birds in the database as follows:

```
bird(crow). bird(sparrow). bird(parrot). bird(penguins). bird(dove).
    bird(robin). bird(turkey). bird(hawk). bird(goose). bird(swallow).
    bird(pigeon). bird(woodpecker).
```

Cut with Failure Ex2

```
can_fly(A) :- bird(A).
```

•The above rule is very general. We cannot ensure that the goal can_fly(penguins) will always fail? So, we are going to change the predicate can_fly definition in this approach as follows:

```
can_fly(penguins) :- fail.
can_fly(A) :- bird(A).
```

•However, the desired result is not provided by the above:

•The head of the first can_fly clause and the goal can_fly(penguins) are matched with each other. In the body of that clause, we are trying to satisfy the goal, and the goal obviously fails. Now, the system looks at the second can_fly clause. The head and the goal match with each other, and the goal is also satisfied in the body of the clause, i.e., bird(A), so the goal can_fly(penguins) succeeds. But this is not the desired result.

•We can achieve the desired result by replacing the clause can_fly by can_fly(penguins) :- !, fail.

```
can_fly(A) :- bird(A).
```

•As before, the head of the first can_fly clause and the goal can_fly(penguins) are matched with each other. In the body of that clause, we are trying to satisfy the goal, the goal obviously fails. But here the cut prevents it from backtracking the system, so the goal can_fly(penguins) fails. •Cut with failure is the combination of fail and goals !.

Backtracking with Failure

While **backtracking** or 'standard' evaluation left-to-right, the **fail** predicate always fails, as the name implies.

We can take advantage of this by combining it with Prolog's automatic backtracking to find all the clauses in the database with a specified property

Searching the database of Prolog dog(pug).

dog(boxer).

dog(rottweiler).

Using the predicate **alldogs**,

OUTPUT:

```
?- alldogs.
pug is a dog
boxer is a dog  rottweiler is a dog.  yes
```

we can process each clause of the **dog** in turn as follows: alldogs :- dog(A), write(A), write(' is a dog'),

nl, fail. alldogs.

Program Explanation

- When we call the **alldogs**, due to this **dog(A)** will be matched with the clauses of **dog** in the database. In the starting, variable A will be bound to atom pug and produced the output as 'pug is a dog'. In the first clause of predicate **alldogs**, the final goal will cause evaluation to fail. Then, Prolog will backtrack over two goals **write** and **nl**, that goals are unsatisfiable until Prolog reaches to **dog(A)**. Then the second time this goal succeeds, and due to this, variable A will be bound to boxer.
- This process will continue until pug, boxer, and rottweiler have all been output, and when evaluation again fails. In the database, there is no further dog, this time call to **dog(A)** will also fail. Due to this, the first clause of **alldogs** will fail. Now the second clause of **alldogs** is examined by Prolog, and this will succeed. After that, the

evaluation will stop.

- Due to this effect, Prolog will generate a loop through the database. To satisfy the goal **dog(A),** Prolog will find all possible values of **A**.

Ex5 Recursion: Loop till condition satisfied

- Recursion Example5
- **Knowledge base**

```
go :- loop(start). loop(end).
loop(A) :- A\=end, write('The value is:'), read(Word), write('Input value is: '),
    write(Word), nl, loop(Word).
```

- **Queries**

```
?- go.
The value is:
Input value is a The value is:
Input value is: hi The value is:
```

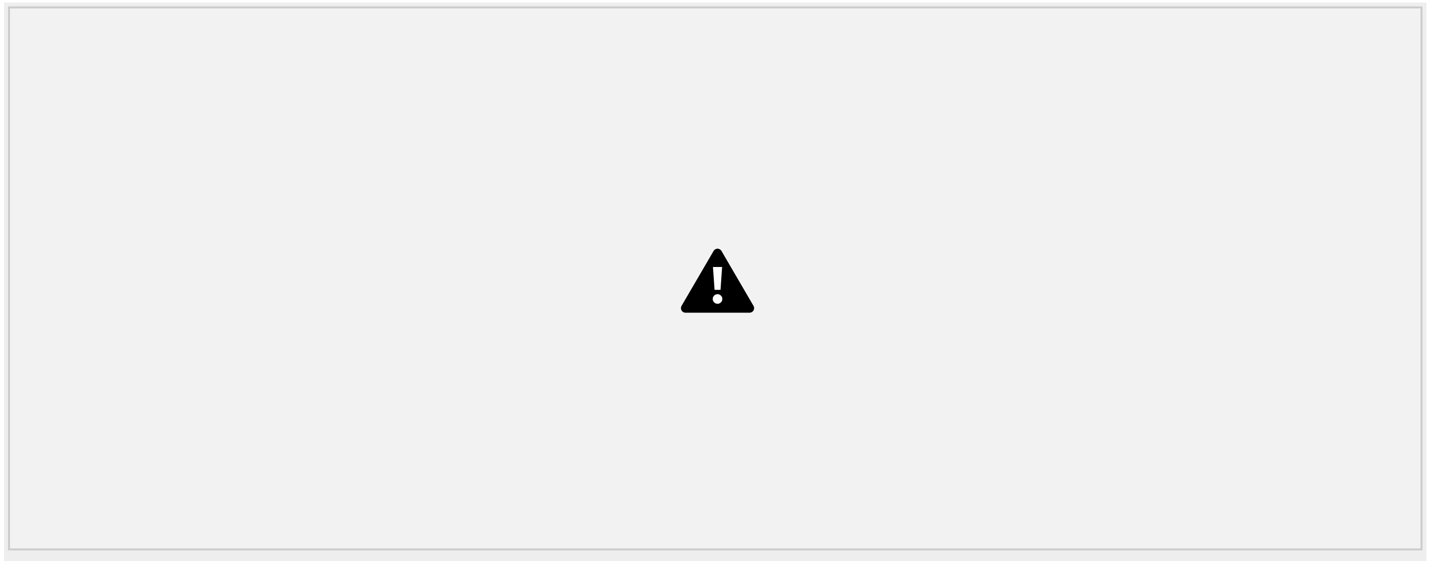
Input value is end

yes

Decision Making

• The decision statements are If-Then-Else statements. So, when we try to match some condition, and perform some task, then we use the decision making statements. The basic usage is as follows –

```
If <condition> is true, Then <do this>, Else
```



Prolog - Conjunctions & Disjunctions

Program

```
parent(jhon,bob).
 parent(lili,bob).
male(jhon).
 female(lili).
 % Conjunction Logic
 father(X,Y) :- parent(X,Y),male(X).
mother(X,Y) :-
parent(X,Y),female(X). %
Disjunction Logic
child_of(X,Y) :- father(X,Y);mother(X,Y).
```

Database Manipulations

• A database is a collection of information which allow organization to access, manage and update the information. • Databases consist of multiple tables which is capable to include different fields.

• Each of these database have different fields which might be relevant to the information stored in the database. A data manipulation provide sublanguage for database such as SQL. Prolog is effective language for database queries.

• Prolog has three database manipulation commands namely retract, asserta, and assertz.

Some IO Actions

write/1

•This clause always succeeds by writing the argument to the standard output.

nl/0

•This clause always succeeds by writing newlineto the standard output

read_number/1

•Reads a number from standard input. Raises an exception if what is typed is not a number. Otherwise, binds the variable to it and succeeds.

PROLOG facilities and deficiencies

- Advantages :
  1. Easy to build database. Doesn't need a lot of programming effort.
  2. Pattern matching is easy. Search is recursion based.
  3. It has built in list handling. Makes it easier to play with any algorithm involving lists.
- Disadvantages :
  1. LISP (another logic programming language) dominates over prolog

with respect to I/O features.

2. Sometimes input and output is not easy.

- Applications :

1. Prolog is highly used in artificial intelligence(AI).

2. Prolog is also used for pattern matching over natural language parse trees.

Deficiencies of Prolog

Resolution Order Control:

•Prolog always matches in the same order – user can control the ordering

•Prolog allows explicit control of backtracking using cut which is actually, a goal and not an operator. It always succeeds but can not be resatisfied through backtracking.

The Closed World Assumption:

•In Prolog truths are those that can be proved using its database •If there is insufficient information in database, to prove a query, it is not actually false, it fails.

•It relates to negation problem.

Deficiencies of Prolog

The negation Problem:

•Stating that two atoms are not equal is not straightforward in Prolog •Bad Solution: adding facts stating that they are not the same. •Alternative: state in the goal that X must not be same as Y sibling(X,Y) :- parent(M,X), parent(M,Y), not(X=Y).

•Here not operator is not logical not means not(not(some_goal)) is not equivalent to some_goal. •The reason is use o horn clause form, prevents any negative conclusions. Intrinsic Limitations:

•Prolog has no idea of how to sort other than simply to enumerate all permutations of given list until it happens to create the one that has the list in sorted order

•We must specify the details of how that sorting can be done, ---which is then not the logic programming anymore.

•Resolution is not capable of doing this sorting

Thank You