Recursive Programs

- Recursion in any language is a function that call itself until a given goal has been succeeded
- In prolog, recursion appear when a predicate contain a goal that refers to itself.
- There are two types:

  Tail Recursion: in which the recursive call is always made Justin the last step before the procedure exits.

  Examples:

  Write a prolog program to find the factorial of 5. i.e. 5!

```
Domain : I=Integer Predicate:
fact(I,I,I).
Clauses:
fact(I,F,F):-!. fact(N,F,R):- F1 is F*N, N1 is N-1,
fact(N1,F1,R).
```

Goal: ? - fact(5,I,F).

Output : F=120.

Non-Trail Recursion : is recursion in which the recursion is not the last step in the procedure call.

```
fact(0,1). fact(1,1).
fact(N,F):- N1 is N-1, fact(N1,F1), F is N*F1.
```

Goal : ?- fact(5,F).

Output : F=120.

Write a prolog program to print number from 1-20.

```
count(20):-!.
count(X):- write(X), X1 is X+1, count(X1).
Goal: count(1).
Output: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Write a program to find the power of any number using tail recursion.

```
power(_, 0, P,P):-!. power(X, Y, Z,P):- Z1 is Z*X, Y1 is Y-1, power(X, Y1, Z1,P),!.
Goal : power(5, 2, 1,P) Output: P=25.
```

Using non-trail recursion

```
power( _, 0, 1):-!. power(X, Y,P):- Y1 is Y-1, power(X, Y1,P1), P is P1*X.
```

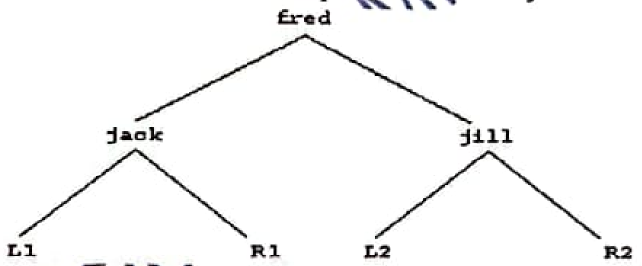
```
Goal : power(5, 2, P) Output: P=25.
```

Write a program to read and write a number of characters until the input character is equal to '#'.

```
Clause:
Repeat.
Repeat: repeat.
Typewriter :- repeat, readchar(C), write(C),nl, C is '#',!.
```

Binary Trees

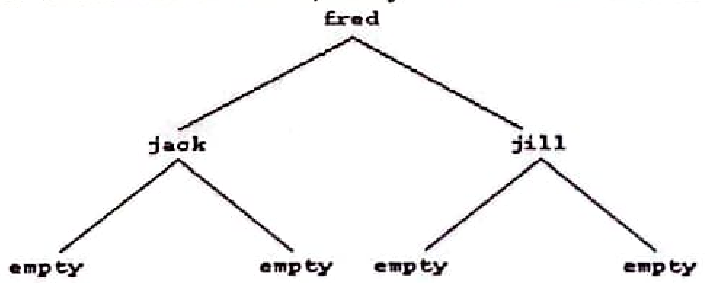
- In the library database example, some complex terms contained other terms, for example, book contained name.
- The following term also contains another term, this time one similar to itself:
- tree(tree(L1, jack, R1), fred, tree(L2, jill, R2))
- The variables L1, L2, R1, and R2 should be bound to sub-trees (this will be clarified shortly).
- A structure like this could be used to represent a "binary tree" that looks like:



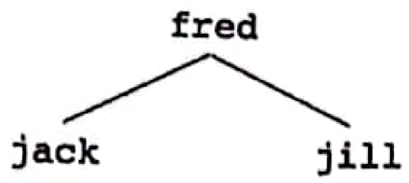
- Binary because each "node" has two branches (our backtrack tree before had many branches at some nodes)

Recursive Structures

- A term that contains another term that has the same principal functor (in this case tree) is said to be recursive.
- Biological trees have leaves. For us, a *leaf* is a node with two empty branches:

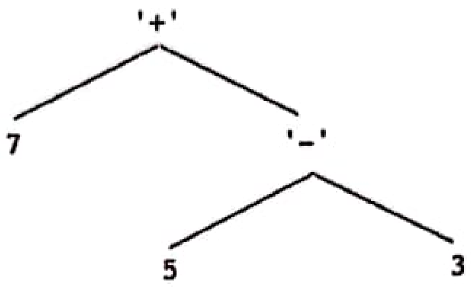


- empty is an arbitrary symbol to represent the empty tree. In full, the tree above would be: tree(tree(empty, jack, empty), fred, tree(empty, jill, empty))
- Usually, we wouldn't bother to draw the empty nodes:



Another Tree Example

```
tree(tree(empty, 7, empty),
    '+',
    tree(tree(empty, 5, empty),
        '-',
        tree(empty, 3, empty)))
```



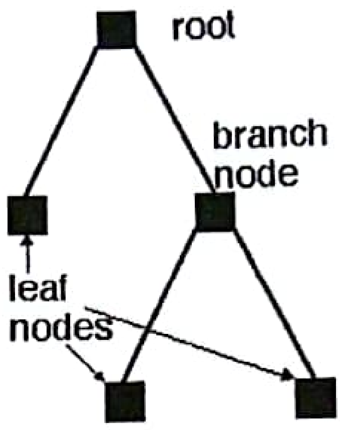
- A binary tree is either empty or contains some data and a left and right subtree that are also binary trees.
- In Prolog we express this as:

```
is_tree(empty).          trivial branch
is_tree(tree(Left, Data, Right)) :-    recursive branch
is_tree(Left),      some_data(Data),
is_tree(Right).
```

- A non-empty tree is represented by a 3-arity term.
- Any recursive predicate must have:
  - (at least) one **recursive branch/rule** (or it isn't recursive :-)) and
  - (at least) one non-recursive or **trivial branch** (to stop the recursion going on for ever).

The example at the heading "An Application of Lists", below, will show how the recursive branch and the trivial branch work together. However, you probably shouldn't try to look at it until we have studied lists.

- Let us define (or measure) the size of tree (i.e. number of nodes): tree_size(empty, 0). tree_size(tree(L, _, R), Total_Size) :- tree_size(L, Left_Size), tree_size(R, Right_Size),
  Total_Size is
  Left_Size + Right_Size + 1.
- The size of an empty tree is zero.
- The size of a non-empty tree is the size of the left sub-tree plus the size of the right sub-tree plus one for the current tree node.
  □ The data does not contribute to the total size of the tree.
- Recursive data structures need recursive programs. A recursive program is one that refers to itself, thus, tree_size contains goals that call for the tree_size of smaller trees **Lists**



Tree of size 5

- A list may be nil (i.e. empty) or it may be a term that has a head and a tail □ The head may be any term or atom.
- The tail is another list.
- We could define lists as follows: is_list(nil).
  is_list(list(Head, Tail)) :-
  is_list(Tail).
- A list of numbers [1, 2, 3] would look like: list(1, list(2, list(3, nil)))
- This notation is understandable but clumsy. Prolog doesn't actually recognise it, and in fact uses . instead of list and [] instead of nil. So Prolog would recognise .(1, .(2, .(3, []))) as a list of three numbers. This is briefer but still looks strange, and is hard to work with.
- Since lists are used so often, Prolog in fact has a special notation that encloses the list members in square brackets:
  [1, 2, 3] = .(1, .(2, .(3, [])))

?- *X = .(1, .(2, .(3, []))).*
X = [1, 2, 3]

List Constructor |

□ Within the square brackets [ ], the symbol | acts as an operator to construct a list from an item and another list.

?- *X = [1 | [2, 3]].*
X = [1, 2, 3].
?- *Head = 1 , Tail = [2, 3], List = [Head | Tail].*
List = [1, 2, 3].

Examples of Lists and Pattern Matching

?- *[X, Y, Z] = [1, 2, 3].* Match the terms on either side of =

X = 1
Y = 2
Z = 3

| separates head from tail of list. ?-

[X | Y] = [1, 2, 3].
X = 1
Y = [2, 3]

So [First | Rest] is the usual way of writing .(First, Rest) in Prolog

?- *[X | Y] = [1].*
X = 1
Y = []

The empty list is written as [] Lists "end" in an empty list!
Note that [1] is a list with one element.

The first several elements of the list can be selected before matching the tail:
?- *[X, Y | Z] = [fred, jim, jill, mary].*
X = fred   Must be at least two elements Y = jim   in the list on the right.
Z = [jill, mary]

Complex List Matching
?- *[X | Y] = [[a, f(e)], [n, m, [2]]].*

X = [a, f(e)]
Y = [[n, m, [2]]]
Notice that Y is shown with an extra pair of brackets: Y is the tail of the entire list: [n, m, [2]] is the sole element of Y.

List Membership

- A term is a member of a list if o the term is the same as the head of the list, or o the term is a member of the tail of the list.
- In Prolog:

trivial branch:

member(X, [X | _]).

a rule with a head but no body

member(X, [_ | Y]) :-   recursive branch
    member(X, Y).

- The first rule has the same effect as: member(X, [Y|_]):- X = Y.
  The form member(X, [X|_]). is preferred, as it avoids the extra calculation.

- Member is actually predefined in Prolog. It is a built-in predicate. There are quite a few built-in predicates in Prolog

```
% length(List, LengthOfList)
% binds LengthOfList to the number of elements in List.
length([OnlyMember], Length) :-
        Length = 1.
length([First | Rest], Length) :-
 length(Rest, LengthOfRest),  Length is
LengthOfRest + 1.
```

This works, but involves an unnecessary unification. It is better for the base case to be length([OnlyMember], 1).
In effect, we take the original version of the base case, and replace Length, in the head of the rule, with the thing that Length is = to. Programmers who fail to do this are usually still thinking procedurally.

Programming Principles for Recursive Structures

- Only deal with one element at a time.
- Believe that the recursive program you are writing has already been written. In the definition of member, we are already assuming that we know how to find a member in the tail.
- Write definitions, not programs!
    - If you are used to writing programs for conventional languages, then you are used to giving instructions on how to perform certain operations.
    - In Prolog, you define relationships between objects and let the system do its best to construct objects that satisfy the given relationship.

Concatenating Two Lists

- Suppose we want to take two lists, like [1, 3] and [5, 2] and concatenate them to make [1, 3, 5, 2]
- The header comment is:

% concat(List1, List2, Concat_List1_List2)
%Concat_List1_List2 is the concatenation of List1 & List2 There are two rules:

- First, the trivial branch: concat([], List2, List2).
- Next, the recursive branch:
  concat([Item | Tail1], List2, [Item | Concat_Tail1_List2]) :- concat(Tail1, List2, Concat_Tail1_List2).
- For example, consider
  ?- concat([1], [2], [1, 2]).
  By the recursive branch:

- concat([1 | []], [2], [1 | [2]]) :-concat([], [2], [2]). and concat([], [2], [2]) holds because of the trivial branch.
- The entire program is:

```
% concat(List1, List2, Concat_List1_List2):
%  Concat_List1_List2 is the concatenation of List1 & List2
concat([], List2, List2). concat([Item | Tail1], List2, [Item |
Concat_Tail1_List2]) :-     concat(Tail1, List2,
Concat_Tail1_List2).
```

An Application of Lists

- Find the total cost of a list of items:

  Cost data:

```
cost(cornflakes, 230).
cost(cocacola, 210).
cost(chocolate, 250).
cost(crisps, 190).
```

- Rules:

```
total_cost([], 0).             % trivial branch
total_cost([Item|Rest], Cost) :-  % recursive branch
cost(Item, ItemCost),    total_cost(Rest,
CostOfRest),    Cost is ItemCost + CostOfRest.
```

Sample query:

?- total_cost([cornflakes, crisps], X).

X = 420

Tracing total_cost ?-
trace.

```
true.
[trace]  ?- total_cost([cornflakes, crisps], X).
  Call: (7) total_cost([cornflakes, crisps], _G290) ? creep
  Call: (8) cost(cornflakes, _L207) ? creep
  Exit: (8) cost(cornflakes, 230) ? creep
  Call: (8) total_cost([crisps], _L208) ? creep
  Call: (9) cost(crisps, _L228) ? creep
  Exit: (9) cost(crisps, 190) ? creep
  Call: (9) total_cost([], _L229) ? creep
  Exit: (9) total_cost([], 0) ? creep
^ Call: (9) _L208 is 190+0 ? creep
^ Exit: (9) 190 is 190+0 ? creep
  Exit: (8) total_cost([crisps], 190) ? creep
^ Call: (8) _G290 is 230+190 ? creep
^ Exit: (8) 420 is 230+190 ? creep
  Exit: (7) total_cost([cornflakes, crisps], 420) ? creep

X = 420

[debug]  ?- notrace.
```

Modifying total_cost

This is an *optional* homework exercise.

What happens if we change the recursive branch rule for total_cost as shown below?

```
total_cost([Item|Rest], Cost) :-    total_cost(Rest, CostOfRest), cost(Item,
ItemCost),
   Cost is ItemCost + CostOfRest.
```

The second and third lines have been swapped around.

You'll find that the rule still works. Try tracing the new version of this rule, work out what happens differently.

Which version do you find easier to understand? Why do think this is the case?

Another list-processing procedure

- The next procedure removes duplicates from a list.
- It has *three rules*. This is an example of a common list-processing *template*.
- Algorithm: ○ If the list is empty, there's nothing to do.
    - ○ If the first item of the list is a member of the rest of the list, then discard it, and remove duplicates from the rest of the list. ○ Otherwise, keep the first item, and again, remove any duplicates from the rest of the list.

```
% remove_dups(+List, -NewList):
% NewList is bound to List, but with duplicate items removed. remove_dups([],
[]).
remove_dups([First | Rest], NewRest) :-
member(First, Rest),    remove_dups(Rest,
NewRest). remove_dups([First | Rest], [First |
NewRest]) :-    not(member(First, Rest)),
remove_dups(Rest, NewRest).

?- remove_dups([1,2,3,1,3,4], X).

X = [2, 1, 3, 4] ;
```

false.

- Note the use of not to negate a condition. An alternative to not is \+.

Singleton Variables

- If Prolog finds a variable name that you only use once in a rule, it assumes that it may be a spelling mistake, and issues a **Warning** about a "singleton variable" when you load the code:

```
% prolog -q -s mycode.pl
Warning: .../mycode.pl:4:
                Singleton variables: [Item]
```

The -q means "quiet" - i.e. don't print the SWI Prolog welcome message. This way, any warnings are easier to notice.

- Here is the code that produced this (with line numbers added): 1 % count(Item, List, Count) counts the number of times the 2 % Item occurs in the List, and binds Count to that number.

```
3
4 count(Item, [], 0).
5 count(Item, [Item | Rest], Count) :-  6
  count(Item, Rest, RestCount),  7     Count is
  RestCount + 1.
   8      count(Item, [Other | Rest],
Count) :-
   9      not(Item = Other),   10
count(Item, Rest, Count).
```

To suppress the warning, put an _ in front of the word Item on line 4 (only). This makes it "don't care" variable. Check for the possible spelling error, first!

```
4 count(_Item, [], 0).
```

Controlling Execution

The Cut Operator/function (!)

- Sometimes we need a way to prevent Prolog finding all solutions, i.e. a way to stop backtracking.
- The cut operator, written !, is a built-in goal that prevents backtracking.

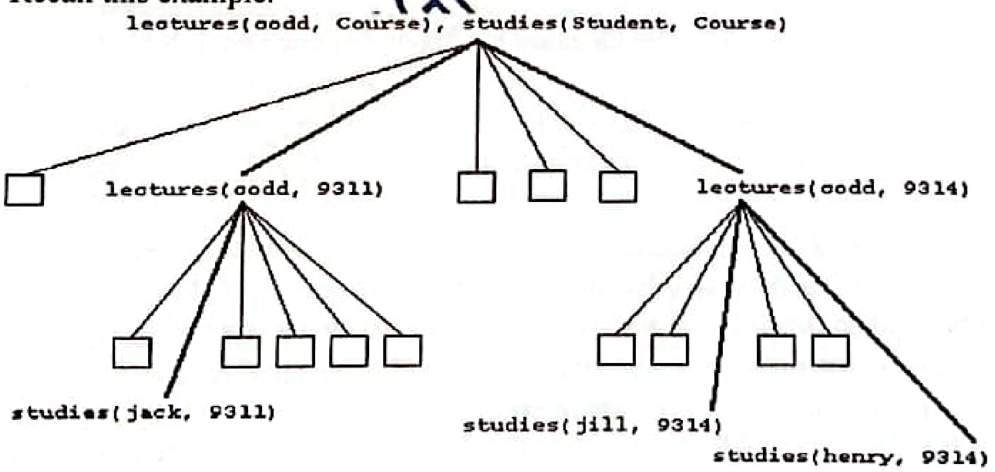
- It turns Prolog from a nice declarative language into a hybrid monster.
- Use cuts sparingly and with a sense of having sinned.

```
% Using cut
no(5):-!.
no(I).
no(10).

| ?- no(X).

X = 5

yes
```

Using cut in the end of the rule.

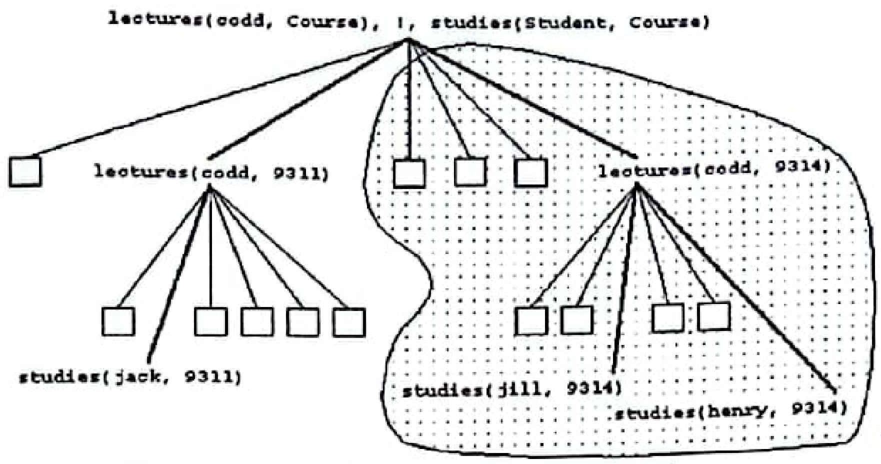
```
a(10).
a(20).
b(a).
b(c).
c(X,Y):- a(X), b(Y), !.

| ?- c(X,Y).
X = 10
Y = a

yes
```

Recall this example:



Cut Prunes the Search Tree

- If the goal(s) to the right of the cut fail then the entire clause fails and the the goal that caused this clause to be invoked fails.



- In particular, alternatives for Course are not explored.

Cut Prunes the Search Tree 2

- Another example: using the facts03 database, try ?- *lectures(codd, X).* X = 9311 ; X = 9314.

?- *lectures(codd, X), !.*
X = 9311.

- The cut in the second version of the query prevents Prolog from backtracking to find the second solution.

Using cuts in later to improve efficiency Recall the code for later:

```
later(date(Y, M, D1), date(Y, M, D2)) :- D1 > D2.  later(date(Y, M1,
_), date(Y, M2, _)) :- M1 > M2.  later(date(Y1, _, _), date(Y2, _, _)) :-
Y1 > Y2.
```

We note that if year and month are the same, all three rules are tried while backtracking. This could be prevented by adding cuts:

```
later(date(Y, M, D1), date(Y, M, D2)) :- D1 > D2, !.  later(date(Y, M1,
_), date(Y, M2, _)) :- M1 > M2, !.  later(date(Y1, _, _), date(Y2, _, _)) :-
Y1 > Y2.
```

This would increase efficiency by eliminating unnecessary backtracking, though it is doubtful if it would be worth bothering about, unless you actually have code that is running too slowly. In that case you should first do an analysis of where the time is being spent, before putting in cuts everywhere! In other cases, adding cuts of this sort to multi-rule procedures might be a useful (if lazy) way of ensuring that only one rule is used in a particular case. Unless it makes the code very clumsy, it is better to use and rely on "condition" goals in each rule (like M1 > M2 in the second rule for later) to specify the case in which it is appropriate. More examples of this are below.

Another cut example

- max, without cut:

```
% max(A, B, C) binds C to the larger of A and B. max(A,
B, A) :-    A > B.
max(A, B, B) :- A =< B.
```

- max, with cut:

```
max(A, B, A) :-
A > B,
  !. max(A, B,
B).
```

- The first version has a negated test in the second rule (=< vs >). The second version substitutes a cut in the first rule for the negated test.
- Remember, no cuts in the first assignment unless they are essential! Hint: the first assignment can be done without cuts.

TOPIC 6: KNOWLEDGE REPRESENTATION AND REASONING

Introduction

- KR and Reasoning is the field of AI that focuses on designing computer representations that capture information about the world that can be used for reasoning, inference, decision-making, and problem-solving.
- It is dedicated to representing information about the world in a form that a computer system can utilize to solve complex tasks such as diagnosing a medical condition or having a dialog in a natural language.
- The justification for KR is that conventional procedural code is not the best formalism to use to solve complex problems.
- KR makes complex software easier to define and maintain than procedural code and can be used in expert systems
- incorporates findings from psychology about how humans solve problems and represent knowledge in order to design formalisms that will make complex systems easier to design and build
- KR goes hand in hand with automated reasoning because one of the main purposes of explicitly representing knowledge is to be able to reason about that knowledge, to make inferences, assert new knowledge, etc.
- Virtually all knowledge representation languages have a reasoning or inference engine as part of the system
- KR and reasoning also incorporates findings from logic to automate various kinds of reasoning, such as the application of rules or the relations of sets and subsets
- Examples of knowledge representation formalisms include: semantic nets, systems architecture, frames, rules, and ontologies.

- Examples of automated reasoning engines include: inference engines, theorem provers, and classifiers

Types of Knowledge:

- Declarative Knowledge: Describes facts about the world (e.g., "The sky is blue").
- Procedural Knowledge: Describes how to perform specific tasks or actions (e.g., "How to ride a bike").
- Meta Knowledge: Describes knowledge about knowledge, including uncertainty, time, and context.

Expressivity and Practicality in KR

- A key trade-off in the design of a knowledge representation formalism is that between expressivity and practicality
- The ultimate knowledge representation formalism in terms of expressive power and compactness is First Order Logic (FOL).
- FOL drawbacks as a knowledge representation formalism - ease of use and practicality of implementation – FOL can be intimidating even for many software developers
- The issue of practicality of implementation is that FOL in some ways is too expressive.
- With FOL it is possible to create statements (e.g. quantification over infinite sets) that would cause a system to never terminate if it attempted to verify them.
- IF-THEN rules provide a subset of FOL but a very useful one that is also very intuitive.
- a subset of FOL can be both easier to use and more practical to implement - a driving motivation behind rule-based expert systems.
- A KR is most fundamentally a surrogate, a substitute for the thing itself, used to enable an entity to determine consequences by thinking rather than acting, i.e., by reasoning about the world rather than taking action in it.
- It is a set of ontological commitments, i.e., an answer to the question: In what terms should I think about the world?
- It is a fragmentary theory of intelligent reasoning, expressed in terms of three components:
  - the representation's fundamental conception of intelligent reasoning; • the set of inferences the representation sanctions; and
  - the set of inferences it recommends.
- It is a medium for pragmatically efficient computation, i.e., the computational environment in which thinking is accomplished
- It is a medium of human expression, i.e., a language in which we say things about the world.

KR and Semantic Web

- Knowledge representation and reasoning are a key enabling technology for the Semantic web
- Languages based on the Frame model with automatic classification provide a layer of semantics on top of the existing Internet.
- Rather than searching via text strings as is typical today it will be possible to define logical queries and find pages that map to those queries
- The Semantic web integrates concepts from knowledge representation and reasoning with markup languages based on XML.
- The Resource Description Framework (RDF) provides the basic capabilities to define knowledge-based objects on the Internet with basic features such as Is-A relations and object properties.
- The Web Ontology Language (OWL) adds additional semantics and integrates with automatic classification reasoners.

Ontology engineering and Ontology language

- In the early years of knowledge-based systems the knowledge-bases were fairly small.
- The knowledge-bases that were meant to actually solve real problems rather than do proof of concept demonstrations needed to focus on well defined problems
- As knowledge-based technology scaled up the need for larger knowledge bases and for modular knowledge bases that could communicate and integrate with each other became apparent
- hence the rise to the discipline of ontology engineering, designing and building large knowledge bases that could be used by multiple projects
- Ontologies provide a formal and explicit specification of a shared conceptualization within a specific domain. They define classes, relationships, and properties to represent the structure of knowledge in a domain.
- A number of ontology languages have been developed. Most are declarative languages, and are either frame languages, or are based on first-order logic(e.g., logic, LISP, etc.)

Reasoning under uncertainty

- Many reasoning systems provide capabilities for reasoning under uncertainty - is important when building situated reasoning agents which must deal with uncertain representations of the world.
- Common approaches to handling uncertainty include the use of:
- certainty factors
- probabilistic methods such as Bayesian inference or Dempster–Shafer theory
- multi-valued ('fuzzy') logic and
- various connectionist approaches

Types of reasoning system

- **Constraint solvers**

- they solve constraint satisfaction problems (CSPs).
- They support constraint programming.
- A constraint is a condition which must be met by any valid solution to a problem
- Constraints are defined declaratively and applied to variables within given domains

- **Theorem provers**
- they use automated reasoning techniques to determine proofs of mathematical theorems.
- They may also be used to verify existing proofs
- In addition to academic use, typical applications of theorem provers include verification of the correctness of integrated circuits, software programs, engineering designs,

- **Logic programs** (LPs)
- are software programs written using programming languages whose primitives and expressions provide direct representations of constructs drawn from mathematical logic.
- An example of a general-purpose logic programming language is Prolog.
- LPs represent the direct application of logic programming to solve problems.
- Logic programming is characterized by highly declarative approaches based on formal logic, and has wide application across many disciplines.

- **Rule engines**
- they represent conditional logic as discrete rules. Rule sets can be managed and applied separately to other functionality.
- They have wide applicability across many domains.
- Many rule engines implement reasoning capabilities.
- A common approach is to implement production systems to support forward or backward chaining.

- **Deductive classifier**
- they arose slightly later than rule-based systems and were a component of a new type of artificial intelligence knowledge representation tool known as frame languages.
- A frame language describes the problem domain as a set of classes, subclasses, and relations among the classes.
- similar to the object-oriented model - Unlike object-oriented models however, frame languages have a formal semantics based on first order logic.
- They utilize this semantics to provide input to the deductive classifier

- **Machine learning systems** –
- evolve their behavior over time based on experience.

- This may involve reasoning over observed events or example data provided for training purposes - For example, they may use inductive reasoning to generate hypotheses for observed facts.
- Learning systems search for generalized rules or functions that yield results in line with observations and then use these generalizations to control future behavior

- **Case-based reasoning systems** (CBR)
- They provide solutions to problems by analysing similarities to other problems for which known solutions already exist.
- They use analogical reasoning to infer solutions based on case histories.
- commonly used in customer/technical support and call centre scenarios and have applications in industrial manufacture, agriculture, medicine, law and many other areas

- **Procedural reasoning systems** (PRS)
- Uses reasoning techniques to select plans from a procedural knowledge base • Each plan represents a course of action for achievement of a given goal
- The PRS implements a belief-desire-intention model by reasoning over facts ('beliefs') to select appropriate plans ('intentions') for given goals ('desires')
- Typical applications of PRS include management, monitoring and fault detection systems.

✓ Introduction to prolog
✓ FOL, predicate, proposition
✓ logic as lag. including logical reasoning.
✓ Elements of prolog.

Atoms, terms, relship, roles, facts, queries.
inbuild function in prolog
Knowledge simulation done using prolog lag.
- Elements as
cat function.
consulting proces
instantiation
resolution
unification

} how they happen in prolog.