Biyani's Think Tank

Concept based notes

Logical and Functional Programming

(B.Tech)

Deepika Shrivastava

Asst. Professor

Deptt. of Engineering

Biyani International Institute of Engineering and Technology

Published by :

Think Tanks

Biyani Group of Colleges

Sector-3, Vidhyadhar Nagar,

Jaipur-302 023 (Rajasthan)

Ph : 0141-2338371, 2338591-95 • Fax : 0141-2338007

E-mail : acad@biyanicolleges.org

Website :www.gurukpo.com; www.biyanicolleges.org

Edition : 2013

Price :

Leaser Type Setted by :

Biyani College Printing Department

Preface

I am glad to present this book, especially designed to serve the needs of the students. The book has been written keeping in mind the general weakness in understanding the fundamental concepts of the topics. The book is self-explanatory and adopts the "Teach Yourself" style. It is based on question-answer pattern. The language of book is quite easy and understandable based on scientific approach.

Any further improvement in the contents of the book by making corrections, omission and inclusion is keen to be achieved based on suggestions from the readers for which the author shall be obliged.

I acknowledge special thanks to Mr. Rajeev Biyani, *Chairman* & Dr. Sanjay Biyani, *Director* (*Acad.*) Biyani Group of Colleges, who are the backbones and main concept provider and also have been constant source of motivation throughout this Endeavour. They played an active role in coordinating the various stages of this Endeavour and spearheaded the publishing work.

I look forward to receiving valuable suggestions from professors of various educational institutions, other faculty members and students for improvement of the quality of the book. The reader may feel free to send in their comments and suggestions to the under mentioned address.

Note: **A feedback form is enclosed along with think tank. Kindly fill the feedback form and submit it at the time of submitting to books of library, else NOC from Library will not be given.**

Author

Syllabus

LOGICAL AND FUNCTIONAL PROGRAMMING: THINK TANK

By : Deepika Shrivastava(Asst.Prof. Btech (CS))

Units

I. PROPOSITIONS AND PREDICATES: Evaluation of constant propositions, Evaluation of proposition in a state. Precedence rules for operators, Tautologies, Propositions a sets of states and Transforming English to propositional form. Introduction to first-order predicate logic, Quantifiers and Reasoning.

II. LOGIC PROGRAMMING USING PROLOG: Constants, Goals and Clauses, Facts,Rules, Semantics, Rules and Conjunction, Rules and Disjunction, Search strategy,
Queries.

III. ADVANCED LOGIC PROGRAMMING USING PROLOG: - Unification ,Recursion, Lists, Cut operator, and Sorting. Data structures, Text strings, Searching
state space, Operators and their precedence, and Parsing in Prolog.

IV. FUNCTIONAL PROGRAMMING: Introduction to lambda calculus-Syntax and semantics, Computability and correctness, Lazy and Eager Evaluation Strategies,
comparison of functional and imperative languages.

V. FUNCTIONAL PROGRAMMING USING HASKELL: Introduction, lists, User-defined data types, type classes, and arrays in Haskell. Input/Ouput in Haskell - type classes IO and Monad, Simple applications/programs in Haskell.

Unit 1 and 2

Propositions , Predicates and Logic programming using Prolog

Q.1 What is Prolog? Explain with uses.

Ans In PROLOG, program logic is expressed in terms of relations, and a computation is initiated by running a query over these relations. Relations and queries are constructed using Prolog's single data type, the term. Relations are defined by clauses. Given a query, the Prolog engine attempts to find a resolution refutation of the negated query. If the negated query can be refuted, i.e., an instantiation for all free variables is found that makes the union of clauses and the singleton set consisting of the negated query false, it follows that the original query, with the found instantiation applied, is a logical consequence of the program. This makes Prolog (and other logic programming languages) particularly useful for database, symbolic mathematics, and language parsing applications. Because Prolog allows impure predicates, checking the truth value of certain special predicates may have some deliberate side effect, such as printing a value to the screen. Because of this, the programmer is permitted to use some amount of conventional imperative programming when the logical paradigm is inconvenient. It has a purely logical subset, called "PURE PROLOG", as well as a number of extra logical features.

Its Implementation is done in 6 stages:

1. ISO prolog
2. Compilation
3. Tail recursion
4. Term indexing
5. Tabling
6. Implementation in hardware

A Prolog program is composed of predicates defined by facts and rules, which are special cases of definite (or Horn) clauses, i.e. clauses with at most one positive literal (the head of the clause).

- A predicate may have several alternative definitions (both facts and rules).

A fact has no negative literals and expresses positive knowledge that is definite (no disjunctions). Example: The knowledge that John is a child of Ann and Alex, and that Ann is a child of Bob, is expressed by two facts, namely

```
child_of(john, ann).
child_of(john, alex).
child_of(ann, bob).
```

A rule has one or more negative literals (the body of the clause), and is used to infer predicates from other predicates. Example: A grand child is the child of the child, i.e.

```
grand_child_of(X,Y) :-
child_of(X,Z),
child_of(Z,Y).
```

Q.2 Explain constant , goals ,clauses fact & rules of prolog program ?

Ans

Constant :- Constants are the value that does not change.constant can be either atom or number.there are two kinds of constant:

1) Atoms
2) Integers

- **Atom**: Strings of characters starting with a lower-case letter or enclosed in Apostrophes. Example of atoms is the name that was given:

  Like Mary john book wine owns jewels can steal
  The special symbols tht prolog uses to denote question "?-" and rules ":-" are also atoms.

- **Numbers**: Strings of digits with or without a decimal point.
  Variables are strings of characters beginning with an upper-case letter or an underscore.
  Structures consist of a functor or function symbol (looks like an atom), followed by a list of terms inside parentheses, and separated by commas.

- **Integer**: integer is used to repesent the number value. Integer are consisting only of digits and may not contain a decimal no.
  For example: 0 1 99 123 457 all are integer no.

Goals :- A goal is an object whose truth or falsity we can check. A goal that turns out to be true is said to succeed. A goal that turns out to be false is said to fail.

In a **compound goal**: all the specified condition must succeed for the goal to succeed . prolog works from left to right in providing the compound goal.once the first part of goal is satisfied ,the variable is bound ,if other part of the goal contain the same variable , they must be satisfied with the same binding .

The purpose of submitting a goal is to find out whether the statement represented by the goal is true according to the knowledge database (i.e. the facts and rules in the consulted program). This is similar to proving a hypothesis - the goal being the hypothesis, the facts being the axioms and the rules being the theorems.

For example:

Like (Mary, food).

Like (Mary, wine).

Like (Mary, food).

Like (Mary, food).

We want to ask a question to prolog "Is there anything that John and Mary both like?

it consist of two goal:

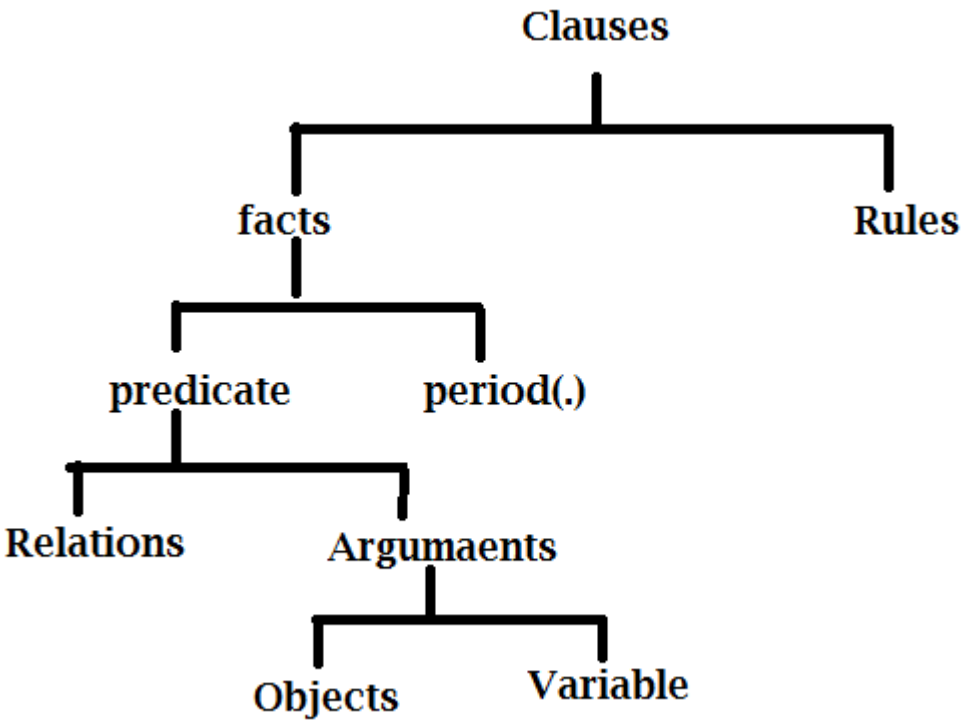
- First, find out if there is some X that mary likes.
- Then, find out if John likes whatever X is.

In prolog two goal would be written as…

Likes(mary, X) , likes(John,X).

Clause:-

In Prolog, a clause represents a statement about objects and relations between objects. Clauses represent the knowledge which the Prolog system can draw upon when processing a query.For Ex. Likes (john, mary) . ……..fact(clause)



Facts:-

A fact must start with a predicate (which is an atom) and end with a fullstop. The predicate may be followed by one or more arguments which are enclosed by parentheses. The arguments can be atoms (in this case, these atoms are treated as constants), numbers, variables or lists. Arguments are separated by commas.

A fact is a predicate expression that makes a declarative statement about the problem domain. Whenever a variable occurs in a Prolog expression, it is assumed to be universally quantified. Note that all Prolog sentences must end with a period.

```
likes(john, susie).        /* John likes Susie */
likes(X, susie).           /* Everyone likes Susie */
likes(john, Y).            /* John likes everybody */
```

- A fact is a Prolog clause without a clause body.

- A fact represents unconditional knowledge, i.e. it describes a relation between objects which always holds true, regardless of any other relations. Thus, provided their clause heads are identical, a fact is equivalent to a rule whose clause body always has the truth value TRUE.

In prolog we can express facts as a symbolic relationship .want to tell prolog the fact that "john likes mary"

For example –suppose we want to tell prolog the fact that "john likes mary" .this fact consist of two objects, called "mary" and "john" and a relationship ,called "likes".

In prolog we can express the fact above as

likes(john,mary).

- In which the name of all relationships and objects must begin with a lower case –letter ,for example ,likes,john,mary.
- The relationship is written separated by commas and the objects are enclosed by a pair of brackets .
- The full stop character "." Must come at the end of a fact .

The factual expression in prolog is called a clauses .if we remove the dot (.) operator from the expression on fact ,then the entire expression before the dot is called a predicate .

For example :

Likes(john,mary). …… … fact[clauses]

Likes(john,mary). ……..predicate

Facts and rules are part of clauses

There are two kinds of clauses: rules and facts.

Rules:

- A rule is a Prolog clause containing a clause body.
- A rule represents conditional knowledge clause body in the equivalent normal structure

The most important operations between subgoals (conditions) in the clause body are the following:

A rule is a predicate expression that uses logical implication (:-) to describe a relationship among facts. Thus a Prolog rule takes the form

left_hand_side :- right_hand_side.

This sentence is interpreted as: left_hand_side if right_hand_side. The left_hand_side is restricted to a single, positive, literal, which means it must consist of a positive atomic expression. It cannot be negated and it cannot contain logical connectives

For example:

John like X if X like wine. Yhis can be expressed as

Like (john, X):-likes (X, wine).

In prolog:-

1) A rule consists of a head and body.
2) The head and body are connected by a symol ":-" which is made up of a colon and a hyper.
3) The rules are also ended with a dot.
4) Head of the rule describes what fact the rule is intended to define.
5) The body describe the conjuction of goal that must be satisfied.

Q.3 How to transform English to prepositional form?

Ans INTRODUCTION:

The purpose of this set of example is to illustrate how informal English lang. description are translated into formulas in prepositional logic and advantage of making such a translation is that formal description can be analysed using formal logic.

Consider a sentence " if it rains,the picnic is cancel". Let identifier r stand for the proposition "it rains" and let identifier pc represent " the picnic is cancelled". Then sentence can be written as r- pc.

As shown by this example, the technique is to represent "atomic parts" of a sentence- how these are chosen is up to the translator- by identifiers and to describe their relationship using Boolean operator.

Example-

r: it rains

pc: picnic is cancelled

wet: be wet

s: stay at home

1. If it rains but I stay at home,I won't be wet:
   (r^s) => ~wet
2. I will be wet if it rains:
   r => wet
3. If it rains and the picnic is not cancelled or I don't stay home, I'll be wet:
   (r^(~pc)v~s) => wet
4. Whether or not the picnic is cancelled, I'm staying home if it rains:
   (pcv~pc)^r => s
5. Either it doesn't rain or I'm staying home:
   ~r v s

Q.4 Write the merits and demerits of Prolog.

Ans MERITS:

- We can compile stand alone program that will execute on a machine that is not running prolog. These stand alone programs can be sold or distributed to users without paying any royalty.
- A functional interfaces to other language is provided allowing procedural languages to support to be added to any prolog system.
- Declared variables are used to provided more secure development control.
- Both integer and real arithmetic editor is provided, making program development compilation and debugging very easily.
- A full compliment of standard predicates for many functions such as string operations, random file access, cursor control,graphics, windowing and sound are available.

DEMERITS:

- Prolog does not allows disjunction ("OR") of facts or conclusion such as "if car does not start and the light does not come on , then either battery is down or problem with ignition or some electric fault".
- Some rules cannot be expressed in prolog.
- Prolog does not allow us to express negative facts or conclusions.

- Prolog does not allow facts, rules having existential quantification.
- Prolog does not allow directly second order logic, but meta level facilities of prolog.

Q5 What do you mean by tautology ,contingency and contradiction ?

Ans **TAUTOLOGY** : A compound statement i.e always true for all possible truth values of its propositional variable is called tautology or valid.

Eg ; (a=>b) ⇔(~ b=>~a)

a	b	a=>b	~b	~a	~b=>~a	(a=>b)⇔(~b=>~a)
T	T	T	F	F	T	T
T	F	F	T	F	F	T
F	T	T	F	T	T	T
F	F	T	T	T	T	T

CONTRADICTION : A compound statement that is always false, is called a contradiction or absurdity.

Eg; (p^q)^~(pvq)

P	q	P^q	pvq	~(pvq)	(p^q)^~(pvq)
T	T	T	T	F	F
T	F	F	T	F	F
F	T	F	T	F	F
F	F	F	F	T	F

CONTINGENCY : A statement that is neither a tautology nor a contradiction is called a contingency.

Eg; (p=>q)^(pvq)

p	q	p=>q	pvq	(p=>q)^(pvq)
T	T	T	T	T
T	F	F	T	F
F	T	T	T	T
F	T	T	F	F

Q.6 Explain quantifier in detail?

Ans QUANTIFIERS:-

A quantifier is an expression that reports a quantity of times that a predicate is satisfied in some class of things .(i.e in a domain) . a quantifier turns a propositional fnction into a proposition without assiging specific values for the variables.

There are two quantifiers :-

1. Universal quantifiers .
2. Existential quantifiers.

- **UNIVERSAL QUANTIFICATION** :- a universal quantification is a type of quantifier , the universal quantification of a predicate P(x) is the proposition (statement)

  "P(x) is true for all values of x in the universe of discourse"

  It is usually denoted by turned A logical oprater symbol ,which, when used together with a predicate variable,is called a universal quantifiers .

- **EXISTENTIAL QUANTIFICATION** :- an existential quantification is a another type of quantifier, the existential quantification of a predicate P(x) is the proposition

  "there exists an element x in the universe of discourse such that P(x) is true"

  It is usually denoted by turned E logical oprater symbol ,which,when used together with a predicate variable ,is called an existential quantifiers .

Q7 Explain the searching state space. What are the different methods of searching state space?

Ans A state space is a graph whose nodes correspond to problem situation, and a given problem is reduced to finding a path in the graph.

In searching state space we use three methods to solve the problem.

- Introductory Concepts
- Depth-first Search
- Breadth-first search

1. **Introductory concepts:**

In this method we use an example to define the problem. The problem is to find a plan for rearranging a stack of block as shown in the figure 1.

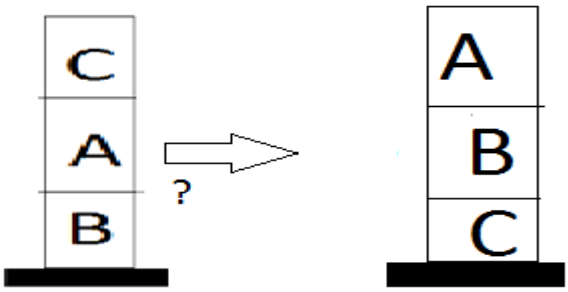


Fig.1

We are only allowed to move one block at a time. A block can be grasped only when its top is clear. A block can be put on the table or on some other block. To find a required plan, we have to find a sequence of moves that accomplish the given transformation.

In the given problem first we put the C in empty stack.

Step: 1

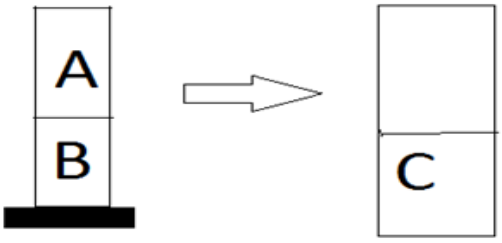


fig. 1.1

Step: 2 In the second step we move A in another empty stack.

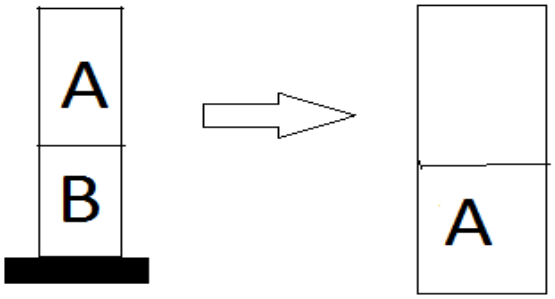
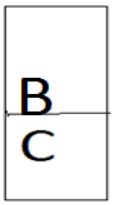
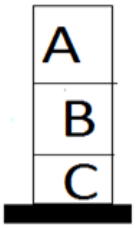


fig. 1.2

Step: 3 In this step we move the B on C in fig 1.1



Step: 4 In this step we move A on B



So we solve the given problem.

As the example illustrates, we have, in such a problem, two type of concept:

1. Problem situation.
2. Legal moves, or action, that transform situation into other situation.

Problem situation and possible moves form a directed graph, called a state space. A state space for our example problem is shown in fig. 2. The nodes of the graph correspond to problem situation, and the arcs correspond to legal transition between states. The problem of finding a solution plan is equivalent to finding a path between the given initial situation (the start node) and some specified final situation, also called a goal node.

The state space of a given problem specifies the 'rules of the game': nodes in the state space correspond to situation, and arcs correspond to 'legal moves', or action, or solution steps. A particular problem is defined by:

- A state space
- A start node
- A goal condition; 'goal node' are those nodes that satisfy this condition.

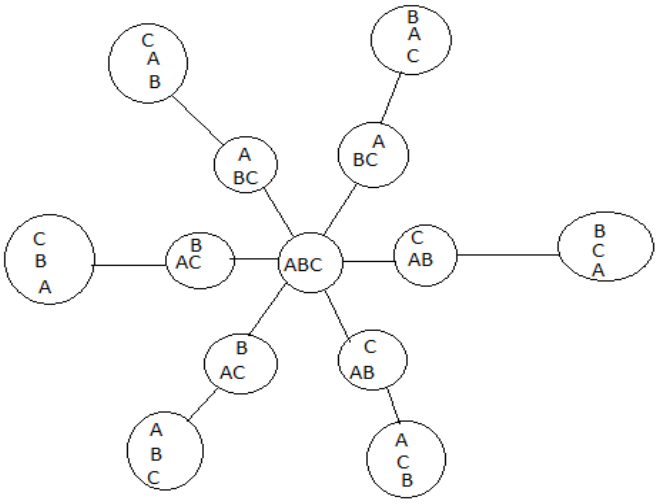


Fig. 2 A state - space respresentation of the block manipulation problem. The indicated path is a solution to the problem in fig. 1

We can attach cost to legal moves or actions.

In cases where costs are attached to moves, we are normally interested in minimum cost solution. The cost of a solution is the sum of the costs of the arcs along the solution path. Even if no costs are given we have an optimization problem: we may be interested in shortest solutions.

Before presenting some program that implement classical algorithm for searching state space, let us first discuss how a space can be represented in a prolog program.

We will represent state space by a relation S(X, Y)

Which is true if there is a legal move in the state space from a node X to a node Y. We will say that Y is a successor of X.

If there are costs associated with moves then we will add a third argument, the cost of the move:

S(X, Y, Cost)

A problem situation can be represented as a list of stack. Each stack can be turn, represented by a list of block in that stack ordered so that the top block in the stack is the head of the list. Empty stacks are represented by empty lists. The initial situation of the problem in fig. 2 can be thus represented by:

[[c, a, b], [], []]

A goal situation is any arrangement with the ordered stack of all the blocks. There are three such situations:

[[a, b, c], [], []]

[[], [a, b, c], []]

[[], [], [a, b, c]]

The successor relation can be programmed according to the following rule: situation2 is a successor of situation1 if there are two stack, stack1 and stack2, in situation1, and the top block of stack1 can be moved to stack2. As all situations are represented as lists of stack, this is translated into prolog as:

```
s(Stacks, [Stack1,|Top1| Stack2 ]| OtherStacks]):- % Move top to Stack2

del ([Top1 | Stack1], Stacks, Stacks1),      %Find first stack

 del (Stack2, Stack1, otherStacks).       %Find Second Stack

del (X,[X|L],L).

del (X, [Y|L], [Y|L1]):-

del (X, L, L1).
```

The goal condition for our example problem is:

```
goal (situation):-

 member ( [a, b, c], situation).
```

We will program search algorithm as a relation

solve (Start , Solution)

where start is the start node in the state space, and solution is a path between start and any goal node. For our block manipulation problem the corresponding call can be:

?- solve ([[c, a, b], [], []], Solution).

As the result of the successful search solution is instantiated to a list of block arrangements. This list represents a plan for transformation the initial state into a state in which all the three blocks are in one stack arranged as [a, b, c].

2 **Depth - first search and iterative deepening:**

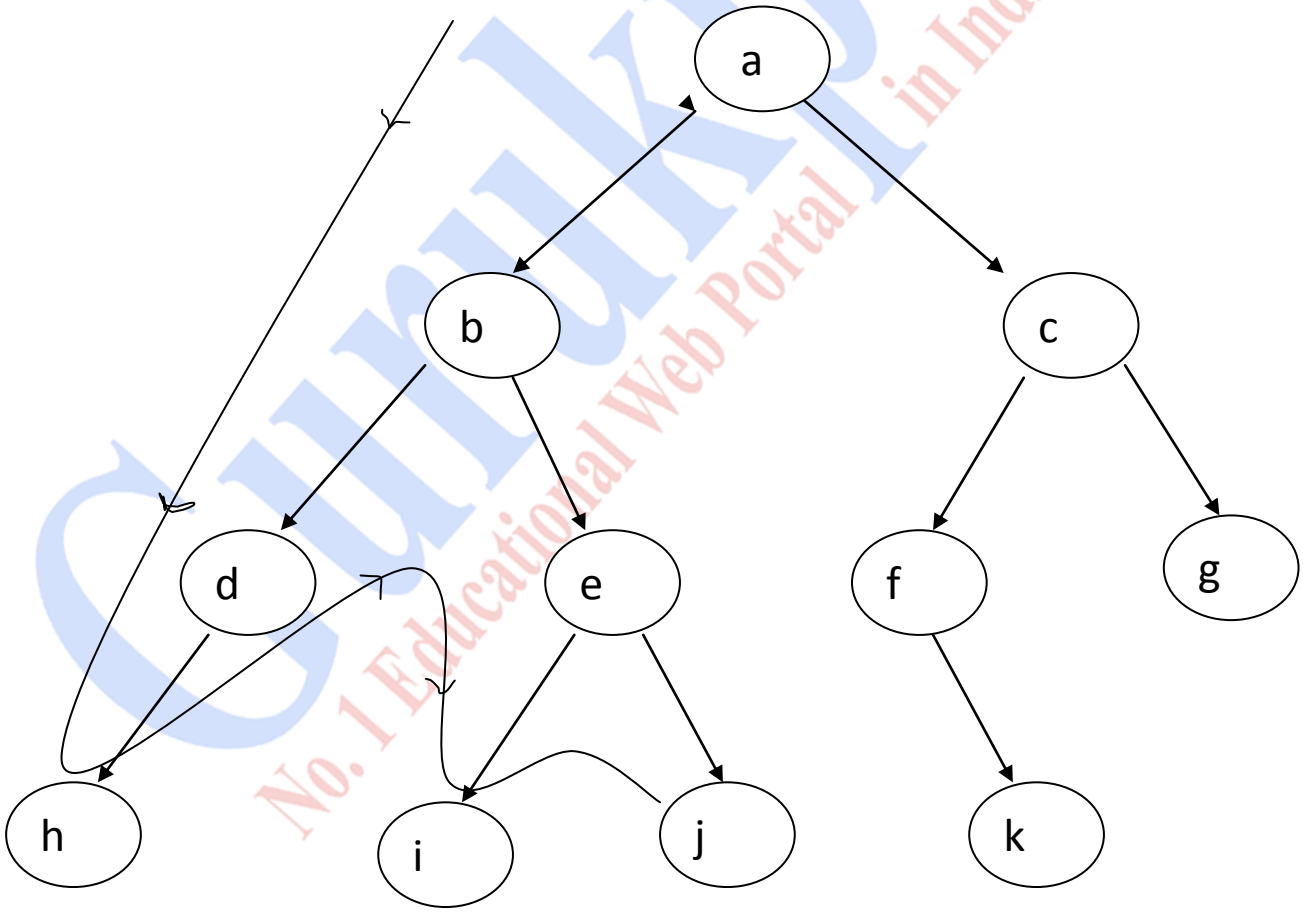
Dfs traversal follows first a path from a starting node to an ending node. Then another path from the start to the end, and so on until all nodes have been visited. Variations in dfs is known as iterative deepening.

Algorithm and its variations:

To find a solution path,sol,from a given ode,N,to some goal node:

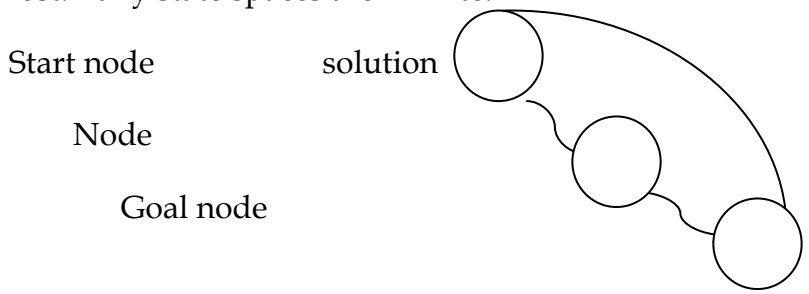
If N is a goal node then sol=[N].

- If there is a successor node,N1,of N,such that there is a path sol1 from N1 to a goal node,then sol=[N | sol1].



This is the order in which nodes are visited.

With the cycle-detection mechanism,our depth-first procedure will find solutions paths in state spaces. There are, however, state spaces in which this program will still easily get lost.many state spaces are infinite.



Relation depthfirst(path,node,solution).

3. **Breadth-first search:**

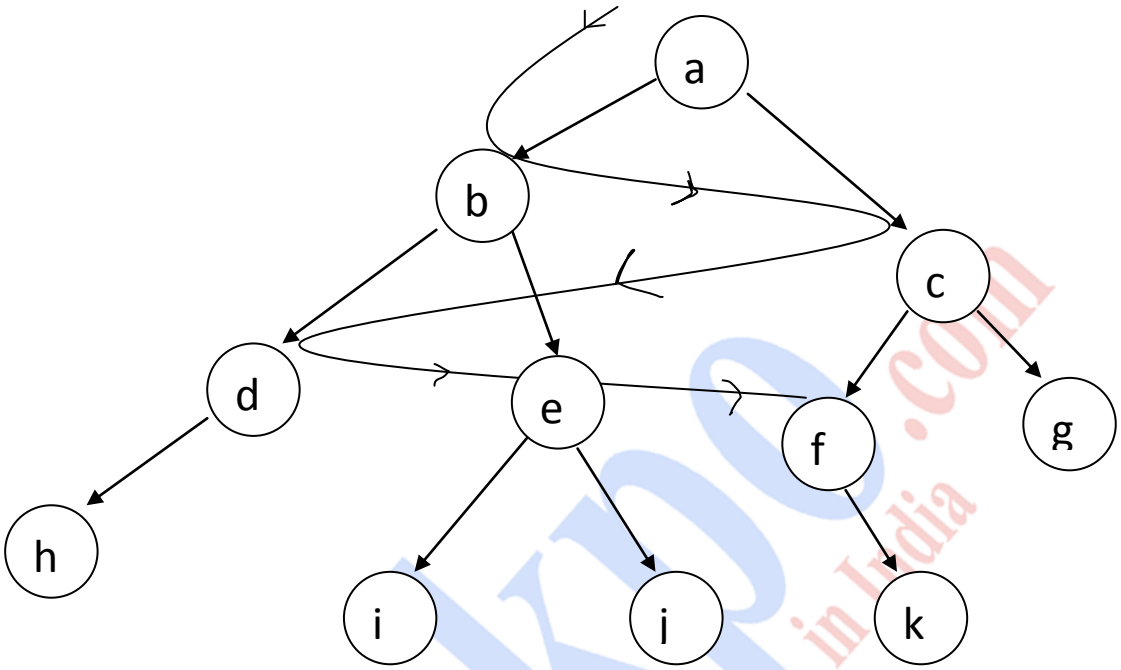
   In a bfs, we will require a starting vertex from which this search will begin. First vertex is selected as the start position,it is visited and printed,and then all unvisited vertices,adjacent to it are visited and printed in some sequential order.finally the unvisited vertices immediately adjacent to these vertices are visited and printed and so on,until the entire graph has been traversed.

   An outline for bfs is:

   To do the bfs when given a set of candidate paths:

   If the head of the first path is a goal node then this path is solution of the problem.

   Remove the first path from the candidate set and generate the set of all possible one-step-extension at the end of the candidate set and execute bfs on this update set.



Unit 3

Advanced Logic Programming Using Prolog

Q.1 Explain data structure using in prolog.

Ans. Basically we use three types of data structure in prolog.

1.Term

2.Unification

3.Operation

TERM:-

Term is a basic data structure in prolog, i.e. , everything is include program and data is expressed in form of term .

There are 4 basic types of terms in prolog:-

- Variables
- Compound terms
- atoms
- numbers

Prolog also provides built-in predicates to access structure of the nonvar terms as well to construct term.

If one needs to copy a term, it is possible to use predicate copy_ term|2 which is built-in in most prolog system.

However, it is straightforward to write a copy_ term in prolog using above mentioned predicates.

During copying one has to remember copies of variable which can be used further during coping. Therefor the register of variable coping is maintained.

UNIFICATION:-

Unification is an engine of prolog. It tries to find most general substitution of variables in two terms such that after applying this substitution to both terms, the terms become the same. To unify terms A and B, one can easily invoke built-in unification A=B. Again, it is straightforward to write prolog code of unification.

We will understand this with the help of this program:-

```
Unify (A,B):-

Atomic (A) , atomic (B), A=B.

Unify (A=B):-

var (A),A=B.

unify (A=B):-

nonvar (A), var(B),A=B.

Unify (A=B):-

nonvar (A), var (B),A=B.

Unify (A=B):-

   Compound (A), compound (B),

A=.. [F | ArgsA], B=.. [F | ArgsB],

Unify_ args (ArgsA, ArgsB).

Unify_ args ([A | TA], [B | TB]):-

   Unify (A,B),

   Unify_ args (TA,TB).

Unify_ args ([], []).
```

Prolog system does not incorporate occurs check because of its time consuming nature. So, occurs check tests the occurrence of the variable X in the term T during unification of X and T.

OPERATORS:-

Writing terms in the form fonctor (arg1, arg2…….) is not often appropriate from the human point of view. Just compare the following two transcription of the same prolog clause.

p(X, Z):- q(X, Y), r(Y,Z),s(Z).

':-'(p(X, Y), (',')(q(X,Y),',')(r(Y,Z),s(Z))))).

Which one do you prefer????

To simplify the entry of the terms, prolog introduces operators which enable "syntactic sugar", i.e., more natural way of writing terms. Operators are used with unary and binary terms only. They enable to set the location of the function (prefix, infix, postfix), the associative feature and, finally, the priority among operations.

Q2. Define the text string in prolog?

Ans A string is a sequence of ASCII characters they are typically used for storing text.

A string is represented by zero or more characters enclosed in double quotes " ".

The body of string may contain any ASCII characters.

Within the body of a string, the following syntactic apply:

Any printable character(one with an ASCII code of 32 or above) except the backslash/ or double quotes " represent itself.

An unprintable character (one with an ASCII code below 32) may be represented by the backslash characters.

b representing backspace

f representing form-feed (new page)

n representing line feed

t representing horizontal tab

" representing double quote character

/ representing backslash itself

A backslash followed by an integer produces the character represented by that ASCII code number, the integer must be to base 10 with not more than three digits.

All character of the string must be of same type.

A string can be of any length.

Order of a character in string is very important.

A string can be empty or null.

Example:-

"Ring the bell : \7 and newline\r\n"

"Note equal:= \008/"

Q3. Explain all operators and their precedence in brief?

Ans. The commonest extension to basic prolog is operator syntax. Operator definitions allow function defines as operators to be used in a different way from normal functions. If we use unextended prolog syntax '+' sign for addition.

OPERATOR TYPE

1. Whether the atom is infix, prefix or posfix;

2.what the" associativity " of the atom is- whether it evaluate right to left or left to right.

An infix operator must have two arguments, and lie between them; a prefix operator must have a single argument, which it must precedence; and a postfix operator must have a single argument, which it must follow (postfix is the rarest of the three).

The associativity of operators is only relevant when parsing a structure which contains several operators and which contains several operators and which can only be parsed unambiguously when the types (prefix, etc..,) andprecedences of these operators are considereded.

OPERATOR PRECEDENCES AND PRECEDENCE NUMBERS

Operator precedences determine the order the order in which operators are carried out in an expression: one operator will always have precedence over another .

The precedence number of an operator is an integer which determine how tightly it binds its arguments. The smaller the precedence number of an operator, the more tightly it binds its arguments. Prolog -2 precedence numbers range from 1 to 1200.

USING OP/3 TO DEFINE OPERATORS

The prolog-2 BIP op/3 defines an atom as an operator. The three arguments are the predence number of the operator, the type of the operator, and the atom we are defining as an operator:

We can also use op/3 to clear an operator declaration and return an operator to ordinary atom status; we do this by making a new operator declaration and return an operator to ordinary atom status : we do this by making a new operator declaration for the atom, but using the atom none as second argument (operator type) to op/3;

RETRIEVING INFORMATION ON OPERATORS

The prolog-2 BIP atomprops/3 will tells us the operator details of an atom.like op/3 ,the first argument is the precedence number of the operator, the second its type,and the third the atom.

Atomprops/3 is resatisfiable; if itused with with three variables arguments it will return all the atoms known to the system.

If there are several declarations for an atom then atomprops will backtrack through the lists and give us them

Q.4 Explain Parsing In Prolog.

Ans. A grammar for a language is a set of rules for specifying what sequences of words are acceptable as sentences of that language.it specifies how the words must group together into phrases and what orderings of these phrases are allowed. If the sequence is indeed acceptable , the process of

verifying this will have established what the natural group of words are and how they are put together.
A Particularly simple kind of grammer is known as "context free" grammer. Rather than give a formal definition of what such a thing is, we will illustrate it by means of a simple example.

Sentence--> noun _phrase, verb _phrase

Noun_phrase --> determiner, noun_phrase

Verb_phrase --> verb

Determiner --> [the]

Noun --> [apple]

Noun -->[man]

Verb --[eats]

Verb --> [sings]

The first rule says that a sentences consists of a phrase called a nonn_ phrase. These two phrases are what are commonly known as the subject and predicate of the sentence:

Informally, a noun phrase is a group of words that names a thing (or things). Such a phrase contains a word, the "noun", which gives the main class that the thing belongs to. Thus " the man" names a man, "the program" names a program and so on. Also according to this grammer , he noun is preceded called a determiner".

Does the sequence decompose into two phrases, such that the first is an acceptable noun_ phrase and the second is an acceptable verb _ phrase?

At the end, if we succeed , we will have looked at all phrases and sub phrases of the sentence, as specified by the grammer , and will have established a structure.

A computer program that construct parse trees for sentences of a language we shall call a parser.Th idea are not confined to application concerned with syntax of natural languages. Indeed the same techniques apply to any problem where we the arrangement of these groups can be specifiedby a set of rules.

Q.5 Explain the following predicates with suitable eg.

1. **Cut Predicate**
2. **Fail Predicate**
3. **Not Predicate**

Ans.

1. Cut Predicate:

Prolog is diff. from other languages as it deals with facts and rules. Cut is used to control the program execution. Cut predicate is the most important and complex feature of prolog. Cut is used to prevent backtracking based on specific condition. Symbol of cut predicate is '!'. Cut predicate do not have any argument. Cut predicate always succeed and after success it block backtracking beyond the cut. If any premise beyond the cut fails, prolog only backtracks upto cut. If the rule itself fails and the cut is the last premise, then no other rules with the same head will be tried. Using cut prolog accept the clause as the decisive clause, so prolog had no need to check other clauses.

Basic purpose of adding cut is to eliminate certain search paths in the problem space.

The cut act like a fence, where it is placed in the program. We cannot backtrack beyond the cut.

Let take an example of Cut Predicate:-

Domains

Name, c name, s name = string

Predicates

Location (c name, s name)

Go

Checkstate (name)

Clause

Go :-

```
        Write (city, state)
        Fail
    Go :-
        Location (city, state)
        Checkstate (state)
        Write (city, state)
        Fail.
    Go :-
        Location ("Jaipur", "Rajasthan")
        Location ("Bikaner","Rajasthan")
        Location ("Surat","Gujrat")
        ! , fail
        Checkstate (-)
  Goal
    go
```

When we run this program, then our goal go will unify with first go clause which is rule. Then city and state will be printed and then the fail predicate forces the backtracking .write predicate always true. Now goal will be unified with next go clause. In unification process, when the checkstate ("Gujrat") fails, the cut predicate prevents prolog from backtracing to the checkstate(state) premise and will try to prove it again. So when checkstate ("Gujrat") conclusion fails, prolog gives upon proving checkstate with "Gujrat" binding so it backtracks to location (city, state) and the next variable binding is tried. The final list contains all but the state with Gujrat address.

2.Fail Predicate

Fail predicate is used for controlling the execution of prologs program. Specially, prologs backtracking mechanism results in unnecessary searching. So it cause inefficiency. For example, some times we want to find out unique solutions of a given goal.

To control this we use a mechanism to control backtracking. This is done with the help of a predicate. Fail predicate is used to force backtracking. Fail predicate forces the backtracking, in an attempt to unify with another clause. When faul predicate is used, the goal being proved immediately fails and the backing is initiated. Fail predicate have no argument. So failing of fail predicate does not depend on variable binding and thus the fail predicate always fails.

For example –

Let take a rule.

Go:-

Go12,

Write ("this line will never execute")

Go12 :-

Fail

Let goal Is go.

Then during matching process, prolog will unify the goal with the head of first rule. Now for the head to be true, prolog tries to prove its premise which is go12. now go12, will unify with the head of second rule. The second rule contains fail, so goal will fail and the prolog will backtrack to the first rule again because fail predicate forces the backtracking.

Now again go goal fails and this the write predicate will never be executed.

The output of this goal will be false.

Let take another example:

Go :-

Write ("this line will be executed")

Go12

Go 12 :-

Fail

Goal : go.

The output of this query will – this line will be executed.

The goal will fail. But this time the string will be printed.

Let take an example of fail predicate:

Domains

Name, color = symbol

Predicates

Location (name, color)

Go

Clauses

Go:-

Fruit (name, color) ,

Write(name), nl,

Write(color), nl

Fail.

Go,

Fruit (apple, green)

Fruit (banana, yellow)

Fruit (mango, yellow)

Goal : go.

The output of this program will be:-

Apple

Green

Banana

Yellow

Mango

Yellow

When our goal is go, then it unifies with the head of first rule. For first rule to be true, all of its premises must be true. In first premises, the location(name, color) predicate forces name to bind with apple and color to bind with green. Then prolog marks this place, in case it needs backtracking. Then the variable names are displayed. Then there is a fail predicate which cause failing of this rule, and forcing the prolog to backtrack. Now writef predicate is always true, so prolog backtracks further at predicate location(city, state). At this time all the two variables fruit and color are freed. Prolog returned to this marked place and tries to find another match. Now, fruit is bind with banana and color is bind with yellow. then again, writef predicate write the values of two variables. Then again fail predicate fails the rule. Again backtracking starts and now fruit and colour are bind with mango and yellow, values of these are then displayed. Again fail predicate, cause failing of rule. Now, the prolog backtrack to second go claws. This is successful.

So from the above program, we can note that the rule using fail predicate always fails. So we have need to add a terminating claws that always succeeds to make the program better. This extra claws that we have to add is called terminating condition.

We can summarize from the above program.

Some imp. Points while using fail predicate:

1) The order of go clause is very imp. Their order defines an algorithm. If we reverse the order of two given go clause then the goal succeeds, but no list printed.
2) The second clause us called the terminating clause. And it is the imp. Part of the definition.
3) Whenever the rule fails , the variables in the clause loose their bindings . and thus backtracking forces new binding.
4) Prolog marks the place of each binding. So it returns and unify with the next clause in the sequence of location clause.
5) Prolog succeeds from left to right, going as far as it can on one path before backtracking and trying another.

3. Not predicate:-

When we want to xpress in the data base, that particular fact is not true, then we use not predicate. The not predicate is used in premise, such as:

Prepare (ram, exam) :-

Not (is, (ram, pass)).

Means ram is to prepare for the exam if ram is not passes.

So if the fact → is (ram, pass) us n then database, then this will fail.

Q6 What is recursion?

Ans Recursion is actually a way of defining functions in which the function is applied inside its own definition. Definitions in mathematics are often given recursively. For instance, the fibonacci sequence is defined recursively. First, we define the first two fibonacci numbers non-recursively. We say that *F(0) = 0* and *F(1) = 1*, meaning that the 0th and 1st fibonacci numbers are 0 and 1, respectively. Then we say that for any other natural number, that fibonacci number is the sum of the previous two fibonacci numbers. So *F(n) = F(n-1) + F(n-2)*. That way, *F(3)* is *F(2) + F(1)*, which is *(F(1) + F(0)) + F(1)*. Because we've now come down to only non-recursively defined fibonacci numbers, we can safely say that *F(3)* is 2. Having an element or two in a recursion definition defined non-

recursively (like *F(0)* and *F(1)* here) is also called the **edge condition** and is important if you want your recursive function to terminate. If we hadn't defined *F(0)* and *F(1)* non recursively, you'd never get a solution any number because you'd reach 0 and then you'd go into negative numbers. All of a sudden, you'd be saying that *F(-2000)* is*F(-2001) + F(-2002)* and there still wouldn't be an end in sight!

Unit 4

Functional Programming

Q1 What is lambda calculus?

Ans The lambda calculus is a formal mathematical system devised by Alonzo Church to investigate functions, function application and recursion. It has influenced many programming languages but none more so than the *functional programming languages*. Lisp was the first of these although only the "pure" Lisp sublanguage can be called a true functional language. Haskell, Miranda and ML are more recent examples. Lambda calculus also provides the meta-language for formal definitions in ***denotational semantics***. It has a good claim to be the prototype programming language.

Binding: free versus bound

λ is a binding operator, just like backwards E ("∃") or upside down A ("∀"). Consequently, it always binds something (a variable), taking scope over some expression that (usually) contains occurrences of the bound variable. More practically, λ always occurs in the following configuration:

(λ var body)

Here, "var" is the variable bound by the lambda operator, and "body" is the constituent that the lambda operator has scope over.

For instance:

(λ x (* (+ 3 x) (- x 4)))

Here "x" is the variable (the one immediately after the binding operator), and the body is "(* (+ 3 x) (- x 4))". The body contains two occurrences of "x", and both are **bound** by the λ operator; any variable token that is not bound is **free**.
(λ x (* (+ y x) (- x z)))

In this form, the tokens of "y" and of "z" are both free.

Freedom is always relative to a particular expression. If the preceding expression is embedded within a larger one, the free variables can come to be bound:

(∃ y (λ x (* (+ y x) (- x z))))

In this case, the "y" has been bound, and only the "z" remains free.

Q.2 What is λ-reduction?

Ans λ-reduction

The key notion of the λ-calculus is that it is possible to arrive at a logically equivalent expression by means of a process called λ-reduction. In the usual case, λ-reduction is actually a combination of three distinct reduction operations, each of which is discussed below. The key operation, the one that does the heavy lifting, is called β-reduction, and that is operation we will discuss first.

By the way, some people say "λ-conversion" instead of λ-reduction; others reserve "λ-conversion" to refer specifically to a single step in a series of reductions. This tutorial is trying not to be overly pedantic, so for present purposes I don't care.

Q.3 What is β-reduction?

Ans β-reduction (the heavy lifting)

Nothing happens until a λ-binding form occurs in construction with an argument, thus:

((λ var body) argument)

Once a λ-based binding form occurs with an argument like this, it is possible to reduce the expression to a simpler form by means of β-reduction (sometimes with the help of α-reduction and η-reduction).

The main idea of β-reduction is to replace every free occurrence of the variable "var" in "body" with "argument". For instance, in the form below, both occurrences of "var" in the body are free.

```
((lambda var ((fn1 var) & (fn2 var))) argument)
```

Consequently, after β-reduction, both occurrences get replaced with "argument", and the result is significantly simpler than the original expression.

Q.4 What is α-reduction?

Ans α-reduction (alphabetic variants)

Unfortunately, applying β-reduction indiscriminately can cause trouble when the body contains binding operators. Consider:

```
((lambda x (lambda y1 (x y1))) :
```

Following the rules of β-reduction, we replace "x" with "z" for a result of "(λ y1 (z y1))", and this result is correct. Note that in the result the λ-operator binds exactly one variable in the body, and the other variable remains free.

But consider what happens in this closely parallel but slightly different situation:

((λ x (λ y (x y))) y)
(λ y (y y))

β-reduction: substitute "y" in for "x" in the body "(λ y (x y))"
==>

Wrong result!

The only difference here from the first example is that the argument is "y" and not "z". But now, blindly following the rules of β-reduction would produce the result "(λ y (y y))", in which the λ-operator binds two variables, not just one, which is not correct. The argument variable "y" is said to have been `captured' by the inner λ-operator; another commonly-used expression for this kind of situation is `variable collision'.

The solution is to make use of alphabetic variants. Roughly, two expressions are "alphabetic variants" if they are identical except perhaps for the choice of variable symbols. For instance, the following expressions are alphabetic variants of one another:

(λ x (λ y (x (+ y x))))
(λ z (λ y (z (+ y z))))

To create an alphabetic variant for an expression of the form "(λ var body)", simply replace each free occurrence of "var" in the expression with "new", where "new" is a variable symbol not occurring anywhere in "body". (Since expressions have finite length, as long as there is an infinite supply of variable symbols, it will always be possible to find a suitable variable to serve the role of "new".) This transformation is called α-

reduction. The crucial property of the reduced form is that each λ operator binds the same number of variables in the same positions within its body.

Some examples of alphabetic variants:

1. ((λ x x) x) ((λ y1 y1) x)	α-reduction on "(λ x x)", substituting "y1" for "x" ==>
2. ((λ x x)(λ x (x x))) ((λ y1 y1)(λ x (x x)))	α-reduction on "(λ x x)", substituting "y1" for "x" ==>
3. (λ x (λ x (x x))) (λ y1 (λ x (x x)))	α-reduction on "(λ x (λ x (x x)))", substituting "y1" for "x" ==>

The third example may look surprising; the key fact is that α-reduction specifically targets only *free* occurrences of the variable in question (free relative to the λ body). Since the second λ binds the last two occurrences of "x", performing α-reduction on the larger form won't touch them.

Now, back to the original problem. The way to deal with "((λ x (λ y (x y))) y)", then, is to first take an alphabetic variant. Because alphabetic variants are guaranteed to be logically equivalent, we can substitute variants in place of the original sub expressions. If we do this cleverly, the "y" and the "z"'s don't interact in the pernicious manner shown above.

1. ((λ x (λ y (x y))) y)	α-reduction on "(λ y (x y))", substituting "y1" for "y" ==>
2. ((λ x (λ y1 (x y1))) y)	β-reduction, substituting "y" for "x" ==>
3. (λ y1 (y y1))	

The result contains exactly one free variable, which is correct. (Note that "y" and "y1" count as entirely distinct variables, as different as "y" is from "x".)

Equivalent steps are taken automatically by the λ-reduction program:

```
((lambda x (lambda y (x y))) y)
```

It usually takes some practice to know when it is necessary to use an alphabetic variant. The safest strategy is to automatically apply α-reduction to every binding operator before each application of β-reduction. See if you can work out the right result for this form before clicking on the "Reduce" button; you will need three applications of β-reduction and at least one application of α-reduction:

```
((lambda x (x x))(lambda x (lambda y (x y)
```

The program uses the safe strategy, blindly performing α-reduction whenever a binding operator occurs inside a body. Experienced human λ-reducers, however, typically apply α-reduction only when it is absolutely necessary to avoid variable collision (it was necessary at least once in the example immediately above). For beginners, though, the best rule is: when in doubt, perform α-reduction!

(Warning:) I have modified the characterization of β-reduction and of α-reduction somewhat in the interests of simplicity. The simplified versions are perfectly sound, and provide the full expressive power of the λ-calculus. If you're interested in the traditional elaboration, consult either the Hankin book or Berendregt book mentioned above.

Q.5 What is η-reduction?

Ans η-reduction

η-reduction says that an expression of the form "(λ x (P x))" is guaranteed to be equivalent to "P" alone, where "P" is any expression (in which x does not occur free). This equivalence is obvious enough, but α-reduction and β-reduction alone do not guarantee it. However, few practical applications of λ-reduction bother to implement η-reduction. In particular, the programs given on this page do not:

```
(lambda x (P x))
```

Clicking on "Reduce" results in no change.

Q6 What is lazy and eager evaluation?

Ans **lazy evaluation** is the technique of delaying an evaluation of any expression until a value is actually being used and also avoid repeated evaluations.

The benefits of lazy evaluation include: performance increases due to avoiding unnecessary calculations, avoiding error conditions in the evaluation of compound expressions, the capability of constructing potentially infinite data structures, and the capability of defining control structures as abstractions instead of as primitives. Lazy evaluation can lead to reduction in memory footprint, since values are created when needed.

Languages that use lazy actions can be further subdivided into those that use a call-by-name evaluation strategy and those that use call-by-need. Most realistic lazy languages, such as Haskell, use call-by-need for performance reasons, but theoretical presentations of lazy evaluation often use call-by-name for simplicity.

The opposite of lazy actions is eager evaluation, sometimes known as *strict evaluation*. Eager evaluation is the evaluation behavior used in most programming languages. **eager evaluation** or **greedy evaluation** is the evaluation strategy used by most traditional programming languages. In eager evaluation, an expression is evaluated as soon as it is bound to a variable. The alternative to eager evaluation is lazy evaluation, where expressions are only evaluated when evaluating a dependent expression. Imperative programming languages, where the order of execution is implicitly defined by the source code organization, almost always use eager evaluation.

One advantage of eager evaluation is that it eliminates the need to track and schedule the evaluation of expressions. It also allows the programmer to dictate the order of execution, making it easier to determine when sub-expressions (including functions) within the expression will be evaluated, as these sub-expressions may have side-effects that will affect the evaluation of other expressions.

A disadvantage of eager evaluation is that it forces the evaluation of expressions that may not be necessary at run time, or it may delay the evaluation of expressions that have a more immediate need. It also forces the programmer to organize the source code for optimal order of execution. Conversely, it allows the programmer to order the source code to control the order of execution.

Note that many modern compilers are capable of re-ordering execution to better optimize processor resources and can often eliminate unnecessary expressions from being executed entirely, if it can be determined that the results of the expressions are not visible to the rest of the program. Therefore, the notions of purely eager or purely lazy evaluation may not be applicable in practice.

Q7 What is functional programming?

Ans **Functional programming** is a style of programming which models computations as the evaluation of expressions. This article is meant to describe it briefly; however, the best way to understand functional programming is to learn the basics of one of the functional programming languages. In functional programming, programs are executed by

evaluating *expressions*, in contrast with imperative programming where programs are composed of *statements* which change global *state* when executed. Functional programming typically avoids using mutable state.

Functional programming requires that functions are *first-class*, which means that they are treated like any other values and can be passed as arguments to other functions or be returned as a result of a function. Being first-class also means that it is possible to define and manipulate functions from within other functions. Special attention needs to be given to functions that reference local variables from their scope. If such a function escapes their block after being returned from it, the local variables must be retained in memory, as they might be needed later when the function is called. Often it is difficult to determine statically when those resources can be released, so it is necessary to use automatic memory management.

Q8 What is imperative programming?

Ans We can define such languages according to the characteristics that they display:

- By default, statements (commands) are executed in a step-wise, sequential, manner.
- As a result order of execution is crucial.
- Destructive assignment - the effect of allocating a value to a variable has the effect of destroying any value that the variable might have held previously.
- Control is the responsibility of the programmer - programmers must explicitly concern themselves with issues such as memory allocation and declaration of variables.

Out of the five principal programming language paradigms the imperative is the most popular, why?

- The imperative paradigm is the most established paradigm.
- It is much more in tune with the computer community's way of thinking.
- Imperative programs tend to run much faster than many other types of program.
- Programmers are prepared to sacrifice some of the advanced features and programming convenience generally associated with higher level languages in exchange for speed of execution.

Unit 5

Functional Programming Using Haskell

Q.1 What is Haskell?

Ans Haskell is a general-purpose purely functional programming language, with non-strict semantics and strong static typing. It is named after logicianHaskell Curry. In Haskell, "a function is a first-class citizen" of the programming language. As a functional programming language, the primary control construct is thefunction. Haskell features lazy evaluation, pattern matching, list comprehension, type classes, and type polymorphism. It is a purely functionallanguage, which means that in general, functions in Haskell do not have side effects. There is a distinct construct for representing side effects, orthogonal to the type of functions. A pure function may return a side effect which is subsequently executed, modeling theimpure functions of other languages.

Haskell has a strong, static type system based on Hindley–Milner type inference. Haskell's principal innovation in this area is to add type classes, which were originally conceived as a principled way to add overloading to the language, but have since found many more uses.

The construct which represents side effects is an example of a monad. Monads are a general framework which can model different kinds of computation, including error handling, nondeterminism, parsing, and software transactional memory. Monads are defined as ordinary datatypes, but Haskell provides some syntactic sugar for their use.

The language has an open, published specification, and multiple implementations exist.

The main implementation of Haskell, GHC, is both an interpreter and native-code compiler that runs on most platforms. GHC is noted for its high-performance implementation of concurrency and parallelism, and for having a rich type system incorporating recent innovations such as generalized algebraic data types and Type Families.

Q. 2 What are function applications and types in Haskell?

Ans:

- **Function application**: Haskell syntax

  • In mathematics, if

  F is a function that takes two arguments and a,b,c,d are values, we write:

  f(a,b) + cd

  • In Haskell the same is written as: f a b + c*d–function application has higher priority than other operators, hence the line above is equivalent to (f a b) + c*d

- **Types in Haskell**

  • Haskell is statically typed

  •we write v :: T to state that value v has type T–for example

  False :: Bool

  not :: Bool > Bool

  not False :: Bool

  • not is a function that takes a Bool as argument and returns a Bool

  • Some basic types (mostly self-explanatory):

  Bool

  Char

  String

  Int

  Integer-Integer represents integer numbers with arbitrary precision Float.

Q.3 Define lists, tuples and function types.

Ans:

- **Lists**

•Lists: sequences of elements of the same type, e.g.:

[False, True, False] :: [Bool]

['a', 'b', 'c', 'd'] :: [Char]

["One", "Two", "Three"] :: [String]

- The empty list: []
- Lists of lists, e.g.: [['a', 'b'], ['c', 'd', 'e']] :: [[Char]]
- Operations on lists (i.e., basic functions that take lists as arguments):

–length xs

• number of elements in list xs

–

head xs

- first element of list xs

–tail xs

- all elements of list xs except first

–xs!!n

- n-th element of list xs(starting from 0) take n xs

• list made of first n elements of list xs

–drop n xs

- list obtained by removing first n elements of list xs , xs ++ ys

• list obtained by appending list ys after list xs reverse xs

• list obtained by reversing the elements of list xs

➢ Tuples

• Tuple: finite sequence of elements of possibly different type,

e.g.:

(False, True) :: (Bool, Bool)

(False, 'a', True) :: (Bool, Char, Bool)

("Yes", True, 'a') :: (String, Bool, Char)

• empty tuple: ()

• More examples:

('a',(False,'b')) :: (Char,(Bool,Char))

(['a','b'],[False,True]) :: ([Char],[Bool])

[('a',False), ('b',True)] :: [(Char,Bool)]

- **Function types**

• Function: mapping from values of a certain type to values of another type, e.g.:

not :: Bool > Bool

isDigit :: Char > Bool

• Using tuples and lists no more than one argument is needed:

add :: (Int, Int) > Int

add (x, y) = x + y

zeroto :: Int > [Int]

zeroto n = [0..n]

• Another way of dealing with multiple arguments: functions that return functions, e.g.:

add' :: Int > (Int > Int)

add' x y = x + y

–

Add is a function that takes an Int as argument, and returns a function that, given another Int, returns an Int

• It works also for more than two arguments: mult :: Int > (Int > (Int > Int))

mult x y z = x y z**

• Functions such as add' and mult that take arguments one at a time are called curried.

Q.4 What are Curried functions and partial application?what are polymorphic types?

Ans:

➢ **Curried functions and partial application**

- Curried functions lend themselves to partial application -this occurs when not all arguments are supplied to the function application the result of partially applying arguments to a curried function is

  another function, e.g.: add' 1 :: Int > Int

- we could also define

  inc :: Int > Int

  inc = add' 1

  then, the result of

  inc 10 is 11

• Note that the function arrow

➢ associates to the right, i.e.

Int > Int > Int > Int

means

Int > (Int > (Int > Int))

• Function application, instead, associates to the left, i.e.

mult x y z means

((mult x) y) z

➢ **Polymorphic types**

• length is a function that can be applied to lists of different types of elements: length [1, 3, 5, 7]

length ["Yes", "No"]

length [isDigit , isLower , isUpper]

• In fact, the type of length is polymorphic(and length is a polymorphic function) as it contains a type variable

:

length :: [a] > Int a is the type variable

• type variables must start with a lowercase letter

•Other examples of polymorphic functions:

fst :: (a, b) > a

head :: [a] > a

take :: Int > [a] > [a]

zip :: [a] > [b] > [(a, b)]

id :: a > a

Q.5 What are Type classes, overloaded types, and methods?

Ans

- **Type class** = a collection of types for example, class,Num contains any numeric types (e.g., Int, Float)
- If a is a type variable and Cis a type class, C ais a class constraint -it states that type a must be an instance of type class C
- **Overloaded type**: a type that includes a class constraint, e.g.,

  3 Num a => a-3 is a constant that is defined for any numeric type a
- Also, +is an overloaded function:

  (+) :: Num a => a > a > a–(+)

  is a (curried) function that can be applied to any pairs of values that belong to an instance of type class Num

• Other examples of overloaded functions:

(−) :: Num a => a > a > a

( ) :: Num a => a > a > a

∗negate :: Num a => a > a

abs :: Num a => a > a

signum :: Num a => a > a

• In general, a type class defines methods, i.e., overloaded functions that can be applied to values of instances of the type class

• For example, type class Eq contains types that can be compared for equality and inequality; as such, it defines the following 2 methods:

(==) :: a > a > Bool

(/=) :: a > a > Bool

–

Bool, Char, String, Int, Integer, Float are all instances of Eq; so are list and tuple types, if their element types are instances of Eq

Q6 How is Class declaration done in Haskell?

Ans

- **Class declaration**
  - A new class is declared using the class

    keyword: class Eq a where

    (==), (=) :: a > a > Bool

    x /= y = not (x == y)

–

this declaration contains a default definition for method /=, so an instance of Eq must only define method ==

• Example o instance of class Eq: instance Eq Bool where

False == False = True

True == True = True_==_ = False

- Classes can be extended to form new classes:

```
class Eq a => Ord a where

(<), (<=), (>), (>=) :: a > a > Bool

min, max :: a > a > a

min x y | x <= y = x  | otherwise =y
```

max x y | x y = y ≤ | otherwise = x

• To define an instance of Ord, we need to provide the definition of methods <, <=, >and >= :instance Ord Bool where False < True = True _ < _ = False

```
b <= c   = (b < c) || (b == c)

b > c    = c < b

b >= c   = c <= b
```

- Other basic classes

• Ord– ordered types–it contains instances of class Eq, whose values in addition are

totally (linearly) ordered–it provides the following six methods:

```
(<)  :: a > a > Bool

(<=) :: a > a > Bool

(>)  :: a > a > Bool

(>=) :: a > a > Bool

min  :: a > a > a

max  :: a > a > a
```

- Bool, Char, String, Int, Integer, Float are instances of Ord; so are list and tuple types, if their elements...
- Show– showable types–types whose values can be converted into strings of characters

  using the following method

  show :: a > String

- Bool, Char, String, Int, Integer, Float are instances of Show; so are list and tuple types, if their elements...
- Read– readable types–types whose values can be obtained from strings of characters

  using the following methodread :: String > a
- Bool, Char, String, sssInt, Integer, Float are instances of Read; so are list and tuple types, if their elements...
  - foldr
- A pattern often used to define a function f that applies an operator ⊕ to the values of a list (with v some value): f [] = v f (x:xs) = x ⊕ f xs
- For example: product [] = 1 product (x : xs) = x product xs∗ and [] = True and (x:xs) = x && and xs
- The (higher-order) foldr function captures this pattern:

  foldr :: (a > b > b) > b > [a] > b foldr f v [] = v
  foldr f v (x : xs) = f x (foldr f v xs)
- Defining functions product and and in terms of foldr:

  product = foldr (*) 1

  and = foldr (&&) True
- For example, if we apply foldr (*) 1 to list 1:(2:(3:(4:[]))) we obtain 1*(2*(3*(4*1)))
- another example of using foldr to define a function: length = foldr (_ v -> 1+v) 0
- In general, the behavior of foldr is: foldr ( ) v [x0,x1,···,xn] = x0 (x1 (···(xn v)···))

  ⊕⊕⊕⊕ essentially, foldr corresponds to the application to the elements of a list of an operator that associates to the right(hence, foldr)26 foldl
- (higher-order) function foldl is the dual of foldr: it corresponds to the application of an operator that associates to the left
- It corresponds to the pattern:

f v [] = vf v (x:xs) =f (v ⊕ x) xs argument v works as an accumulator, which evolves by applying operator ⊕ with the value of the head of the list

- This is captured by the following definition: foldl :: (b > a > b) > b > [a] > b

  foldl f v [] = v foldl f v (x:xs) = foldl f (f v x) xs

- In general, its behavior is:

  foldl ( ) v [x0,x1,···,xn] = (···((v x0) x1)···) xn ⊕⊕⊕⊕

- Examples of definitions using

  foldl: product = foldl (*) 1 and = foldl (&&) True product and can be defined either with foldr

  or with foldl since they are both associative

- Another example:

  reverse = foldl (\xs x > x:xs) []

for example the result of reverse [1, 2, 3] is 3 : (2 : (1 : [ ]))

➢ Composition of functions

- The (higher-order) composition operator . takes two functions f and g as arguments, and applies f to the result obtained by applying g to its argument the type of the argument of g must be the same as the type of the result of f

- In other words:

  (.) :: (b > c) > (a > b) > a > c

  f . g = \x > f (g x)

- The composition operator can be used to make some definitions more concise: instead of

  odd n = not (even n)

  twice f x = f (f x)

  sumsqreven xs = sum (map (^2) (filter even xs))

we can define:

odd = not . even

twice = f.f

sumsqreven = sum.map (^2).filter even the last definition works because composition is associative:

f . (g . h) = (f . g) . h

- the identity function

  id = \x > x is the unit for ., i.e., for any function f we have f.id = id.f

- We exploit id and foldr to define a composition of lists of functions:

  compose :: [a > a] > a > a

  compose = foldr (.) id

➢ Type declarations

- The simplest way to declare a type is as a synonym of an existing type:

  type

  String = [Char]

  type

  Pos = (Int, Int)

  type

  Board = [Pos]

- Type parameters are admitted:

  type

  Assoc k v = [(k,v)]

  Assoc represents, through a list, a lookup table of <key,value> pairs, where the keys are of type k, and the values of type v,a function that, given a key and a lookup table, returns the value associated with the key:

```
find :: Eq k => k > Assoc k v > v

find k t = head[v | (k', v) < t, k == k']
```

- In Haskell one can also declare entirely new types, through a data declaration, e.g.: data

```
Bool = False | True
```

- New types can be used in functions:

```
data

Move = Left | Right | Up | Down

move :: Move > Pos > Pos

move Left (x,y)  = (x1, y)

move Right (x,y) = (x+1, y)

move Up (x,y)    = (x, y+1)

move Down (x,y)  = (x, y1)

moves :: [Move] > Pos > Pos

moves [] p       = p

moves (m : ms) p = moves ms (move m p)
```

➢ Type declarations with parameters

- Constructors in data declarations can have parameters: data

```
Shape = Circle Float | Rect Float Float
```

- Examples of functions on type

```
Shape:

square :: Float > Shape

square n = Rect n n

area :: Shape > Float

area (Circle r) = pi   r ^ 2
```

*

area (Rect x y) = x y

*

notice that we can use pattern matching with the constructors

- Circle and Rect are constructor functions: their results are values of type Shape Circle 1.0 is a value onto itself, of type Shape, it is not evaluated any further
- data declarations can also have (type) parameters, e.g.:
data Maybe a = Nothing | Just a type Maybe represents optional values (i.e., values that may fail): if the value is undefined, then its value is Nothing, otherwise it is Just v(with v a value of type a)example of use of type Maybe: "safe" functions that, in case of errors, simply return a Nothing

value:

safediv :: Int > Int > Maybe Int

safediv _ 0 = Nothing

safediv m n = Just (m `div` n)

safehead :: [a] > Maybe a

safehead [] = Nothing

safehead xs = Just (head xs)

30. Recursive types

- Types defined through data declarations can be recursive: data

Tree a = Leaf a | Node (Tree a) a (Tree a)

functions on type Tree:

occurs :: Eq a => a > Tree a > Bool

occurs v (Leaf x) = v == x

occurs v (Node t1 x t2) = v == x | |

occurs v t1 | |

occurs v t2

flatten :: Tree a > [a]

flatten (Leaf v) = [v]

flatten (Node t1 v t2) = flatten t1 ++ [v]

++ flatten t2

- Extended example: tautology checker

- (subset of) Propositional logic: constants: False, True

  propositional variables: A, B, C, ... Z

  connectives: ¬,(not), ∧,(and), ⇒ (parentheses: (,)

- examples of formulas:

  A ∧¬A

  (A ∧B) ⇒A

  A ⇒(A ∧B)

- tautology: a formula that is true, no matter the values of the propositional variable

  for example, (A ∧B) ⇒A is a tautology

- We declare a type for propositions: data

  Prop = Const Bool

  | Var Char

  | Not Prop

  | And Prop Prop

  | Imply Prop Prop

- The value of a proposition depends on the values assigned to its variables; we use a type

  Subst to represent possible assignments of values to variables, through a lookup table:

type Subst = Assoc Char Bool example of assignment: [('A', False),('B',True)]

➢ Tautology checker (cont.)

- We define a function that, given a proposition and an assignment of values to variables, returns the value of the proposition in that assignment:

```
eval :: Subst > Prop > Bool

eval _ (Const b) = b

eval s (Var v) = find v s

eval s (Not p) = not (eval s p)

eval s (And p1 p2) = eval s p1 && eval s p2

eval s (Imply p1 p2) = eval s p1 <= eval s p2
```

- We define a function that returns all variables in a proposition:

```
vars :: Prop > [Char]

vars (Const b) = []

vars (Var v) = [v]

vars (Not p) = vars p

vars (And p1 p2) = vars p1 ++ vars p2

vars (Imply p1 p2) = vars p1 ++ vars p2
```

- A function that removes duplicates from a list:

```
rmdups :: Eq a => [a] > [a]

rmdups [] = []

rmdups (x:xs) = x : rmdups (filter (/= x) xs)
```

➢ Tautology checker (end)

- A function that, given a number n, returns all possible combinations of Booleans of length n: bools :: Int > [[Bool]]

```
bools 0   = []

bools n+1 = (map (False:) bss)++(map (True:) bss)
```

where bss = bools n e.g., the result of bools 2 is
[[False,False],[False,True],[True,False],[True,True]]

- A function that generates all possible assignments to the variables of a proposition: substs :: Prop > [Subst]

  substs p = map (zip vs) (bools (length vs))

  where vs = rmdups (vars p)

- finally, the desired function: isTaut :: Prop > Bool
  isTaut p = and [eval s p | s < (substs p)]

- In fact the result of isTaut (Imply (And (Var 'A') (Var 'B')) (Var 'A')) is True.

Q7 Implement quicksort in Haskell.

Ans Quicksort in Haskell

• qsort [] = []

qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger

where

smaller = [a | a < xs, a <= x]

larger = [b | b < xs, b > x]

- The two equations define that quicksort is a function that is applied to sequences of values and that:

  –if qsort is applied to an empty sequence, the sequence is already sorted otherwise, let us call x the first element of the sequence, and xs the rest of the sequence; then, if smaller is the subsequence of xs that contains all elements no bigger than x, and larger is the subsequence of xs that contains all elements bigger than x, the sorted sequence is given by concatenating smaller, x and larger

- Example of execution:

  qsort [3, 5, 1, 4, 2]= { applying qsort}

  qsort [1, 2] ++ [3] ++ qsort [5, 4]= { applying qsort}

(qsort [] ++ [1] ++ qsort [2]) ++ [3] ++ (qsort [4] ++ [5] ++ qsort [])

= { applying qsort, since qsort [x] = [x]}

([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [])

= { applying ++ }

[1, 2] ++ [3] ++ [4, 5]

= { applying ++ }

[1, 2, 3, 4, 5]