PANJAB UNIVERSITY

Provisional Admit Card for Under Graduate Examinations (Semester) - February,2021

Examination	**B.E. (C.S.E.)**
Semester	5
Session	February, 2021
Application No.	2061288241
Roll No.	**CO18311**
Candidate Name	**Anshul Gupta**
Father's Name	Devi Chand Gupta
Mother's Name	Sudha Rani
Regd. No.	37018000023
Subject	CS 501 CS 551 CS 502 CS 552 CS 503 CS 553 CS 504 CS 505 CS 556

Answer Sheet for On Line Examination

Final Examination

Panjab University

Feb./March, 2021

i) University Roll No. (in figures) .....CO18311.....

(in words) ..CO.EIGHTEEN..THOUSAND..THREE..HUNDRED.AND.ELEVEN....

ii) Name of the Student : ..ANSHUL..GUPTA..... iii) Class: ..B.E..C.S.E.....

iv) Semester :.....5..... v) Subject: CS504: PRINCIPLES OF PROGRAMMING LANGUAGES

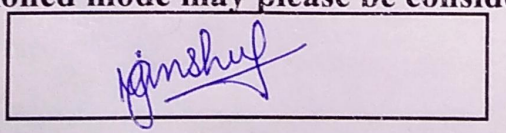
vi) Subject Code : .....6789..... vii) Paper Code :....0917....

viii)Total No. of Pages Written :.....22..... ix)Date of Exam: 10./03/2021

x) Undertaking (Only for the students of College) :

I am submitting my Answer Sheet through .....ONLINE..... (Online / ~~Hard Copy~~) mode and will not submit the same through other mode. Answer sheet submitted only through above mentioned mode may please be considered for evaluation.

xi)Signature →



Answer Sheet for On Line Examination
Final Examination Feb./March, 2021

i) University Roll No. (in figures) .....C018311.....

(in words) C.O.EIGHTEEN..THOUSAND...THREE..HUNDRED...AND...ELEVEN.....

ii) Name of the Student : ANSHUL....GUPTA..... iii) Class: ..B.E...C.S.E.....

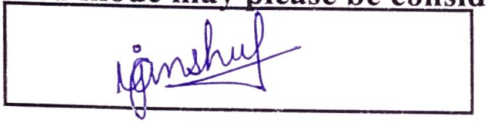
iv) Semester :.....5..... v) Subject: CS504: PRINCIPLES OF PROGRAMMING LANGUAGES

vi) Subject Code : .....6789..... vii) Paper Code :....0917....

viii)Total No. of Pages Written :.....22..... ix)Date of Exam: 10./03./2021

x) Undertaking (Only for the students of College) :

I am submitting my answer sheet through .....ONLINE..... (Online/~~Hard Copy~~) mode and will not submit the same through other mode. Answer sheet submitted only through above mentioned mode may please be considered for evaluation.

xi)Signature → 

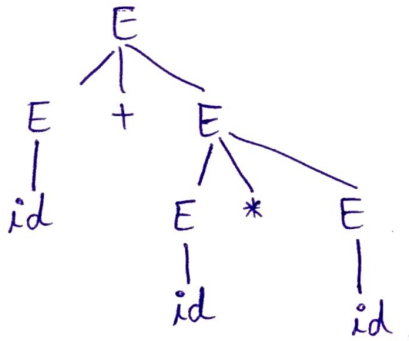
Ans1.)a) According to definition of ambiguous grammar:
A content free grammar is said to be ambiguous if there exists more than one derivation tree for the given input string i.e. more than one left most derivation tree or rightmost derivation tree.

∴ There can be more than one distinct parse tree for a grammar to be ambiguous.

e.g. $E \rightarrow E+E \mid E*E \mid id$

for $w = id+id*id$

Derivation is, $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$
& its derivation tree is given by.

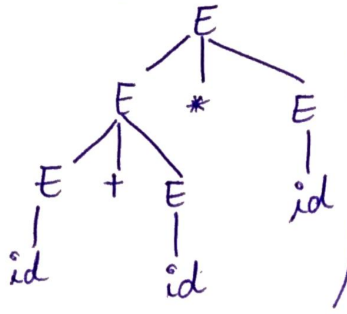


Parse Tree 1 from LMD

Derivation 2:

$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$

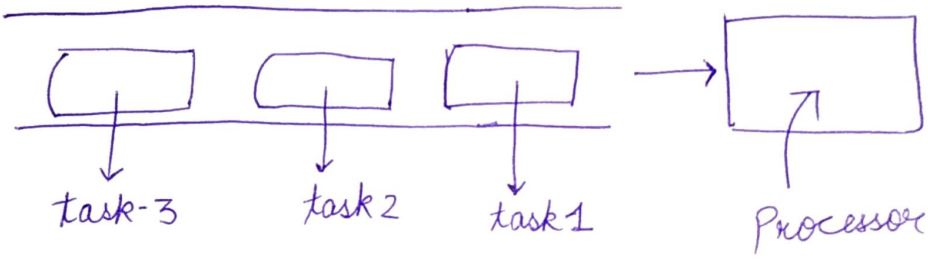
whose derivation tree is given by:



Parse Tree 2 for another LMD.

∴ as in our grammar, we got more than one parse tree for the string id + id * id which implies the grammar is ambiguous.

b.) Concurrent programming is a technique in which two or more processes start, run in an interleaving fashion through content switching and complete in an overlapping time period by managing their access to shared resources. Eg: In a core of CPU, we have more than one task in an interleaved manner.



Concurrent programming is needed to:

(i) Reduce user response time: Concurrent programming ultimately reduces the user response time as the process run in an interleaved manner.

(ii) Efficient Resource Utilization: Computation devices have resources like CPU cycle, access time, IO devices and we can use them effiently by concurrent programming.

(iii) Economical: Economically, concurrent programming is more feasible as on network system we have to handle many users at the same time.

(iv) Availability, Speed and distribution → if one processor is busy other may help → to implement processes → processors in different locations can collaborate to solve a problem.

C.) **Independent Compilation** : Independent compilation is the compilation of some of the units of a program separately or independently from the rest of the program, without the benefits of interface information.

Example of independent compilation in Java is as follows:

file: Add.java

```
class Add {
    int add(int x, int y){
        return x+y;
    }
}
```

file: Sub.java

```
class Sub {
    int sub(int x, int y){
        return x-y;
    }
}
```

file: Sample.java

```
class Sample {
    public static void main (String[] args ){
        int x = 5; y = 5;
        int sum, subt;
        Add a = new Add a();
        Sub b = new Sub b();
        sum = a.add(x, y);
        subt = b.sub(x, y);
        System.out.println ("Add: " + sum);
        System.out.println ("Sub: " + subt);
    }
}
```

→ C, C++, Java or python follow independent compilation.

In the above example, if we complile the above program in sample.java the files are compiled in any order independent of other files, and as and when required but separately.

```
javac. Sample.java
javac. Add.java
javac. Sub.java
```

d.) Following are the reasons which makes type conversion more flexible to the user:

(i) Explicit type not needed: If there is type conversion in compilers, then programmers do not need to explicitly specify any data type to variable.

(ii) It provides compatibility between heterogenous data types.

↳ Narrowing conversion: converts an object to a type that that cannot include all of the values of the original type. e.g. float to int

↳ Widening conversion: that converts an object to a type in which an object is converted to a type that can include at least approximations to all of the values of original type e.g. int to float.

(iii) User can use it as per its need and thus providing more flexibility.

e) Functional programs is simply an expression, and thus in functional programming, executing a program is equivalent to Evaluating the expression. It is relatable to the imperative view as:

$$\nabla_n = E[\nabla_n]$$

The reason why pure functional programming does not have any assignment operation is that there is no variable in functional programming, thus no state is there in the program and since there are no variables, we have nothing to assign the values for. The major reason being having nothing to assign.

Therefore, there is no assignment operation in pure functional programming.

Ans III.

Overloading refers to the ability to use a single identifier to define multiple methods of a class that differ in their input and output parameters. Overloaded methods are generally used when they conceptually execute the same task but with a slightly different set of parameters.

↳ It is a concept used to avoid redundant code where the same method name is used to avoid redundant code where the same method name is used multiple times but with a different set of parameters. The actual method that gets called during runtime is resolved at compile time, thus avoiding runtime errors.

↳ Overloading provides code clarity, eliminate complexity and enhances runtime performance.

Example: An identifier F denotes both a function f1 of type S1 → T1 and a function f2 of the type S2 → T2.

This overloading can be categorised into the following two types:

Context - Independent Overloading

↳ It requires that S1 and S2 are non equivalent in the case of above example.

↳ Overloading is based on the parameter types.

↳ Consider the function call F(E), if the actual parameter E is of type S1, then

Content Dependent Overloading.

↳ It requires only that S1 and S2 are non-equivalent or that T1 and T2 are non-equivalent.

↳ Overloading is based on the parameter types as well as the return type.

↳ Consider the function call F(E) where E is of the type S1 (equivalent to S2). If the

F here denotes f1 and the result is of type T1.
If E is of the type S2, then F here denotes f2 and result is of type T2.

function call occurs in a context where an expression of type T1 is expected, then F must denote f1; if the function call occurs in a context where an expression of type T2 is expected, then F must denote f2.

In context dependent overloading, if S1 and S2 are non-equivalent, the function to be called can be identified as above. If S1 and S2 are equivalent but T1 and T2 are non-equivalent, context must be taken into account to identify the function to be called.

Function Overloading VS Operator Overloading

↳ We can have multiple definitions for the same function name in the same scope.

↳ The definition of the function must differ from each other by the types and/or the number of arguments in the argument list.

↳ We cannot overload function declarations that differ only by return type.

↳ Example:

```
int fun (int x, int y) {
    return (x+y); }
double fun (double x, double y){
    return (x+y); }
```

Both the functions perform the same operation but on different data types.

↳ This means redefining or overloading of most of the built-in operators available in C++ (refers to as using same operator for different purposes).

↳ We can use an operator as a function to do any task other than what it is supposed to do.

↳ Syntax:

```
ret-type class_name::operator X (argument)
{
    // perform operations
}
```

ret-type → return type
class_name → class in which operator is overloaded.
X → the operator to be overloaded (e.g. +, -, *)
In brackets → the arguments to be passed.

Method Over-riding:

↳ For method overriding,

(i) The argument list should be exactly the same as that of the overridden method.

(ii) The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.

(iii) Constructors and methods declared static or final cannot be overriden.

(iv) If a method cannot be inherited, then it cannot be overriden.

Method Overriding in Java:

```
class Human {  //overridden method
    public void eat(){
        System.out.println("Human is eating");
    }
}

class Boy extends Human {  //overriding method
    public void eat(){
        System.out.println("Boy is eating");
    }
    public static void main(String args[]){
        Boy obj = new Boy();
        obj.eat();
    }
}
```

Output : Boy is eating.

Overriding in C++ :

```
class Base {
    public:
        void display (){
        cout << "Function of Base Class";
    }
};
class derived: public Base {
    public :
        void display () {
        cout << "Child Class";
    }
};
int main () {
    derived derives;
    derives . display ();
    return 0;
}
```

Output : Child Class.

Ans IV

a.) **Backtracking :**

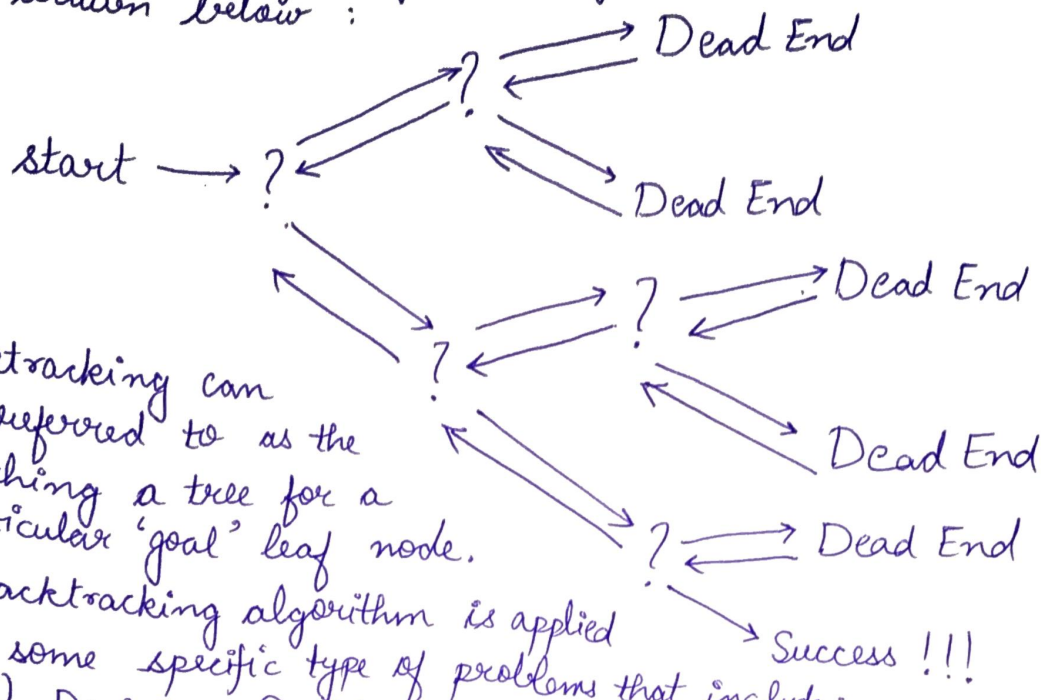
↳ It is a general algorithm for finding all (or some) solutions to some computational problems, that ~~solutions~~ ~~to~~ ~~a~~ incrementably builds candidates to the solution and abandon each partial candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

→ The idea is that we can build a solution step by step ~~by~~ using recursion; if during the process we realize that is not going to be a valid solution, then we stop computing that solution and we return back to ~~the~~

step before i.e. backtracks.

→ Basically, it is a systematic way of trying out different sequences of decisions until we find one that 'works'.

→ The working may be easily understood by the tree drawn below :



Backtracking can be referred to as the searching a tree for a particular 'goal' leaf node.

↳ Backtracking algorithm is applied to some specific type of problems that include :

(i) Decision Problem : In this, we search for a feasible solution of problem.

(ii) Optimization Problem : In this, we search for the best solution that can be applied.

(iii) Enumeration Problem : In this, we find all feasible solutions (or the set of all feasible solutions) of the problem.

Cut :

↳ The cut symbol (written as !) is a special sub-goal used to provide a limited form of control over backtracking.

↳ Usage of backtracking algorithm is beneficial but uncontrolled backtracking can be a nuisance.

↳ Prolog (Logic Programming Language) uses cuts to provide control over backtracking, by committing the prolog interpretor to a particular portion of the

↳ potential search space.

↳ The cut, in prolog, is a goal, written as !, which always succeeds, but cannot be backtracked.

It is the best used to prevent unwanted backtracking, including the finding of extra solutions by prolog and to avoid unnecessary computations.

↳ A cut is like a one-way door that lets you out but doesn't let you back in.

Advantages or the needs of Cuts is as follows:

(i) The program will run faster. No time wasting on attempt to resatisfy certain goals.

(ii) The program will occupy less memory as lesser back-tracking points to be remembered.

Ans IV

b.) Synchronization primitives: These are simple software mechanisms provided by a platform (e.g operating system) to its users for the purpose of supporting thread or process synchronizations. They are usually built using lower level mechanisms (e.g. atomic operations, memory barriers, spinlocks, etc.). Muten, conditional variables and semaphores are all synchronization primitives, so are shared as exclusive locks.

(i) Semaphores → These are lower level synchronization mechanisms that are mainly used when interprocess communication occurs via shared variables. It is a data object that can assure an integer value and can be operated on by the primitives P and V. The definitions of P and V are:

P(s) → if S > 0, then S = S − 1 else suspend current process.

V(s) → if there is a process suspended on the semaphore, then wake up process else S = S + 1.

The primitives P and V are assumed to be atomic operation the semaphore has an associated data structure where the descriptors of process suspended on the semaphore is recorded. Data structure is usually a queue.

Example of Semaphores:

Implementing mutual exclusion.

a) Semaphore has an initial value of 1.
b) P() is called before a critical section.
c) V() is called after critical section.

```
Semaphore Lock 1;
P(Lock);
// critical section
V(Lock);
```

Lock = 1

Lock = 1 → 0

Lock = 0

Lock = 0 → 1

(ii) Muter: It is a mutual exclusion object that synchronizes access to a resource. It is created with a unique name at the start of a program. The muter is a locking mechanism that makes sure only one thread can acquire the muter at a time and enter the critical section. This thread only releases the muter when it exists with the critical section.

(iii) Event: It is a primitive that any number of processes can open independently. After opening an event, a process may wait or signal on the event. If a process calls wait, it will become blocked

until the event is signalled by some other process. If any process calls signal, all waiting process should be immediately unblocked. Calling signal with zero processes waiting does nothing. When a process is done using an event, it should close it, and if all processes close the event, the event gets destroyed.

(iv) Lock: A lock is an object that can be held by atmost one thread at a time. Only the thread that last acquired a lock is allowed to release that lock. Locks are useful for guarding critical sections. Multiple methods can be a part of the same critical sections.

(v) Conditional Variable: A conditional variable is an object used in combination with its associated lock to allow a thread to wait for some condition while its inside a critical section. It is a place to wait, sometimes called a rendezvous point. Operations on conditional variables include:

→ wait (c) : release monitor lock so that if somebody else can get in wait for somebody else to signal condition. Thus it have wait queue.

→ signal (c) : wake up at most one waiting process / thread. if no waiting processes, signal is lost this is different than semaphores

→ broadcast (c) : wake up all waiting processes / threads

(vi) Monitors and Signals: A monitor is a programming language construct that supports controlled access to the shared data. It addresses the very usability issues synchronization of different concurrent processes. It helps in enforcing semaphores. Only one monitor procedure active at any given time:

```
monitor example
    integer i;
    condition c;
    procedure p1( );
        .
        .
    end
    procedure p2( );
        .
        .
    end
end monitor
```

A monitor encapsulates :

- Shared data structure
- procedures that operate on the shared data.
- Synchronization between concurrent threads that invoke those procedure.

Ans VI

a.) Polymorphic Data Types

Generic pointers that store only a byte address and no information about the type of data stored at that memory addresses can be used to implement polymorphic data types.

e.g. function (void * a , void *b) , Here a and b are generic pointers which can hold int, float (or any other) value as an argument. In C++ the address of a variable of one data type cannot be assigned to a pointer of another data type.

e.g.
```
int * ptr;
double d = 13.0
ptr = &d.
```

The error occured because the address is a double type variable. However, the pointer is int type.

Here, we can use the void pointer in C++ . e.g.

```
Void *ptr
double d = 9.0
ptr = &d
```

The void pointer is a generic pointer that is used when we dont know the data type of variable that the pointer points to.

↳ Function templates are another example of polymorphic data types. Functional templates are special functions which can operate with generic types. This allows to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

A template has the following declaration declaration:

```
template <typename T>
T function name (T parameter1, T parameter2, ---){
    //code
}
```

Here T is a template argument which accepts different data types (int, float etc.) and typename is a keyword. When an argument of a data type is passed to function Name() the compiler generates a new version of function Name() for the given data type.

Example:

```
# include <iostream>
using namespace std;
template <typename T>
T add (T num 1, T num 2){
```

```
return (num1+ num2);
}

int main ( ){
    int result1;
    double result2;
    Cout << "6 + 7 = " << result1 << endl;
    result2 = add <float> (6.7, 7.6);
    Cout << "6.7 + 7.6 = " << result2 <<
        endl;

    return 0;
}
```

OUTPUT: — 6+7 = 13

6.7 + 7.6 = 14.3

Ans VI b) In simpler words,

↳ Type inference is the ability of the compiler to find out the type of an expression depending upon the operation being performed.

↳ It involves analyzing a program and then inferring different types of some or all expressions in that the program so that the programmer

does not need to explicitly input and define data types everytime variables are used.
Type inference is a compiler's ability to look at each method invocation and corresponding declaration to determine the type of argument / arguments that make the invocation applicable.

— The ability to infer types automatically makes many programming tasks easier, leaving the programmer free to omit type annotations while still permitting type checking.

— Type inference is a feature of functional programming languages. The compiler or interpreter needs only minimal information in order to figure out what the data type of the variable or expression is.

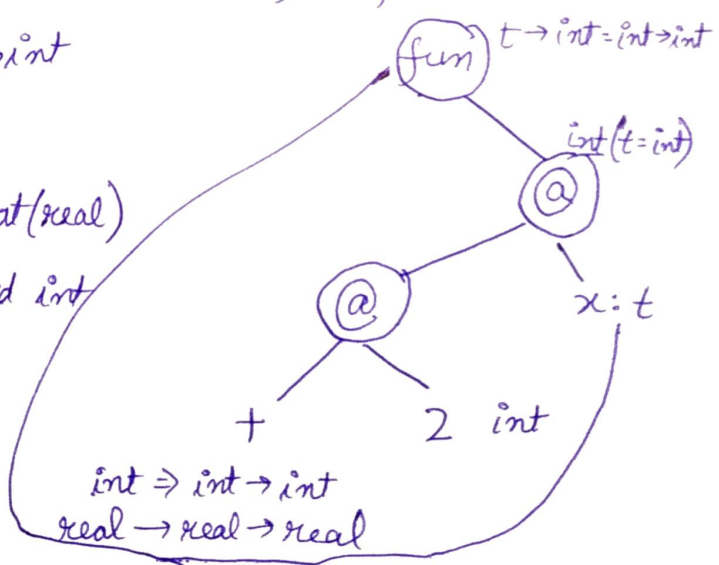
Example: (showing working of type inference)

→ It follows the main steps of as mentioned below.
(i) Assign types to leaves
(ii) Propagate to internal nodes & generate constraints.
(iii) Solve by substitution.

e.g. - fun f(x) = 2 + x;
> val it = fn: int → int

Here, 2 has type int,
'+' has type int, float (real)
∴ the resultant is used int

Graph representation for f(x) = 2 + x



A Type Checking

↳ The contextual installation process is performed by unification & is the basis of polymorphic typechecking.

↳ An expression of type can legally appear in all contexts where an instance of that type can appear. The typechecking process consist of matching type operators and instantiating type variables.

↳ There are two types of type checking:

(i) Static Type Checking: It means that the correctness of using type is performed at compile time.

(ii) Dynamic Type Checking: It means that the correctness of using types is performed at run time.

In principle, type checking can always be performed at run time if the info about types of values in program is accessible in executing code. Dynamic checking takes more time and lowers the reliability of compiled code.

↳ A functional programming language is called strongly typed, if the type of each expression can be determined at compile time, guaranteeing that the type related errors cannot occur in object program.

↳ Example: Pascal is strongly typed language. However user for Pascal some checks can be performed only dynamically. Consider the following definitions:

table: array [0 ... 255] of char;
i: integer.

Also in Example:

x = 1 + "hey" would result in a type error as + doesn't allow addition of integers and strings. This is done by type checking mechanism.

Ans VII

a.) Procedural Abstraction

↳ C++ supports function procedures, constructors and methods. Methods differ from ordinary functions only in that they are attached to objects. The result type of a function or method may be void or any other type.

↳ C++ supports both copy in and reference parameters. Unlike C, C++ does not force us to pass an explicit pointer to achieve the effect of a reference parameter.

↳ C++ supports content-independent overloading of functions, constructors and methods. In other words, two or more functions may share the same identifier in the same scope provided only that they differ in their parameter types or numbers.

↳ C++ treats its operators exactly like functions, we may overload any existing operator by providing an additional definition. But we may

invent new operator symbols.

Example: C++ Overloaded function:

1. void put(ostream str, int i);
2. void put (ostream str, double u);

] C++ overloaded functions.

3. int put (ostream str, double x); // illegal.

↳ illegal because it differs from the second function only in its result type.

Generic Abstraction

↳ C++ supports generic classes. A generic class can be parameterized with respect to values, variables, types and functions on which it depends.

↳ C++ also supports generic functions. A generic functions give us an extra level of abstraction.

Example: C++ generic function

```
template <class Item>
void swap(Item &x, Item &y){
    Item z = x;
    x = y; y = z
}
```

We can call this function without first instantiating it:

```
int a[];
:
swap(a[i], a[j]);
```

The above function defines a swapping function that is parameterized with respect to the type Item of the values being swapped.

Data Abstraction

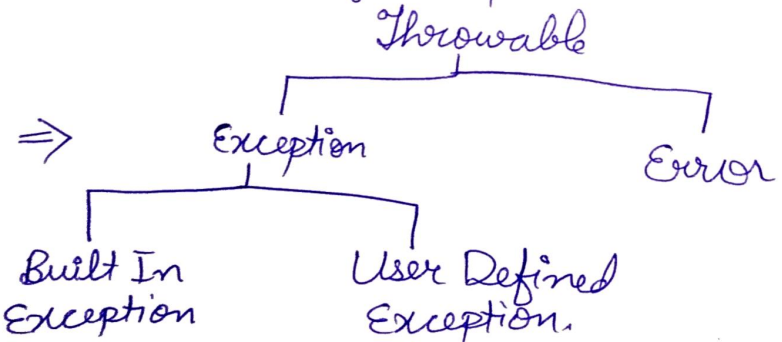
↳ C++ support data abstraction by means of classes.
↳ In C++ terminology, a subclass is called a "derived class", a superclass is called as "base class", and a method is simply called a function.
↳ Class declaration also distinguishes between public, private and protected components which determine the level of abstraction.
↳ The class declaration also distinguishes between two kinds of variable components i.e. instance variable & class variable (distinguished by specifier static).
↳ Similarly, there are instance methods & class methods

Example: C++ class declaration:

```
class Example {
    public:   // access specifier (public, protected or private based on level of data abstraction)
    int num;
    char str[20];   } attributes
};
```

Ans VII b) Exceptions are the unwanted errors or bugs or events that restrict the normal execution of a program. Each time exception occurs, program execution gets disrupted.

Different classes of Exceptions in Java:



The parent class of all exception in Java is "java.lang.exception"

2 types of Exceptions are:

- Checked Exception (Built-In Exception)
- Unchecked Exception (User Defined Exception.

A checked exception is a compile time exception, checked during the compilation process, whereas an unchecked exceptions are defined by user to describe a certain situation.

Examples of Checked exceptions include:

- Arithmetic exception
- Array index out of bounds exception.
- Class not found Exception
- IO Exception
- Null Pointer Exception
- Interrupted Exception
- File Not found Exception, etc.

While an unchecked Exception example is:

```
class MyException extends Exception
    MyException () { }
  MyException (String str) {
    super (str);
  }
```

Ans VII c.) Features of Prolog that helps in classifying it as a logic programming are:

(i) The facts and the rules of the program are used to determine which substitutions for variable in the query called unification,

example:

Unification ↓ $a(x, B) = c(y, b)$

Returns $x = y$, & $B = b$

(ii) The rules and queries used in prolog is similar to that required in Logic Programming Language (semantics).

(iii) Backtracking: When a task fails, prolog traces backwards & tries to satisfy previous state as the case of Logic programming language.

(iv) Recursion: Prolog Uses recursion to give results for various queries just similar to a logic programming language.