1)

Assert is a Java keyword used to define an assert statement.

An assert statement is used to declare an expected boolean condition in a program.

If the program is running with assertions enabled, then the condition is checked at runtime.

If the condition is false, the Java runtime system throws an AssertionError.

Syntax

```
assert expression;

assert expression1 : expression2;
```

Example

```
import java.util.Scanner;

public class AssertionExample{

 public static void main(String[] args){

  Scanner scanner = new Scanner( System.in );
  int x,y;

  x = scanner.nextInt();
  y = scanner.nextInt();

  assert y==0:" Not possible to divide by zero";

  int ans = x / y;

  System.out.println("The answer is "+ans);
 }
}
```

Execution Command

By default assertions are not enabled in Java and hence we need to invoke them manually.

To enable assertion -ea switch of Java has to be used.

Execution Command - java -ea AssertionExample

Disadvantages

There are some situations where assertion should be avoid to use. They are:

1. According to Sun Specification, assertion should not be used to check arguments in the public methods because it should result in appropriate runtime exception e.g.IllegalArgumentException, NullPointerException etc.
2. Do not use assertion, if you don't want any error in any situation.

---

2)

Generics in Java is similar to templates in C++.

The idea is to allow type (Integer, String, … etc and user defined types) to be a parameter to methods, classes and interfaces. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well.

Advantanges

1. **Type-safety :** We can hold only a single type of **objects** in generics. It doesn't allow to store other objects.

2. **Type casting is not required:** There is no need to typecast the object.

3. **Compile time checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

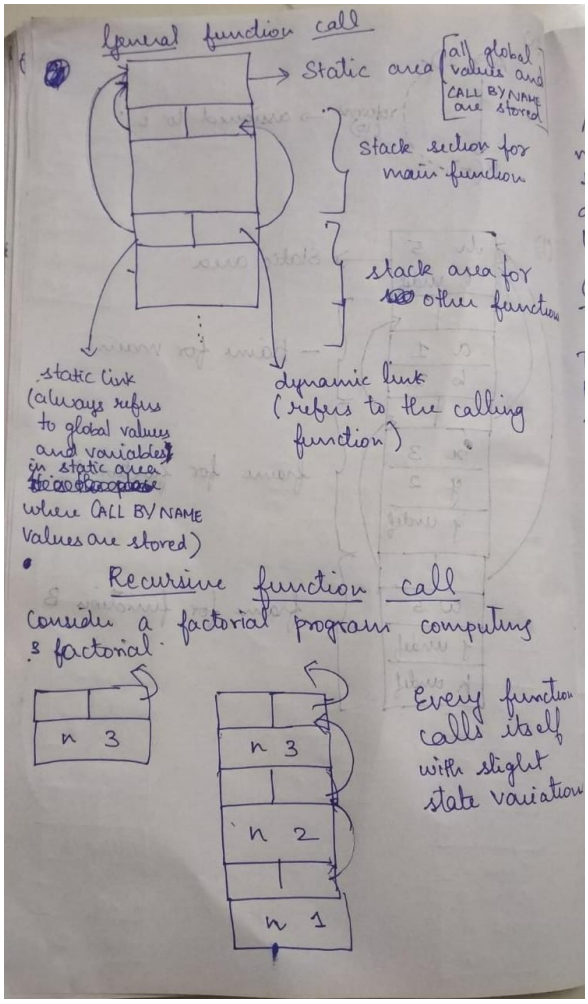
Example

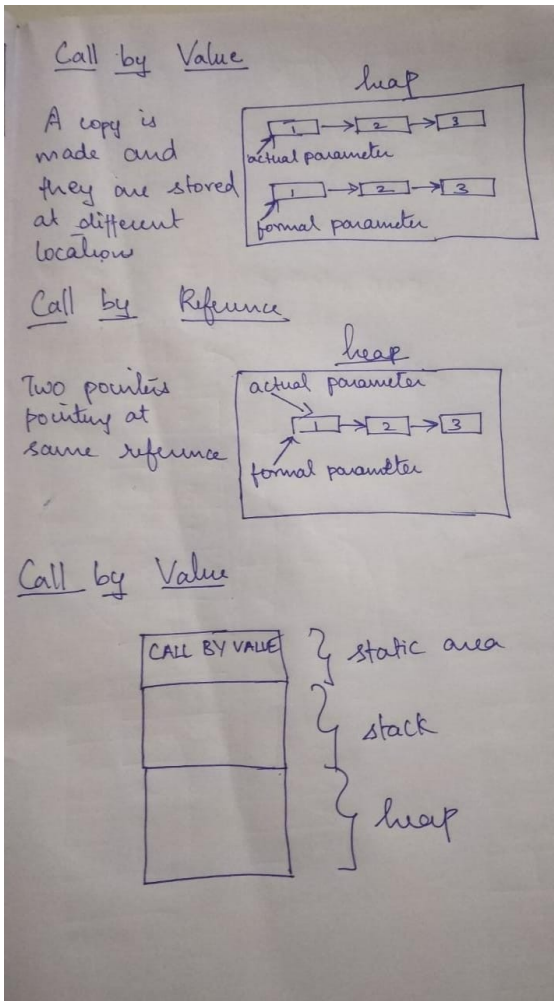
```
import java.io.*;

import java.lang.*;

import java.util.*;

class TemplatePrototype <T>{

    public T val;


    TemplatePrototype(T obj){

        this.val = obj;

    }
```

```
    T printData(){
        return this.val;
    }
}
public class Template{
    public static void main(String[] args){
        TemplateePrototype<String> tt = new
TemplatePrototype<String>("string template");
        System.out.println("The data is " + tt.printData());
        TemplatePrototype<Integer> tt1 = new
TemplatePrototype<Integer>(15);
        System.out.println("The data is " + tt1.printData());
    }
}
```

---

3)





4)

C/C++

```
#include<iostream>
using namespace std;
void callFunc( void(*fp)(string) ){
     (*fp)(“Welcome”);
     (*fp)(“Bye”);
}
void dispData(string n){
     cout<<n<<endl;
}
int main(){
     callFunc(dispData);
     return 0;
}
```

Python

```
def square_root(func,val):
     root_val = val ** (1/2)
     return func(root_val)
def si(val):
     return abs(val)
print(square_root(si,56))
```

JavaScript

```
function alertUser( val){
     alert(val)
}
function alertMsg(func){
     func(“Welcome to Chennai”)
}
alertMsg(alertUser)
```

Other languages like C#,Java,etc also support this feature.

5)

Memory management is the process of allocating new objects and removing unused objects to make space for those new object allocations.

There are 3 basic concepts in Memory Management

- The Heap and Nursery
- Object Allocation
- Garbage Collection

The Heap and Nursery

Java objects reside in an area called the heap. The heap is created when the JVM starts up and may increase or decrease in size while the application runs. When the heap becomes full, garbage is collected. During the garbage collection objects that are no longer used are cleared, thus making space for new objects.

Note that the JVM uses more memory than just the heap. For example Java methods, thread stacks and native handles are allocated in memory separate from the heap, as well as JVM internal data structures.

The heap is sometimes divided into two areas (or generations) called the nursery (or young space) and the old space. The nursery is a part of the heap reserved for allocation of new objects. When the nursery becomes full, garbage is collected by running a special young collection, where all objects that have lived long enough in the nursery are promoted (moved) to the old space, thus freeing up the nursery for more object allocation. When the old space becomes full garbage is collected there, a process called an old collection.

The reasoning behind a nursery is that most objects are temporary and short lived. A young collection is designed to be swift at finding newly allocated objects that are still alive and moving them away from the nursery. Typically, a young collection frees a given amount of memory much faster than an old collection or a garbage collection of a single-generational heap (a heap without a nursery).

In R27.2.0 and later releases, a part of the nursery is reserved as a keep area. The keep area contains the most recently allocated objects in the nursery and is not garbage collected until the next young collection. This prevents objects from being promoted just

because they were allocated right before a young collection started.

Object Allocation

During object allocation, the JRockit JVM distinguishes between small and large objects. The limit for when an object is considered large depends on the JVM version, the heap size, the garbage collection strategy and the platform used, but is usually somewhere between 2 and 128 kB. Please see the documentation for -XXtlaSize and -XXlargeObjectLimit for more information.

Small objects are allocated in thread local areas (TLAs). The thread local areas are free chunks reserved from the heap and given to a Java thread for exclusive use. The thread can then allocate objects in its TLA without synchronizing with other threads. When the TLA becomes full, the thread simply requests a new TLA. The TLAs are reserved from the nursery if such exists, otherwise they are reserved anywhere in the heap.

Large objects that don't fit inside a TLA are allocated directly on the heap. When a nursery is used, the large objects are allocated directly in old space. Allocation of large objects requires more synchronization between the Java threads, although the JRockit JVM uses a system of caches of free chunks of different sizes to reduce the need for synchronization and improve the allocation speed.

Garbage Collection

Garbage collection is the process of freeing space in the heap or the nursery for allocation of new objects. This section describes the garbage collection in the JRockit JVM.

- The Mark and Sweep Model
- Generational Garbage Collection
- Dynamic and Static Garbage Collection Modes
- Compaction

The Mark and Sweep Model

The JRockit JVM uses the mark and sweep garbage collection model for performing garbage collections of the whole heap. A mark and sweep garbage collection consists of two phases, the mark phase and the sweep phase.

During the mark phase all objects that are reachable from Java threads, native handles and other root sources are marked as

alive, as well as the objects that are reachable from these objects and so forth. This process identifies and marks all objects that are still used, and the rest can be considered garbage.

During the sweep phase the heap is traversed to find the gaps between the live objects. These gaps are recorded in a free list and are made available for new object allocation.

The JRockit JVM uses two improved versions of the mark and sweep model. One is mostly concurrent mark and sweep and the other is parallel mark and sweep. You can also mix the two strategies, running for example mostly concurrent mark and parallel sweep.

Generational Garbage Collection

The nursery, when it exists, is garbage collected with a special garbage collection called a young collection. A garbage collection strategy which uses a nursery is called a generational garbage collection strategy, or simply generational garbage collection.

The young collector used in the JRockit JVM identifies and promotes all live objects in the nursery that are outside the keep area to the old space. This work is done in parallel using all available CPUs. The Java threads are paused during the entire young collection.

Dynamic and Static Garbage Collection Mode

By default, the JRockit JVM uses a dynamic garbage collection mode that automatically selects a garbage collection strategy to use, aiming at optimizing the application throughput. You can also choose between two other dynamic garbage collection modes or select the garbage collection strategy statically. The following dynamic modes are available:

(a)**throughput**, which optimizes the garbage collector for maximum application throughput. This is the default mode.

(b)**pausetime**, which optimizes the garbage collector for short and even pause times.

(c)**deterministic**, which optimizes the garbage collector for very short and deterministic pause times. This mode is only available as a part of Oracle JRockit Real Time.

The major static strategies are:

(a)**singlepar**, which is a single-generational parallel garbage collector (same as parallel)

(b)**genpar**, which is a two-generational parallel garbage collector

(c)**singlecon**, which is a single-generational mostly concurrent garbage collector

(d)**gencon**, which is a two-generational mostly concurrent garbage collector

Compaction

Objects that are allocated next to each other will not necessarily become unreachable (“die”) at the same time. This means that the heap may become fragmented after a garbage collection, so that the free spaces in the heap are many but small, making allocation of large objects hard or even impossible. Free spaces that are smaller than the minimum thread local area (TLA) size can not be used at all, and the garbage collector discards them as dark matter until a future garbage collection frees enough space next to them to create a space large enough for a TLA.

To reduce fragmentation, the JRockit JVM compacts a part of the heap at every garbage collection (old collection). Compaction moves objects closer together and further down in the heap, thus creating larger free areas near the top of the heap. The size and position of the compaction area as well as the compaction method is selected by advanced heuristics, depending on the garbage collection mode used.

Compaction is performed at the beginning of or during the sweep phase and while all Java threads are paused.

---

6)

Ex 7.1

```
j = 0;
i = 3/j;
for(i=1;i>-1;i++)
     i--;
```

In this above snippet, although syntactically it is right, there are two semantic errors that occur here.

Since j is zero,and when there is a divison by zero,the operation is undefined.

In the loop construct, due to the increment and decrementation of the control variable, the loop will never terminate.

Both these are semantic errors which can be handled only in run time and result in run time errors.

Ex 7.2

Some common examples that we can consider of statements which are syntactically correct but semantically wrong are listed below

- Division by zero

  Eg i = 0; cout<<3/i<<endl;

- Non terminating loop

  Eg while(1){}

- Dangling else problem

```
if(cond1)
    if(cond2)
        if(cond3)
    else{}
```

- Using float value as second parameter in a statement

  Eg a%2

- Data Underflow(Accessing non existant value)
- Data Overflow(Unable to allocate location for data)
- Pointer access (Orphans and widows)

```
Eg p = new Node(5);
   q = p
   delete(p)
```

Ex 7.3

(a) Java supports the idea of infinity only for the data types double and float and not for int. There exists two types of infinty.

- Positive Infinty

  public static final double POSITIVE_INFINITY
  A constant holding the positive infinity of type double. It is equal to the value Double.longBitsToDouble(0X7ff0000000000000L)

- Negative Infinity

  public static final double NEGATIVE_INFINITY
  A constant holding the negative infinity of type double. It is equal to the value Double.longBitsToDouble(0Xfff0000000000000L)

- NaN

  public static final double NaN
  A constant holding a Not-a-Number (NaN) value of type double. It is equivalent to the value returned Double.longBitsToDouble(0x7ff8000000000000L).

The method .isInfinite() is used to check whether the double or float value is infinite or not.

(b) If we consider the statement i = 3/j if j == 0 then it produces an Arithmetic Exception which i caught and the exception is displayed on terminal but if the numerator is a float value then there is no exception instead infinity is printed on the screen.

In case of double/float division, the output is Infinity, the basic reason behind that it implements the floating point arithmetic algorithm which specifies a special values like “Not a number” OR “infinity” for “divided by zero cases” as per IEEE 754 standards but in the case of integer it throws an Arithmetic Exception

---

Ex 7.4

Type of x	Type of y	Type of return value
int	int	int
int	float	float
float	int	float
float	float	float
char	char	int

char	int	int
int	char	int
char	float	float
float	char	float

Ex 7.5

```
import java.io.*;
import java.lang.*;
class sample{
    public static int factorial(int val){
        int f = 1;
        int i = 1;
        while(i<val){
            i++;
            f = f * i;
        }
        return f;
    }
    public static void main(String[] args){
     Scanner sc = new Scanner(System.in);
     int n = sc.nextInt();
     System.out.println(factorial(n));
     }
}
```

For any n bit representation the max value that can be represented is 2^n -1.

In that case the integer data type can hold values upto (2^32)-1.

The range for this integer datatype is from -(2^31) to (2^31)-1.

If we consider unsigned integer datatype the range is from 0 to (2^32)-1.

In both the cases of signed and unsigned integer the datatype can hold only upto factorial of 12.
Beyond 12 it cannot hold values.

In that case the integer datatype cannot hold those values.
In case of unsigned integer some garbage value within the range gets thrown up.
In case of signed integer negative value or some positive garbage value get thrown up.

Ex 7.6

It does not throw any error while typecasting it. Instead it displays a value cooresponding to the ascii value of 1.

It is because the ascii values are repeated in a cycle whose period is from 0 to 255. In other words, it works like number mod with 256 and displays the ascii value of the mod value on the screen.

257 mod 256 is 1 and value of ascii 1 is displayed on screen.

So,there are no errors in the code.

Ex 7.7

It does not throw any error while typecasting it. Instead it displays a value cooresponding to the ascii value of 1.

It is because the ascii values are repeated in a cycle whose period is from 0 to 255. In other words, it works like number mod with 256 and displays the ascii value of the mod value on the screen.

65537 mod 256 is 1 and value of ascii 1 is displayed on screen.

So,there are no errors in the code.

Ex 7.8

In both the cases of C++ and Java the intital and final values are not the same. This is because of the design of the compilers and interpreters are such. Whenever an integer is converted to a float, the float value is always represented in terms of scientific notations and in that case there will be only ones place and followed by six decimal places and an exponent value at the end.Since the float data type has a precision of only 6 digits, the last 2 digits incase of a 8 digit integer will be rounded off and stored in the decimal part and hence there will be

a change of value and hence when we convert them back to integer, we do not get back the same values. So that is why there is a difference in the values before and after the conversion.

The same output is obtained in both Java and C++.

Code

```
import java.io.*;
import java.lang.*;
class sample{
    public static void main(String[] args){
        int val = (int)Math.pow(2,30) + 65534;
        System.out.println(val);
        float vv = (float)val;
        System.out.println(vv);
        int vvv = (int)vv-1;
        System.out.println(vvv);
    }
}
```

Output

```
1073807358
1.07380736E9
1073807359
```

Here, 1073807358 gets rounded off to 1.07380736 and this value is then converted to int in reverse process as 1073807360 and 1 is subtracted from it to make it 1073807359.

Ex 7.12

In C, there are no specific errors corresponding to the error mentioned in the table given.

The action of these errors are returned as values specific to the functions in that are being called.

The error actions may be sometimes returned as signals to the processes,exit value of a process or hardware interrupts.

Ex 7.13

(a)Differences between C++ Exceptions and Java Exceptions

C++	Java
In C++, all types (including primitive and pointer) can be thrown as exception.	In Java,only throwable objects (Throwable objects are instances of any subclass of the Throwable class) can be thrown as exception.
In C++, there is a special catch called “catch all” that can catch all kind of exceptions	In Java,there is no explicit “catch all” but “Exception” object can catch any type of exception.
In C++, there is no finally block.	In Java, there is a block called finally that is always executed after the try-catch block. This block can be used to do cleanup work.
In C++,all exceptions are unchecked.	In Java,exceptions can be checked or unchecked.
In C++, there is no throws keyword, the same keyword throw is used for this purpose also.	In Java, a new keyword throws is used to list exceptions that can be thrown by a function.

(b) Differences between Ada Exceptions and Java Exceptions

Ada	Java
An exception is said to be raised at the point of occurence and handled at the point where control is transferred.	An exception is said to be thrown at the point of occurence and handled at the point where control is transferred.
An exception can be decalred anywhere with a specific name and word exception to indicate it. Transmission_Error: exception;	The programamer decalres an exception as a subclass of the class Exception public class TransmissionError extends Exception {...}
There are four pre-defined exceptions: (a)Constraint_Error, (b)Program_Error, (c)Storage_Error, (d)Tasking_Error.	There is a large number of exception classes predefined in the package java.lang. Object | Throwable

	/ \ Error Exception / | \ / | \
The failure of a language-defined check raises one of the four predefined exceptions.	The failure of a language-defined check throws an exception of one of the predefined subclasses of Error or RuntimeException
By using the pragma Suppress, i.e. a directive to the compiler, permission can be given to an implementation to omit certain languagedefined checks.	Languagedefined checks are always performed, and cannot be suppressed.

begin
  ... -- call operations in File_System exception
when End_Of_File =>
  Close (Some_File);
when File_Not_Found =>
  Put_Line (“Some specific message”));
when others =>
  Put_Line (“Unknown Error”); end;
	try {
	  int a[] = new int[2]; a[4];
	}
	catch (ArrayIndexOutOfBoundsException e) {
	System.out.println (“exception:” + e.getMessage());
	e.printStackTrace();
	}

Ex 7.14

```
BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));

int count = 0;

int number;

while(count<3){

   try{

      System.out.println("Enter number: ");

      number = Integer.parseInt(in.readLine());

      break;

    }

    catch(NumberFormatException e){
```

```
        System.out.println("Invalid number,please re-enter");
        count++;
        System.out.println(3-count +" attempts left");
    }
    catch(IOException e){
        System.out.println("Input error,please re-enter");
        count++;
        System.out.println(3-count +" attempts left");
    }
}
```

Ex 7.15

```
import java.io.*;
import java.lang.*;
class sample{
     public static void main(String[] args){
BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
int count = 0;
int sum = 0;
float avg = 0;
int number;
while(true){
   try{
      System.out.println("Enter number: ");
      number = Integer.parseInt(in.readLine());
      if(number >= 0){
        sum+=number;
        count++;
      }
```

```
      else{
         if(count != 0)
           avg=(float)sum / count;
           break;
         }
      }
      catch(NumberFormatException e){
         System.out.println("Invalid number,please re-enter");
      }
      catch(IOException e){
         System.out.println("Input error,please re-enter");
      }
   }
   System.out.println("SUM : " + sum);
   System.out.println("AVERAGE : " + avg);
  }
}
```

Ex 7.16 (a),(b) and Ex 7.17

```
import java.lang.*;
import java.util.*;
import java.io.*;
class StackUnderflowException extends Exception{
    public StackUnderflowException(){
        super();
    }
    public StackUnderflowException(String s){
        super(s);
    }
}
```

```
class StackOverflowException extends Exception{
    public StackOverflowException(){
        super();
    }
    public StackOverflowException(String s){
        super(s);
    }
}
class Stack{
    private int stack[];
    private int top = 0;
    public Stack(int n){
        stack = new int[n];
    }
    public void push(int val) throws StackOverflowException{
        if(top >= stack.length)
            throw new StackOverflowException("Push on Full
Stack");
        stack[top] = val;
        top++;
    }
    public int pop() throws StackUnderflowException{
        if(top<=0)
            throw new StackUnderflowException("Pop on Empty
Stack");
        return stack[--top];
    }
    public int peep() throws StackUnderflowException{
        if(top == 0)
             throw new StackUnderflowException("Peep on Empty
Stack");
```

```
        return stack[top-1];
    }
    public static void main(String[] args){
        Stack s = new Stack(3);
        try{
            int val = 1;
            //s.peep();
            s.push(val);
            val++;
            System.out.println("STACK TOP: " + s.peep());
            s.push(val);
            val++;
            System.out.println("STACK TOP: " + s.peep());
            s.push(val);
            val++;
            System.out.println("STACK TOP: " + s.peep());
            s.push(val);
        }
        catch(Exception e){
            System.out.println(e);
        }
    }
}
```

Ex 16(c)

```
import java.lang.*;
import java.util.*;
import java.io.*;
class StackAssert{
    private int stack[];
```

```
private int top = 0;
public StackAssert(int n){
    stack = new int[n];
}
public void push(int val){
    assert top < stack.length:"Push on Full Stack";
    stack[top] = val;
    top++;
}
public int pop(){
    assert top > 0:"Pop on Empty Stack";
    return stack[--top];
}
public int peep(){
    assert top!=0:"Peep from empty stack";
    return stack[top-1];
}
public static void main(String[] args){
    StackAssert s = new StackAssert(3);
    int val = 1;
    s.push(val);
    val++;
    System.out.println("STACK TOP: " + s.peep());
    s.push(val);
    val++;
    System.out.println("STACK TOP: " + s.peep());
    s.push(val);
    val++;
    System.out.println("STACK TOP: " + s.peep());
    s.push(val);
```

```
    }
}
```

---

7)

Ex 8.1

(a) The increment operator would be of two types: pre-increment and post-increment. In the case of the type checker it needs to check whether the operand on the left side or right side must not be an undef. The type of the operand can be int, float or char (increment yields the next ascii value character) and that the values must be at least one less than the max value each type can hold.

(b) The meaning of the increment statement is that it takes a source variable in the current state and increments the value by one and assigns it to the source variable itself thus changing its state.

(c) Add the functio to clite interpreter:

```
State M(IncrementStatement i, State sigma)
{
StaticTypeCheck.check(Sigma.containsKey(I.op), “refernce to undefined variable”);
return state.onion(i.op, M(i.op+1, sigma));
}
```

(d) The meaning of an expression is the value that it produces and it guarantees that there is no state change that is taking place. But an increment operator is similar to an assignment operator where the value of source variable in current state is retrieved and incremented by one and assigned to the source variable itself thus causing a state change violating rules of expression.

Ex 8.2

(a) Concrete Syntax:

```
Cin --> cin statement;
statement --> >> val statement;
```

val --> identifier | constant

Cout --> cout statement;

statement --> << identifiers statement;

val --> identifier | constant

(b) Abstract Syntax:

CIN = cin >> ; Identifier | constants list

CIN = cout << ; Identifier | constants list

(c) cin and cout are added to the lexical analyser. The >> and << operator are added to the operator list and type checking is restricted to input stream on left most side for >> operator and output stream on left most side for << operator while the right hand side would be variables that have been declared.

(d) In the Clite interpreter the above changes are made. The only differences is the right hand side variables need not be declared to be used in cin or cout.

Ex 8.3

Allowing and disallowing b=a where a is undef:

Disadvantage of disallowing:

When comes to pointer arithmetic there may be situations of one pointer referencing another which may be undef in current state but may be defined in a future state so that the target pointer can also follow reference the source.

Advantage of disallowing:

When such a statement is allowed handling of states of variables when performing future operations becomes more tedious and erroneous as there are chances of encountering more and more undefined variables in expression calculations.

If we allow such statements the changes in exprssion should be made as follows:

```
Value M (Expression e, State state) {I
     if (e instanceof Value)
          return (Value)e;
```

```
        if (e instanceof Variable) {
             StaticTypeCheck.check( !(Value)(state.get(e)),
        “refernce to undefined variable”);
             return (Value)(state.get(e}};
        }
        if (e instanceof Binary) {
             Binary b = (Binary)e;
             return applyBinary (b.op, M(b.terml, state),
        M(b.term2, state)):
        }
        if {e instanceof Unary) {
             Unary u = {Unary)e;
             return applyUnary(u.op, M(u.term. state));
        }
        throw new IllegalArgumentException(“should never reach
        here”
    };
```

Ex 8.4

(a) ModuloOperation --> Term ModuloOp Term

ModuloOp --> %

Term --> Identifier | ArithmeticExpression

(b) ModuloOperation = BinaryOp %; Expression term1, term2

(c) Lexer is modified by adding % to the operator set and retrun the token as MOD while the parser uses the concrete syntax to parse the modulo operation.

(d) Add the following to the apply Binary function:

```
        if(v1.type() == Type.INT)
             if(op.val.equals(Operator.MOD))
```

```
            return new IntValue(v1.intValue() %
        v2.intValue());
```

Ex 8.5

a && b = if a then if b then true else false else if b then false else false

a || b = if a then if b then true else true else if b then true else false

In this definition of the grammar short circuiting can be avoided.

Ex 8.8

Step	Before statement	Variables		
		i	a	Z
1	3	undef	undef	undef
2	4	5	undef	undef
3	5	5	2	undef
4	6	5	2	1
5	7	5	2	1
6	8	5	2	1
7	9	5	2	2
8	10	2	2	2
9	6	2	4	2
10	7	2	4	2
11	9	2	4	2
12	10	1	4	2
13	6	1	16	2
14	7	1	16	2
15	8	1	16	2
16	9	1	16	32
17	10	0	16	32
18	6	0	256	32
19	12	0	256	32

Ex 8.9

(a) Assignment --> Identifier = Expression

Expression --> Expression * Identifier

Expression --> Identifier

Identifier --> a | d

(b) Assignment --> (Int) Identifier = (Int) Expression

Expression --> (Int)((Int) Expression*(Int) Identifier)

Expression --> (Int) Identifier

Identifier --> (Int) a | (Int) d

(c) The expression z*a is computed by calling ApplyBinary(*,z,a). Here the op.val will equal Operator.INT_TIMES and the values od z and a are extracted and multiplied and the result is returned as an integer.

Ex 8.15

(a) {(x,1),(y,5),(z,3)}

(b) {(x,1),(y,2),(z,3),(w,1)}

(c) {(y,5),(w,1)}

(d) {(y,5)}

(e) Null set

(f) Null set

(g) {(x,1),(y,2),(z,3),(w,1)}

Ex 8.16

(a) M((z+2)*y,{(x,2),(y,-3),(z,75)})

M((z+2)*y,{(x,2),(y,-3),(z,75)}) = ApplyBinary(*,A,B)

A = M(z+2,{(x,2),(y,-3),(z,75)})

B = M(y,{(x,2),(y,-3),(z,75)})

B = -3 as meaning og B is -3

A can be further split as follows

M(z+2,{(x,2),(y,-3),(z,75)}) = ApplyBinary(+,C,D)

C = M(z,{(x,2),(y,-3),(z,75)})

D = M(2,{(x,2),(y,-3),(z,75)})

Here D = 2 and value of C = 75 as meaning of C is 75

M(z+2,{(x,2),(y,-3),(z,75)}) = ApplyBinary(+,75,2)

= 77

M((z+2)*y,{(x,2),(y,-3),(z,75)}) = ApplyBinary(*,77,-3)

= -231

(b) M(2*x+3/y-4,{(x,2),(y,-3),(z,75)})

M(2*x+3/y-4,{(x,2),(y,-3),(z,75)}) = ApplyBinary(-,A,B)

A = (2*x+3/y,{(x,2),(y,-3),(z,75)})

B = (4,{(x,2),(y,-3),(z,75)})

Here,B = 4

A can be further split as follows

M(2*x+3/y,{(x,2),(y,-3),(z,75)}) = ApplyBinary(+,C,D)

C = (2*x,{(x,2),(y,-3),(z,75)})

D = (3/y,{(x,2),(y,-3),(z,75)})

C can be further split as follows

M(2*x,{(x,2),(y,-3),(z,75)}) = ApplyBinary(*,E,F)

E = M(2,{(x,2),(y,-3),(z,75)})

F = M(x,{(x,2),(y,-3),(z,75)})

Here E = 2 and F = 2

D can be further split as follows

M(3/y,{(x,2),(y,-3),(z,75)}) = ApplyBinary(/,G,H)

G = (3,{(x,2),(y,-3),(z,75)})

H = (y,{(x,2),(y,-3),(z,75)})

Here G = 3 and H = -3

M(2*x,{(x,2),(y,-3),(z,75)}) = ApplyBinary(*,2,2)

= 4

M(3/y,{(x,2),(y,-3),(z,75)}) = ApplyBinary(/,3,-3)

= -1

M(2*x+3/y,{(x,2),(y,-3),(z,75)}) = ApplyBinary(+,C,D)

= 3

M(2*x+3/y-4,{(x,2),(y,-3),(z,75)}) = ApplyBinary(-,3,4)

= -1

(c) M(1,{(x,2),(y,-3),(z,75)}) = 1

Ex 8.17

M(z=2*x+3/y-4,{(x,6),(y,-12),(z,75)})

M(z=2*x+3/y-4,{(x,6),(y,-12),(z,75)}) =

{(x,6),(y,-12),(z,75)} U {z,M(2*x+3/y-4,{(x,6),(y,-12),(z,75)})}

M(2*x+3/y-4,{(x,6),(y,-12),(z,75)}) is found as follows

M(2*x+3/y-4,{(x,6),(y,-12),(z,75)}) = ApplyBinary(-,A,B)

A = (2*x+3/y,{(x,6),(y,-12),(z,75)})

B = (4,{(x,6),(y,-12),(z,75)})

Here,B = 4

A can be further split as follows

M(2*x+3/y,{(x,6),(y,-12),(z,75)}) = ApplyBinary(+,C,D)

C = (2*x,{(x,6),(y,-12),(z,75)})

D = (3/y,{(x,6),(y,-12),(z,75)})

C can be further split as follows

M(2*x,{(x,6),(y,-12),(z,75)}) = ApplyBinary(*,E,F)

E = M(2,{(x,6),(y,-12),(z,75)})

F = M(x,{(x,6),(y,-12),(z,75)})

Here E = 2 and F = 6

D can be further split as follows

M(3/y,{(x,6),(y,-12),(z,75)}) = ApplyBinary(/,G,H)

G = (3,{(x,6),(y,-12),(z,75)})

H = (y,{(x,6),(y,-12),(z,75)})

Here G = 3 and H = -12

M(2*x,{(x,6),(y,-12),(z,75)}) = ApplyBinary(*,2,6)

= 12

M(3/y,{(x,6),(y,-12),(z,75)}) = ApplyBinary(/,3,-12)

= 0

M(2*x+3/y,{(x,6),(y,-12),(z,75)}) = ApplyBinary(+,C,D)

= 12

M(2*x+3/y-4,{(x,6),(y,-12),(z,75)}) = ApplyBinary(-,12,4)

= 8

M(z=2*x+3/y-4,{(x,6),(y,-12),(z,75)}) =

{(x,6),(y,-12),(z,75)} U {z,8}

M(z=2*x+3/y-4,{(x,6),(y,-12),(z,75)}) = **{(x,6),(y,-12),(z,8)}**

---

8)

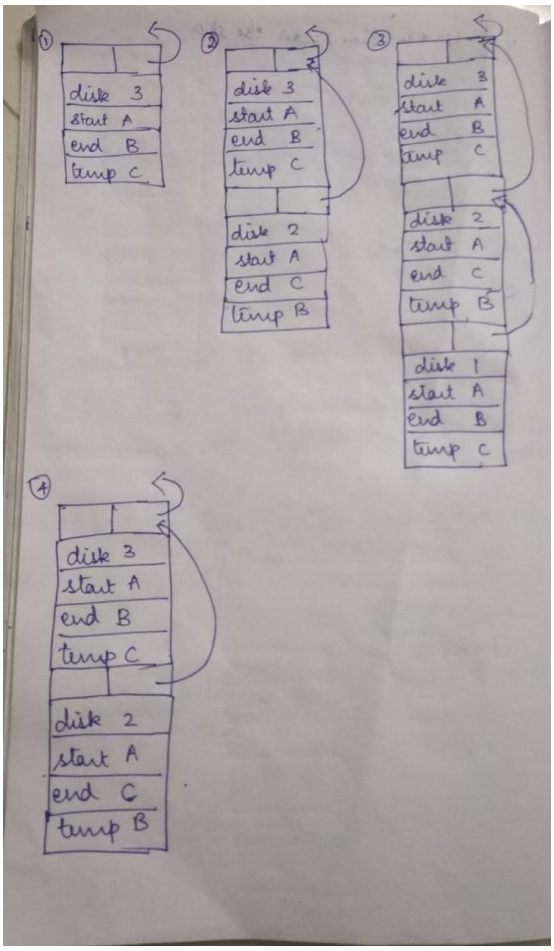
Ex 9.1

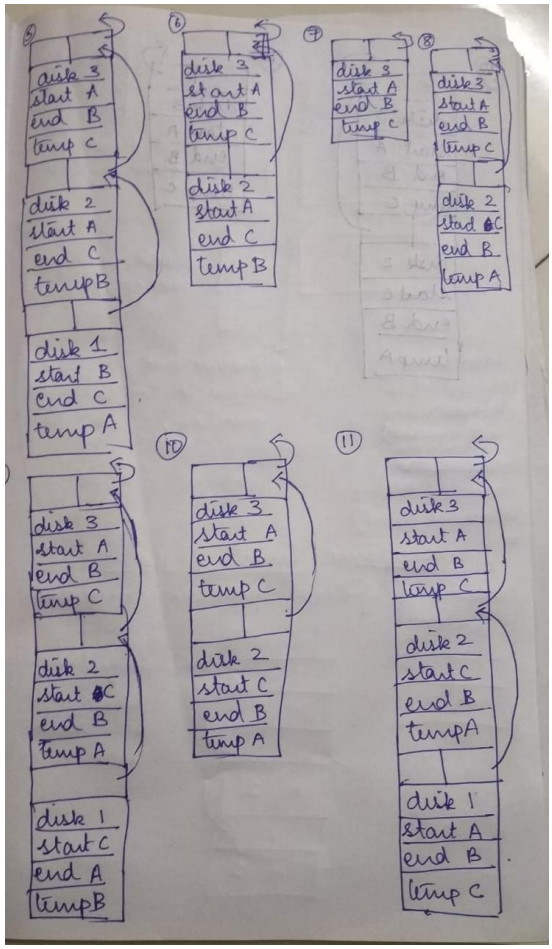
(a) 5 2 4

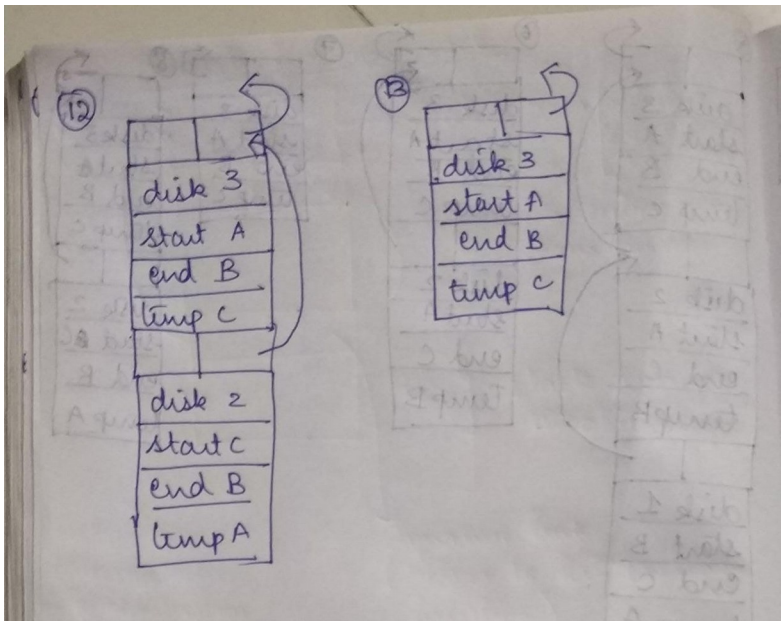
(b) 5 4 2

(c) 5 4 2

Ex 9.4


1
disk 3
start A
end B
temp C
2
disk 3
start A
end B
temp C
disk 2
start A
end C
temp B
3
disk 3
start A
end B
temp C
disk 2
start A
end C
temp B
disk 1
start A
end B
temp C
4
disk 3
start A
end B
temp C
disk 2
start A
end C
temp B


5
disk 3
start A
end B
temp C
disk 2
start A
end C
temp B
disk 1
start B
end C
temp A
6
disk 3
start A
end B
temp C
disk 2
start A
end C
temp B
7
disk 3
start A
end B
temp C
8
disk 3
start A
end B
temp C
disk 2
start C
end B
temp A
10
disk 3
start A
end B
temp C
disk 2
start C
end B
temp A
disk 1
start C
end A
temp B
11
disk 3
start A
end B
temp C
disk 2
start C
end B
temp A
disk 1
start A
end B
temp C

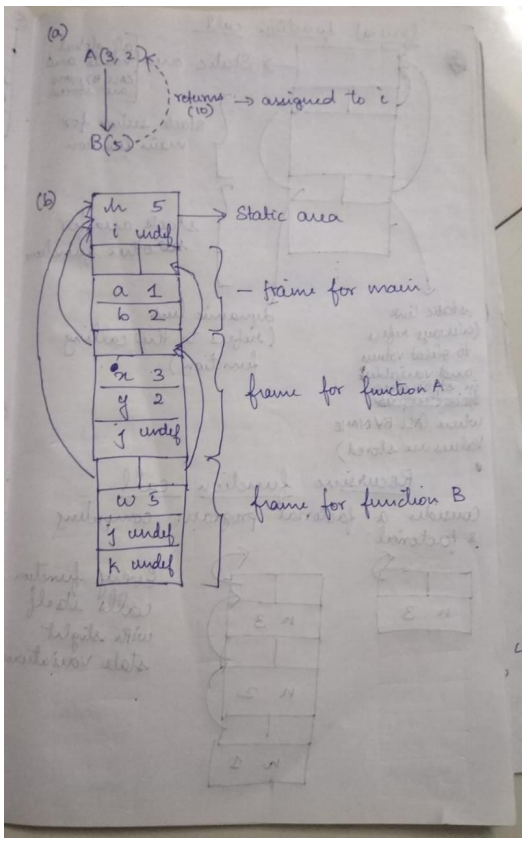

12
disk 3
start A
end B
temp C
disk 2
start C
end B
temp A
13
disk 3
start A
end B
temp C

Ex 9.5

Modified Source Code for the figure 9.1

```
int h,i ;
int B(int w)
{
     int j, k;
     return 2*w;
}
void A(int x, int y) {
     bool j;
     i= B(h);
}
int main()
{
     int a, b;
     h=5; a = 3; b=2;
     A(a,b);
}
```

(a) and (b)



(c) If the local declaration from the function A had not been removed then the function would allocate the value returned by b to the local variable i (typecasting the returned value to

boolean) rather than the global variable i which doesn’t intend to solve our purpose

---

Ex 9.8

```
#include<stdio.h>
void swap(int arr[],int a,int b){
    int temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}
void sort(int arr[],int m,int n){
    int i,j,key;
    if(m < n){
        int part;
        int pi = arr[n];
        int i = m - 1;
        int j;
        for(j = m;j<=n-1;j++){
            if(arr[j]<=pi){
                i++;
                swap(arr,i,j);
            }
        }
        swap(arr,i+1,n);
        part = i+1;
        sort(arr,m,part-1);
        sort(arr,part+1,n);
    }
}
void quicksort(int arr[],int len){
```

```
    sort(arr,0,len-1);
}
int main(){
    int arr[] = {4,5,3,6,1};
    int n = 5;
    quicksort(arr,5);
    int i;
    for(i = 0;i<5;i++)
        printf("%d ",arr[i]);
    printf("\n");
    return 0;
}
```

There are the three functions and an array(list) is passed by refernce to all the functions so that the changes are reflected in each case.

---

9)

Ex 10.1

(a)concrete syntax: ReturnStatement->return Expression; | return;
abstract syntax:

(b)
Type rule 10.4:
A return statement must appear in the body of non void functions except main,and its expression must have same Type as that of function

Type rule 5:
Return statement may appear in the body of void function with expression having void type being assigned to void or no return statement and no type in the function call.

Ex 10.2

```
int h,i;
void C(char h)
{
     h=h+1;
}
void B(int w){
```

```
     int j,k;
     i=2*w;
     w=w+1;
     C(i);
}
void A(int x,int y){
     bool i,j;
     B(h,i);
}
int main()
{
     int a,b;
     h=5,a=3,b=2;
     int q=A(a,b);
}
```

Ex 10.5

The meaning rule 10.1 can be modified to accomodate functions without side effecs only if it functions cannot have global variabes,static variables and has no parameter of reference type which practically seems to be difficult.

---

10)

Ex 11.1

Feature	new/delete	malloc/free
Memory allocated from	Free Store	Heap
Returns	Fully typed pointer	void* (unless typecasted)
On failure	Throws(never returns NULL)	Returns NULL
Required size	Calculated by compiler	Must be specified in bytes
Handling arrays	Has an explicit version	Requires manual calculation
Reallocating	Not handled intuitively	Simple(no copy constructor)
Call of reverse	Implementation defined	No
Low memory cases	Can add a new memory allocator	Not handled by user code
Overridable	Yes	No

Use of constructor/destructor	Yes	No

Ex 11.2

The existing garbage collector in Perl is **Reference Counting.** It is already available to the programmers. The problem that will arise in Reference Counting is circular references.

Perl 6 has a pluggable garbage collected scheme (the underlying VM will have multiple garbage collection options and the behavior of those options can have an effect on Perl). We can choose between various garbage collectors, or implement our own garbage collector.

In perl it is easy to create circular references and the programmers need to remove it explicitly as the garbage collector by defaualt doesn't remove it.

Example

//Creating a circular reference

```
sub create_pair {
    my %val1;
    my %val2;
    $val1{other} = \%val2;
    $val2{other} = \%val1; //Circular reference is created.
    return;
}
```

Weaking the circular reference

The solution available for circular reference in perl is using weaken available in Scalar::Util

```
use Scalar::Util qw(weaken);
sub create_pair {
    my %val1;
    my %val2;
    $val1{other} = \%val2;
```

```
    $val2{other} = \%val1;
    weaken $val2{other}; //Weakening the reference
    return;
}
```

In reference counting all the widows will be automatically cleared and in the case of orphans refernce count will become zero and all such nodes are added to the free_list which is done automatically without programmer intervention.

Ex 11.3

- ArrayIndexOutOfBoundsExceptions
- NegaiveArraySizeExceptions
- NullPointerExceptions
- OutOfMemoryError
- InternalError

Ex 11.4

ArrayIndexOutOfBoundsExceptions

In the scenario stated above, the stack contains the heap address of the first element of array in the dope vector for the array and the usage of a negative index in that case might result in going to a heap index that is in an unused state and an error might be throw up and abrupt disruption of the execution takes place.The same scenario comes again if we use an index beyond the range of the array.

NegaiveArraySizeExceptions

This is not an indexing error but in case of a negative size in the decalration of the array,the exception is raised ane exited.

Ex 11.6

Dynamic arrays can be allocated in stack during runtime in C Lite.

There are certain problems as well as advantages in terms of accession and size of declaration in declarations involving stacks.

In terms of declarations it would be like this

```
State initialState (Declarations d){ I
     State state= State();
          for (Declaration decl : d){
               val = changeState();
               state.put(decl.v+val, Value.mkValue(decl.t));
          }
          return state;
}
```

In terms of accession(referencing) the semantics could be a the same as that as local variable stored in a stack but with the presence of offsets and the data is stored continuously.

a.target in a declaration would be rechanged as a.target + offset value

---

Ex 11.7

Based on space we can easily say that the allocation of dynamic array in a heap is much more advantageous because in such a case, the stack will contain only the dope vector and hence the memory in the stack is also not wasted but if the dynamic array is implementd on the stack, then there would be huge memory problems as the content is directly written to the stack and condiserably occupies the stack space and the stack also has a limted space and hence we would not be in a position to use the stack if such a scenario occurs.Furthermore,after allocating a dynamic array in a stack ,if the array is passed as a parameter to another function, then the entire array is duplicated on the stack again and hence the stack becomes full and is not effectively utilized.

Based on time, we can say that stack is better than heap because in heap before insertion , there might be cases where garbage collector needs to be run and the running of algorithm of garbage collector varies depending upon the algorithm that is being used whereas in a stack there is no such problem like a garbage collector and data is directly written into it.

Space – Heap Time – Stack.
