11th July'17
Tuesday

Principles Of Programming Languages

(Q) Why to study?

(1) → to improve the ability to develop efficient algorithms.

(2) → To improve the use of existing programming language.

(3) → to improve the vocabulary of various programming constructs.

(4) → to learn new programming languages.

(5) → to learn a better choice of language.
Eg: to use JAVA, when reqd. to study internet based application.

(6) → to design a new language.

* Generation of languages → first, second, third gen. (in detail).

Language Evaluation:

(1) Readability :- ease with which some code (programs) can be understood & read.

factors affecting ~~read~~ readability of a program →

(A) Simplicity :
should contain min. no. of data structures and data constructs.

(B) Orthogonality : ~~Orth~~ It affects readability of program.

→ Orthogonality means the min. no. of smaller data constructs reqd. to form a bigger & simple data construct.

OR

* relative no. of programming (primitive) constructs combined in relatively smaller no. of ways to form new constructs.

→ Too much orthogonality not reqd otherwise it'll make program complex.

Ⓒ Control Statements : affects readability

Eg: Goto statement is not used in programs. ∵ ↓ readability of programs.

Eg- if-else, while, for, etc.

Ⓓ DATA TYPE AND STRUCTURE →

Eg: seatreserved = 10
" = TRUE (better)

Ⓔ SYNTAX : should be user-friendly & not restricted / constrained.

→ A language that allows user to use complete words will have better understanding.

3

12th July '17

Wednesday.

C++

Structured programming

OOP.
- Class, Object
  - Attributes
    - Private
    - Public
    - Protected
  - Inheritance
  - Encapsulation
  - Constructor / Destructor.
  - Friend Fn (in Inheritance)
  - Cohesion → self study.
  - Coupling
  - Abstraction
  - virtual Fn.
  - Abstract class.

* **Inheritance:-**
is used for reusing the code.
4 Types →
  1. Single level
  2. Multi-level
  3. Multiple
  4. Hierarchial.

* Structured C++ Programs →

4

Addition of 2 nos →

C

```
#include<stdio.h>
#include<conio.h>

int main()
{
   int a, b, sum;

printf("Enter a & b=");
scanf("%d %d", &a, &b);

sum = a+b;

printf("sum=%d", sum);

getch();
}
```

C++

```
#include<iostream.h>
#include<conio.h>
using namespace std;
int main()
{
int a, b, sum;
printf("Enter a & b=");
scanf("%d %d", &a, &b);
sum = a+b;
printf("sum
cout<<"Enter a and b=";
cin>>a>>b;

sum = a+b;

cout<<"sum of nos="<<sum;

getch();
}
```

5

Armstrong Number : (C++) ; Using Structured P.L.

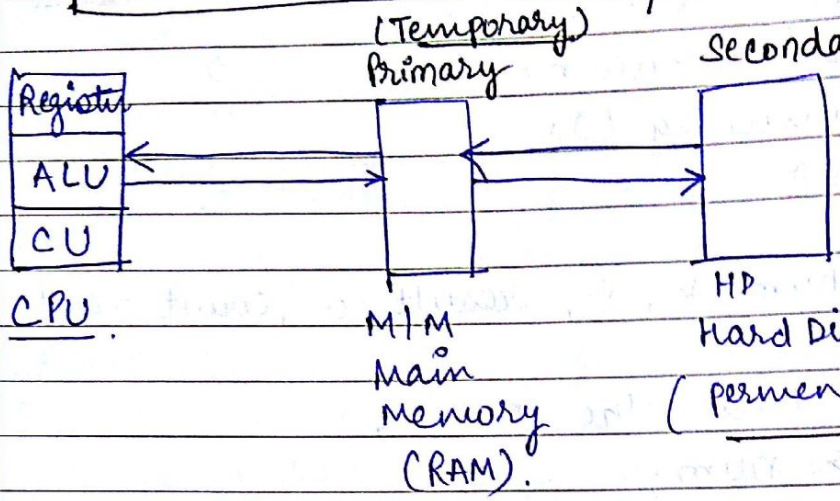
```
# include <iostream.h>
# include <conio.h>
void armstrong ();
int main()
{
int num, k, i, result=0, count=0, t, rem;

cout<<" Enter the no=";
  cin >> num;
  k = num;
  while (k!=0)
  {
    k = k/10;
    count++;
  }
cout<<" No. of digits is =" <<count<<endl.
armstrong();
getch();
}
void armstrong()
{
t = num;
for (i=0; i<count; i++)
  {
  rem = t%10;
  t = t/10;
  result = result + rem * rem;
  }

if (result == num)
{
  cout<<" No. is 'Armstrong' ";
}
else {
  cout<<" Not Armstrong No.\n";
}
}
```

18th July '17
Tuesday

FILE HANDLING



→ Registr : Size large.
→ When computr shutdown, data not saved & lost.

→ ∴ By using file handling, we store data in the hard-disk permenantly.

→ Using concept of FH, data entered by the user can be stored in permenant storage device for future use. even if we powroff the system.

[Q:] What's a File ?
Ans:→ File's a collection of related data that a computr treats as a single unit.

→ computer stores file to secondary (perm) memory, so that content of file remains intact, when a computr shuts down.

→ when a computer opens the file, it copies the file from secondary to primary storage, and when file is closed, it transfers the data from primary to secondary memory.

STEPS FOR FH. OPERATION IN CPP → Programming

In C++, FH is achieved by header file <fstream.h>, while in C, it's achieved by <stdio.h>.

① Declare a file handling variable.

Syntax:
fstream var_name;

② Declare the file name. // to find in which file operations are performed.
→ cout<<"Enter file name:";
→ cin>> ; // User input.

③ Open the file.

Syntax:
var_name.Open (File_name, Mode) → which opn to perform read, write, append

→ For performing any operation in file, 1st var_name is used.

* Mode 3 Types → write
read
append.

⑥ Closing the File → .var-name . close();

		C++	C.
(1)	Write	ios :: out	w
w	ios : : out	ios : : out	
(2)	Read	ios : in	R
R	ios : : in		
(3)	append	ios :: app	a
a	ios : : app		

Scope Resolution Operator (SRO).

- Till now, data was stord in registr, now using write — we can store it in file.

④ → To get / fetch/take value entred from the keyboard.

get	cin
put	cout

Ⓐ cin . get() → only single 'char' value can be input & not string

To print : Ⓑ cout . put() : prints char by char value ;
It prints value on the display unit.

9 (Next class: For string storing data in the file).

★ Store a 'char' in C., so that value's printed.

(C) cout.put(c) ↑

(D) var_name.get() → It takes the value from file.

file में value store करेगा, is still left →.

(E) var_name.put(c) → Put value in file char by char. ↕ OR

cin.get();

(Q.) To know, file has ended when we read?

(5) EOF(); (End of file) ASCII EOF → is 26.

→ It's (EOF) a special character named as 'end of file'. It's inserted automatically after last char in the file, to mark the end of file.

→ Its ASCII value's 26.

★★.

(Q.1) WAP to insert char by char data in the file? (ths.cpp)

```
# include <iostream.h>
# include <conio.h>
# include <fstream.h>

void main()
{
  clrscr();
```

NOTE: char fn[13]; exact 13 size because

8 char + 1 + 3 + 1 = 13.
.cpp \0

```
fstream a;          // declare varname fh:
~~a.open();~~
char fn[13];        // file name
char c;
~~a.open();~~
cout<<"Enter the file name=";
cin>>fn;
a.open(fn, ios::~~ios~~out);  // To open file

cout<<"Enter the text until # is pressed=";
while((c=cin.get())!='#')
{
  a.put(c);
}
a.close();          // program ends
getch();
}
```

HW.
Q) WAP char by char 'read'

11

Q. (Next) cin.get() → keyboard input.

To copy data of 1 file to another.

27th July '17

Thursday.

~~fh.cpp & fh1.cpp~~

(Q.2) ✓ Write a program, to read a file; 'char' by 'char'.

fh_practice.cpp & fh1.cpp

→ means already a file would've been created

→ We need to open a file.

→ Then, read a file.

ios::in (read).

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>

void main()
{
  fstream a; // filehandling variable.
  char fn[13];
  char ch;
  cout<<"enter the name of file =";
  cin >> fn;
  a.open(fn, ios::in);
  // Read the file until EOF is reached.
  while(!a.eof())
  {
  ch=a.get();
  // To be displayed on monitor screen.
  cout.put(ch);
  }
  a.close();
  cout<<" File Read successfully \n";
  getch();
}
```

ch=a.get();

cout.put(ch);

(Q.3) WAP to copy the content of one file to another file?. [fh_practice1.cpp]

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
using namespace std;
void main()
{
  fstream a,b;
  char fn1[13], fn2[13];
  char ch;
  cout<< "Enter name of file = ";
          to be opened in read mode
  cin >> fn1;

  a.open (fn2, ios::in); // To open a file a.open is used.
                         --> // To Read a file.
Syntax // var_name.put (ch, filename);

~~while~~

  cout <<" Enter filname to be created =";
  cin >> fn2;

  b.open ( fn2 ~~ios~~ ios::out); // write in file.

   while (!a.eof())
   {
     ch=a.get();
     b.put(ch);
   }
   a.close();
```

```
b.close();

cout<<" File copied successfully \n";
getch();
}.
```

28th July '17
Friday.

Imp. Topics For Mini-Test :-

1. Insert, Read in File Handling.
2. Static & Dynamic Scoping.

(I) STATIC SCOPING →

If the declaration of any variable's found inside the function (local variables) then, use this declaration. If, the declaration's not-found inside the fn then, refer to the global variable.

(II) Dynamic Scoping →

In a fn, if a local variable's found then use it. If not found, then refer the variable of that fn, from which this fn is called.

Program 1 =>.

```
int x = 2;
void P();
void Q();

void main()
{
int x=4;
P();
getch();
}
P()
{
```

15

* assembler → high level to assembly language.
* compiler → high level to machine language

Static → global.
dynamic → fn.

(Inline fun)

```
int x = 6.
printf ("%d", x);
a();
}
a()
{
printf ("%d", x).
}
```

⇒ Its Implementation is done by using stack.

Static Output : 6, 2 (global variable)
Dynamic Output : 6, 6 (x value in P())

☆ Phases of Compilation →

(1) Translation ⇒ * converting the language of 1 form into another form is called translation.

★ Types of Translator →

(1) Assembler → converts the high-level language into assembly language.

(2) Compiler → converts the high-level language into machine language.

process: High level lang → Assembly language → machine language

③ **Interpreter** : also converts the high-level language to machine language.

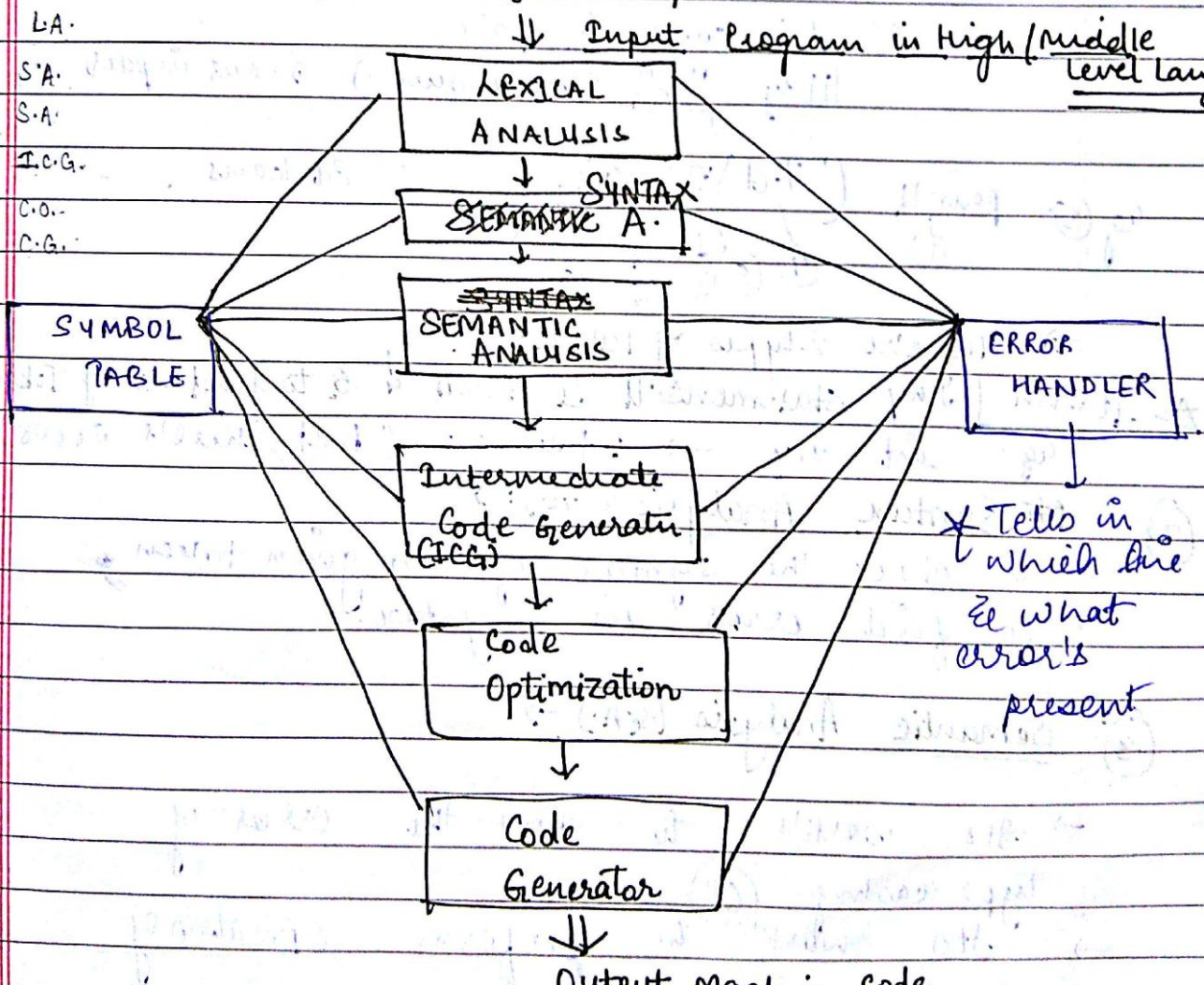
→ High-level language :- C++, human-~~like~~ language friendly.

→ C : middle level language

→ Low-level lang : In 0, 1. (Binary)

PHASES OF COMPILATION :-

- There are 6 phases of compilation ⇒



↗ connector → to token conversion.

① Lexical Analysis (L.A.): (Only token identification)
- In lexical Analysis errors are not identified/checked.
- It identifies the lexims or token in a program
→ converts high level language to tokens.

(Q.) (from) statement's given and no. of tokens're to be found?

(A) User-Defined And (B) Keywords.

eg①: int a, b; : There are 5 tokens.
① ② ③ ④ ⑤

eg②: printf ("Hello"); : There are 5 tokens ⇒ printf ("Hello") ;
① ② ③ ④ ⑤

~~(A) USER-DEFINED TOKENS :-~~

∵ ( ) come in pair
" " (double quotes) occur in pair.

eg③: printf ("%d\n", a); :- 8 tokens.
① ② ③ ④ ⑤ ⑥ ⑦ ⑧

→ There're 2 types of token.

★ [Exam] Imp: statements'll be given & Q to find no of Tokens?
eg: int x,y → 4 tokens. (But, there's error)

② ~~Sy~~ Syntax Analysis (S.A.) →
→ To check the syntax of a given token
→ To find error in Syntax

③ Semantic Analysis (S.A.) → work is to perform op of type casting
→ ~~Its work's to show the error of type casting.~~ (OR)
→ Its work's to perform operation of type casting

④ **Intermediate Code Generator (ICG)**

→ convert the given code into Assembly language.

⑤ **Code Optimization**:

→ It optimizes the code.

→ How To Optimize code?

```
i = 2 * 2;
i = 2 + 2;
```

→ Use the basic operator, in place of derived operator.

→ eg: Addition, subtraction, multiplication, etc. achieved by this.

```
for (i=1; i<10; i++)
{
    x = y+2;
    m = i+2;
}
```

* Code Optimization Techniques - Read HW.

⑥ **Code Generation**:

→ Final machine code is generated.

Error Handler:

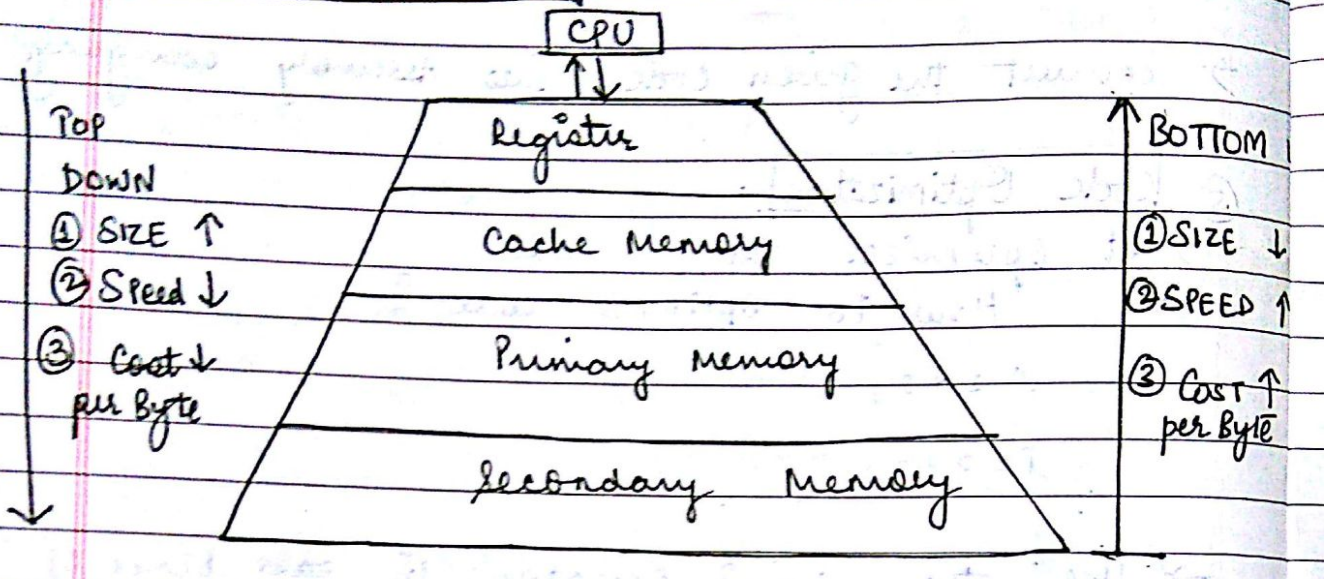
→ It'll do keyword mapping by symbol table

→ It'll find errors, in what line.

Memory hierarchy → CPU
↓
Registers
Cache memory
Primary memory
Secondary memory

R.
CM
PM
SM

MEMORY HIERARCHY =>



* Cost per Byte → least for hard disk.

1Tb = 1024 Gb

Cache → कोश

1st Aug '17
Tuesday

No. of tokens in following ?

printf ("Hello World");

→ 5 tokens present.
→ Hello World (string) considered as a single token.

Assembly level language → has symbols

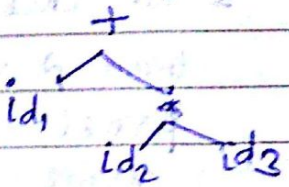
(Q.) Pg/. Consider difference blw compiler & interpreter.

- Convert Higher level language - to → identifiers Then it passes through different phases.

- Then, syntax analyser converts code in passe to tree (2nd layer).

Eg: a+b*c

$id_1 + id_2 * id_3$



- Concept of grammars. (read as terminals)

Grammar is a collection of 4 touples.

$G = \{ V, T, P, S \}$

P → Production.

S → Starting symbol.

T → Terminals

V → Variables

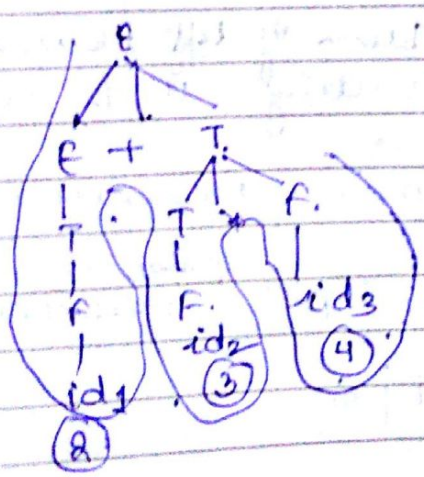
- In semantic analyser takes passe tree as input. checks if grammatical errors are present or not.

- E: variables.

$E \rightarrow E+T / T$. (Tokens).

$T \rightarrow T * F / F$

$F \rightarrow id$ (identifier)

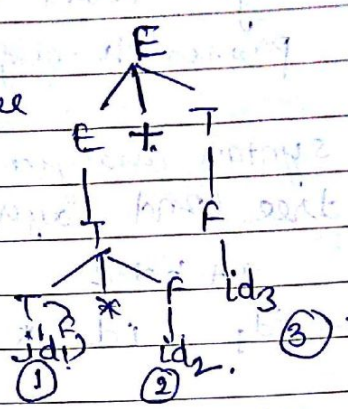


eg: 2+3*4.

More than 2 pass tree

↓

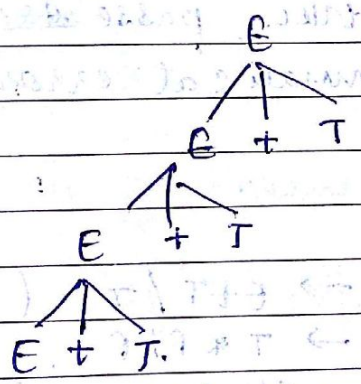
* No ambiguous (more than 2 result) problem should arise



1*2+3.

* Left Recursive grammar:
in which we can move to left recursively.

eg: $E \rightarrow E + T \mid T$



(Q) How To solve the problem of left Recursive Grammar? By converting it to right recursive grammar.

Ans: Conversion to right recursive.

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$ ← Epsilon is NULL value.

Q. $E' \rightarrow T \mid \epsilon$.

→ left most.

→ More than 1 LMD ~~&~~ or RMD, then it's called ambiguous problem; it should be unique.

* ICG:
Intermediate Mnemonics.

* SYMBOL TABLE: All values, stored.
It'll be predefined which all terminals're used:
- Also, all infos stored for high → low conversion.

a + b * c.
Register R1 = b * c.
Register R2 = a + R1.

(*) LOADER AND LINKER (LIKE IN TURBO C)
[Loader] → First division in modules takes place.

[Linker] → These modules're merged together.

* Generations =>.
1) magnetic drums, → 1st gen
2) transistor - 2nd gen.
3) IC (Integrated circuits) 3rd gen.
4) BLSI := 4th gen (large scale).
5th gen → Artificial intelligence.

23

2nd Aug '17

Wednesday.

AMBIGUOUS GRAMMAR =>

$E \rightarrow E+E$

$| E * E$

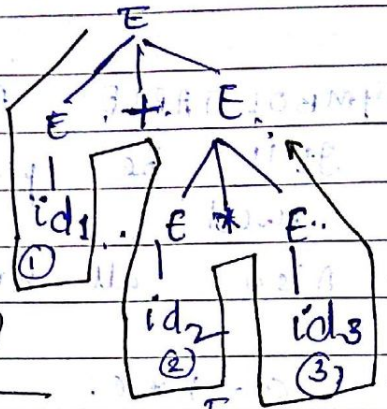
$| id$

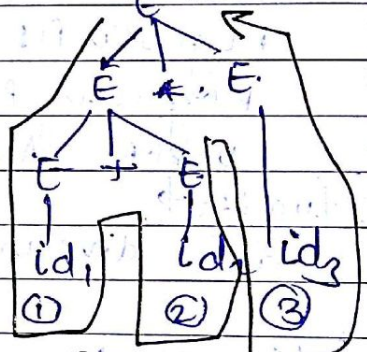
$id = id_1 + id_2 * id_3$

$= 1 + 2 * 3 = 7$ — (I)

Eg: $2 + (3*4) = 14$ — (I)

$= (1+2) * 3 = 9$ — (II)





→ If 2 or more parse tree can be made, then problem's ambiguous.

~~If an~~.

Eg: $(2+3) * 4$

$= 20$ — (II)

(Q.) How To make it unambiguous?

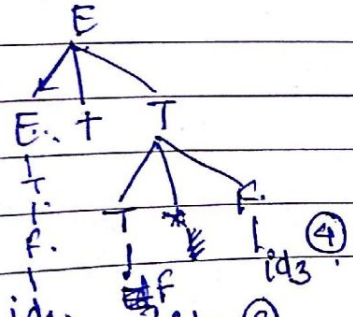
(expressions)

$E \rightarrow E+T \mid T$ (Terms)

$T \rightarrow T*f \mid f$ (Factors)

$f \rightarrow id$ (identifiers)

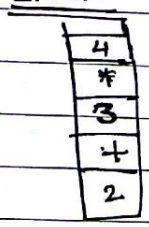
Parse Tree.



$2 + (3*4)$

$= 14.$

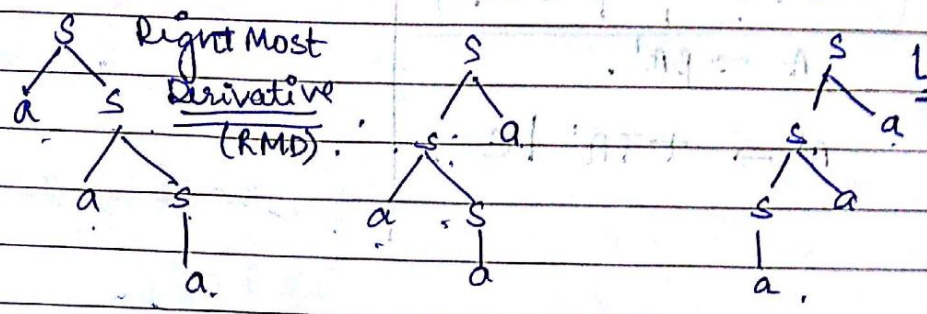
Stack



(Q.) S → as | sa | a. How many parse tree can be made?

w = aaa (Result should always be aaa).

Soln →

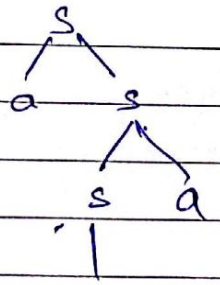


aaa aaa. aaa

→ Structure of parse tree are different, but final result / ans is same aaa.

→ This is called ambiguous grammar. ∴ But,

~~→ But~~ → above is unambiguous grammar ∴



AMBIGUOUS ⇒ a.

More than 1 LMD (left most derivative) & RMD (right most Derivative) for a given string, LMD, RMD represent different parse tree.

UNAMBIGUOUS →

There exists 1 LMD & 1 RMD.
Both LMD & RMD represent same parse tree result & different / unique parse tree structure.

classmate Date Page

$A \rightarrow A\alpha \mid \beta$.

$\alpha \rightarrow +T$, $\beta \rightarrow T$.

$A \rightarrow \beta A'$.

$A' \rightarrow +TA' \mid \epsilon$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$.

(Q1) Eliminate the left recursion.

(Q2) Design the parse tree for :-

$S \rightarrow asbs$.

$s \rightarrow bsas \mid \epsilon$.

$w = abab$.

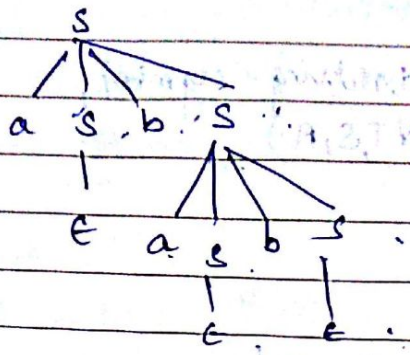
Soln: $T \longrightarrow fT'$

$T' \longrightarrow *fT' \mid \epsilon$

⇒ We converted left recursive to right recursive. & right recursive's acceptable. Parser is in syntax analyser.

→ Left recursive isn't acceptable, it has to be converted to right recursive

Soln 2:



~~E → E * E~~

E + E * E

id + E * E

id + id * E

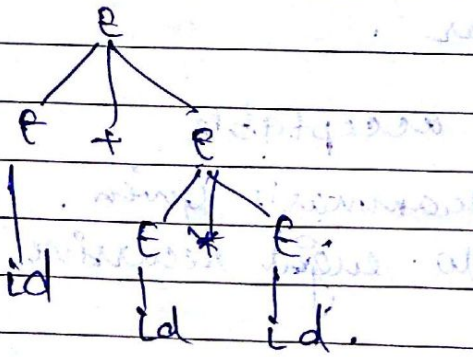
id + id * id

16th Aug'17
Wednesday

(Q.) Design parse tree for checking ambiguous/ unambiguous grammar.

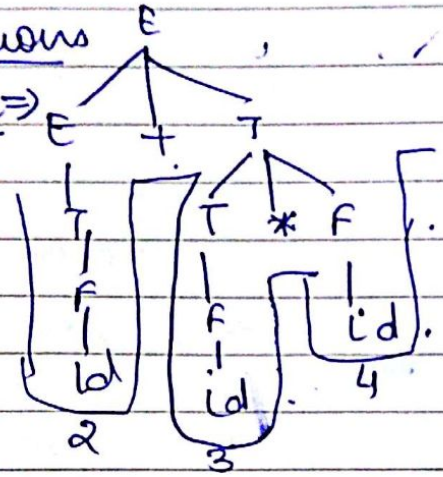
E → E + E | E * E | id .

x = id + id * id .



* Grammar means :
Variable, terminals, starting symbol, production.
(*, +) (V, T, S, A)

Unambiguous Grammar ⇒



E → E+T | T

T → T*F | F

F → id

2 + 3 * 4 = 14

V → (V + T)*

variable → Terminal

id > * > +

example =>

A → Aα | β

A → αA | β : is right recursive grammar.

→ Right Recursive grammar's acceptable.
→ So, if left recursive grammar's given. Then, convert it to right recursive.

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

$E \to TE'$

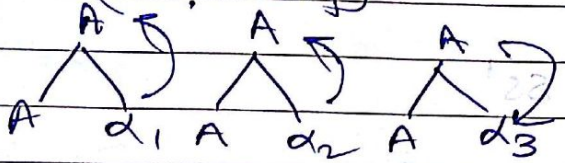
$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

#2 Types — Deterministic & Non-Deterministic grammar

(Back Tracking)



$A \to A\alpha_1 \mid A\alpha_3 \mid A\alpha_3$

Q. How to remove non-Deterministic ? to Deter.. Grammar

$A \to A\overset{\alpha_3}{A'}$

$A' \to \alpha_1 \mid \alpha_2 \mid \alpha_3$

Q. $S \to (L) \mid x$. This's all fine !

$L \to L(S) \mid (S)$. → How to Remove ?

α β

$L \to SL'$

$L \to ,SL' \mid \epsilon$

29

~~¬P ∧ (¬P ∨ Q)~~ ¬P ∧ (¬P ∨ Q)

(¬P) ∨ (¬P ∧ Q).

(¬P ∧ ¬P) ∨ (¬P ∧ Q)

¬P ∨ (¬P ∧ Q).

(Q.) Foll's exp is non-deterministic convert to deterministic?

$S \rightarrow \underbrace{iE+S}_{A} \mid \underbrace{iE+SeS}_{A} \mid a$

$E \rightarrow b$

Soln.

$A \rightarrow AA'$

$A' \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3$

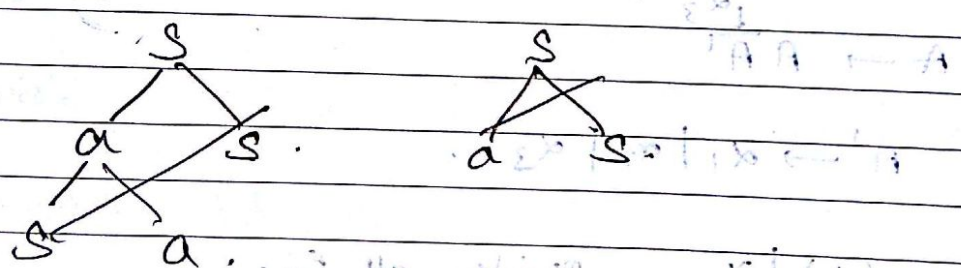
$S \rightarrow iE \rightarrow SS'$

$S' \rightarrow E \mid SiCS \mid a$

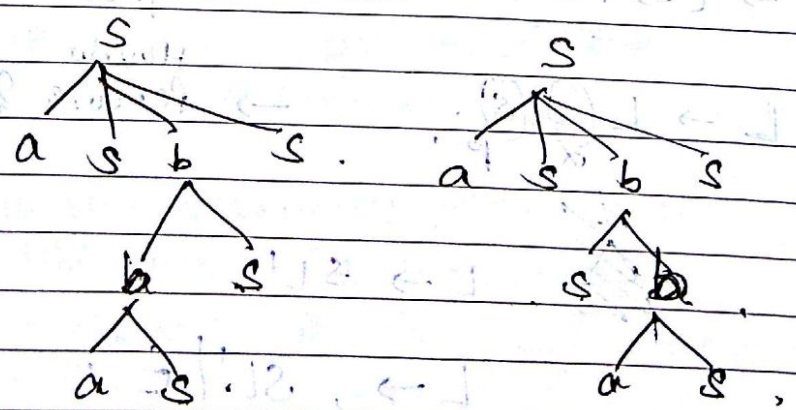
Q. ~~$S \rightarrow aS \mid Sa \mid \epsilon$~~ Determine?

$S \rightarrow asbs \mid bs \mid as \mid \epsilon$

$w = abab$



Soln →



30

22nd Aug'17

Tuesday

- Regex.

In JAVA → Regex is a class
Regular expression

Regex reg = new Regex (" . "); // giving memory to this object.
(will accept any 1 character)

matches is Objects

~~Regex~~

boolean tool. = reg.matches ("a");

escape character \

gmail: "^.+@.+\..+$";

(Q.) Find no. of Tokens.

eg: void main()
{
int a = 15;
printf("The number is %d", a);
/* ⟷ Multi line comment
}*/

Total no of Tokens = 19

30th Aug 17
Wednesday

Scope And Binding :

Scoping variables & binding < static / dynamic .

Static | Dynamic scoping

```
int a=20;
Void main()
{
  int a=30;
  printf("%d", a);
  swap(a,b);
}
void swap (int a, int b)
{
  int b = 10;
  printf("%d %d", a, b);
}
```

STATIC	DYNAMIC
20, 10	30, 10
global()	main()

* garbage Value : arbitary value,

(*) Static Scoping also called Lexical scoping.

STATIC : When free variable are resolved from global variables.
— Global variables by dyault init from 0.

DYNAMIC : when free variables are resolved from previous fn call.
— var in fn have garbage value by dyault.

Example: void main() {

one time ← 1. static int var = 3; → value 3's always retained..
init 2. printf("%d", var--);

1. ✗ not executed.
2. ✓ ∴ one time init.

3. if(var)
4. main();
}

Olp: ~~∞ loop~~ 3, 2,

* If static removed

void main()
{ ≡ int var = 8;
auto int var = 3; → every time var = 3.
by default printf("%d", var--);
if(var)
main()
}

∞ loop

* When, STATIC is not used:

(1) ⇒ Local variable 'var' will be declared every time & fⁿ is called. It'll be initialized & executed everytime when fⁿ is called.

* global variable - always static / one time initialized.

(2) int i=10; if globally declared then,
≡ static int i=10; always static.

```
int a, b, c = 0;        → even if we don't write it.
void printfun();
main()
{
    static int a = 1;
    printfun();
    a = a + 1;
    printfun();
    printf("%d %d", a, b);
}
void printfun()
{
    static int a = 2
    int b = 1;
    a += ++b;
    printf("%d %d", a, b);
}
```

it's ok.

O/p: 4, 2
4, 2
2, 0

auto ⇐ static

Register ⇐ static

a = 3.
a = 4, b = 2

a = 1 2
b = 1 2
a = a + 2
= 4.

Register ⇒ fast

a = 2

a = 1 2
b = 1 2
a = a + 2
= 4 → 5

a = 2, b = 2.

a = a + 3

free variable = 5.
means static scoping
Takes place
∴ p = 0

Compile-time binding → static Binding
Run-time binding (Late binding) → Dynamic Binding

After Replacement → O/p: 4, 2.
6, 2 Register'll update.
2, 0

Python, LISP → Dynamic Scoping.
Also Google.

* Example 2:

```
int a = 5; // global variable.
main()
{
  int a = 10;
  B();
}
B()
{
  int a = 20;
  C();
}
C()
{
  int a = 30;
  D();
}
D()
{
  int a = 40
  printf("%d", a); // 40
}
```

if not present (static scoping) → 5

Dynamic scoping // 40 → 30 (if a not defined in D)

* Example 3:

```
main()
{
1. static int i = 5; // only 1 time initialization.
2. if(--i)
```

i = 5

if(4) 3, 2, 1, 0.

3. main(); // recursive fn.
}
4. printf ("%d", i);
}
}

~~No output~~

0 0 0 0 ∵ main fn is called 4 times
∴ 4 zeroes printed.

Example 4:

~~main()~~
Var x, y; // global variable
~~main()~~
procedure P(var, int)
Begin
n = (n+2) | (n-3)
End
void main(){
Begin
x = 7, y = 8;
Q();
printf ("%d", x);
end.
}
}
procedure Q()
Begin
Var x, y.
x = 3, y = 1;
printf ("%d", x); P(Y);
End.

31st Aug '17
Thursday

Methods Of calling →

① call by value. (No change of actual paran)
② call by Reference. (Change Also of actual para).
③ call by name → replacement by ~~Macros~~. Macros.
④ callByResult

(Q.1) What's the written value of f(P,P) .?

p = 5 // (initialized)

```
int f (int &x, int c)
{
    c = c-1;
    if (c == 0)
        return 1;
    x = x+1;
    return f(x, c) * x;
}
```

x: 5 → 5 ; c: 5 → 5 → ... → 0

x = 6.

f(6, 4) * 6 .

~~f t~~

f = (6, 4) * 6.
f = (7, 3) * 7
f = (8, 2) * 8.
f = (9, 1) * 9.
return 1.
1 * 9.
= 9.

f(7, 3) * 7 . .

it all pts to address

Ans :- O/p: 4 times 9 occurs.
O/p: 9 * 9 * 9 * 9

recursion//

Storage classes - classes
- external
- global.

Storage Type	Scope / Visibility	Life Time
1) Local (garbage value)	within the function	within the fn (default: garbage)
2) global	entirety program.	default value = 0 entire program.
3) Static	within the program	Across the function call
4) Extern external. (Default value=0)	Across the file similar to global variable	
5) Constant garbage value = default & static variable — write once only — read many times.	applied to local, global	eg: const int *p = 1; write only once. eg: const int i = 0; i = i + 1; printf("%d\n", i); ↓ Compile-time error.

* external (extern) → used for linking purpose
  extern file1.c
  file2.c

* Variable: is an abstraction of memory cell.
* Name of variable: it's an identifier that refers to the variable. &

Variable created dynamically i new or malloc.

* Address:

— location in memory where variable is stored.

b

i = 1000X [Hexadecimal form representn of address].

— Address: sometimes also called as l-value

— whenever, L-value error occurs then, it means address is not assigned.

It's always LHS of assignement

Value: value is "content" of variable, also called as R-value.
It is RHS of assignement.

Lifetime:

Period of time variable exists.

Scope: portion of code where variables can be accessed.

Binding: is association b/w entity & attribute. (properties).
eg: employees. (real world objects)

→ Real world objects are called entity.
attribute: is property.

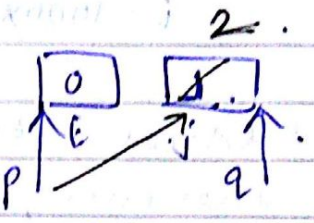
eg: student works on a project.
entity.

Its deadlnis → are properties.

Call By Name
Call By Reault } Next class.

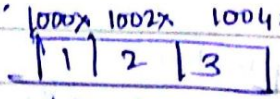
5th Sept '17 Tuesday

```
void f(int *p, int *q)
{
    p=q;
    *p=2;
}
int i=0, j=1;
int main()
{
f(&i, &j);
printf("%d %d \n", i, j);
}
```



p=q
(p copies adress of q)
*p=2

* Data structure's are abstract data-type.
(1) Primitive
(2) Non - Primitive.



a[i][j]
n m.
Base address = 5000

* Addrss a_{ij} = 5000 + [(i-1)m + j - 1] x 2

Adv of Abstract Data Types =>
- actual implementation is hidden in it.
- Encapsulation in C++.

Adv of Data abstraction —

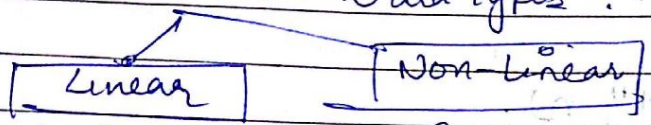
Data Types

(I) Primitive (II) Non-Primitives

- Abstract Data Type:

(I) Primitives — int, float, double.
- Its size depends on compiler size, processor.

(II) Non-Primitive Data Types:



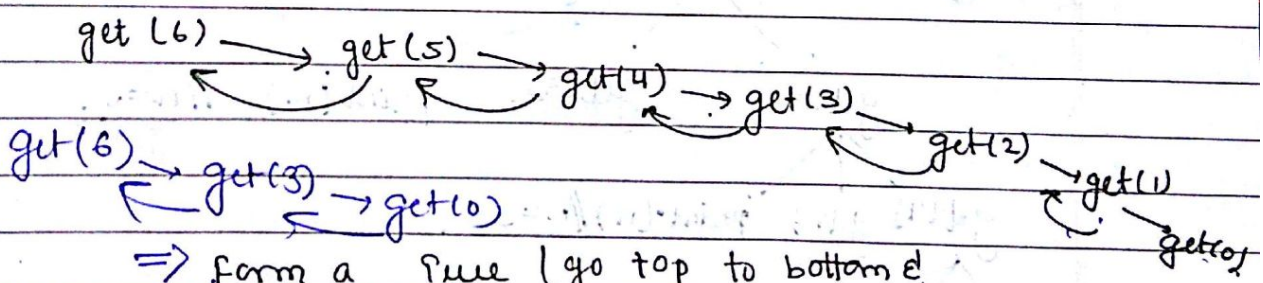
Stack, Queue, array, linkedlist — Tree, Graph.

(Q.) get (int n) // for n=6.

```
{ if(n<1) return;
  get(n-1);
  get(n-3);
}
```

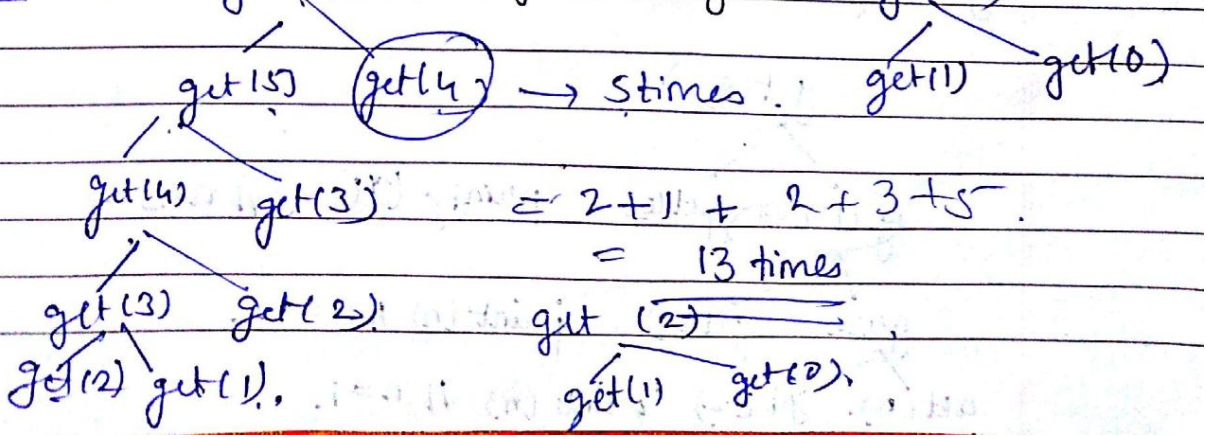
Without stack you can't tell how many times get fn is called.

Ans: ~~6~~ ~~8~~ 8. Ans: 13.



⇒ form a Tree (go top to bottom, left to right).

Soln: ✱



get(4) → 5 times.

= 2+1 + 2+3+5.

= 13 times

```
int f(int n){
    if(n<0)
        return 1;
    f(n-1);
    f(n-3);
    printf("%d", n);
}
```

in prev eg :- 25 times called.

f(2)

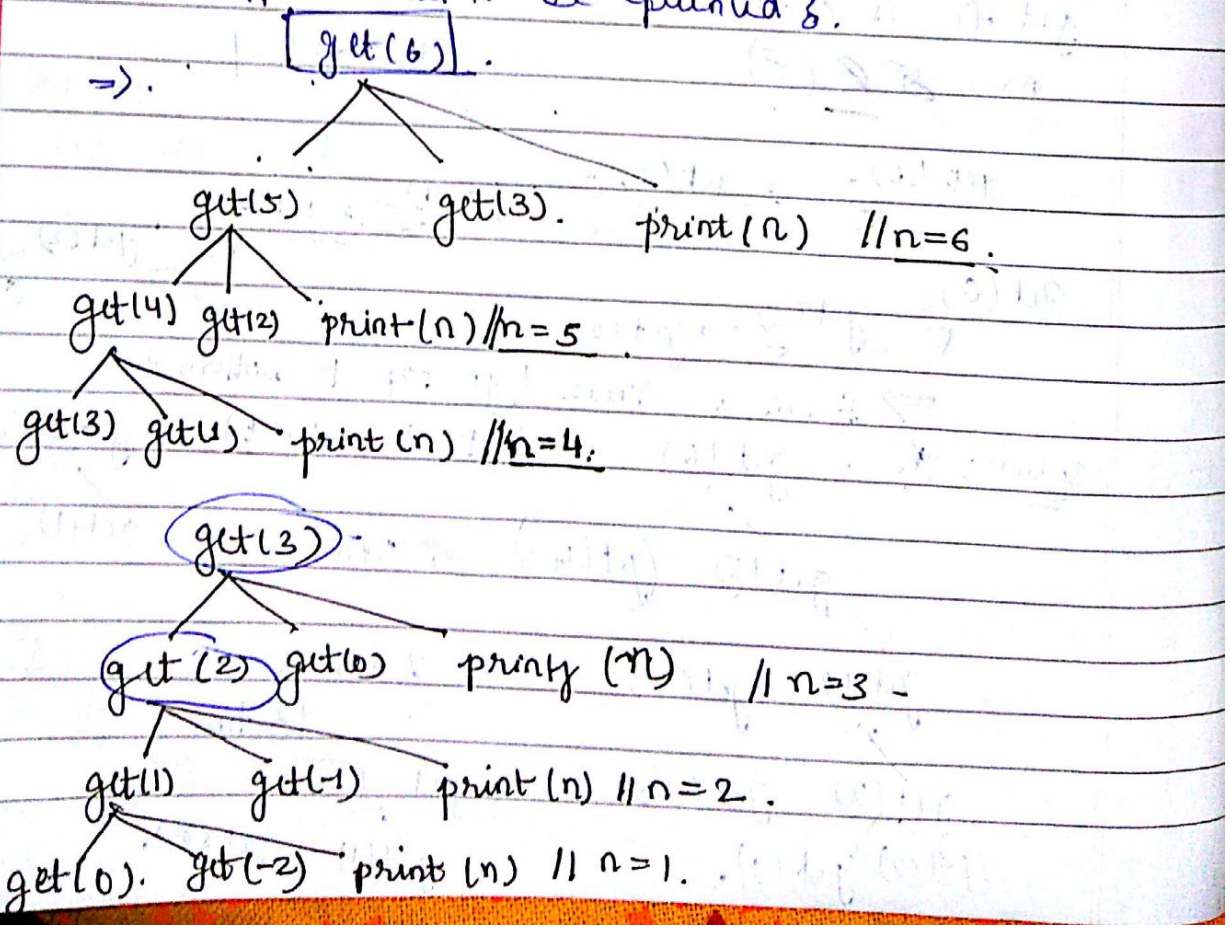
3 + 2 + 5 + - - - - f(1) f(0)

f(0) f(-1)

~~2 - 1 - 1 - 1~~

* Now in o/p what'll be printed ?

=>



(This's called dynamic programming).

1 2 3. 4 5.

↓ gives 2. ↓ gives 3rd value.

* We can reuse this value of get (2) when needed same value, stack also.

• Stack's used for recursion.

(*) **Abstract Data Type** (ADT): doesn't specify how the data type's implemented. It's defined in terms of interface. ^ type and set of operations on that type.

This implementation details are hidden from the user of ADT, and protected from the outside access, this concept is referred to as encapsulation.

→ Class that support ADT is C++, C#, JAVA, Python.

* **Eg:** while using sql. software or etc, user doesn't know internal implementation.

Only user view's seen.

3 Types of View
- → User : how's structure
- → Logical :
- → Physical : how data's stored in memory

* Conceptual Levelling.

eg: Bank account:
only a subpart can be seen.

Real life Example of ADT →

→ If we as a driver, drive the car but, ~~for a test~~ don't know the internal car mechanism, is example of abstract Data Types. (is hidden).

→ for a list of integers, for particular position in the list, insert a new integer at part pos, return true value if list is empty, reinitialization of the list.

– Return no. of intg currently in the list.

– Delete the integer at part pos in list.

5th Sept '17 Wednesday

Exception Handling

→ at run-time, exception error can come.

2 Types: (1) Logical Error
(2) Syntax Error.

int i=20, j=10, k=10.

result = i/(j-k) → // Division By 0. ∴ Exception.

∴ it's not executed ← result = i/(j+k) // Here run-Time error occurs.

* To handle this problem, we study exception Handling.

→ In Prev. Eg, we have arithematic, error.

* try, catch, throw. used.

eg: Even stack overflow is run-time error.

2 Types of errors
- Compile Time (syntax)
- Run Time Error // Exception Handling

```
try
{
  ---
  ---
}
catch (arithmeticException e)        ← type of error.
  System.out.println(e);
```

↳ means print line

(*) Java code:

```
class Test
{
  static void main (String args[]);
  {
    int a=20, b=10, c=10;
    int x = a/(b-c);        // ∞ value ∴ overflow stack
    System.out.println("x = " + x);
    int y = a/(b+c);
    println("y = " + y);
  }
}
// Run Time Arithematic Exception (Error)
```

Put this in try{}

```
try {
    x = a/(b-c);
    println ("x=", +x);
}
catch (arithmatic error e)
    {
        println (e);
    }
```

Example: Null Pointer Exception:

```
string p = Null;
println (string.length());
```

Eg: array out of bound array.

↳ when input elements > size given.

⇒ We can use multiple catches for throw.

Advantages -

① Exception handling: handles the abnormal error.

In c++ ⇒

```
int double (int a, int b){ //fn.
    if (b==0)
    { throw (b/0) (b);
    catch
        return (a/b);
}
```

- Exception: is an abnormal condition.
  → are run-time error, that program may encounter while, executing.

- try: Represents block of code that can throw an exception.

- catch: Represents the block of code that is executed when a particular exception is thrown.
  - Atleast 1 catch present, we can use multiple catches.

- Throw: is used to throw an exception.

Object
↓
Throwable
↓
Exception | Error.

Exception:
- ClassNotFound — When we've not included package (by import) in JAVA(eg)
- IllegalAccessException.
- InterruptedException
- RunTimeException
  - IllegalArgumentException
    - ArrayIndexOutOfBound
    - StringIndexOutOfBound.
  - IndexOutOfBoundException
  - NullPointerException.

Error.

* Exception Handling's a method to found error. found 2/0, SQL.

→ Adv. of E.H. is to maintain normal flow of application.

→ eg: String s = "abc";

int i = Integer.parseInt(s);

↓

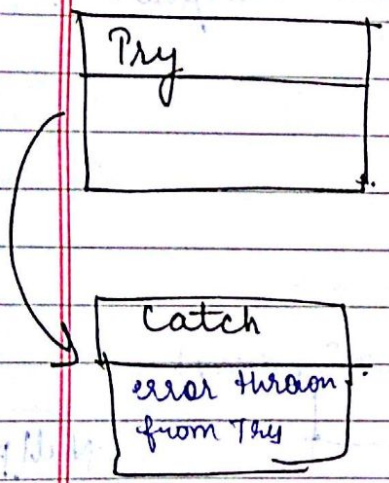
Number format exception.

(∵ string passed in integer)

eg: int a[] = new int[5];

a[10] = 50;

→ Array Index Out of Bound exception.



Creator of exception

statement that handles exception

* Abstract class: complete info. not given.
→ Pure virtual fn in JAVA

eg: class A
{ public:
virtual void swap() = 0;
};

* Object of ~~this~~ abstract class can't be

int
float.

created (never), (but called from Base class).

```
void main()
{
P(x);
}
Class Q : public P
    public : swap()
    {
    cout<<" Inside child class\n";
    }
```

When void main()
{ Q x
x.swap();
Only then it will be printed.

7th Sept'17

Thursday. char → 1 Byte (= 8 bits) If (= 8 Bytes) → means $2^8 = 256$ elements can be stored

* signed char range: -128 to +127.
* 2's complement: used to represents (-ve) numbers.

0 to 255. (unsigned char Range).

$2^{10} = 1024$
$2^{15} = 1024 \times 32$
2048
30720
32768

$-(2^{15}-1)$ to $(2^{15}-1)$.
-32768 to $+32768$

double → 8 Byte
long double → 10 Byte

S	exp	mantissa
←	16	→

IEEE 754 Bit Repr.

(2) * User Defined Data Type → typedef float fe;

(3) Derived Data Type → array, string, structure, struct, union

* Lab Assignment →

Ans → Q.1)

```
void main()
{
    unsigned int i, j;
    i=1;
    j=1;
    i++;
}
printf("Max. value of int j = %u", j);
```

O/P = ?

* For JAVA & C ?? : Diff. of Data Type?

↓

Byte, boolean → extra data types + Range may also change.

Q.2) In Q.① find no. of tokens?

" ____ "

↳ Counted as single token in string

| void | main | (|) |

{

| unsigned | int | i | , | j | ; |

| i | = | 1 | ; |

↳ Tokens

j = i;

i++;

}

* E → variables (on LHS present.

id → identifier // Terminal.

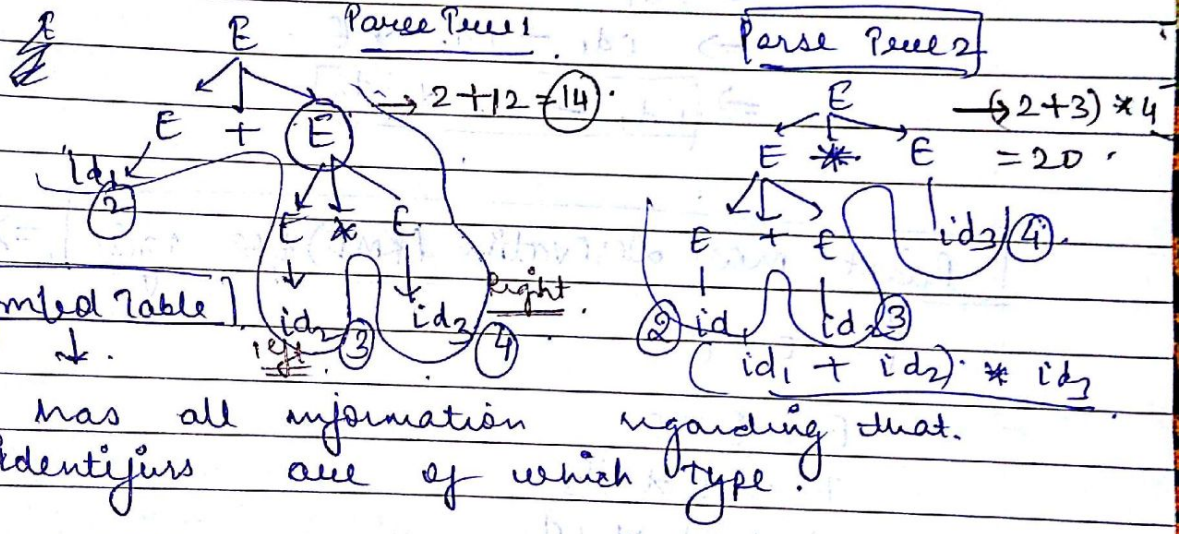
Q.3. E → E+E | E*E | id . : This is our grammar?

→ Here both + & * have same precedence ∴ we get ambiguity.

Generation: $id_1 + id_2 * id_3$.
2 3 4

We'll generate it by Parse Trees.

This grammar's called an ambiguous grammar. (∃ > 2 parse Trees).



Symbol Table ↓.

has all information regarding that. Identifiers are of which type.

In Lexical Analyser (1st layer) → it removes the white-spaces, headers & converts to tokens.

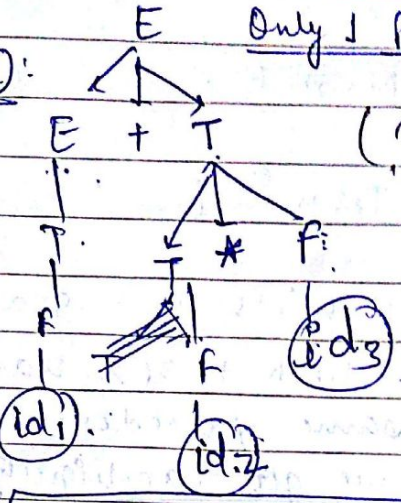
(*) Inorder, Preorder, Post order Traversals of Trees

(*) Problem'll be resolved - if we predefine that '*' has more precedence than '+'.

Eg :- E → E+T | T.
T → T * F | F
F → id .

Only 1 Parse Tree can be made.

Example (1):



(This is left recursive grammar)

$\Rightarrow 2 + (3 * 4)$.

$2 + (12)$

$= 14$.

Left most Derivate (LMD) for Eg (1) $\Rightarrow$

$E \rightarrow E * E$

$\rightarrow E + E * E$

$\rightarrow id_1 + E * E$

$\Rightarrow id_1 + id_2 * E$

$\Rightarrow id_1 + id_2 * id_3$

Right most Derivative (RMD) for eg (1) $\Rightarrow$

$E \rightarrow E + E$

$E + E * E$

$E + E * id_3$

$E + id_2 * id_3$

$\Rightarrow id_1 + id_2 * id_3$

* (Make it Right Recursive) :- grammar. How ?

$A \rightarrow A\alpha \mid \beta$

Example :- $A \rightarrow A\alpha \mid \beta$ Problem of left recursive grammar removed here.

$A \rightarrow \beta A'$.

$A' \rightarrow \alpha A' \mid \epsilon$.

$A \rightarrow A\alpha \mid \beta$.

$A \rightarrow \beta A'$.

*. Since, only right recursive grammar's acceptable.
∴ we've to remove left recursion.

$E \rightarrow E+E \mid E*E \mid id$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$

α mean :-

$E \rightarrow E(+T) \mid T$

α

anything after E
Or 1st element.

α can be anything :
Variable + Terminal.

$\alpha = (V+T)^*$.

TE'

$E \rightarrow +TE' \mid \epsilon$.

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$.

q: a^* means ∞, (combination of characters) :
$a^* = \{\epsilon, a, aa, aaa, \ldots\}$.

q: $(a+b)^* = \{\epsilon, ab, ba, a, b, \ldots, aabb\}$
many combinations can be formed infinitely.
If all acceptable Then, complete string's available in the machine.

☆ Turing machine (Theory of Comput) → most powerful :

$a^n b^n$. → means no. of a's & b's are equal.
↳ ∴ in this aab string not acceptable in this
↳* It's called context free - grammar, stack's also used.
Push Down Automator (PDA).

53

(*) P : P is prime number.

1, 2, 3, 5, 7, - - - prime nos
↳ not equal interval.

* not acceptable

Syntax & Semantics → PDF Book