- First Programming Language: FORTRAN
- First Programmar: Augusta Lovelace

- Types Of Programming Language:

Object - Oriented Programming → C++, Java, C#
Procedural → C, C++
Functional → Haskel, Scala, Excel
Scripting → Ruby, Python, Shell, Perl, JS
logical → Prolog, Datalog

Oldest:
FORTRAN → Scientific Purpose
COBOL → Business Oriented

- Ratio Of Usage Of Languages Globally

Python - 14%.
C - 15%
Java - 18%
C++ / C# - 12%.
Fortran - 0.96%.
Cobol - 0.76%

PROGRAMMING LANGUAGE:

Grammatical rules or instructions which help in communication b/w human & machine.

Imperative vs Declarative

- Steps / From scratch
- Results by executing each steps

- Focus on Result
- MYSQL is the example

Language Translators:

Efficient → Time
→ Space

Compiler:

- Language Translator
- At a time full-translate (Execute)
- 2 Steps → Object code
  → .exe file
- COBOL, C, C++, C#
- Faster than Interpreter

Interpreter:

- Interprete line by line (Interprete
- 1 step → line by line scripts
- Python, PHP, JS, Ruby

Hybrid:

- Interpreted & Compiled

COMPILER (Compilation Process)

Source Code → Compile → Executable code

Steps Of Code Compilation:

1 - Lexical Analysis
2 - Syntatic Analysis
3 - Semantic Analysis

-> LEXICAL ANALYSIS :

+ Divide the source code in tokens (lexemes)
+ Ignores whitespaces
+ Identifies -> Collection of characters
+ ASCII can be counted as token.
+ Identifier / variable -> main, func e.t.c.
+ First phase of compiler construction.

-> SYNTATIC ANALYSIS:

+ Grammar of Programming Language.
+ Check Lexemes

-> SEMANTIC ANALYSIS:

+ Overall Meaning of Source code
+ Parse Tree.
+ Either we are achieving overall meaning of source code or not. (checks ^)

Preprocessors:

Linking:
Embeds pre-defined modules / libraries to the source code. The .exe file does not depends on any dependencies.
pre-defined modules -> #include

Factors to Assess a Programming Language:

1 - Readability
2 - Cost
3 - Writeability
4 - Portability (Reusability)
5 - Reliability

Models Of Computation:

- Data Calculation
- Memory Management
- To compute / estimate a problem theoretically
  - ↳ numbers
- To compute / estimate → need maths function.

"A model which may take input and suggests some output".

1 - Sequential Model:

- Step by step / Sequentially
- Example:
  - FSM (Sequentially but no memory)
  - PDA (Sequentially, memory stack)
  - TM (Sequentially, memory RAM)

2- Functional Model:

- Structured Model and hierarchy model (Representation)
- Lambda
- Abstract Rewriting system
- Combinational Logic

↓ used in

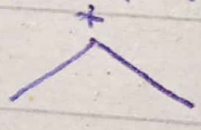
e.g: Predicate Logic

$\delta \rightarrow \vee$ } Relational

$\vee \rightarrow \cup$ } Mechanism

e.g Tree representation of mathematical expression

$2 + 3 * 4 - 1$



2 $\quad$ $3 * 4$

Keywords:

Mathematical formulas

Tree representation/structure

Relational Mechanism

3 - CONCURRENT MODEL:

- Example -> ① Email Mechanism
  - Actor Initiates Task
  - Individual actions

② KPN (Kahn Process Network)

General Attributes for a Good Programming Language:

1- Clarity, Simplicity, Unity (follows)

- Algorithm completely fits the source code.
- Instruction / postulate of algorithm impacts the source code.

2- Orthogonality -> Perpendicular
--> | | not strikes

- No Effects if we combine the two functions.

- Premitives / Constraints

↓

Data Structures

+ Python is not completely orthogonal

List -> set (we will lose the data)

list = [1, 1, 2, 3, 4, 4] => set(list)

set = (1, 2, 3, 4)

- Degree of Freedom will increase in orthogonal languages.

- Non-orthogonal -> C-language / C++
- Partially orthogonal -> Python Java.
- Completely orthogonal -> Scala, Algol, Lisp

Also known as Quasi-Orthogonal

H.w (Examples of these. How they are completely orthogonal)

3- Support For Abstraction

- Data hiding / Encapsulation / Inheritance

4- Programming Environment

5- Portability Cost

6- Program Verification

Examples Of Orthogonality:

- Algol 68:

1- 3*(a:=c+4)+2;

2- X:= IF a< 2 THEN 3.2 ELSE 5.2 FI;

3- IF a<2 x THEN y FI:=3.5;

One can execute a print statement like:

print (IF a<2 THEN

"a is less than two"

ELSE-IF a=2 THEN

"a equals two"

ELSE

"a is greater than two"

FI)

· SCALA :

Combines Object-Oriented & Functional programming.

We can use functions just like all other objects as local variables, fields or parameters to other functions. In doing so, we experience no side effects.

For instance, passing a function as an argument doesn't invoke it.

Input parameter name → callback input types

```
def sayHello(callback:()=>Unit){
  callback()
}
```

callback's return type

↓

Function that is passed in is invoked here.

Currying -> we have more than one variable in function

LAMBDA CALCULUS:

Model of computation which expres an expression in abstract form.

1) Variables — x, y, z

2) Abstraction — $(\lambda x . y)$ (x: variable/argument, y: value)

$f(x) = y$

3) Application — $(M\ N)$

Expressions:

a) $\lambda x . x \Rightarrow f(x) = x$

b) $\lambda x (\lambda y . x) \Rightarrow f(x, y) = x$ (Currying)

c) $(\lambda x . (\lambda y . x))\ z \Rightarrow (\lambda y . z)$

($\lambda x . (\lambda y . x)$: Function/head; z: value)

d) $(\lambda x.(\lambda y.x\ y))(\lambda z.z)(z)$

$\Rightarrow (\lambda y.(\lambda z.z)\ y)(z)$

$\Rightarrow (\lambda z.z)(z)$

$\Rightarrow z$

e) $(\lambda x.x)(\lambda y.y)(\lambda z.z)$

$\Rightarrow (\lambda y.y)(\lambda z.z)$

$\Rightarrow \lambda z.z$

f) $(\lambda x.\lambda y.y)\ a\ b$

x ↓ y ↓

$\lambda x.\lambda y.y \rightarrow f(x,y)$

$\Rightarrow b$

Church Encoding:

$\lambda f.\lambda x.x \longrightarrow$ Zero

$\lambda f.\lambda x.fx \longrightarrow$ one

inner expression

$\Rightarrow [\lambda n.\lambda f.\lambda x.f(nfx)]$

$[\lambda n.\lambda f.\lambda x.f(nfx)](\lambda f.\lambda x.x)$

$\Rightarrow \lambda f.\lambda x.f((\lambda f.\lambda x.x)fx)$

$\Rightarrow \lambda f.\lambda x.f((\lambda x.x)x)$

$\Rightarrow \lambda f.\lambda x.fx$

Expression used to increment or decrement a quantity.

$+yy \Rightarrow y+y$

1) $((\lambda x.((\lambda y.(*2y))(+xy)))y)$

$\Rightarrow (\lambda y.(*2y)(+yy))$

$\Rightarrow \lambda y.(*2y)(2y)$

$\Rightarrow *2(2y)$

$\Rightarrow 4y$

2) $((\lambda x.(\lambda y.+xy)5)((\lambda y.-y3)7))$

$\Rightarrow (\lambda y.+7y)5)(\lambda y.-y3)$

$\Rightarrow +7(5)(\lambda y.-y3)$

$\Rightarrow 12\ (\lambda y.-y3)$

$\Rightarrow -(12)(3)$

$\Rightarrow 9$

Simplest Expression

$[(\lambda x.xx)(\lambda y.yy)]$

↳ repeat hota jaiga, koi meaningful expression nhi aiga

$(\lambda y.yy)(\lambda y.yy)$

$(\lambda y.yy)(\lambda y.yy)$

↳ repeatition

WHAT IS SYNTAX ?

Structure of
- -> expressions
- -> statements
- -> constraints
- -> premitivies e.t.c

WHAT IS SEMANTIC?

Meaning of Structure.

* Semantic is more difficult to monitor than syntax.

/ * (*)* ~~(**)~~

/ * (non*)* (*)*

Regular Expression:

Language Identification / Recognizer / Validation

*0 => Zero or more instances

+0 => one " "

+a.1 => multiple instances of a and one instance of 1.

Database of collection of instances of source code. Checks whether the characters, operators e.t.c are part of language or not.

H.w => Validation

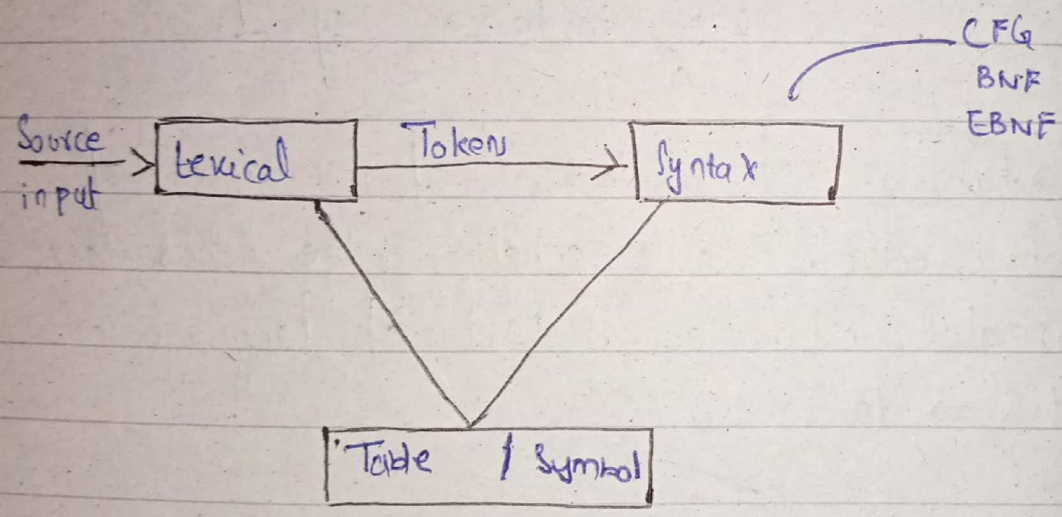
Ramzan Vacations H.w

Develop RE identifer in C.

Validation with JFLAP.

LEXICAL ANALYSIS / ANALYZER (RE):

- Regular Expression
- Grammars:
  - CFG
  - BNF
  - EBNF



RE --> Alphabet --> String --> Language

- Divide source code in token for syntax analyzer.
- Store them in 3 informations
  1 - Identifier
  2 - ID
  3 - include => value

How To Make Token Tables?

① Fast Lexical Analysis Tools -> Fastest but least efficient.

② C - Language

③ Machine Language. (Most Accurate results can be obtained)

character followed by
" " "
" "

Count = 123

Count (small)

recognize -> number followed by number followed by number

count k token ki value

value = Count

identifier = Count

rule = =

Const_integer =

Count (capital) -> either capital or small followed by either capital or small

Letter => [a-z A-z _]

Letter => [a-z A-z _]. — dot allows single instance

Letter => [a-z A-z _]+ — atleast one or more

Letter => [a-z A-z _]* — Zero or more

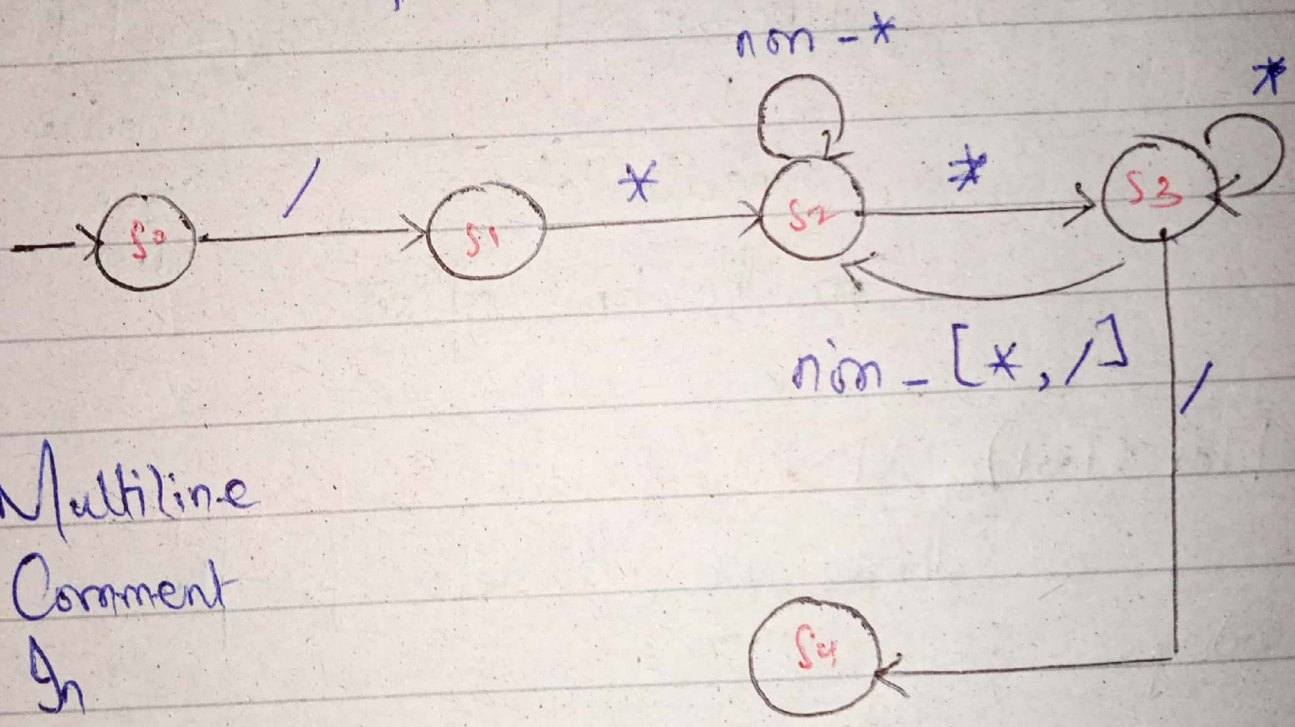
digit = 0|1|2|3|4 — Allow only one instance from 0 to 4

1005 ->
10005 ->
105 ->
} -> [0-5]+

① Letter = [a-z A-Z _]*

② Digit = [0-9]*

③ Identifier =



Multiline Comment

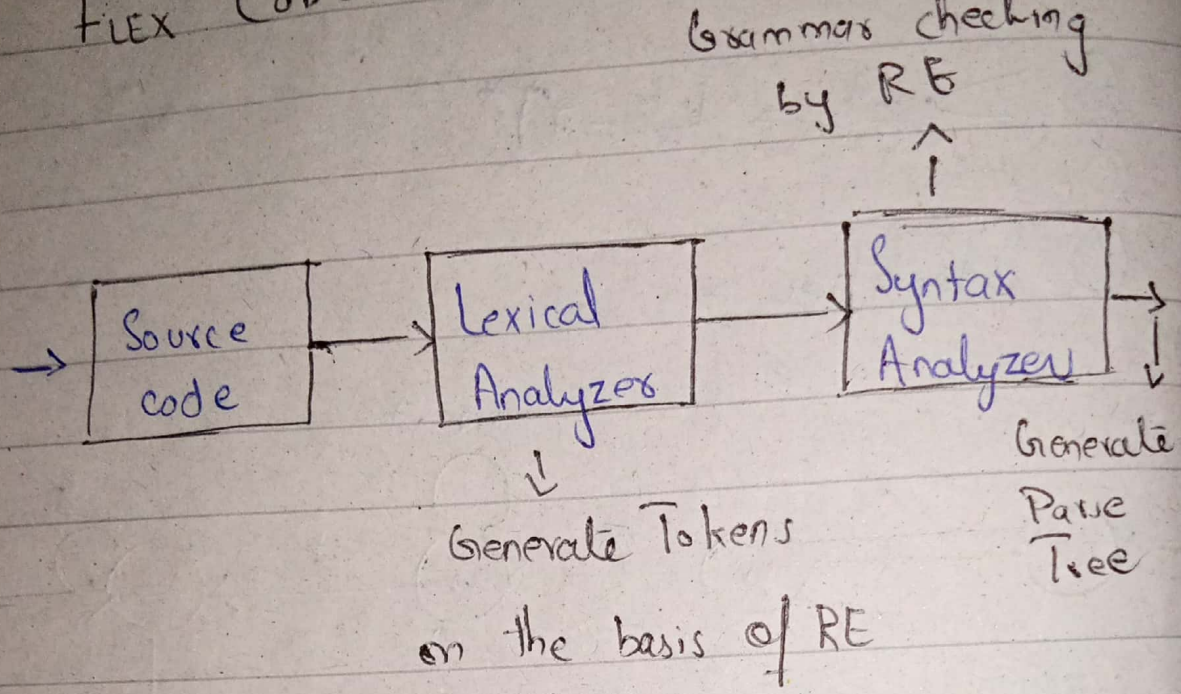
/ a

* b

non * c

non / d

FLEX CODE



Flex (Tool): Developed in C-language

There are 3 rules for flex coding:

① Definition — % (opening) Header files % (closing) with single %
Var initialization

② Rules / Pattern — %%

%%

③ UDS (User Defined section)
main( ) functions

```
main() {
    yylex();
}
```

calls → for flex coding to start work.

single instance

RE

Action

```
%%
[\n] -> {Printf("Hello, After Ramdan")}
.    -> {Printf("Wake-up");}
```

Count Alphabets Flex Coding:

If RE matches with input string, perform/ action fire

* YYtext is a variable used by YYlex to take input from user.

```
%{
int count = 0;
%}
%%
[A-Z] -> {Printf("Detected");
          count++;}
.     -> {Printf("Not Detected");}

[\n] -> {return 0;}
%%
main(){
   yylex();
   Printf("The value of count", count);
}
```

Count Vowels Hex Coding:

```
%{
count = 0
vowels = 0
%}
%%
[A a E e I i O o U u] ->
        { printf ("Vowel Detected")}
            vowel++;
[A-Z a-z] -> {printf ("Alphabet
Detected");
  count++; }

. -> {printf ("Not Detected");}

[\n] -> {return 0;}
%%
main( ){
    yylex( );
    printf (" The value of vowels: ", vowels);
    printf (" The value of consonants: ",
     count - vowels);
}
```

cmd 2,- -> 2 or more

Count Specified Strings Flex Coding:

{aa} [{aa}] or [a{2}] or [a{2,-3}]

① {aa} {ccccc} {bbb}

```
%.
    counta = 0
    countb = 0
    countc = 0
%.
%.%.
    [a{2}] -> {printf("aa Detected");
                counta++;}
    [c{5}] -> {printf("ccccc Detected");
                countc++;}
    [b{3}] -> {printf("bbb Detected");
                countb++;}
    .  -> {printf("Nothing Detected");
    [\n] -> {return 0;}
%.%
main() {
    YYlex();
    printf("The value of aa: ", counta);
    printf("The value of bbb: ", countb);
```

```
    printf("The value of c=%c", count);
}
```

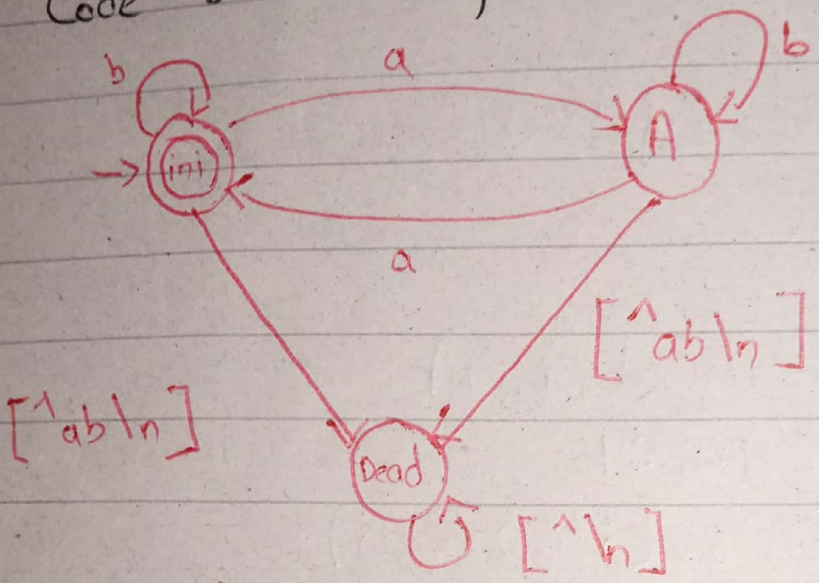
① Check whether user input numbers or characters.
② input in string → convert in numbers

Check whether even or odd Flex coding:

② Even or odd

```
%{ bool isEven = false
%}
%%
[0-9]+  {i = atoi(yy)
          i = yy
          if (i % 2 == 0){
              printf("Even"); isEven = true;
          } else{
              printf("odd"); isEven = false
          }
%%
main(){
  YYlex();
    if (isEven){
        printf("Even");
    }
    else{
        printf("Odd");
    }
}
```

Code DFA in flex



%{ S A Dead (states as label)

%% Rule section

<initial>a BEGIN A;
<initial>b BEGIN <initial>;
<initial>[^ab\n] BEGIN Dead;
<initial>\n BEGIN <initial>; {printf("Accepted");}

Agr b input kra phr 'enter (\n) input krenge jb se Accept ka msg aiga.

See directory -> dir

<A> a BEGIN <initial>;
<A> b BEGIN <A>;
<A> [^ab\n] BEGIN Dead;
<A> \n BEGIN <initial>; {Printf ("NA"); }

<Dead> [^\n] BEGIN <Dead>;
<Dead> \n BEGIN <initial>; {Printf ("NA"); }

flex hello.l -> extension (command)
|
v
file name

If successful compilation:
gives file lex.yy.c
extension

else:
error

Compiling file
gcc lex.yy.c

If success:
gives a.exe

else:

error

File Execution:

a.exe

↳ enter

Syntax Analysis:

Syntax: Structure of → Statements
→ Expressions
→ forms

Semantic: Overall meaning of structure of source code.

Syntax is determined with the help of Grammars:

- CFG → 4 tuple devise
- CSG
- BNF
- EBNF

CFG (4 tuple devise)

G (T, P, V, S)

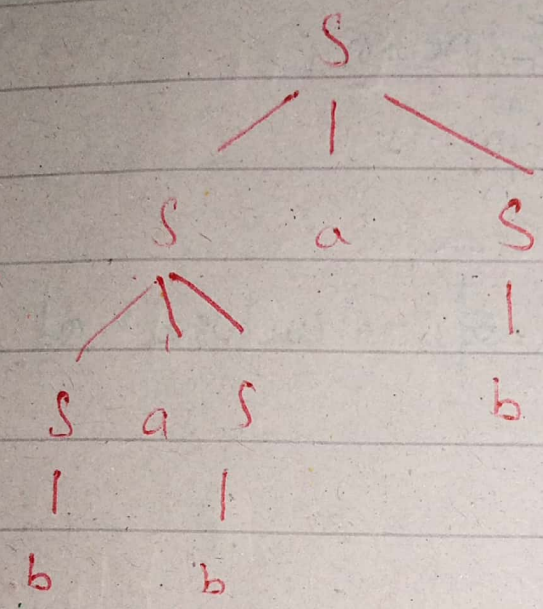
Chomskey Hierarchy: Type - 0
Type - 1
Type - 2
Type - 3

Recursive

$S \rightarrow SaS$

$S \rightarrow b$

{ b, bab, bababab }

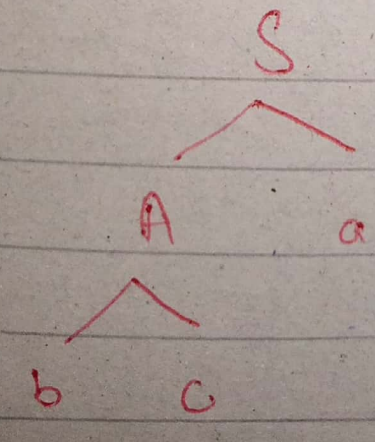


Non - Recursive

$S \rightarrow Aa$

$A \rightarrow b|c$

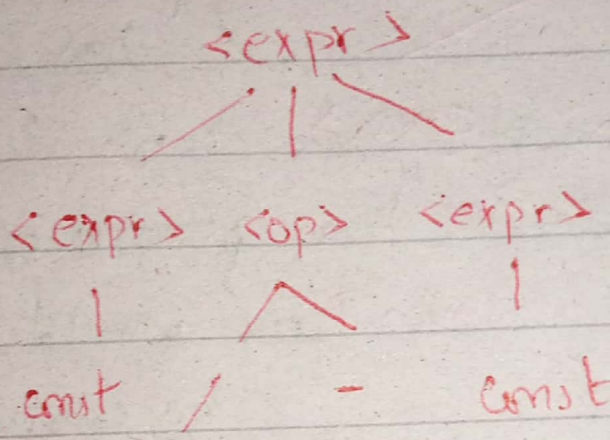
{ ba, ca }

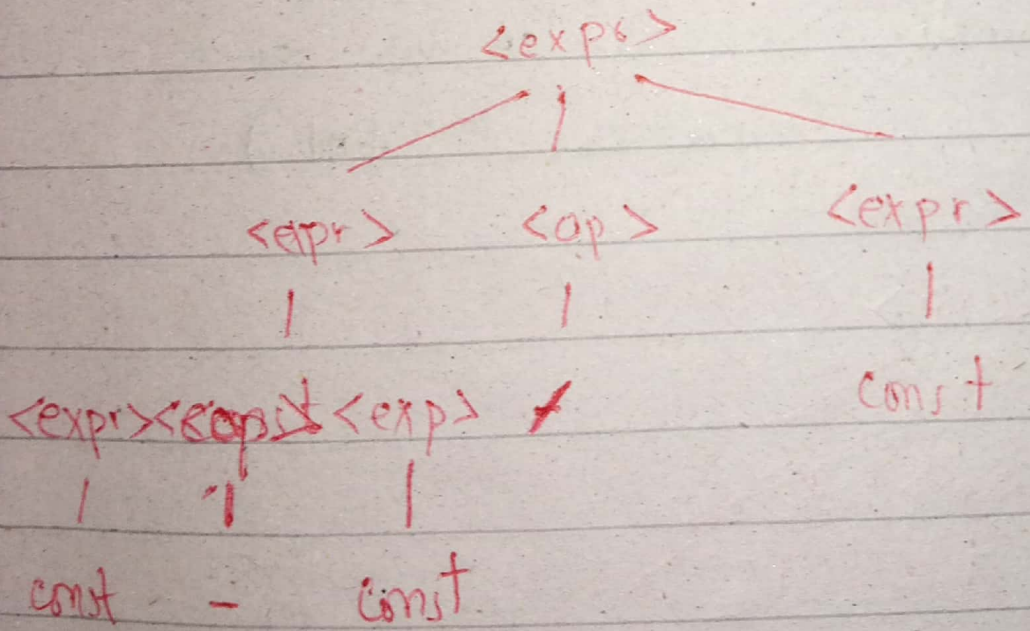


<exps> → <exps> <op> <expr> | const

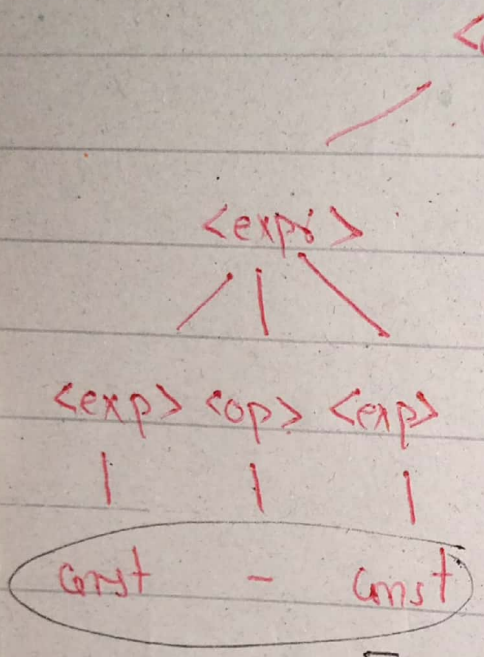
<op> → / | -

=> const - const / const





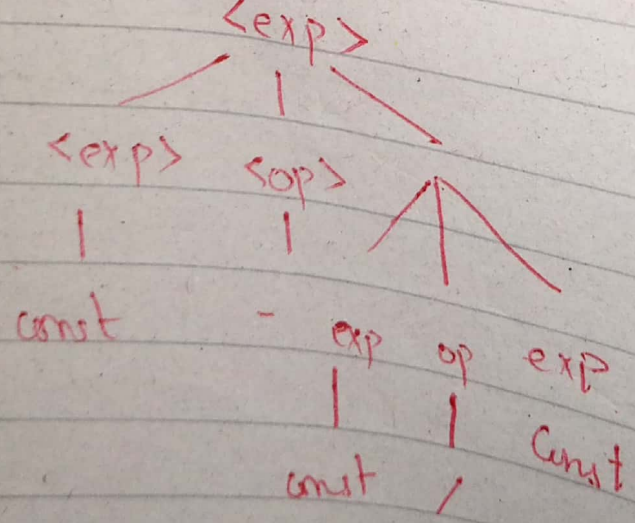
L.R : (left recursive)



Terminals.

Wrong bcz has high precedence / should be computed first (Terminals are read first)

R.R :



If ambiguity is found in grammar we cannot go into semantic phase.

To remove ambiguity we must know a rule. we will take

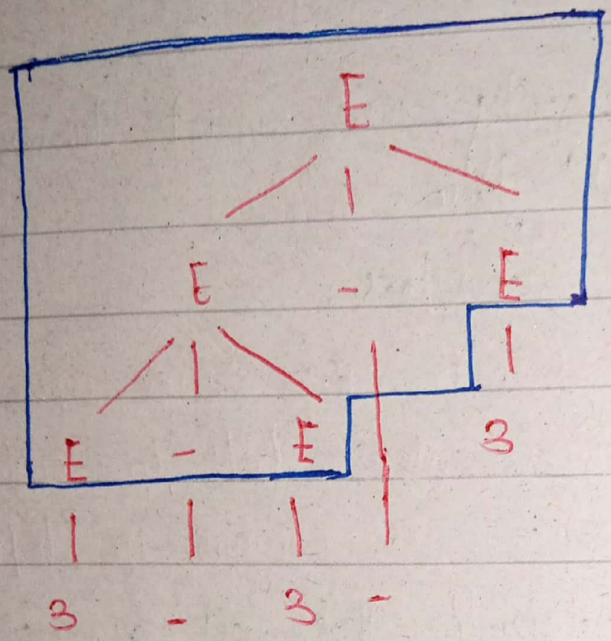
- precedence of operators.
- Associativity (same precedence operator either solve left to right or Right to left)

- It is not necessary that all grammars become unambiguous.

E -> E - E | id

-> 3 - 3 - 3

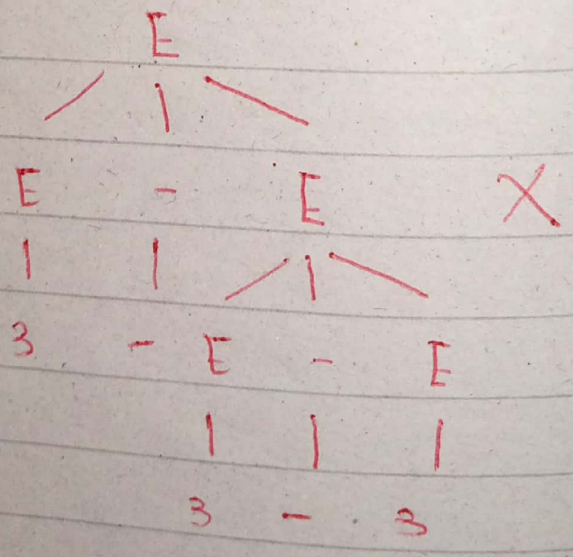
L.R



a/c to L -> R

((3-3)-3) = -3 (Results are different)

R.R



X

(3-(3-3)) = 3

Same precedence L→R

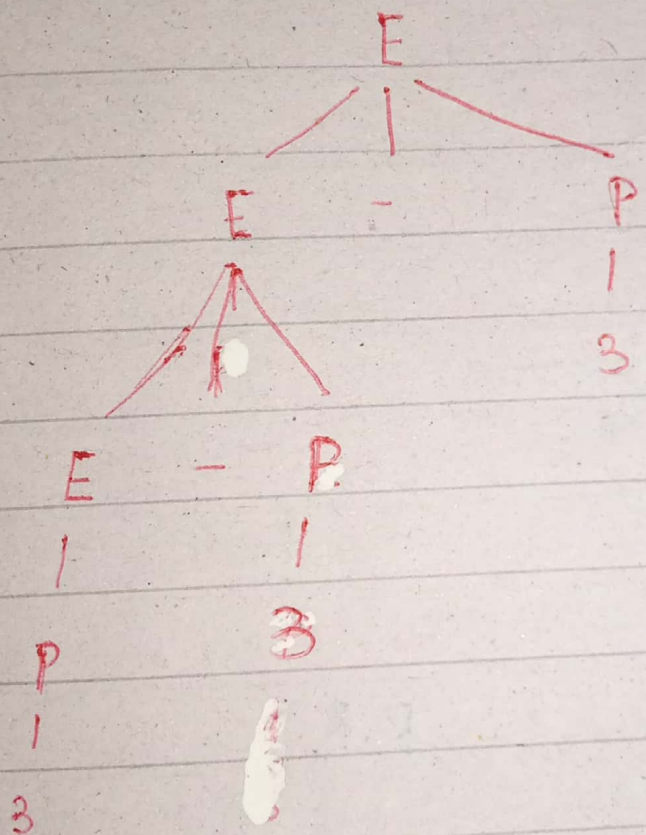
Removing Ambiguity from grammar

E -> E - E | id

E → E - P | P

P → id.

Sirf L-R banega



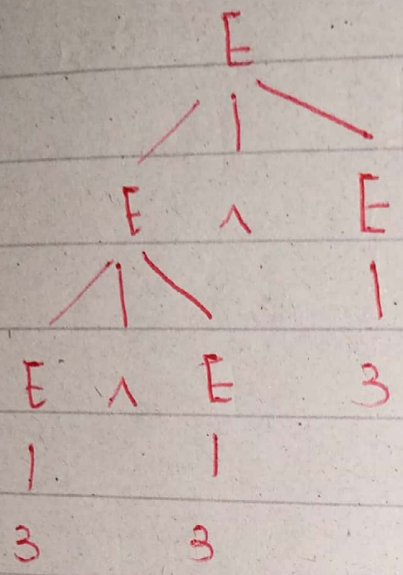
Removed Ambiguity by making grammar L-R.

$E \rightarrow E \wedge E \mid id$

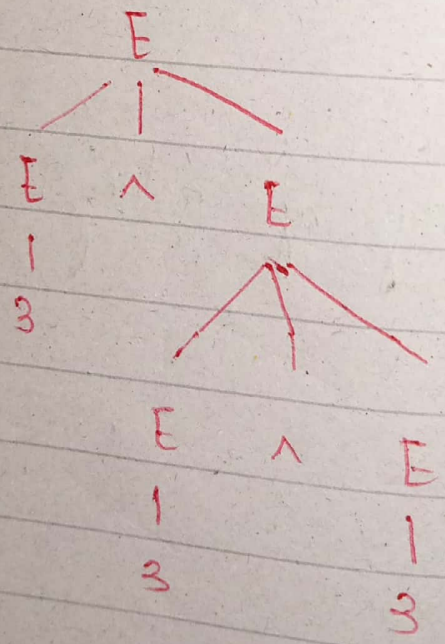
$3 \wedge 3 \wedge 3$

3^{3^3}

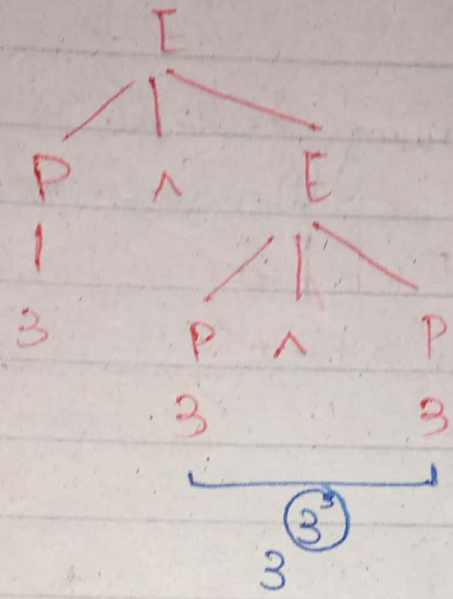
L.R



R.R



Remove Ambiguity by making grammar R->L

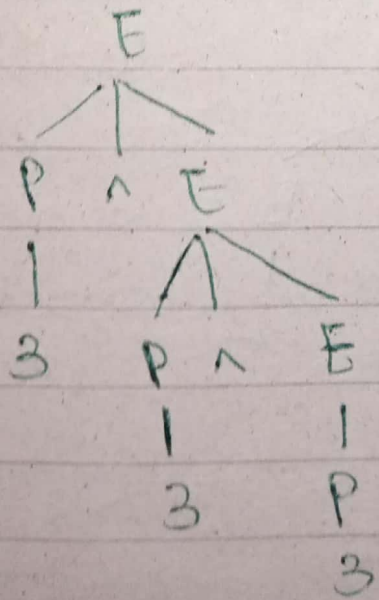


E = P^E | P

P = id

E -> P^E | P

P -> id

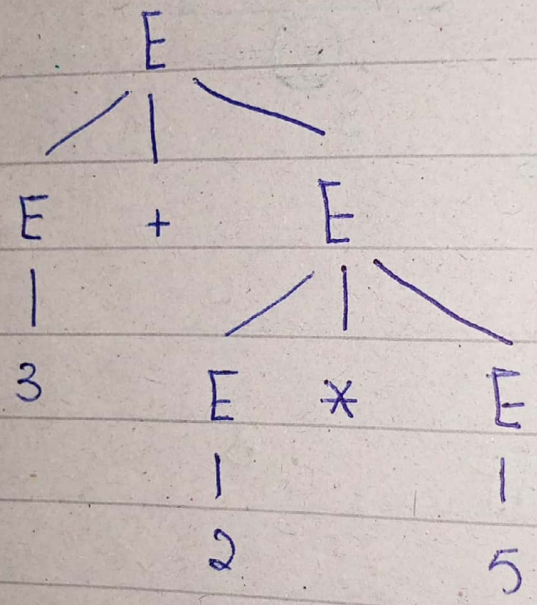


Removal Of Ambiguity:

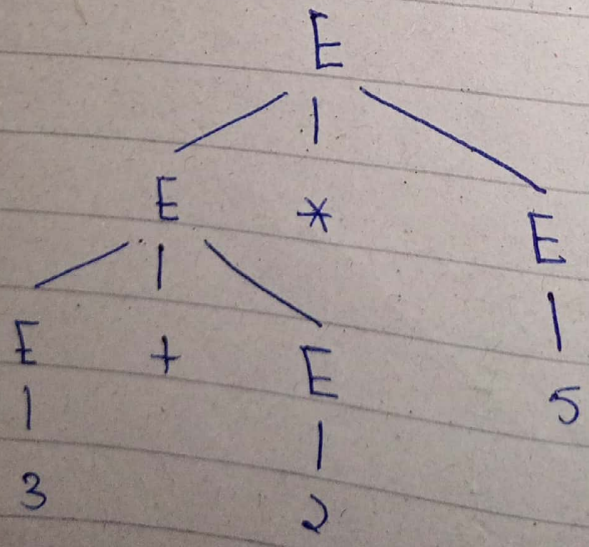
Check the following Grammar: (Ambiguous / Unambiguous)

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

Input String : 3+2*5



R.R ✓



L.R

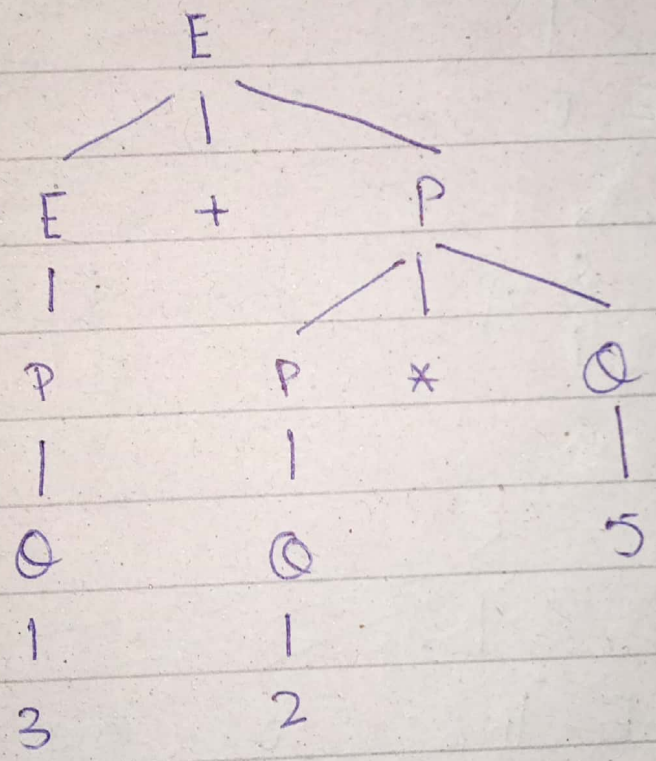
✗ a/c to operator precedence

Unambiguous Grammar.

E -> E + P | P

P -> P * Q | Q

Q -> id

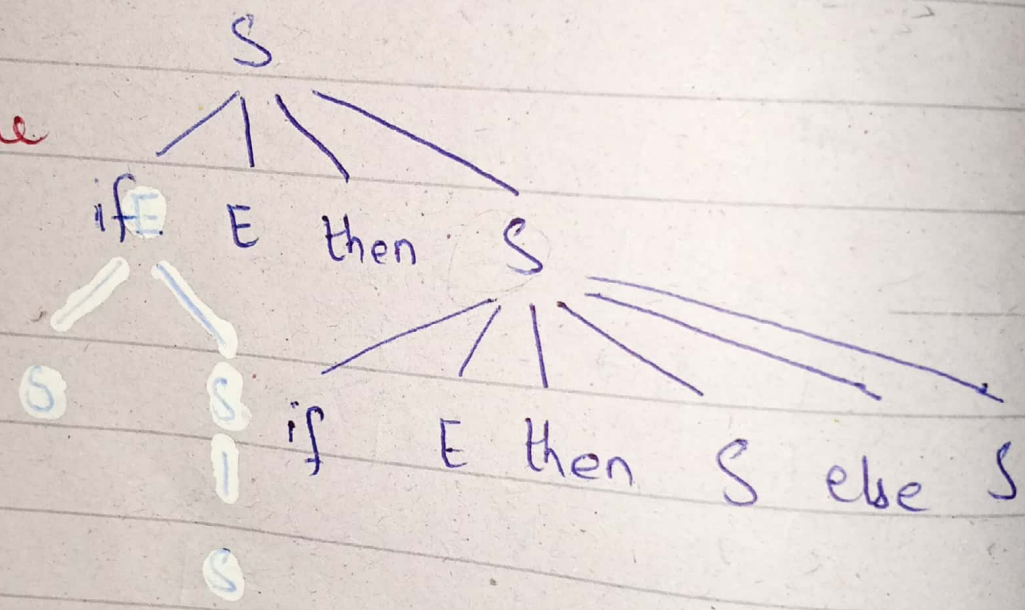


Grammars :

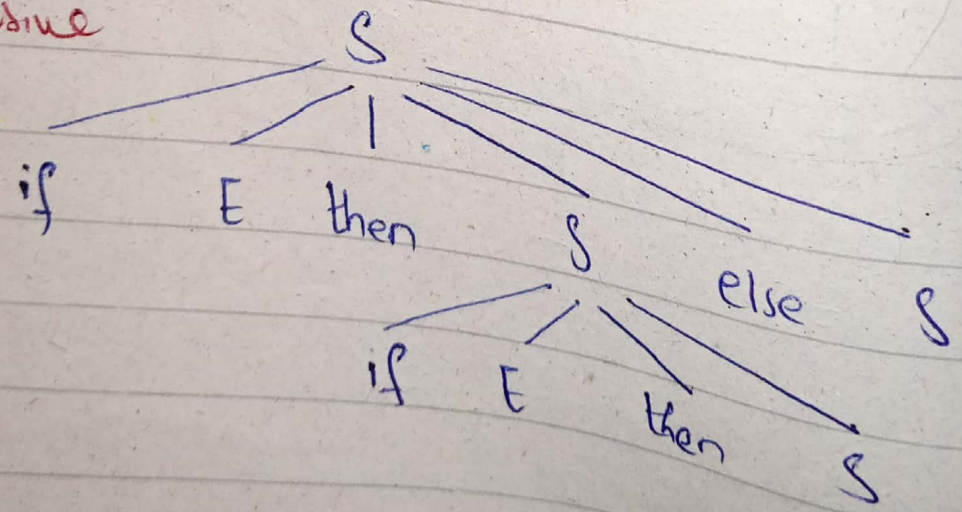
S → if E then S | if E then S else S

W → if E then if E then S else S

Right Recursive



Generic Recursive



Generic Form Of Conditional Statements:

- Matched
- Unmatched

} labels (Restrict Tree creation)

1. Start → match | un-match
2. match → if expr then match else match
3. match → non-alternative
4. un-match → if expr then stmt
5. un-match → if expr then match else un-match.

if - then - else Grammar:

<if-stmt> --> if(<logic_expr>)<stmt> |
if (<logic_expr>)<stmt>
else <stmt>

Unambiguous:

<stmt> --> <match> | <unmatch>

<match> --> if(<logic_expr>) <stmt> |
a <non-if stmt>

<unmatch> --> if <logic_expr> <stmt> |
if <logic_expr> <match> else <unmatch>

$((\lambda x.((\lambda y.(*2y))(+xy))1)y)$

$\Rightarrow (\lambda y.(*2y))(+yy)$

$\Rightarrow (\lambda y.(*2y))(+2y)$

$\Rightarrow *2(2y)$

$\Rightarrow 4y$

$((\lambda x.(\lambda y.+xy)5)((\lambda y.-y3)7))$

$\Rightarrow ((\lambda x.(\lambda y.+xy)5)(-73))$

$\Rightarrow (\lambda x.(\lambda y.+xy)5)(4)$

$\Rightarrow (\lambda y.+4y)55$

$\Rightarrow +4.554 5$

$\Rightarrow 9$

$(\lambda u.(\lambda y.y\,u)(\lambda z.u\,z))(\lambda y.yy)$

$\Rightarrow (\lambda y.y(\lambda y.yy))(\lambda z.(\lambda y.yy)z)$

$\Rightarrow (\lambda z.(\lambda y.yy)z)(\lambda y.yy)$

$\Rightarrow (\lambda y.yy)(\lambda y.yy)$

$((\lambda u.((\lambda y.(*2uy))(+uy)))y)$

$\Rightarrow (\lambda y.(*2yy))(+yy)$

$\Rightarrow *2(+yy)(+yy)$

$\Rightarrow *2(2y)(2y)$

$\Rightarrow 8y^2$

$(\lambda f \cdot f\ 7)((\lambda x \cdot x\ x)\ \lambda y \cdot y)$

$\Rightarrow (\lambda f \cdot f\ 7)((\lambda y \cdot y)(\lambda y \cdot y))$

$\Rightarrow (\lambda f \cdot f\ 7)(\lambda y \cdot y)$

$\Rightarrow (\lambda y \cdot y)\ 7$

$\Rightarrow 7$

$((\lambda x \cdot (x\ y))(\lambda z \cdot z))$

$\Rightarrow (\lambda z \cdot z)\ y$

$\Rightarrow y$

$((\lambda x \cdot ((\lambda y \cdot (x\ y))\ x))(\lambda z \cdot w))$

$\Rightarrow (\lambda x \cdot (x\ x))(\lambda z \cdot w)$

$\Rightarrow (\lambda z \cdot w)(\lambda z \cdot w)$

$$((\lambda x.((\lambda y.(xy))x))(\lambda z.w))$$

$$\Rightarrow \lambda y.((\lambda z.w)y)(\lambda z.w)$$

$$\Rightarrow (\lambda z.w)(\lambda z.w)$$

$$\Rightarrow w$$

$$(((\lambda f.(\lambda g.(\lambda x.((fx)(gx))))(\lambda n.(nm)))(\lambda n.z))p)$$

$(\lambda f.[(\lambda g.((ff)g))(\lambda h.(kh))])(\lambda x.(\lambda y.y))$

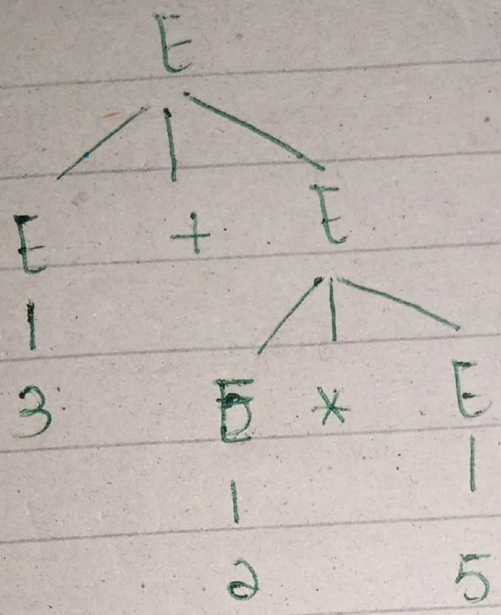
$\Rightarrow \lambda g.((((\lambda x.(\lambda y.y))(\lambda x.(\lambda y.y)))g)(\lambda h.(kh)))$

$\Rightarrow ((\lambda x.(\lambda y.y))(\lambda x.(\lambda y.y)))(\lambda h.(kh))$

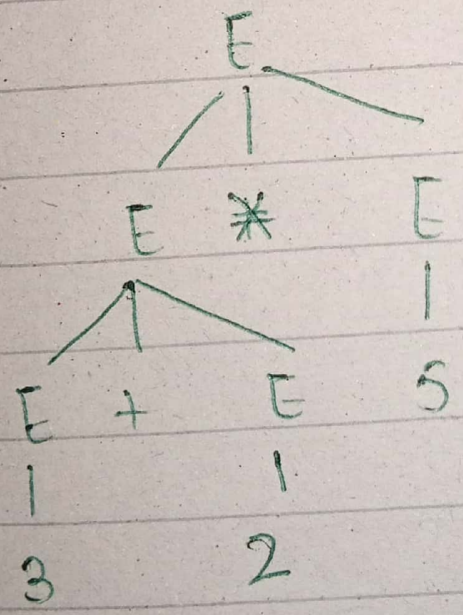
$\Rightarrow (\lambda y.y)(\lambda h.(kh))$

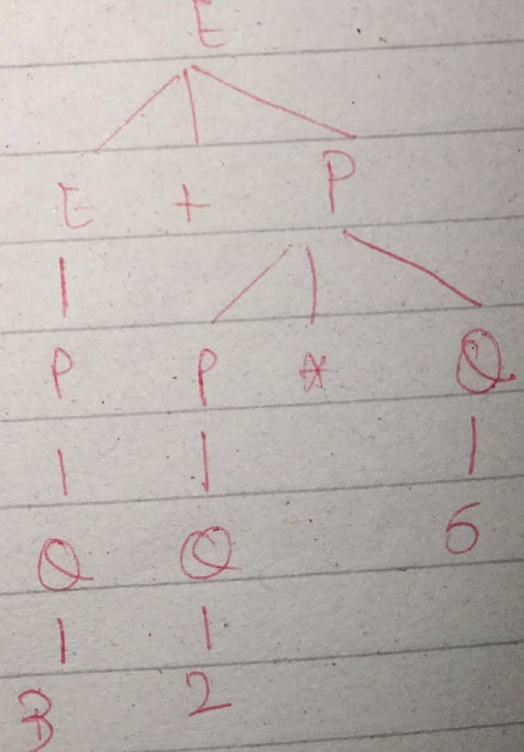
$\Rightarrow \lambda h.kh$

$E \rightarrow E+E \mid E*E \mid id$ $(3+(2*5))$



wrong a/c to associativity

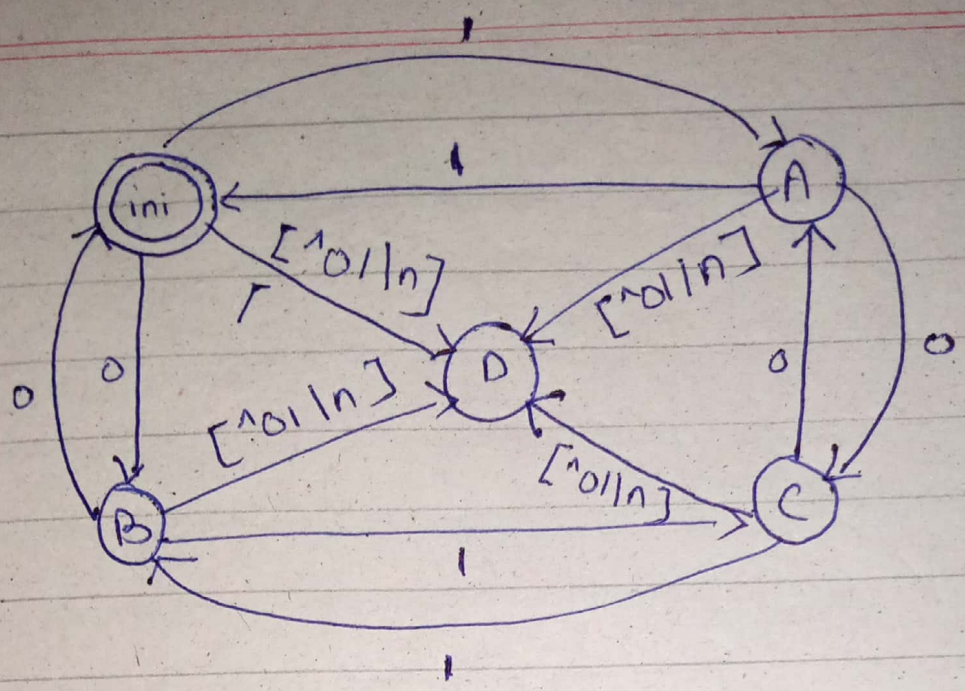




$E \rightarrow E+P \mid P$

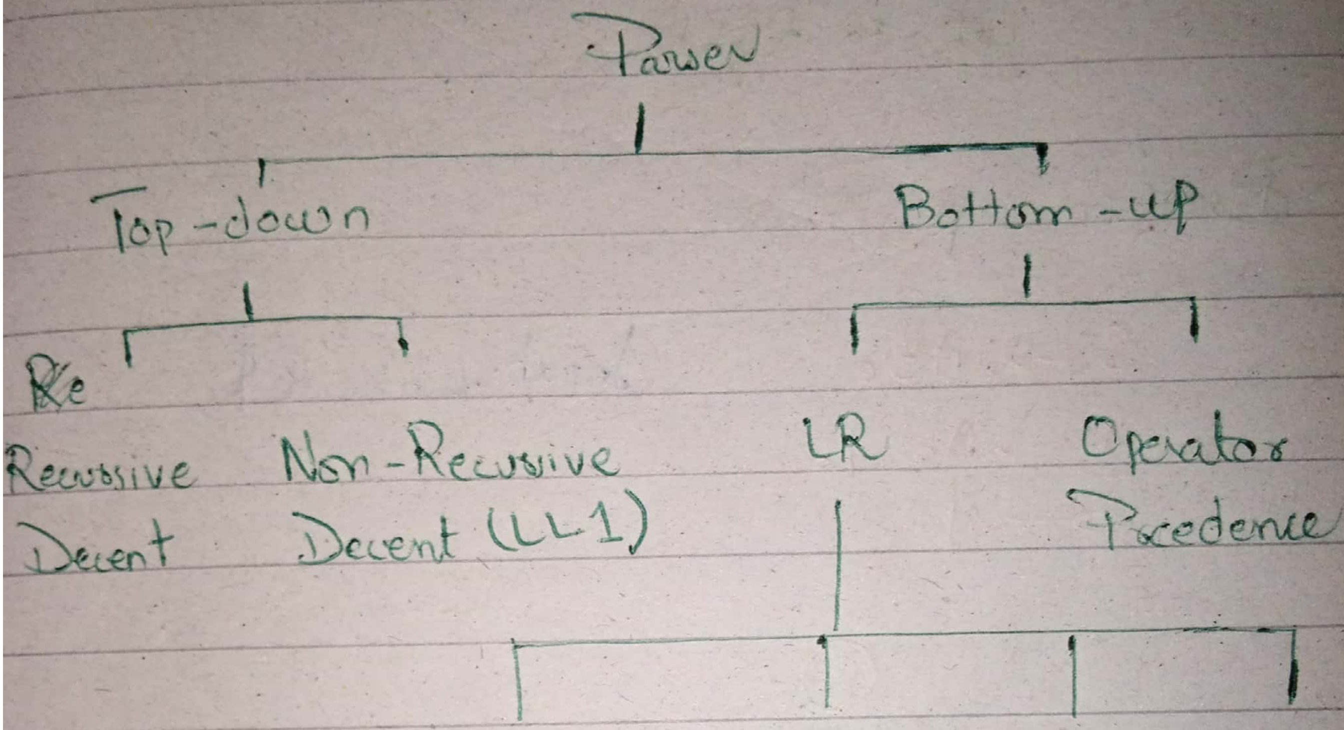
$P \rightarrow P*Q \mid Q$

$Q \rightarrow id$




```
%{
%}

%s A B C D

%%

<INITIAL> 0   BEGIN  B;
<INITIAL> 1   BEGIN  A;
<INITIAL> [^01\n] BEGIN D;
<INITIAL> \n  BEGIN INITIAL; {printf("
                  Accepted \n"); }

<A> 0 BEGIN C;
<A> 1 BEGIN INITIAL
<A> [^01\n] BEGIN D;
<A> \n BEGIN INITAL; {printf("Not Accepted"); }
```

Types of Parser:



Top-Down Parser:

- Recursive Decent Parser
  - Backtracking parser
    Jis rule ko fire krchuky thy lekin usse input nhi milsha, wapis start per jaengy.
- Non-Recursive Decent Parser
  - Predicting / Dynamic
  - No Back Tracking.

Left Most Derivation

L L 1

No. of look-ahead 1 symbol agy apk ps konsa terminal h.

Left to Right

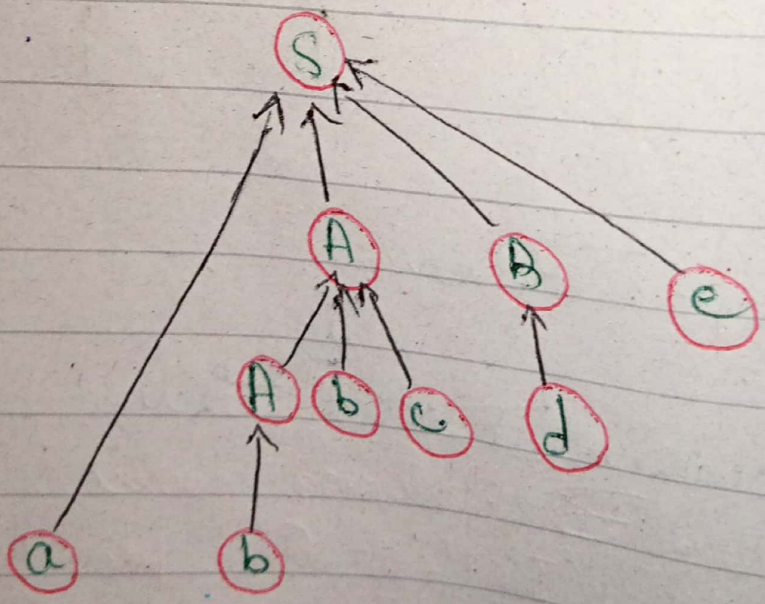
Bottom-up Parser:

S -> aABe

A -> Abc | b

B -> d

Input: abbcde$



- LR Parser

  L -> Left to Right

  R -> Right most derivation

DESCRIBING SEMANTIC ANALYSIS:

* Methodology And notation for semantics:
  -
  -
  -
  -
  -

* Attribute Grammar is a type of Grammar which is used to make semantics easy (watch video)

→ CFG with some extra information

CFG + Extra Grammar = Attribute Grammar

* Type checking
  - Type mismatch
  - Undeclared variables
  - Reserved identifier misuse

Jb undeclared variables use kra the hun. Keywords hu ghlt use kra

★ Functions:
- Type Checking
- label Checking
- Flow control Checking

Operational Semantics:

- Plain → natual semantics
- small tasks to build plain → Operational Semantics

- Types/Levels:
  1. Natural " (Big Step)
  2. Structural " (Small Step)

→ operational semantics

Eg: ERP systems

Denotational Semantics:

- based on recursive functions/numbers theory
- also known as Mathematical Semantics
- Denotion of real world problem with mathematics
- bring real world problem to functions/numbers/range/domain

Axiomatic Semantics:

- Axiom / Inference / predicate logic / calculus
- PROLOG
- Relational logic -> Assertions
  $a > b \ \& \ (a < 0)$

Use of relational operator.

- Axiom / Inference -> rules on the basis of which we develop relational logic / relation.
- Conditions
- Preconditions -> Assertions -> Resultant / Postconditions.

Assertion:

$a = b + 1$ $\{a > 1\}$

$b \neq 0$ -> Precondition (Weak) Post condition

$b \neq -1$ ->

$\{b > 0\}$ -> Weakest precondition.

$\{b > 10\}$ -> Strongest precondition.

Top-down Parser

Bottom-up Parser

PARSER USING PDA:

When input is accepted or rejected the input array and stack is empty. Acceptance by empty stack.

E.g $\{0^n 1^m 0^n \mid m,n >= 1\}$

Logic: push 0's, skip 1, pop 0's

$\delta(q_0, 00111100, Z)$ Instantantaneous Description (ID)

↓ ↓ ↓

State given i/p stac

↳ Turngstile Notation (process initiate)

$\vdash \delta(q_0, 0111100, 0Z)$

$\vdash \delta(q_0, 111100, 00Z)$

$\vdash \delta(q_0, 11100, 00Z)$

$\vdash \delta(q_0, 1100, 00Z)$

$\vdash \delta(q_0, 100, 00Z)$

$\vdash \delta(q_0, 00, 00Z)$

$\vdash \delta(q_0, 0, 0Z)$

$\vdash \delta(q_0, \epsilon, Z)$

ACCEPT

Input String & Stack is empty, String is accepted

Known As ACCEPTANCE BY EMPTY STACK

• Design Of Top-Down Parser:

PDA has following four types of transitions:

-

-

-

-

Top-down parser for expression $x+y*z$ for Grammar:

$S \rightarrow S+X \mid X,\ X \rightarrow X*Y \mid Y,\ Y \rightarrow (S) \mid id$

States are eliminated.

in stack
$Z = I$

Solution:

If the PDA is $(Q, \Sigma, S, \delta, q_0, I, F)$, then top-down parsing is:

$(x+y*z, I) \vdash (x+y*z, SI) \vdash (x+y*z, S+XI)$
$\vdash (x+y*z, X+XI) \vdash (x+y*z, Y+XI)$
$\vdash (x+y*z, x+XI) \vdash (+y*z, +XI) \vdash (y*z, XI)$

$\vdash (y*z, X*Y1) + (y*z, y*Y1) \vdash$
$(*z, *Y1) \vdash (z, Y1) \vdash (z, z1)$
$\vdash (\epsilon, 1)$

Design Of Bottom-up Parser:

Expression "x+y*z"

Grammar:

$S \rightarrow S+X \mid X, X \rightarrow X*Y \mid Y, Y \rightarrow (S) \mid id$

Solution:

If " " bottom-up parsing is:

$(x+y*z, 1) \vdash (+y*z, x1) \vdash (+y*z, Y1)$
$\vdash (+y*z, X1) \vdash (+y*z, S1) \vdash (y*z, +S1)$
$\vdash (*z, y+S1) \vdash (*z, Y+S1) \vdash (*z, X+S1)$
$\vdash (z, *X+S1) \vdash (\epsilon, z*X+S1) \vdash$
$(\epsilon, Y*X+S1) \vdash (\epsilon, X+S1) \vdash (\epsilon, S1)$

Tree will be same either created from top-down approach or bottom-up approach.
The difference is b/w strategy/computation.

PL -> Programming language

NAMES, BINDING AND SCOPES:

Identifier Association Variables
of
relation

*NAMES:

1- Design issues for name:

- Case sensitive? PL have capability to distinguish b/w capital or small letters. case sensitivity in identifiers.
- special words, reserved words & keywords?
  Those words which are parts of PL.

2- Length

- meaningful length -> Explain (imp)
- If too short, cannot be connotative.
- Software designers impose this.
- Restriction in Variable names -> Php, Perl, Ruby.

instance -> time specific
↳ that changes in the territory of class

instance
Sir kamran at 9 Sir Nayeed at 10

3- Special Characters:

- PHP: begin with $ sign.
- Perl: begin with special character specifies Variable's type.
- Ruby: Variables begin with:
  - @ -> instance Variable
  - @@ -> class variables

→ Scalar quantity (Variable begin with $)

1 - Array (@)
2 - Array Associative (%)

Example of each How

Instance -> specific to class Variable changes in the territory of class

Class Variable -> class specific
-> changes outside the class

SecA has TPL course
SecB " "

4- Case Sensitity:
- Disadvantage (Names that lookalike are different)

5- Special Words:

- Reserved words -> language restricts / reserved.
-> cannot be used as a user-defined name

e.g: functions in libraries used by other, cannot use those function names as user-defined name.

- Keyword -> user restricts word context specific.
-> PI = 3.144
#Beauty = 9.8
-> cannot be used as variable name.

eg: COBOL (reserved words)

6 - Variables:

- Abstraction of memory
- Characterized as six tuples of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

- Attributes of Variable:
  - Name: not all variables have names
  - Address: address of variable

method to access variables:

dynamic -> Address (Pointer, reference variables, heap trees, graph) (dynamic)
-> Name
-> Indexing

-> Aliasis (variable name ko chota karna)
L-> readability issue

- Type: determines range of values of variables

- Value: contents of location with which the variable is associated.

signed int
-ve to +ve
range

unsigned int
0 to (some large value)
range.

l-value: address
r-value: value

- Abstract Memory Cell:
  Physical cell or collection of cells associated with a variable.

★ Binding: (Association / Relation)

- Association b/w entity & an attribute, such as b/w variable and its type or value:
  - name & type
  - variable & value

int a = 0

- Binding time → time at which binding takes place.
  - Compile time
  - run time

- Possible Binding times
  -
  -
  -
  -
  -

- Static & Dynamic Binding

- Explicit & Implicit Declaration

Explicit
hi chz
bta di variable
li
type btana

Implicit
Var, type
nku btana

Adv: writability Disadv: readability

Important

STORAGE BINDING:

Association of values with respect to memory.

CATEGORIES OF STORAGE BINDING:

allocation deallocation

- STATIC BINDING:

- Assign a value to variable at compile time.

- Execution at run time.
  Not change at run time.

```
Const int count = 100;
```

Before Allocation	Alloc Dealloc Start end	End / After Execution

- Deallocation at the end
- Allocation through out and start of execution.

printf(" ");
"
"
"
int count = 0; Allocation

+ Drawback: Donot provide flexibility in programming.

+ Advantage: Fast

STACK

- DYNAMIC BINDING: simple array

- Variables that are made on stack.
- Changes at run time.

int count = 10;

Before Execution	Allocation at time of declaration	Deallocation End	After Execution

+ Advantage: Change values during execution as permissible.

+ Drawback: History sensitive. Donot retain previous values. Slower than previous.

- Explicit Heap Dynamic:

new int count = 10;

- use pointer to make its reference.

Bef Exe	Allocation Deallocation During Execution apni mrzi se	Aft Exe

+ Advantage: Save memory, expand during runtime.

+ ~~I~~ Disadvantage: Slower than Ptrs 2.

- Implicit Heap Dynamic:

```
st = ['Python', 'C']
```

```
st = 5
```

- Change type implicitly } Leads to expand or collapse.
- Change data implicitly }

Before Exe

Allocation Deallocation
During Execution
apni marzi sey.
Expand & Collapse
due to type
change.

After Exe

Scope: (Environment)

Environment -> multiple variables (multiple var

Scope -> initialization of a var (single va
declaration " " "

C-Specific word for scope is block

↳ is block ke andr yeh var hain.

Scope	Env	bbck
single var	List of var	List of var inside a block of code

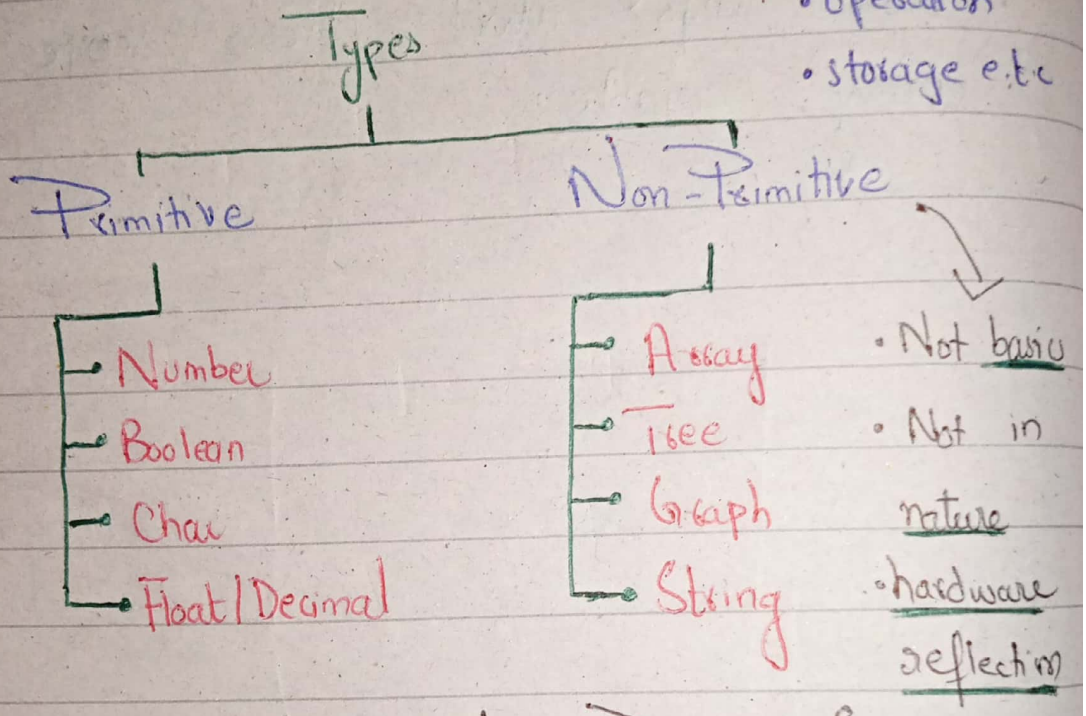
-> local var
-> global var

Search Process:

- related to scope / env / block.
- managed by compiler.
- stores info about the scope of every variable.

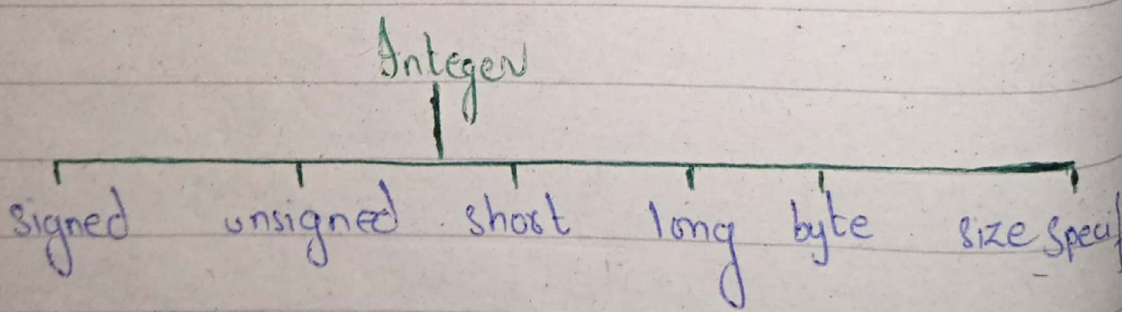
Data Types:

Descriptor: Collection Specified/over all specs of a datatype.
- operation
- storage etc



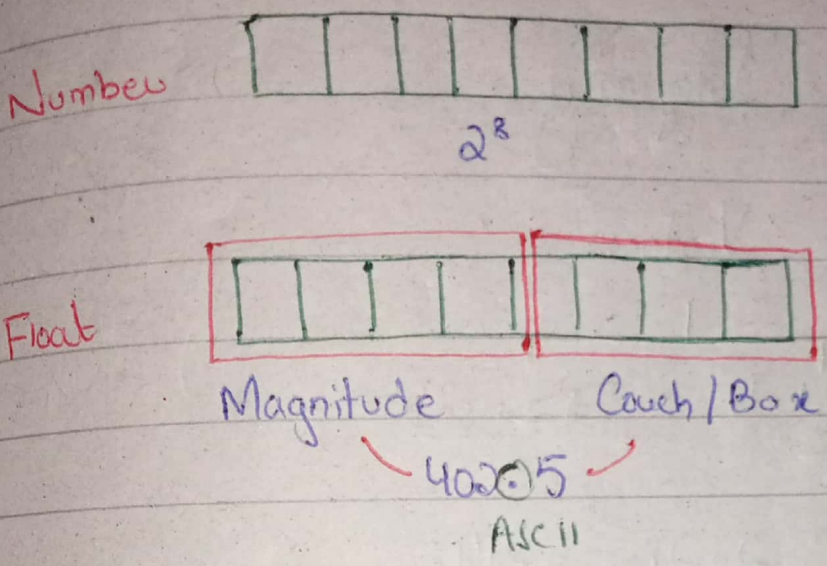
Design Issues In Data Types:

- Type Checking



- Range limit

- How floats are stored in memory?



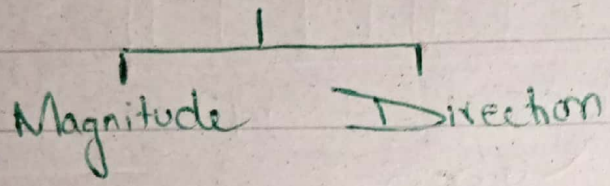
Two blocks will be maintained for Magnitude and couch/box. ASCII code for decimal. Both blocks will be placed in a single block.

IEEE 754 (Standard) for floating points

- Design Issues In Floating Points:
  - Accuracy
  - Precision
  - Completness

- Complex Numbers:

  - Real + Imaginary
  - Used in Vectors



  - Example: 2+2j , 3-14j
  - Python allows to write vectors in as it is format.

C99 was the basic version of C-language

- Memory wastage

A language supports decimal places upto 10th digit.

e.g 1.2000000000

Memory wastage

- Boolean:

Yes -> 0 No -> 1

or Yes -> 1 No -> 0 ≤ preferred, languages supports

Yes -> bits will be high (1) in numbers other than 0.

No -> All bits will be low (0)

100 gives -> Yes

- Characters: String

- String (Array / Collection / Sequence of characters)

- Non-primitive

- Due to hardware reflection, it is non-primitive.

- Doesnot comply nature

- Storage can be : • Static
  • Dynamic
  } -> depends on size
  ↓
  depends on storage

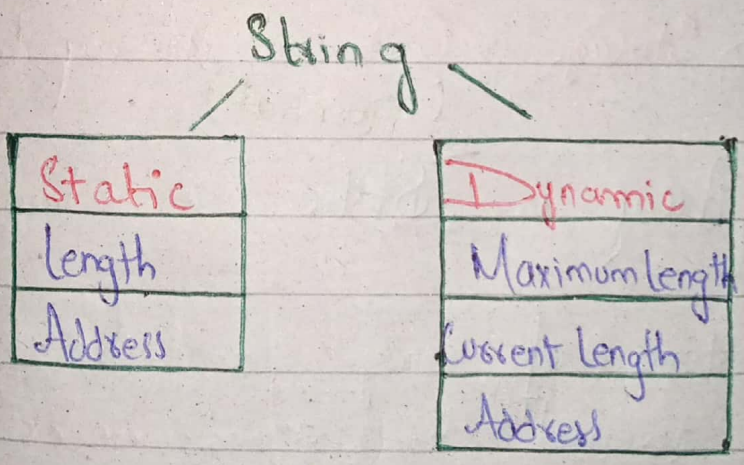
- We can add/^append the strings in terms of concatenation.

- We can use comparison operators in string.
  - Comparision is done character by characters.
    "Karachi" == "Islamabad"
  - e.g Comparision for searching

- Sub-String Reference or Sub-List Search
  - Can be used in pattern matching.

What features Python and Basic C gives for String?

Python	Basic C
• split • reverse • sub-string searching	String as a data type is not the part of C. So, comparison cannot be done.



Descriptor of String

Tells about
- Storage
- Length of String

Array:

- Collection of Items having same Data type. (Homogenous data types)

- Now a days, definition changes Collection of Items having different Data type. (Heterogenous data types)

Design Issues Regarding Arrays:

+ Size -> Storage -> Contiguous Memory location (sequential)

- Basic Array is Static in nature.

- Data items of arrays can be randomly accessed by giving: • index or • Address

+ Indexing

Legal subscript -> name [5]

Illegal subscript -> name [c]

Bound the programmer, to use numbers in the indexing of array.

- Operations in Arrays:
  - Insertion
  - Deletion
  - Sorting
  - Searching

+ Out of bound exception.
  (Index out of range)

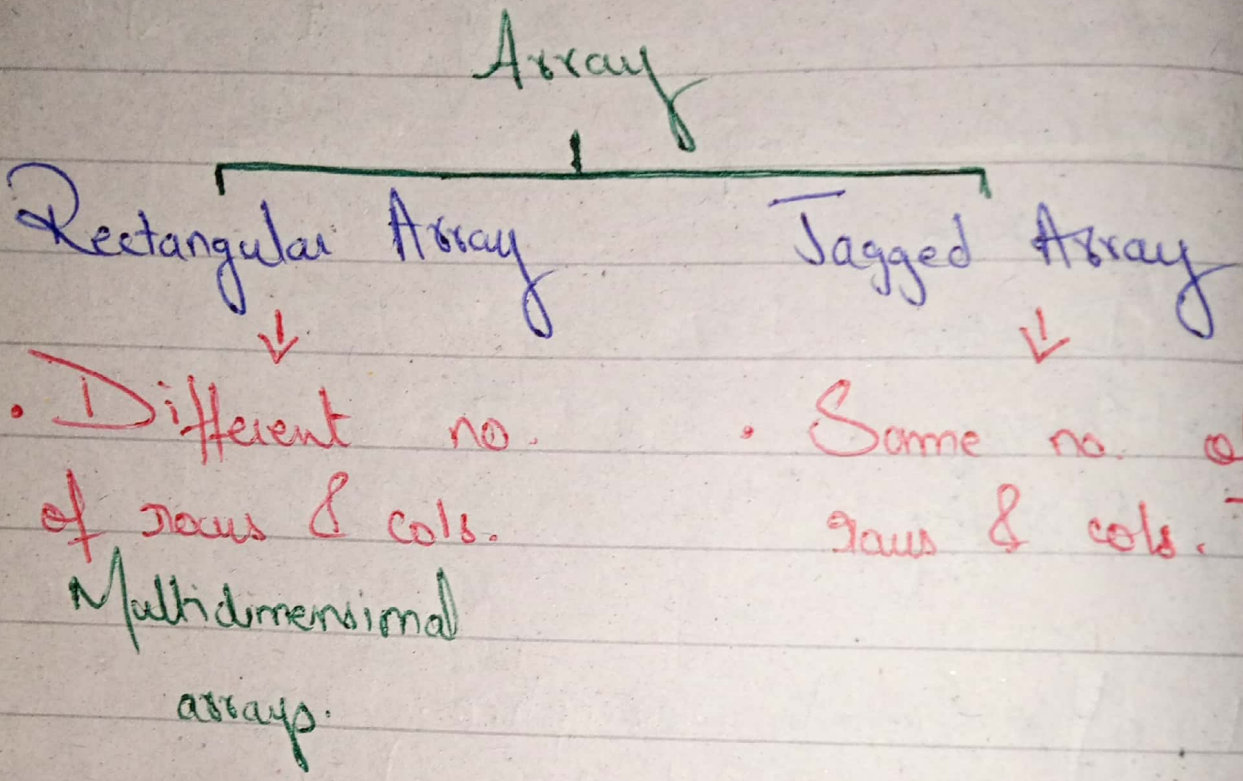
- Syntax of Arrays:

  A[10] → Python / Java e.t.c
  A(10) → Fortran / Ada e.t.c

- Type Checking In Arrays:
  of index Values [10] is Mandatory.

- Languages that supports heterogenous datatypes:
  Python, JS, PHP, Pearl e.t.c
  Ruby

• Types of Arrays



encloses the data type of a variable

DESCRIPTORS IN ARRAY:

Array name
Type of Element
Index Type
Lower Bound
Upper Bound
Address

Single Dimensional Array (1D) Descriptor

Array name
Element Type
Index Type
Number of dimensions
Index range 1
⋮
Index n
Address

MultiDimensional Array Descriptor.

Also study: Dynamic Array Descriptor,
Tuple Descriptor,
List, Record,
Set, Union,

POINTER:

To Point the Dynamic Memory Location!

```
int a = 10;
Float *ptr = NULL;
ptr = *a;
```

wrong as int & float are diff data types having different no. of bytes for memory.

* The data type of variable, and data type of pointer should be same.

Types of Pointers:

Initialize

1- Dangling (Allocate then Deallocate its
2. NULL (Pointer directing to NULL)
3- Wild (Initialized but not used)

Problem

code

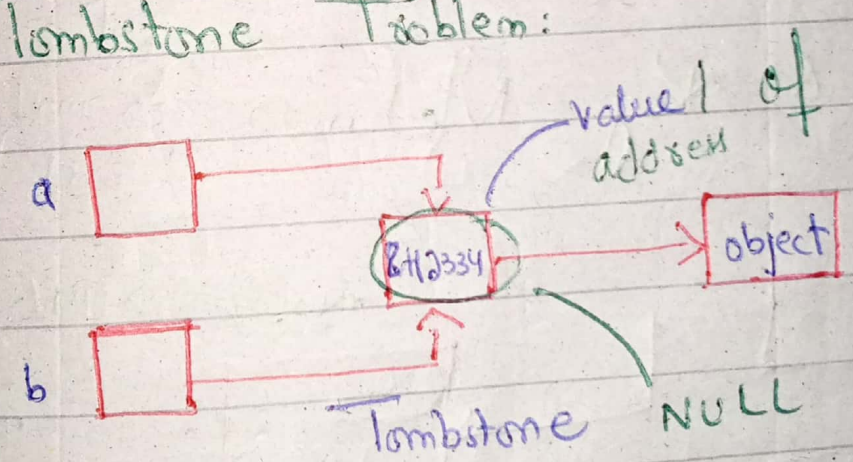
→ If used after deallocation it will be clashed with other locations. because garbage collection is not done properly & will direct to a location not accessible to you.

```
func() {
    int x =
    :
    return *x;
}
```

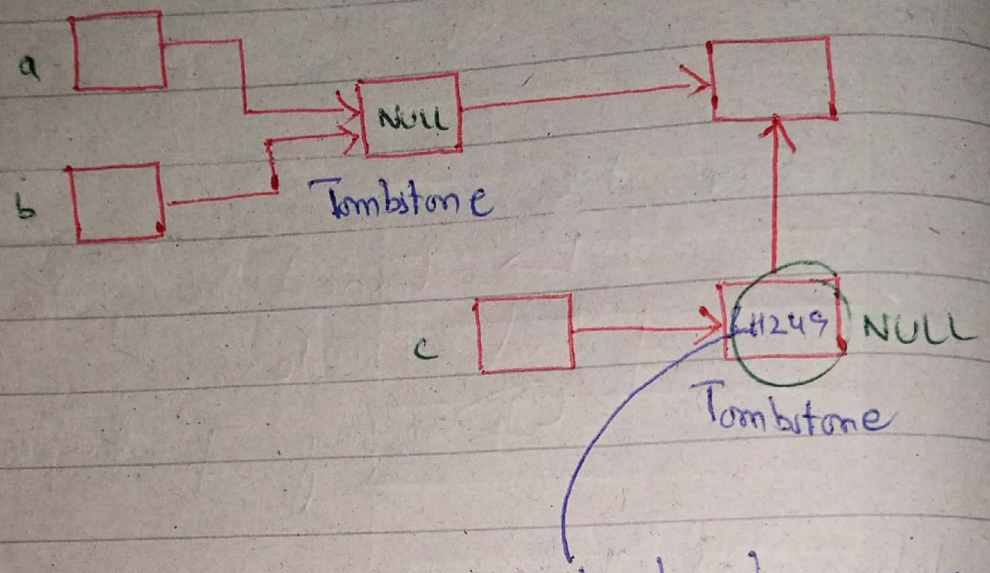
↳ returned the address of local variable.
(Dangling pointer problem)

Solution of this problem:

- Tombstone Problem:

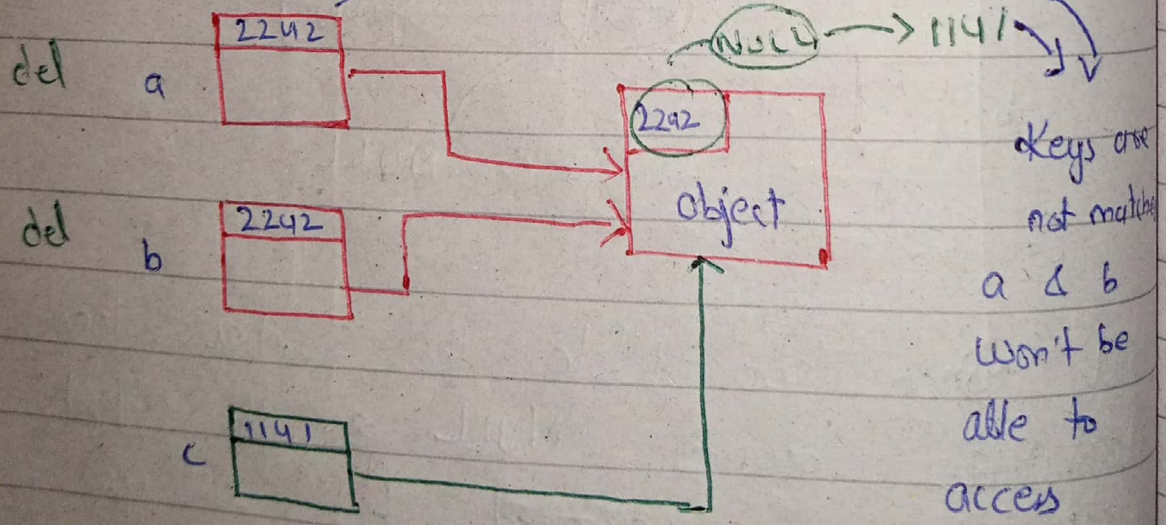


But when we delete the pointer then Tombstone becomes NULL & the object will never be accessed but it is stored in memory.



overhead on memory in terms of resource.

. Lock & Key Mechanism (In most of PL)

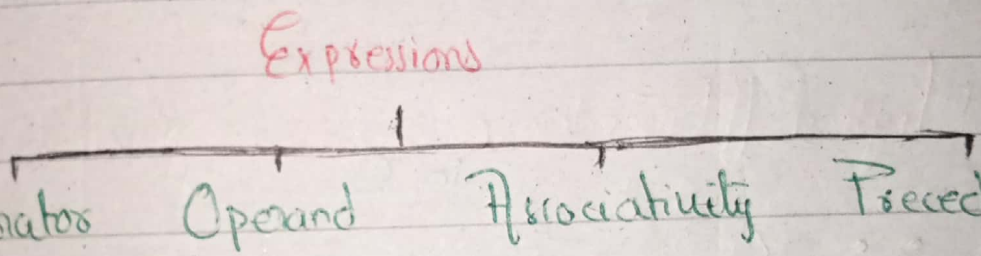


Keys are not matched a & b won't be able to access

Expressions Assignment:

For any type of problem of real whatever we compute something from computer, the output is expression.

Used in Imperative Paradigm.



Design Issues Related to Expressions:

1) Precedence Rule
2) Associativity Rule
3) Evaluation of Operand
4) Overloading of Operating
5) Side effects.
6) Mixed mode expression
   - Expression with combo of diff types of operators
     Logical, arithemetic, bitwise etc.

Types of Operators:

1) Unary (increment / decrement)
2) Binary (Arithemetic operators)
3) Ternary (if else (conditional) ?
three operands Var x = (y>10) ? true : false)

Operator Precedence: → Language Dependent

Highest Precedence in C

:: Scope Resolution Operator.

Ruby: Operators are not available.

We have Methods for computation instead of operators.

Scheme or Lisp: Change the way of writing exp (Preorder approach)

Both have same paradigms.

a+b*c

to write this in scheme/lisp:

(+a(*bc)) → Preorder

average = (count==0)? 0 : sum/2

ternary exp

Types of operands:
- variables / ident
- Const

Evaluation Order:

Find that is unknown

1) Variable / Identifier
2) Constant
3) Paranthesis
4) Function call

- Same precedence will be given to user-defined and built in functions

Side Effects:

Something that is not desired with output but also comes as output. is sideeffect.

impacts → a = 10 ← reference address to a.

→ b = a + fun(&a)

→ sideeffect

have capability to change value of a

Solution:

problem: Call function in the end.

→ variables in expression are freezed

if variables in expressions are same and

the same variable is used in function either passed by ref or value in that same exp. Freeze that variable. then compute fonctions at the end.

Referential Transparency: Same answers to the same nature exp.

result1 = (func(a) + b) / (fun(a) - c)

tmp = fun(a); -> why we call it RTransparent bcz it has this common factor (tmp).

result2 = (tmp + b) / (tmp * c)

result1 = result2

then both exp result1 & result2 comply Referencial Transparency.

If there is no Referencial Transparency in exp then we have side effects in exp.

universal ground -> 0 volts
Stabilize

SHORT-CIRCUIT EVALUATION:

The estimation of an expression to be evaluated as zero. This is said to be short-circuit evaluation.

```
a = 0
(13 * a) * (12 / 3) -> 0
```

- Readability increased
- Syntax analysis work will reduce.
- Semantic " " "

Compiler construction Saves time & computational resource

C, C++, Java offers shortcircuit

Bitwise operators donot offer shortcircuit.

Assignment operator

= := (in Ada)

a = a+1 a = 10 a, b, c = 10, 11, 12

a += 1 ternary expressions.

a = input("Enter name: ")

PERL & RUBY (multiple assignments)

Sub-program:

Some selected set of instructions that can run independently.

- Module
- Subroutine
- Part of Main Program
- Function
- Procedure
- Method

Return type Function( )
↓
Req of Func ← parameter

A set of instruction that can be executed independently

Procedure & Function are two broad categories.

- Process Oriented
- Not Data Specific

- Implementation of Mathemedical model
- Data Specific

* Abstraction - Not interested to know hidden

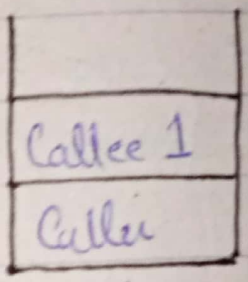
- Process Abstraction (Function hidden)
- Data Abstraction (Object hidden)

Callee 1

Whenever a function / Sub program will be called in the main program.

Caller

Execution control goes to callee and the data of caller goes into stack.



Callee 1 calls another func. Callee 1 goes in stack and execution control goes ~~init~~ to callee 2.

Callee 2 execution completes.
Execution control goes to stack.
Callee 1 resume. its data (pop) from stack.

1 Design Issues:

① Return type Function (Parameter) ② → Req of Func

Req but not necessary.

① Declaration of function -> Signature
② Definition
③ Call

Parameter

formal
Same in ① Declaration
② Definition

Actual
① Call
either constant or variable.

```
int Add(x,y);  -> Signature
```
↳ parameters / formal

```
int main(){
    input x,y
    print(add(x,1)
}
```
↳ actual / arguments

↳ Parameters / formal

```
int Add(int a, int b){
    return a+b
}
```

Parameter:

Design Issues:

=> Data type — Parameter, Return Type, Arguments

=> Type Conversion

=> Global / local -> Scope

=> Positional Arguments

=> Keyword Arguments

=> Default value of arguments

=> Variable no. of parameters — Fixed or Variable — extend or collapse no. of parameters.

=> Optional / Required Arguments

=> Scope

Local

Function Design Issues:

=> Data type — Parameter, Return Type, Arguments

=> Type Conversion

=> Fixed or Variable Parameter

=> Scope

Local — Static, Dynamic

Global — Static, Dynamic

=> Calling Method of Function:
Argument Passing Method.

=> Function Should be generic.

=> Side Effect.

=> Function overloading / overriding

=> Dedicated Task

=> Recursion / Nesting functions

=> Position of function definition
- Either in main
- in header file
- above main
- after main

=> Labels / flags | goto above:
Labels

like interrupt in OS.

return goto

Parameter Passing Modde:

* IN mode Caller -> Callee
* OUT mode Callee -> Caller
* IN-OUT mode Both

IN mode:

```
Add (x, 1)
int Add(int a , int b )
   { return a+b ; }
```

* a and b are local to function.
* cannot access or change data.

OUT mode:

```
Add (x, 1)
int Add (int a , int b){
    int c; — inside vicinity/ of functio
    c = a+b;          scope
    return c;
}
```

* c can bring data to you main program (to caller)
* can access but cannot change.

IN-OUT mode:

* parameters modified & return.

C -> Pass by value allowed
" reference (using pointers)

C++ -> Pass by reference (Special type of pointer)

Diff in mechanism

Python -> Both allowed (value & reference)

Java -> Pass by value
Pass by obj (reference)

SQL -> Pass by ~~refer~~ ref / value

Typical:
Comments, Reviews, Suggestions, Recommendation

① What is TPL
② Paradigms
③ Compilation
  ① Lexical } Numerical Flex & bison
  ② Syntax } 90% syntax should be correct
  ③ Semantics -> Type

Easy:
Explain, Recommend, describe
- 5 mark 1.5 page
- Must mention diagrams if necessary
- Explaination valid

④ Type of Semantics
Numerical of lexical

- 12 Questions (6 Typical, 6 Easy)
- Skip Lecture Control Structures

⑤ Parsing imp
⑥ Quiz Questions
  • must justify (is the grammar amb not amb)
  • justification imp.