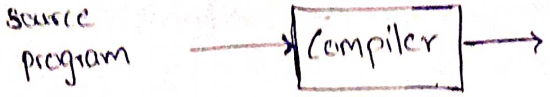
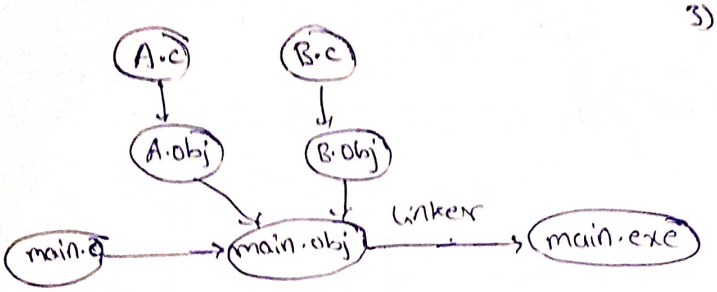
16/06/20

Compiler Design

Compiler: It is a language translator which translates from one language to other



1) Assembly language
2) Relocatable machine language
3) Absolute machine code.



Contents:

1) Lexical Analysis
2) Parsing } ★ 70%
3) S.D.T }
4) Intermediate code generation

4 marks:
2 - 1 marks
1 - 2 marks

Reference: Ullman

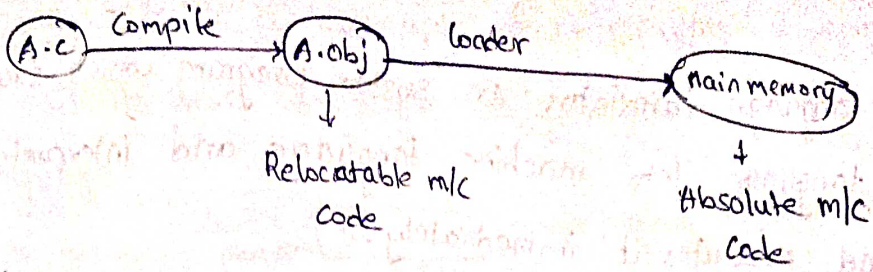
Introduction:

Compiler: It is a language processor which translates source program into object program.

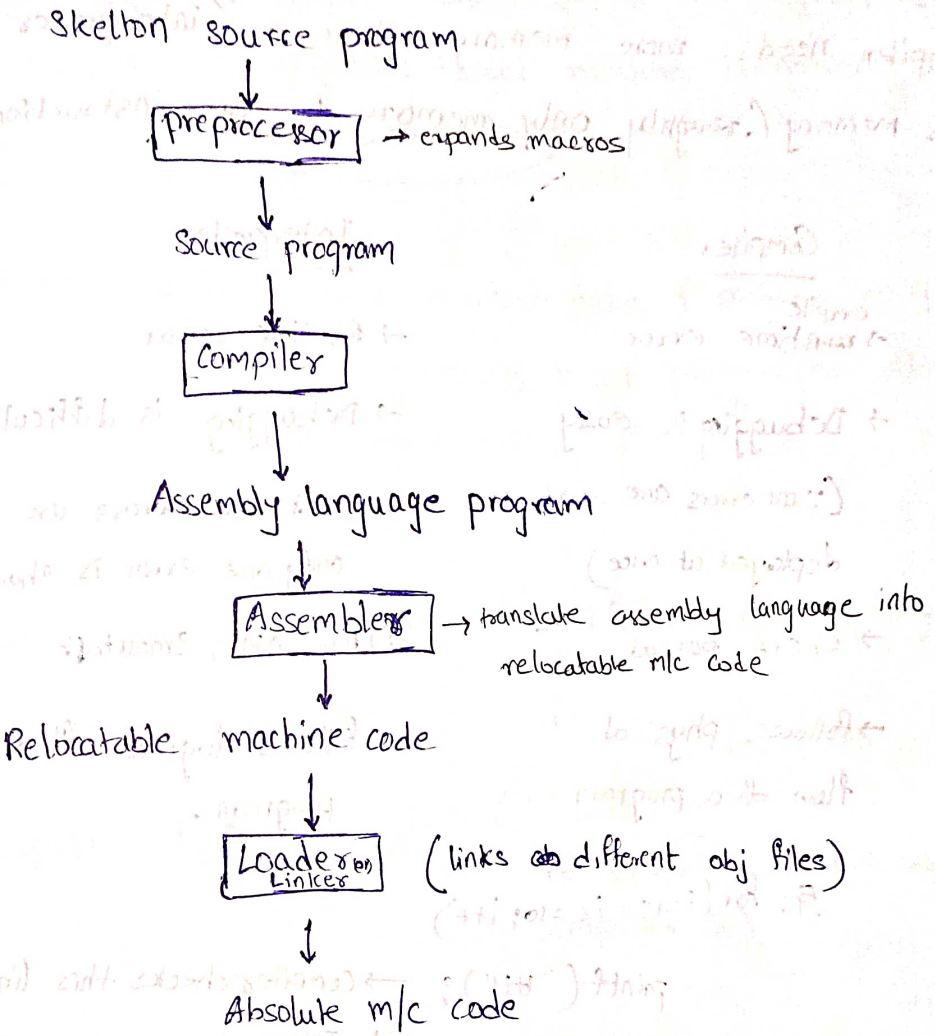
Object program may be any of below 3:

(i) Assembly language
(ii) Relocatable machine code
(iii) Absolute machine code

→ where?



Language Processing system:



Linker: It makes a single program from several files of relocatable machine code.

Loader: It takes relocatable machine code and altering the relocatable addresses and placing code & data into memory. This process is called loading.

Interpreter:

It is a language translator or system program which translates a high level language into machine language and interprets each instruction and executes it immediately.

→ Execution of compiled languages is faster than interpreted languages.

→ Compiler needs more memory where as interpreter needs less memory (roughly only memory for one instruction)

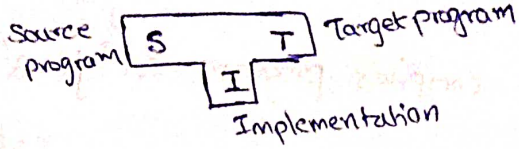
Compiler	Interpreter
→ compile time error	→ Runtime error
→ Debugging is easy (∵ all errors are displayed at once)	→ Debugging is difficult (not all errors are shown only one error is shown).
→ C, C++, pascal	→ LISP, APL, Smalltalk
→ follows physical flow of a program	→ follows logical flow of a program.

Eg: `for(i=1; i<=10; i++)`
`printf("Hi");` → compiler checks this line only once
→ Interpreter considers the line 10 times

→ Interpreters are mainly used when size of main memory is less (so that we can execute each instruction line by line)

→ All web based programming languages are interpreted programming languages because they need to be run different devices with different memory.

Bootstrapping: It is a concept of compiling a compiler program in its own language to generate the compiler for a new language.

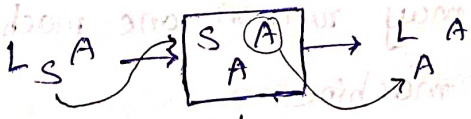


Existing compiler: S_A^A → Compiler which takes 'S' program and generates target program for m/c A. which runs on A.

Required compiler: L_A^A → Compiler which takes L program as i/p and generates target program for m/c A and which runs on A.

Step 1: Write a compiler program for 'L' language in language 'S' targeted for m/c A.

Step 2: Compile L_S^A in S_A^A (∵ L is written is S to compile L we use compiler S)



↓ compiling a compiler program in same language

Eg:

existing: C win Compiler for 'C' for generating target program for windows and it runs on windows
win

Required: Consider we need a compiler for java

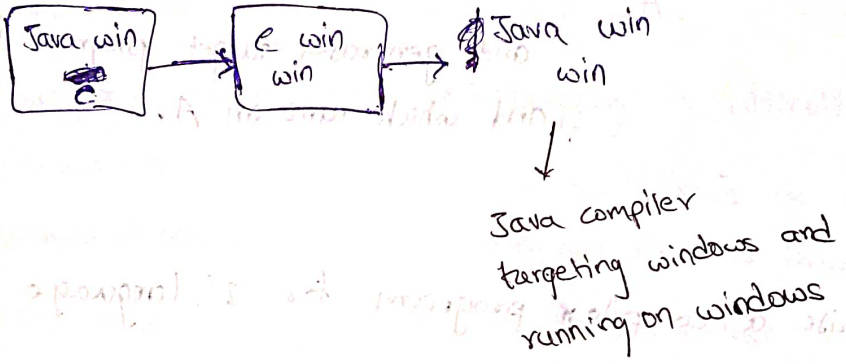
i.e., java win
win

Step1: Write a compiler program for "java" using 'c' targeted for window

java win
C

Step2: Compile

Java win using C win
C win



Self-hosting Compilers:

It is compiler which is capable of compiling its own compiler program.

Cross compiler:

A compiler that may run on one machine and generates target program for another machine.

Existing: $L_A{}^A$ (compiler for L language, targeted for m/c 'A' and runs on machine A)

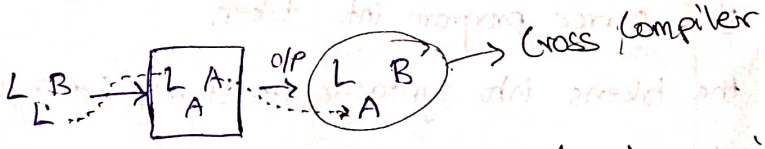
Required: $L_B{}^B$ (compiler for L language, targeted for m/c 'B' and runs on machine B)

step 1 : write compiler program for [redacted] 'L', targeted for machine B using L.

ie., L_{L}^{B}

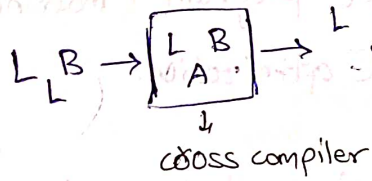
step 2 : Compiler

L_{L}^{B} using L_{A}^{A}



↳ compiler for language 'L' targeted for m/c 'B' and runs on m/c 'A'

step 3 : Compile L_{L}^{B} into L_{A}^{B}



above entire process is known as cross compilation.

Existing: Java win / win

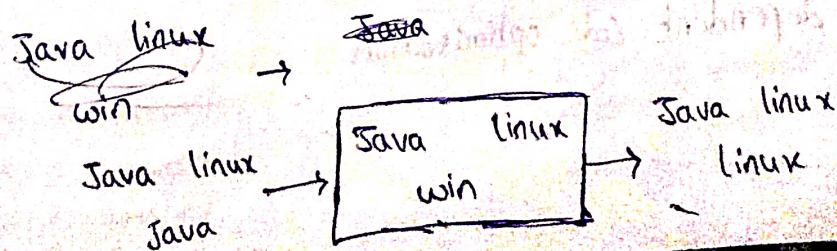
Req: Java linux / linux

step 1 : write a compiler program · Java linux / Java

step 2 :

↳ cross compiler

step 3 :



Compilation Process:

It is divided into 2 parts

1) Analysis part
2) Synthesis part

Analysis part

(i) divide the source program into token
(ii) group the tokens into syntactic structures (~~syntax~~ parse trees)
(iii) verify the semantics
(iv) generate intermediate code
(v) m/c independent code optimization

Synthesis part:

1) generate the target program (machine dependent)
2) M/c dependent code optimization.

→ Each task in analysis and synthesis part is represented as phase.

phase: It is a logically cohesive operation
i.e, logically interlinked operation.

Phases in Compilation:

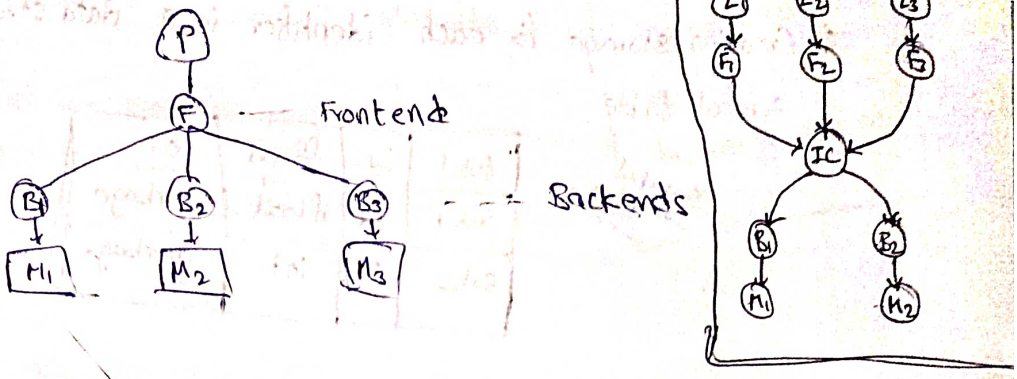
1) Lexical Analysis
2) Syntax Analysis
3) Semantic Analysis
4) Intermediate Code generation
5) M/c independent optimization (only if required)

} front end

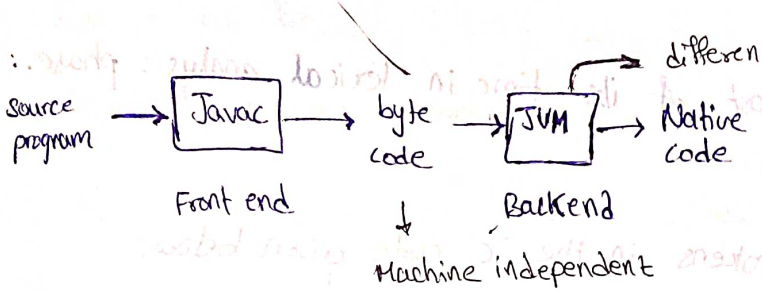
6) Code generation
7) M/c dependent code optimization

} back end

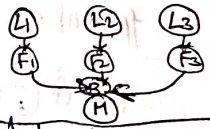
* Consider we need to make a compiler of language 'P' for 3 machines M_1, M_2, M_3. We can design one front end and design 3 backends for each machine. This process is called retargeting.



Eg:



Similiarly if we have different source languages compilers with same target machine code, we can design different front ends producing same IC and have one backend.



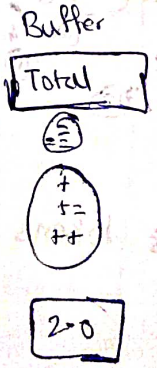
1) Lexical Analysis:

* also known as linear analysis, scanner, token recogonizer

* Lexical analysis is a phase in which stream of character making the source program is read and grouped into tokens that are sequences of characters having a collective meaning such as identifiers, keywords, constants, special characters etc.

Eg: Total = sub1 + sub2 * 2.0 ;

total ← identifier
= ← operator
sub1 ← identifier
+ ← operator
sub2 ← identifier
* ← operator
2.0 ← number
; ← special character



function of LA:

1) Removes whitespaces and comments (if there are more than 1 consecutive whitespaces, then it is reduced to one whitespace)

↳ single spaces are not removed

2) divides the program into tokens.

3) Creating storage for each identifier in a data structure called symbol table.

symbol table

total	id	float	0.0
sub1	id	float	garbage
sub2	id	int	garbage

Note:

* Compiler spends most of its time in lexical analysis phase.

Q1) Find the no of tokens in the 'c' code given below:

```
main()
{
  // printing the message
  printf("Hello students");
}
```

∴ 10 tokens

Q2) Find no of tokens in the below program

```
while(i>=0)
{
  printf("%d", i);
  i++;
}
```

∴ 18 tokens

Note:

Scanning the i/p stream, (source program) the LA groups the characters into meaningful sequences called Lexemes. For each lexeme, a token is produce as an o/p. The form of token is

<token-name, attribute-value>

token-name is an abstract symbol that is used in syntax analysis. Attribute-value points to an entry in symbol table for this token.

(Q3) which of the following can be recognized as a token without depending on next symbol

a) + b) < c) >> d) none

Sol:

for +, += or ++ may be possible

for <, <= may be possible

for >>, >>= may be possible

∴ none.

The process of lexical analysis is done by input buffering scheme

(Q4) which of the following can be recognized as a token without depending on next symbol

a) int b) average c) char d) all e) none

(Q5 6-2000) Find no of tokens in the following 'c' code

```
printf("i=%d, &i=%x", i, &i);
```

a) 3 b) 26 c) 10 d) 21

Sol:

& is an operators

So it must be counted seperately

printf (" i = %d , &i = %x " , i , &i) ;

① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

∴ 10 tokens

Difference b/w lexeme and token:

→ lexeme is token name in the source program

→ token is of form <token-name, attribute value> which is generated by Lexical Analyzer

E.g.: position = initial + rate * 60;

For the lexeme position, token is <id, 1>

id is for identifier

1 points symbol table entry for position.

Sly for lexeme inital, the token is <id, 2>

For lexeme =, the token is <=>. It does not need any attribute value.

token of form <assign> can also be generated

Design of Lexical Analyzer:

→ We can either write a manual program for lexical analyzer or use tool (LEX tool).

Manual design:

Consider we have a token of identifier type

token ⟶ id

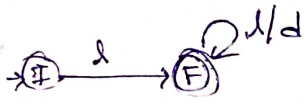
represented by ↓

pattern ⟶ starts with letter and followed by a letter or digit.

represented by ↓

Reg. Exp ⟶ $l(l|d)^*$

represented by ↓

Finite Automata ⟶ 

represented by ↓

Transition table ⟶

	l	d
→ I	F	∅
(F)	F	F

Implemented by ↓

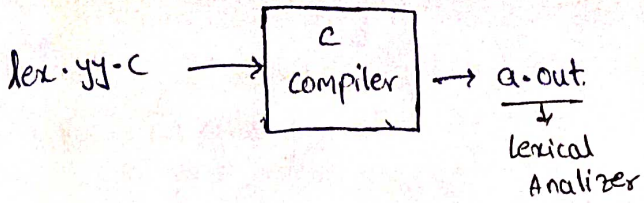
Transition function ⟶ ID() → for this we write a C function.

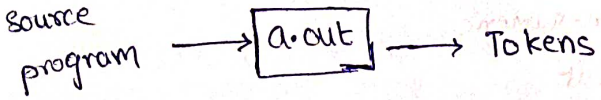
Designing with tool (not needed for gate)

compiler.l → LEX Compiler → lex.yy.c

compiler.l: here we specify token type and regular exp of it

lex.yy.c: It contains FA, its transition table and transition function





Types of Lexical errors:

(i) Unterminated Comments error:

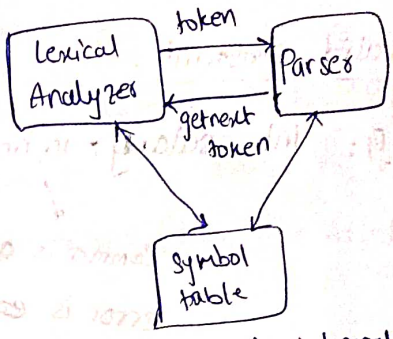
Eg:

```
/* Line1
   Line2
   Line3
```

Here comment is not closed

Hence unterminated comment error occurs.

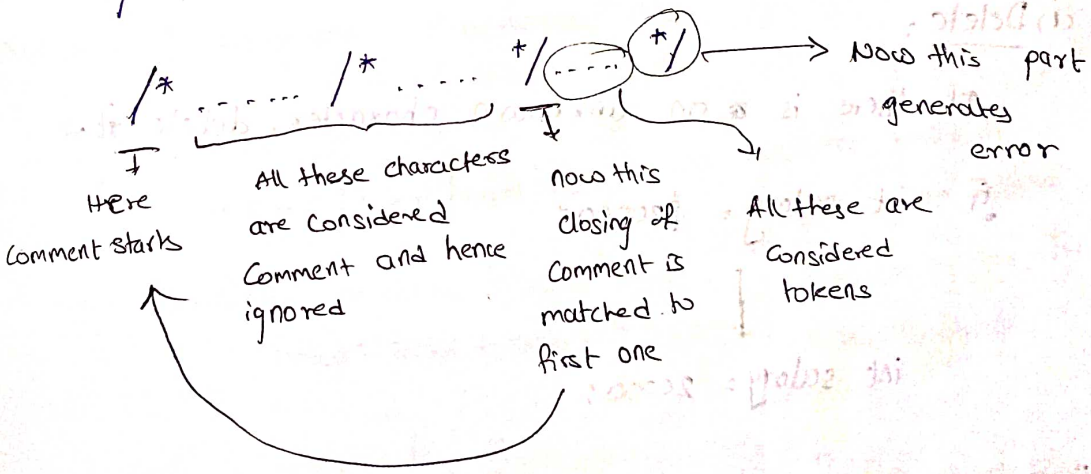
Process of lexical analysis



Thus parser and lexical analyzer run in parallel if needed. If needed all front end can be implement in one module There no definite boundary blw phase of front end

(ii) Nested Comments:

Eg:



(iii) Length of longest identifier:

Eg:

```
int interest_on_pricaple_amount_and_compound_interest;
```

generally in c programming length of identifier shouldn't exceed 31 characters. So we consider above line as error.

(iv) Invalid Identifier

Eg. int 1st salary;

reading 1 → lexical analyser assumes it to be a constant

After reading other non-numeric characters it generates error.

(v) Invalid constant:

Eg: int salary = 10,000;

Comma is operator on hence error is generated

(vi) Illegal symbols:

Eg: int salary = $20,000;

Error Recovery in L.A:

(i) Delete:

If there is an unknown character, delete it.

Eg: int salary = $20000;

↓

int salary = 20000;

(ii) Insert:

Eg: /*..... (unterminated comment)

↓ correction

/*.....*/ → where to insert is can't be found though.

③ Transpose:

Eg: int 1st salary;

(1st swapped)

↓

int s1tsalary;

④ Replace:

Eg: int salary = $20000;

↓

It is replaced by a character allowed in the language.

If no appropriate character is found then it just replace '$' with empty space (i.e., like delete).

Syntax Analysis (parser):

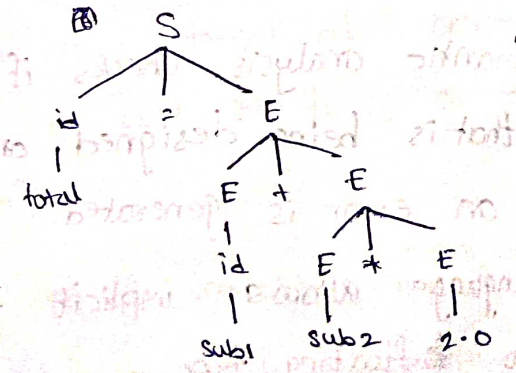
* To define the syntax of a language, CFG is used
* Consider below grammar to recognize certain mathematical assignment statements.

Eg: S → id = E

E → E+E | E*E | num | id } → grammars used in parsing are called Concrete syntax for the language

now consider deriving below stmt.

total = sub1 + sub2 * 2.0;



Derivation tree (or) Parse tree. (or) Concrete Syntax tree

* The o/p of syntax analysis is either parse tree or syntax tree.

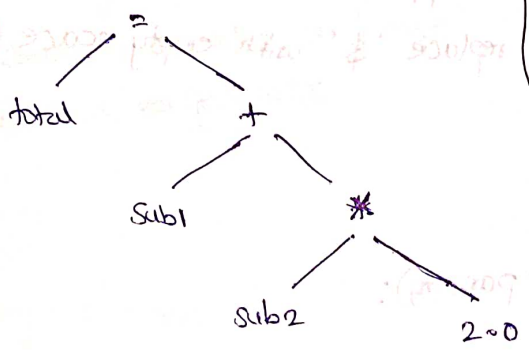
* Parse tree and syntax tree are different from each other.

Syntax tree:

It is a compressed form of parse tree.

→ In syntax tree all the operators are present at internal nodes and operands are present at leaf nodes.

The syntax tree of previous example's parse tree is as shown below



Syntax tree
(or)
Abstract syntax tree

Abstract syntax tree contains programming constructs as interior nodes, where as parse contains nonterminals

This syntax tree is given as input to semantic analysis.

Semantic Analysis:

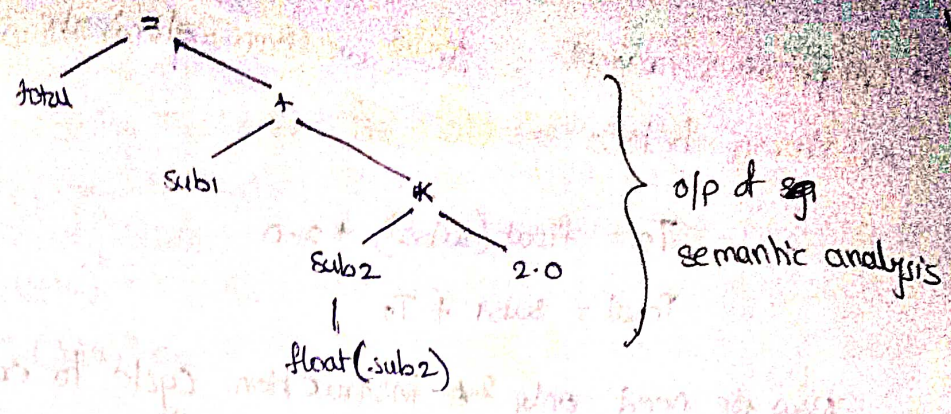
* Important component of semantic analysis is typechecking

Eg: At the bottom most level of the previous syntax tree we are multiplying sub2 with 2.0.
Assume data type of sub2 is integer.
2.0 is real number.

→ Now semantic analysis checks if this is permitted by the compiler that is being designed or not.
If not, an error is generated.

→ If the language allows implicit type conversion, sub2 is converted into floating type.

→ errors related to scope, undeclared or multiple declaration are detected at semantic analysis.



→ o/p of semantic analysis is called optimized syntax tree or ~~syntaxtic~~ semantically verified syntax tree.

→ this o/p is given as i/p to intermediate code generation phase.

Intermediate Code Generation:

* There are different ways of representing IC.
* Three Address code (TAC) is mostly used representation of IC.

TAC's statements contain atmost 3 memory addresses.

Eg: The IC generation for previous example is as shown

T_1 = float (sub2);

T_2 = T_1 * 2.0;

T_3 = sub1 + T_2

Total = T_3

Code Optimization:

optional phase

~~optimization is not there in gate~~

→ This can be performed after target code generation or before target code generation or both times or even we can leave it (i.e., no optimization).

→ Optimizing is reducing no of variables, no of instructions and some other operations without changing the logical meaning.

→ In previous example's IC we can eliminate T_3 and T_1 as shown below

$T_2 = \text{float}(sub2) * 2.0$

$Total = sub1 + T_2$

Now we need only 2 instruction cycle to execute above code. Also we have eliminate two temporary variable (T_1, T_2) which frees some memory.

optimization should balance below 3 things:
(i) faster execution
(ii) shorter code
(iii) less power consumption

Code Generation:

Code generation is not needed for gate

Here we generate object code (in this say assembly language program).

The below is assembly code for optimized IC

```
MOVF   sub2, R0
MULF   2.0, R0
MOVF   sub1, R1
ADDF   R1, R0
STORE  R0, Total
```

Later this is given as i/p to assembler.

Two Additional phases:

Symbol table is constructed by lexical, syntax & semantic analysis phases. Creating entries in symbol table is mostly done by syntax analyzer

(i) Symbol table:

It is a data structure containing a record for each identifier along with some other attributes for storing information. Procedure names are also stored in symbol table.

(ii) Error Handler:

* Errors are possible at every phase.
* After detecting an error, the phase must deal with error and compilation should not terminate and should continue the rest of the process of compilation.

Q6 / G-18

A lexical analyzer uses the following patterns to recogonize three token T_1, T_2, T_3 over the alphabet $\{a, b, c\}$.

T_1 : a?(b|c)*a

T_2 : b?(a|c)*b

T_3 : c?(b|a)*c

Note that 'x?' means 0 (or) 1 occurence of symbol x. Note also that the analyzer outputs the token that matches the longest possible prefix. If the string bbaacabc is processed by the analyzer, which one of the following is the sequence of tokens it outputs?

a) $T_1 T_2 T_3$ b) $T_1 T_1 T_3$ c) $T_2 T_1 T_3$ d) $T_3 T_3$

Sol:

a) $T_1 T_2 T_3$

bba|acab|c
T_1 T_2 T_3

b) $T_1 T_1 T_3$

bba|aca|bc
T_1 T_2 T_3

c) $T_2 T_1 T_3$

bb|a a|cab c
T_2 T_1 T_3

d) bbaac|bc
T_3 T_3

Since we need to consider longest possible prefix

∴ opt (d)

Q7 / G-20

Consider the following statements.

I. Symbol table is accessed only during lexical analysis and syntax analysis

II. Compilers for programming languages that support recursion necessarily need heap storage for memory allocation in the runtime environment

III. Errors violating the condition 'any variable must be declared before its use' are detected during syntax analysis.

Which of the above statements is/are true?

a) I only b) I and III only

c) II only d) None of I, II and III

Sol:

I: Symbol table is used in all the phases

II: Stack is enough to support recursion

III: This error is detected during semantic analysis.

∴ None

Syntax Analysis:

* This phase is also known as parsing.
* The syntactic rules are defined using a CFG.

↓

Thus syntax analyzer do only the thing the can be defined by CFG.

CFG:

$G = (V, T, P, S)$

form of production is

$A \to \alpha$ where $A \in V$

α is $(V+T)^*$

Eg: P:

$S \to AB$

$A \to bA \mid \epsilon$

$B \to c$

Now it is represented as

$G = \{\{S, A, B\}, \{b, c, \epsilon\}, P, S\}$

Let bbc be i/p to above grammar

~~$S \to AB$~~

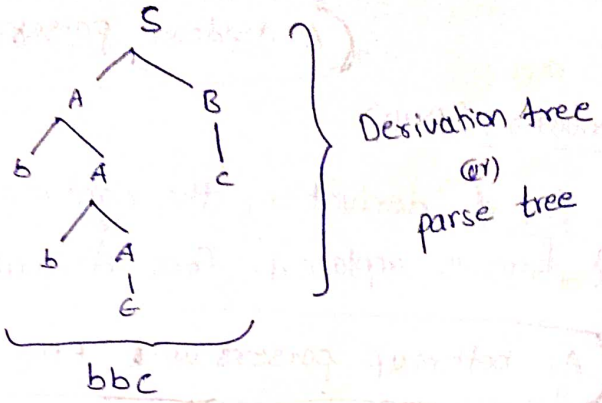
⇒

$S \rightarrow AB$
$\rightarrow bAB$
$\rightarrow bAc$
$\rightarrow bbAc$
$\rightarrow bbc$

} Derivation

Here derived string bbc is called sentence.

we can represent the derivation pictorially by derivation tree (Parse tree)



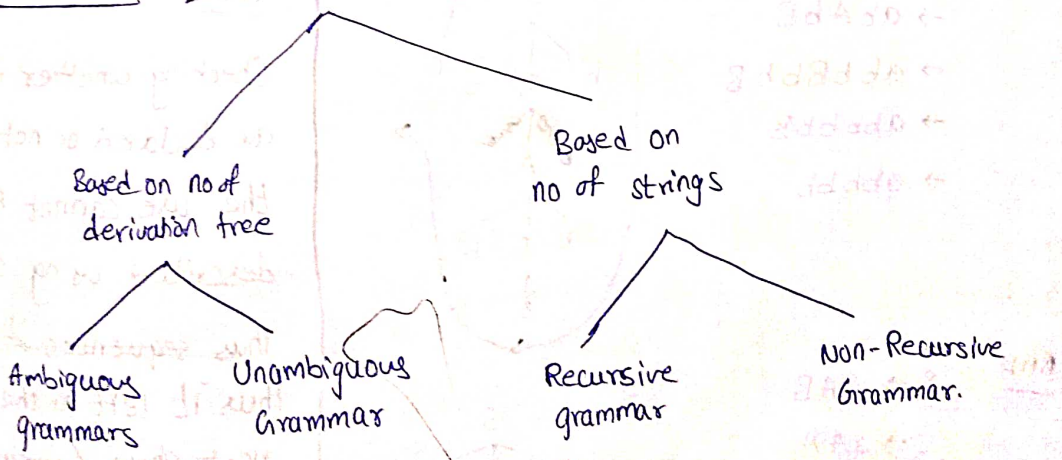
Derivation:

Derivation provides a means of generating the sentences of a language.

Derivation tree (or) Parse tree:

A parse tree is an equivalent form of showing a derivation which represents derivation graphically or pictorially

Classification of CFG:



Ambiguous Grammars: An ambiguous grammar is one that produces more than one left most derivation or more than one right most derivation for some sentence of a language.

Types of derivations:

There are two types of derivations:

(i) **Left most derivation (LMD)**

In each step of derivation, the left most non-terminal in the sentential form is replaced. This derivation is called LMD.

All top-down parsers use LMD

(ii) **Right most derivation (RMD):**

In each step of derivation, the right-most non-terminal in the sentential form is replaced. This derivation is called RMD.

All bottom up parsers used "RMD in reverse"

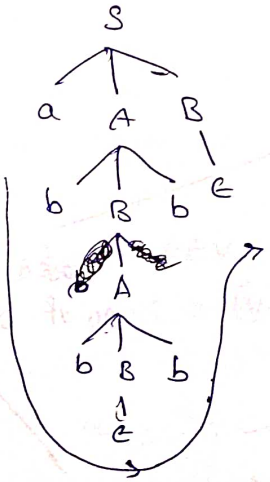
Eg: $S \rightarrow aAB$
$A \rightarrow bBb$
$B \rightarrow A|\epsilon$

i/p: abbbb

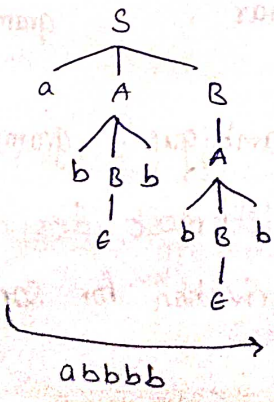
Construct LMD & RMD for above string.

LMD:

$S \rightarrow aAB$
$\rightarrow abBbB$
$\rightarrow abAbB$
$\rightarrow abbBbbB$
$\rightarrow abbbbB$
$\rightarrow abbbb$



RMD: $S \rightarrow aAB$
$\rightarrow aAA$
$\rightarrow aAbBb$
$\rightarrow aAbb$
$\rightarrow abBbbb$
$\rightarrow abbbb$



Note:

Checking whether identifiers are declared or not before the use cannot be described using CFG. Thus ~~sequences of~~ to Thus it left to the next phase (semantic analysis). Thus sequence of token accepted by a parser forms a superset of the programming language.

(Q8) Find LMD & RMD for the following grammar

$S \to aB \mid bA$

$A \to a \mid as \mid bAA$

$B \to b \mid bS \mid aBB$

I/P : aaabbabbba

LMD :

$S \to aB$
$\to aaBB$
$\to aaaBBB$
$\to aaabBB$
$\to aaabbB$
~~$\to aaabbbS$~~
~~$\to aaabbba$~~
$\to aaabbaBB$
$\to aaabbabB$
$\to aaabbabbS$
$\to aaabbabbbA$
$\to aaabbabbba$

RMD :

$S \to aB$
$\to aaBB$
$\to aaBaBB$
$\to aaBaBbS$
$\to aaBaBbbA$
$\to aaBaBbba$
$\to aaBabbba$
$\to aaaBBabba$
$\to aaaBbabba$
$\to aaabbabba$

(Q9) Consider the following grammar

$E \to E+E \mid E*E \mid id$

Check whether the grammar is ambiguous or not

Sol:
Consider string id + id * id

LMD :

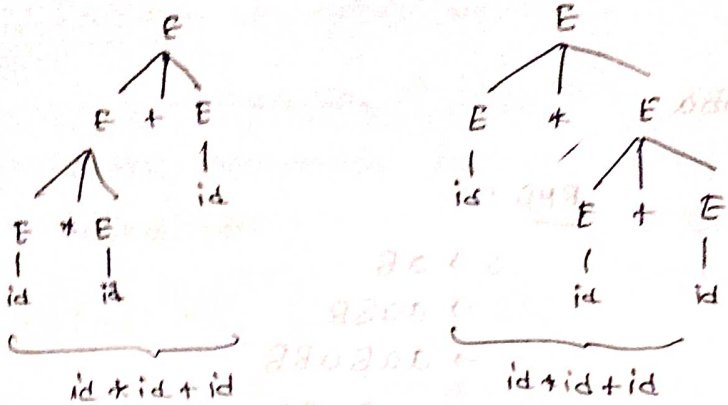
$E \to E+E$
$\to id+E$
$\to id+E*E$
$\to id+id*E$
$\to id+id*E$

$E \to E*E$
$\to E+E*E$
$\to id+E*E$
$\to id+id*E$
$\to id+id*id$

we have 2 left most derivations for above string

∴ Ambiguous.

Generally for this type of questions try to construct two different parse tree for same input.



Observation:

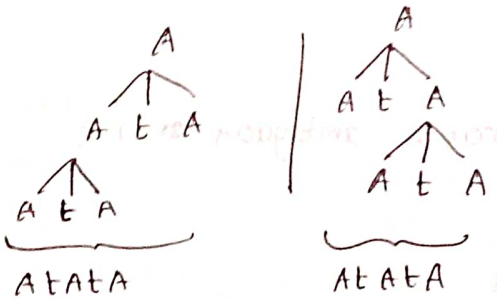
If we have a production of form

$A \rightarrow AtA$, t is terminal

then the grammar is ambiguous.

These productions are called ambigous productions.

This ambigous because we can expand left A once and right A once.



(I) 2M

Q.10 Which of the following terminal strings has more than one parse tree when parsed according to the grammar below. 4M

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

a) id + id + id + id

b) id + (id * (id * id))

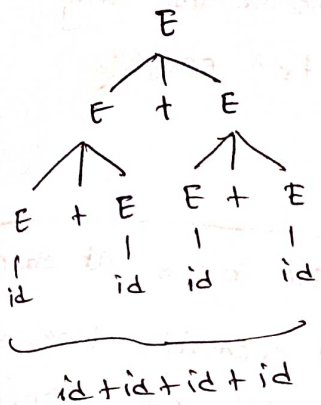
c) (id * (id * id)) + id

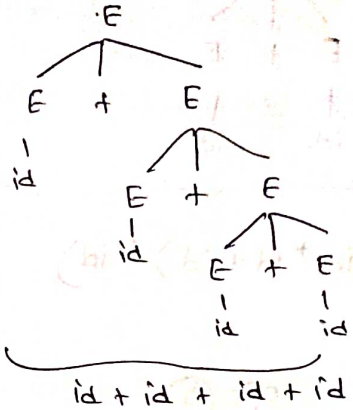
d) ((id * id + id) * id)

For the correct string with more than one parse tree obtained as solution to the above question how many parse trees are possible?

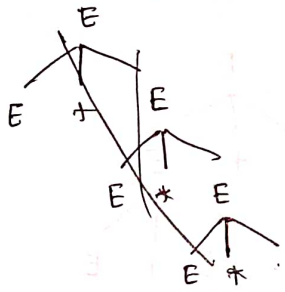
Sol:

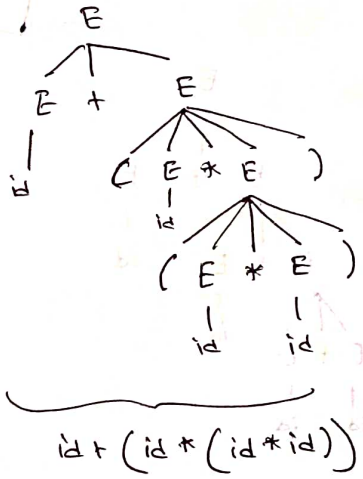
1) a) id + id + id + id





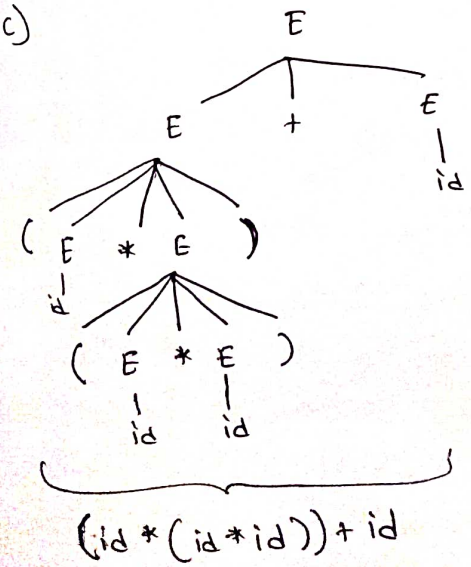
b)





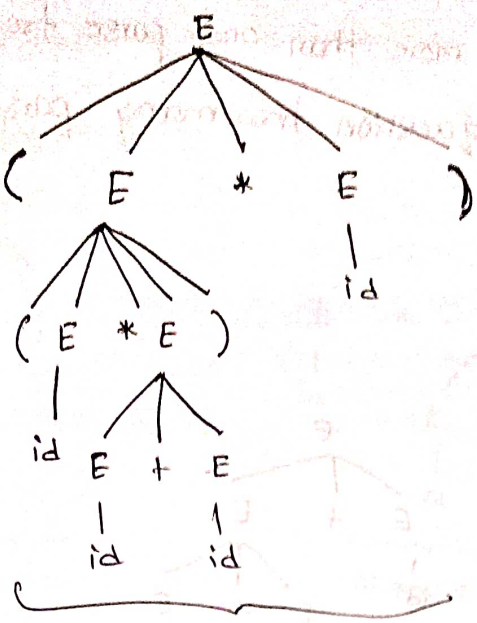
This is the only derivation tree possible

c)



This is the only tree possible

d)



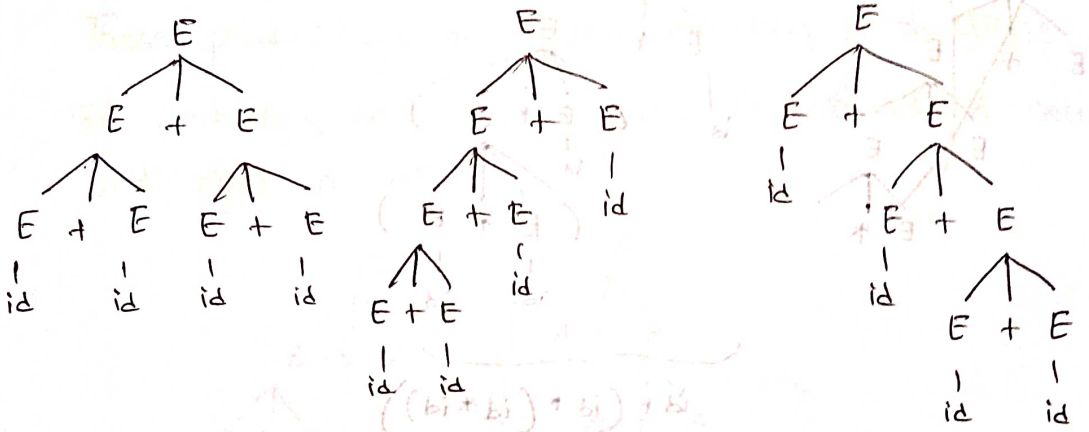
((id * id + id) * id)

This is the only tree possible

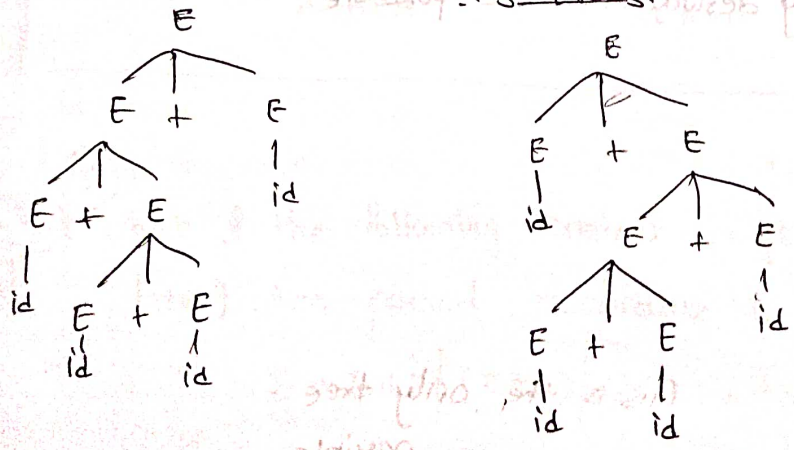
∴ opt (a)

For option b, c, d at each level of expanding tree, we can find only one option (way) ∴ they have only one parse tree

II)



~~3 trees~~



(011) Consider the following grammer

$S \to aS/A$

$A \to aAb/bAa/\epsilon$

(i) which of the following string is generated by the above grammar?

a) aabbaba b) aabaaaba c) ababbb d) aabbaa

(ii) For the correct answer to the above question, How many steps are required to derive the string and how many parse trees are possible

a) 6 & 1 b) 6 & 2 c) 7 & 2 d) 4 & 2

Sol:

(i) d) aabbaa

observing grammar we can say it accepts

any no of a's followed by aAb

bAa

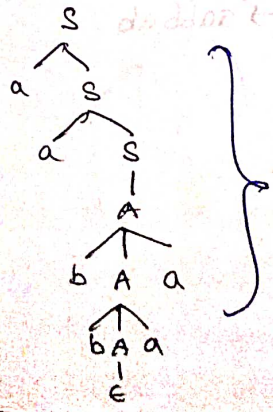
only opt (d) is of this form

∴ opt-(d)

(ii) d : aabbaa

$S \to aS$
$\to aaS$
$\to aaA$
$\to aabAa$
$\to aabbAaa$
$\to aabbaa$

} 6 steps



At each level of tree's expansion we can find only one possibility

∴ only 1 tree

∴ 6 & 1

Q12 / G-07 Consider the CFG with $\{S, A, B\}$ as the non-terminal alphabet, $\{a, b\}$ as the terminal alphabet. S as the start symbol and the following set of production rules

$S \to aB$ $\quad S \to bA$

$B \to b$ $\quad A \to a$

$B \to bS$ $\quad A \to aS$

$B \to aBB$ $\quad S \to bAA$

(i) which of the following strings is generated by the grammar?

a) aaaabb b) aabbbb c) aabbab d) abbbba

(ii) for the correct answer to above question, how many derivation trees are there

a) 1 b) 2 c) 3 d) 4

Sol:

i) $S \to aB \mid bA \mid bAA$

$A \to a \mid aS$

$B \to b \mid bS \mid aBB$

a) aaaabb

$S \to aB$

$\to aaBB$

$\to aaaBBB$

$\to aaaaBBBB$

α

b) aabbbb

$S \to aB$

$\to aaBB$

$\to aabSB$

$\to aabSbS$

~~$\to aabbbA$~~

$\to aabbbAbA$

α

c) aabbab

$S \to aB$

$\to aaBB$

$\to aabB$

$\to aabbS$

$\to aabbaB$

$\to aabbab$

$\therefore$ opt (c)

(ii) Now we check for different possibilities of derivation

$S \to aB$

$\to aaBB$

$\to aabB$ — $aabSB$ — $aabbAAB$

$\to aabbS$ — $aabbAB$ — $aabbaAB$

$\to aabbaB$ — $aabbaB$

$\to aabbab$ — $aabbab$

$\therefore$ 2 trees.

Note:

→ No algorithm exists to verify the ambiguity of CFG.

→ No algorithm exists to convert ambiguous grammar into unambiguous.

→ By using trail and error method some of the ambiguous grammars can be converted to unambiguous grammars.

Eg: ① $E \to E + E \mid id$ is ambiguous grammar.

⇓

② $E \to E + T \mid T$

$T \to id$

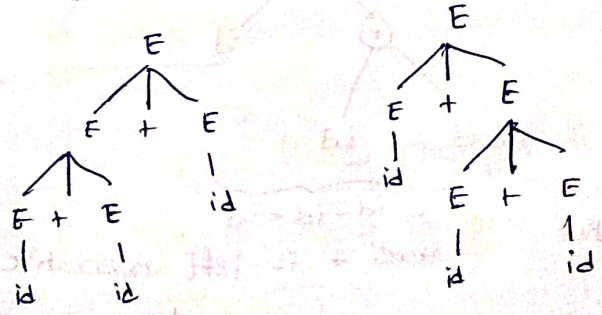
(or)

③ $E \to id + E \mid id$

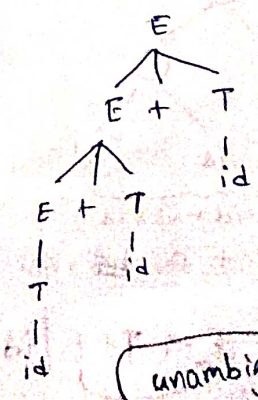
} unambiguous grammar

Consider $id + id + id$

deriving with grammar ①



deriving with ②



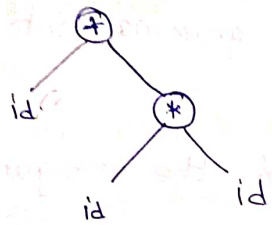
unambiguous

→ By redefing the grammar using precedence and associativity we can convert some of the ambiguous grammars can be ~~can~~ converted in unambiguous.

→ Some of the ambiguous can never be converted into unambiguous grammar. Such grammars are called inherently unambiguous grammar.

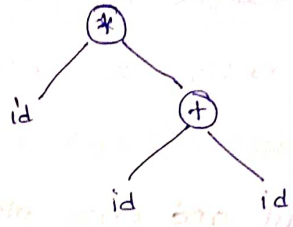
Converting ambiguous grammars into unambiguous using precedence and associativity:

→



id + id * id

P(*) > P(+)



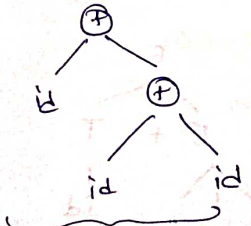
id * id + id

P(+) > P(*)

P-Precedence

* The operator that appears at the bottom of the tree has higher precedence.
* The operator that appears at the top of the tree has lower precedence

→

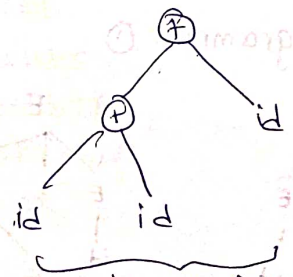


id + id + id

Here + is right associative

E → T + E | id

right most symbol of RHS is equal L.H.S



id + id + id

Here + is left associative

E → E + T | id

leftmost symbol of R.H.S is equal to L.H.S.

(13) Convert the following ambiguous grammars into unambiguous using precedence and associativity.

(i) $E \to E + E \mid E * E \mid id$

precedence: $* > +$

both are left associative

Sol:

'+' w.r.to E & T:

since + has least precedence, it should appear on the top of tree

i.e., '+' should be derived first.

$\therefore E \to E + E$

Since '+' is left associative

$E \to E + T \mid T$

'*' w.r.to T & F:

precendence of * is higher than +.

So now we write production for deriving *

$\therefore T \to T * T$

But '*' is left associative

$\therefore T \to T * F \mid F$

'id' w.r.to F:

E is finally changed to T

T is finally changed to F

$\therefore$ we add production

$F \to id$

∴ The modified grammar is

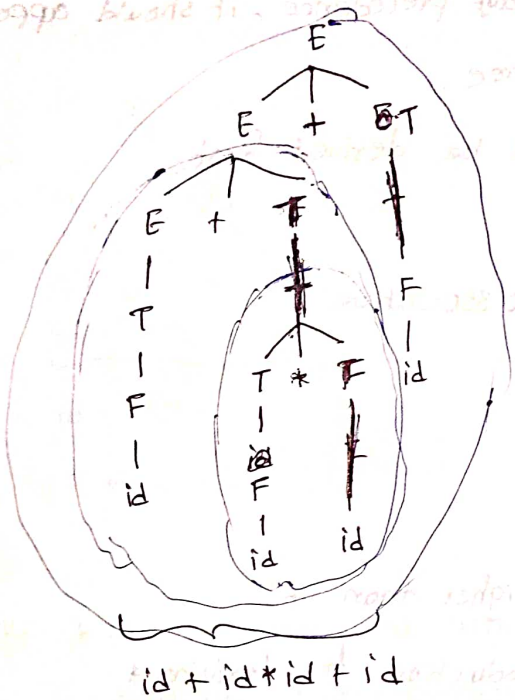
$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id$

Now this grammar obtained by considering precendence and associativity is unambiguous.

Consider the string id + id * id + id



id + id * id + id

Here the expression evaluates to

$id + (id * id) + id$

1st

2nd

3rd

(ii) $E \to E+E \mid E*E \mid E-E \mid E \div E \mid E \wedge E \mid id$

Precedence : $\wedge > (* = \div) > (+ = -)$

Associativity : $\wedge$ — right associative

$+, -, *, \div$ — left associative

+, - should be on top of tree

$$E \to E+E \mid E-E$$

Associativity is left

$$\therefore E \to E+T \mid E-T \mid T$$

*, ÷ should be next

$$T \to T*T \mid T \div T$$

Considering associativity

$$T \to T*F \mid T \div F \mid F$$

^ should be next

$$F \to F \wedge F$$

associativity is right

$$F \to G \wedge F \mid G$$

Next we have id

$$\therefore G \to id$$

So the required grammar is

$$E \to E+T \mid E-T \mid T$$
$$T \to T*F \mid T \div F \mid F$$
$$F \to G \wedge F \mid G$$
$$G \to id$$

(Q14) Write precedence and associativity of the operators for the following grammar

(i) $E \to E*F \mid F$
$F \to F \wedge T \mid T$
$T \to P+T \mid P$
$P = id$

Precedence : (+) > (^) > (*)

Associativity : * - left
^ - left
+ - right

(ii) A → @ A @ B | B

B → C # B | C ^ B | C

C → C $ D | C + D | D

D → E - D | E * D | E

E → id

Precedence : (- , *) > ($, +) > (#, ^) > (@)

Associativity : @ - left
- ~~left~~ right
^ - ~~left~~ right
$, + - left
-, * - right

Q15 / G-

E → E * F | F + E | F

F → ~~E~~ F - F | id

Choose the correct answer based on precedence

a) * > + b) - > * c) + = - d) + > *

Sol:

~~opt (a)~~

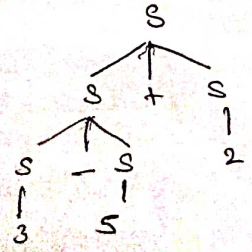
P(*) = P(+)

P(-) > P(*)

∴ opt (b)

Q16 / G- ★

Consider the following parse tree



which of the following based on precedence

a) (+) = (-) b) (+) > (-)

c) (-) > (+) d) none

Here question is according to the parse tree

and it is clear that '-' is being evaluated 1st

$\therefore (-) > (+)$

$\therefore$ opt (c)

Q17 Consider the following grammar

$S \rightarrow S+S \mid S*S \mid id$

Choose correct option on precedence

a) $(+) > *$ b) $* > +$ c) $+ = *$ d) none

Note:

→ A production which is both left and right associative is said to be ambiguous grammar

Eg: $S \rightarrow S+S$ (right associative, left associative)

Q18 Consider the following left associative operators in decresing order of precedence $-, *, \$$(exponent)

what is the result of following expression

$3-2*4\$1*2\$3$

Sol:

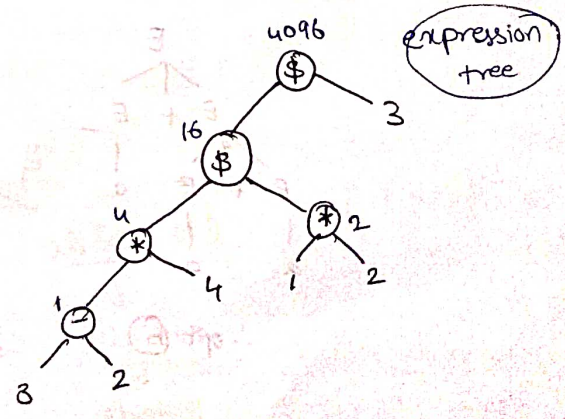
$3-2*4\$1*2\$3$

$1*4\$1*2\$3$

$4\$1*2\$3$

$4\$2\$3$

$16\$3 = 4096$



expression tree

Q19 / G-15 / 2M

The operators in some programming language are given below

+	high precedence	left associative
−	medium precedence	Right associative
*	low precedence	left associative

the value of the expression 2-5+1-7*3 in this language is ____

sol:

2 - 5 + 1 - 7 * 3

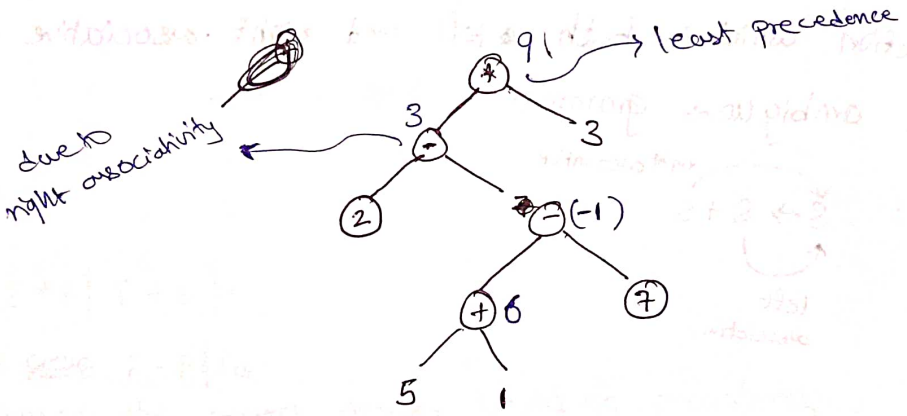
2 - 6 - 7 * 3

2 - (-1) * 3 (∵ right associative)

3 * 3

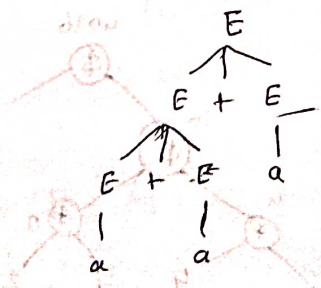
9

Solving by expression tree



Material problems on parsing techniques:

P/1 Since it is given that it is left associative



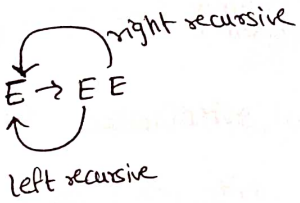
∴ opt (b)

Q2 since we have 'ε' we have infinite derivations

$S \to SS$	$S \to SS$
$\to SSS$	$\to SSS$
$\to aSS$	$\to SSSS$
$\to aaS$	$\to aSSS$
$\to aaa$	$\to aaSS$
	$\to aaaS$
	$\to aaa$

$\therefore \infty$

Q3

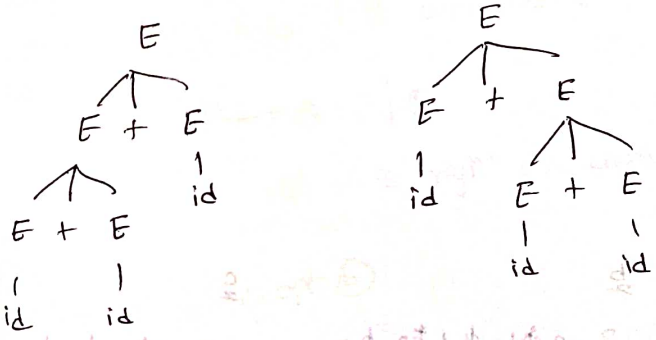


$\therefore$ ambiguous

Q4 observing grammar we can say for 'a' at right end we have 'b' at left end and viceversa. And in middle we have b.

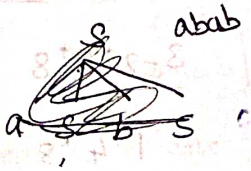
This satisfied by opt (d)

Q5



$\therefore$ 2 trees.

Q6



$S \to aSbS$

abS → $abaSbS$ → $ababS$ → $abab$

$abSaSbS$ → $abaSbS$ → $ababS$ → $abab$

$\therefore$ 2 derivations

$\therefore$ 2 parse trees.

P/9

$E \to E \uparrow T \mid T$

$T \to F + T \mid F$

$F \to i$

precedence

too
↑ - left
\+ - right
low

precedence : $+ > \uparrow$

∴ opt (d)

P/10

$P(+) > P(\uparrow)$

∴ opt (c)

P/11

$[P(*) = P(+)] < P(-)$

∴ opt (b)

P/12

$3 - 2 * 4 \$ 2 * 3 \$ 2$

$1 * 4 \$ 2 * 3 \$ 2$

$1 * 16 * 3 \$ 2$ (∵ $ is left associative)

$1 * 16 * 9$

$16 * 9$

144

P/13

a)

$3 - 2 * 4 \$ 1 * 2 \$ 3$

$1 * 4 \$ 1 * 2 \$ 3$

$4 \$ 1 * 2 \$ 3$

$4 \$ 2 \$ 3$

$16 \$ 3$

$a = 4096$

$= 2^{12}$

b)

$3 - 2 * 4 \$ 1 * 2 \$ 3$

$1 * 4 \$ 1 * 2 \$ 3$

$1 * 4 \$ 2 \$ 3$

$4 \$ 2 \$ 3$

$4 \$ 8$

$b = (2^2)^8 = 2^{16}$

c)

$3 - 2 * 4 \$ 1 * 2 \$ 3$

$3 - 2 * 4 * 2 \$ 3$

$3 - 2 * 4 * 8$

$1 * 4 * 8$

$4 * 8$

$c = 32$

∴ $c < a < b$

Q20 / GATE-07

Consider the grammar below with two operators * and +

$S \rightarrow T * P$

$T \rightarrow U \mid T * U$

$P \rightarrow Q + P \mid Q$

$Q \rightarrow id$

$U \rightarrow id$

which one of the following is TRUE?

A) + is left associative, while * is right associative

B) + is right associative, while * is left associative

C) Both + and * are right associative

D) Both + and * are left associative

Sol:

$P(+) > P(*)$

$T \rightarrow T * U \mid U$

$\therefore$ * is left associative

$P \rightarrow Q + P \mid Q$

$\therefore$ + is right associative

$\therefore$ opt (b)

Q21 / GATE

If G is a grammar with productions

$S \rightarrow SaS \mid aSb \mid bSa \mid SS \mid \epsilon$

where S is the start symbol, then which one of the following strings is not generated by G?

a) abab b) aaaab c) abbaa d) babba

a) abab

S → SS
→ asbS
→ abS
→ abasb
→ abab

b) aaab

S → SS
→ SSS
→ SaSSS
→ SaSaSS
~~→ SaaSS~~
→ aSaSS
→ aaSS
→ aaS
→ aaaSb
→ aaab

∴ opt (d)

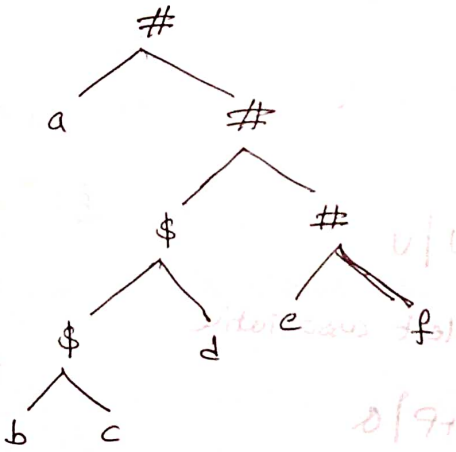
c) abbaa

S → SS
→ asbS
→ abS
→ abbSa
→ abbSaSa
→ abbasa
→ abbaa

d) babba

S → bSa
→ ⊘
This has more b's than a's
∴ not possible

(Q22 / G) Consider the following parse tree for the expression a#bcd#e#f involving two binary operator $ and #



which one of the following is correct for the given parse tree?

a) $ has higher precedence and is left associative; # is right associative.

b) # has higher precedence and is left associative; $ is right associative

c) $ has higher precedence and is left associative; # is ~~right~~ left associative

d) # has higher precedence and is right associative; $ is left associative

Sol:

The expression is $a \# [((b \$ c) \$ d) \# (e \# f)]$

$ is at bottom

∴ $ has higher precedence than #

also in bcd

b$c is evaluated first

∴ $ is left associative

From expression it is clear the # is right associative

∴ opt (a)

20/06/20

CFG based on number of strings:

1) Non-Recursive grammar:

A grammar which generates finite no of strings is said to be non-recursive grammar.

Eg: S → Aa | Bb

A → c | e

B → d | f

ca, ea, db, fb are only string generated by the grammar.

∴ It is non-recursive grammar.

2) Recursive Grammar:

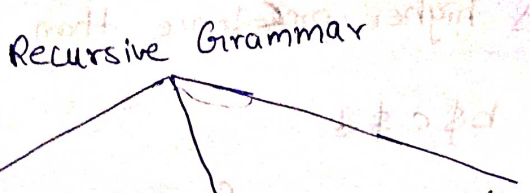
A grammar which generates infinite no of strings is called recursive grammar.

Eg: S → Sa | b

The grammar generates b, ba, baa, baaa....

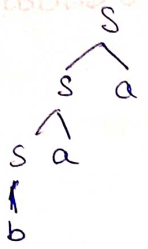
∴ Recursive grammar.

Classification of Recursive grammars :



Left Recursive grammar

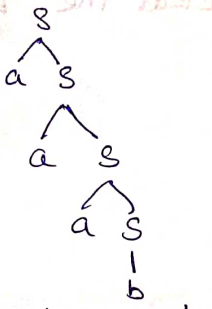
* $S \rightarrow Sa|b$



* Here tree expands on the ~~right~~ left side

Right recursive grammar

* $S \rightarrow aS|b$

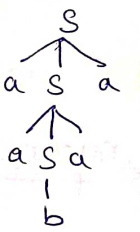


* Here tree expands on the right side

General Recursive grammar

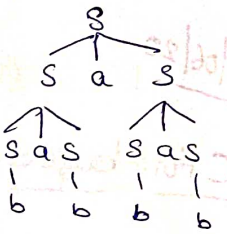
→ It is a grammar which neither left recursive nor right ~~recursive~~ recursive or both.

Eg: $S \rightarrow aSa|b$



neither left nor right

$S \rightarrow SaS|b$



both left and right

Note: A grammar which is both left & right recursive is said to be ambiguous grammar

→ Non-Recursive grammars are not used in the programming languages because we need to generate infinite string in programming language.

→ ∴ Recursive languages are suitable for programming language.

→ ~~Thoughh~~ It is not that all recursive languages are suitable for every parsing technique.

→ If a recursive ~~Langua~~ grammar is not suitable we need to make appropriate modifications on the grammar.

Modifying the Grammar:

① Left recursive:

A grammar is left recursive, if the leftmost symbol in the RHS is same non terminal as that in the LHS.

(or)

A grammar is said to be left recursive if it has a non-terminal 'A' with production $A \to A\alpha/\beta$ for $\alpha = (VUT)^*$

* Consider the production

$$A \to A\alpha / \beta$$

$A \to A\alpha$
$\to A\alpha\alpha$
$\to A\alpha\alpha\alpha$
$\to A\alpha\alpha\alpha \cdots \alpha$
$\to \beta\alpha\alpha\alpha \cdots \alpha$ → Here A' generat $\alpha^n, n \geq 0$
$\to \beta A'$

The left recursion is removed by adding new non-terminal A' as follows

$$A \to A\alpha/\beta \equiv \begin{matrix} A \to \beta A' \\ A' \to \alpha A'/\epsilon \end{matrix}$$

A grammar is said to be left recursive if it has atleast one non-terminal from whose derivation we can obtain a sentential form such that the leftmost symbol of the sentential form is same as that of L.H.S

$A \xrightarrow{*} A\alpha$

α is $(VUT)^*$

$\xrightarrow{*}$ mean series of steps in derivation

Eg: Eliminate left recursion from below grammar

$E \to E+T/T$
$T \to T*F/F$
$F \to id$

Sol:

$E \to E+T/T$

$\Rightarrow E \to TE'$
$E' \to +TE'/\epsilon$

$T \to T*F/F$

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

$F \to id$ is not recursive

∴ Required grammar is

$E \to TE'$

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

$T \to FT'$

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

$F \to id$

(Q23) Eliminate left recursion from the below grammar

$S \to Sa \mid b \mid d$

$S \to bS' \mid dS'$

$S' \to aS' \mid \epsilon$

$A \to A\alpha_1 \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \beta_m$

⇓ Eliminating left recursion

$A \to \beta_1 A' \mid \beta_2 A' \mid \dots \beta_m A'$

$A' \to \alpha_1 A' \mid \epsilon$

~~$A \to \alpha_1 A$~~

$A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \beta_m$

⇓

$A \to \beta_1 A' \mid \beta_2 A' \mid \dots \beta_m A'$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \alpha_n A' \mid \epsilon$

(Q24) Eliminate left recursion from below grammar

$A \to Aa \mid Ab \mid c \mid d$

⇓ eliminating left recursion

$A \to cA' \mid dA'$

$A' \to aA' \mid bA' \mid \epsilon$

Indirect Left recursion:

Consider the grammar

S → Aa | b

& A → Sd | e

Consider derivation

S → Aa → Sda

Indirect left recursion.

Eg: A → Ba

B → Cda | a

C → Ae

Also has indirect left recursion

A → Ba → Cdaa → Aeaa

In this case try to draw parse tree thinking parser for string aeaa and see how left recursion effects

→ Direct left recursion is also called immediate left recursion.

→ Indirect left recursion should also be eliminated if needed. for topdown parsers.

Eliminating indirect left recursion

→ We first convert indirect left recursion to direct left recursion.

Then we eliminate it.

Eg: Consider the below grammar

S → Aa | b

A → Ac | Sd | e

Here S has indirect left recursion

A has direct & indirect left recursion

Eliminating indirect left recursion of S

S → Aa | b

A → Ac | Aad | bd | e

↳ elimination of indirect left recursion of S.

With this indirect left recursion of A is also eliminated.

Now we remove direct left recursion of A

$S \to Aa \mid b$

$A \to bdA' \mid eA'$

$A' \to cA' \mid adA' \mid e$

Thus the above grammar is now free from both direct and indirect left recursion.

(Q25) Eliminate left recursion from below grammar.

$S \to Aa \mid b$

$A \to Bd \mid e$

$B \to Sf \mid g$

Sol:

$S \to Aa \to Bda \to Sfda$

$\therefore$ S has indirect left recursion

$A \to Bd \to Sfd \to Aafd$

$\therefore$ A has indirect left recursion

$B \to Sf \to Aaf \to Bdaf$

$\therefore$ B has indirect left recursion

Eliminating indirect left recursion of S

$S \to Aa \mid b$

$A \to Bd \mid e$

$B \to Aaf \mid bf \mid g$

$A \to Bd \to Aafd$

Eliminating indirect left recursion of A.

$S \to Aa \mid b$

$A \to Bd \mid e$

$B \to Bdaf \mid eaf \mid bf \mid g$

Now we need to eliminate direct left recursion of B

$S \to Aa|b$

$A \to Bd|\epsilon$

$B \to eaB'|bfB'|gB'$

$B' \to dafB'|\epsilon$

Note: Here all we did is ~~meeting~~ making B direct recursive and eliminating it.

~~Similla~~ Similarly we can do the same for S or A.

Thus we don't have unique answer for this kind of question.

8.26 / G-16 Which of the following grammers is free from left recursion.

a) $S \to AB$
$A \to Aa|b$
$B \to C$

b) $S \to Ab|Bb|c$
$A \to Bd|\epsilon$
$B \to e$

c) $S \to Aa|B$
$A \to Bb|Sc|\epsilon$
$B \to d$

d) $S \to Aa|Bb|\epsilon$
$A \to Bd|\epsilon$
$B \to Ae|\epsilon$

Sol:

a) ~~S~~ $A \to Aa$

∴ left recursive

b) $S \to Ab \longrightarrow Bdb$
$\searrow Bb \to eb$

∴ no left recursion

c) $S \to Aa$ ~~B~~ $\to Sca$

∴ left recursive

d) ~~$S \to Aa \to Bda \to Ae$~~
$A \to Bd \to Aed$

∴ left recursive

∴ opt (b)

Q27 / G-17 Consider the following expression grammar G1:

$E \to E-T \mid T$

$T \to T+F \mid F$

$F \to (E) \mid id$

which of the following grammars is not left recursive, but is equivalent to G1?

a) $E \to E-T \mid T$

$T \to T+F \mid F$

$F \to (E) \mid id$

b) $E \to TE'$

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

$T \to T+F \mid F$

$F \to (E) \mid id$

c) $E \to TX$

$X \to -TX \mid \epsilon$

$T \to FY$

$Y \to +FY \mid \epsilon$

$F \to (E) \mid id$

d) $E \to TX \mid (TX)$

$X \to -TX \mid +TX \mid \epsilon$

$T \to id$

Sol:

opt (a) & (b) has direct left recursion

So they can be eliminated.

c) $E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

But in question X is used as E'

Y is used as T'

∴ opt (c) is correct

d) Though the grammar has no left recursion, it produces different grammar.

Material problems

(P/7)

$A \to Sa \mid b$

$S \to Sc \mid Ad$

$\Downarrow$

$A \to Sa \mid b$

$S \to Sc \mid Sad \mid bd$

$\Downarrow$

$A \to Sa \mid b$

$S \to bdS'$

$S' \to cS' \mid adS' \mid \epsilon$

opt (a)

(P/8) same as (Q17)

Left Factoring :

It is a process which isolates the common parts of two or more productions into a single production.

Any production of form

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n$$

→ Here if string starts with 'α' the parser doesn't know which of derivations to choose. So we take α as common an create new non-terminal. This process is called factoring.

can be modified as

$$A \to \alpha A'$$

$$A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

→ Left factoring helps parser make accurate decisions and reduces ambiguity in taking decision.

→ Also left factoring reduces backtracking.

Eg: Left factor the following grammar:

$S \to iEtS \mid iEtSeS \mid a$

$E \to b$

Sol:

iEtS is common

$S \to iEtSS' \mid a$

$S' \to eS \mid \epsilon$

$E \to b$

Eg: $A \to abB \mid abC \mid ad$ (or)

$A \to aA'$

$A' \to bB \mid bC \mid d$

$A \to aA'$

$A' \to bA'' \mid d$

$A'' \to B \mid \epsilon$

$A \to abA' \mid ad$

$A' \to B \mid C$

$A \to aA''$

$A'' \to bA' \mid d$

$A' \to B \mid C$

Parsing:

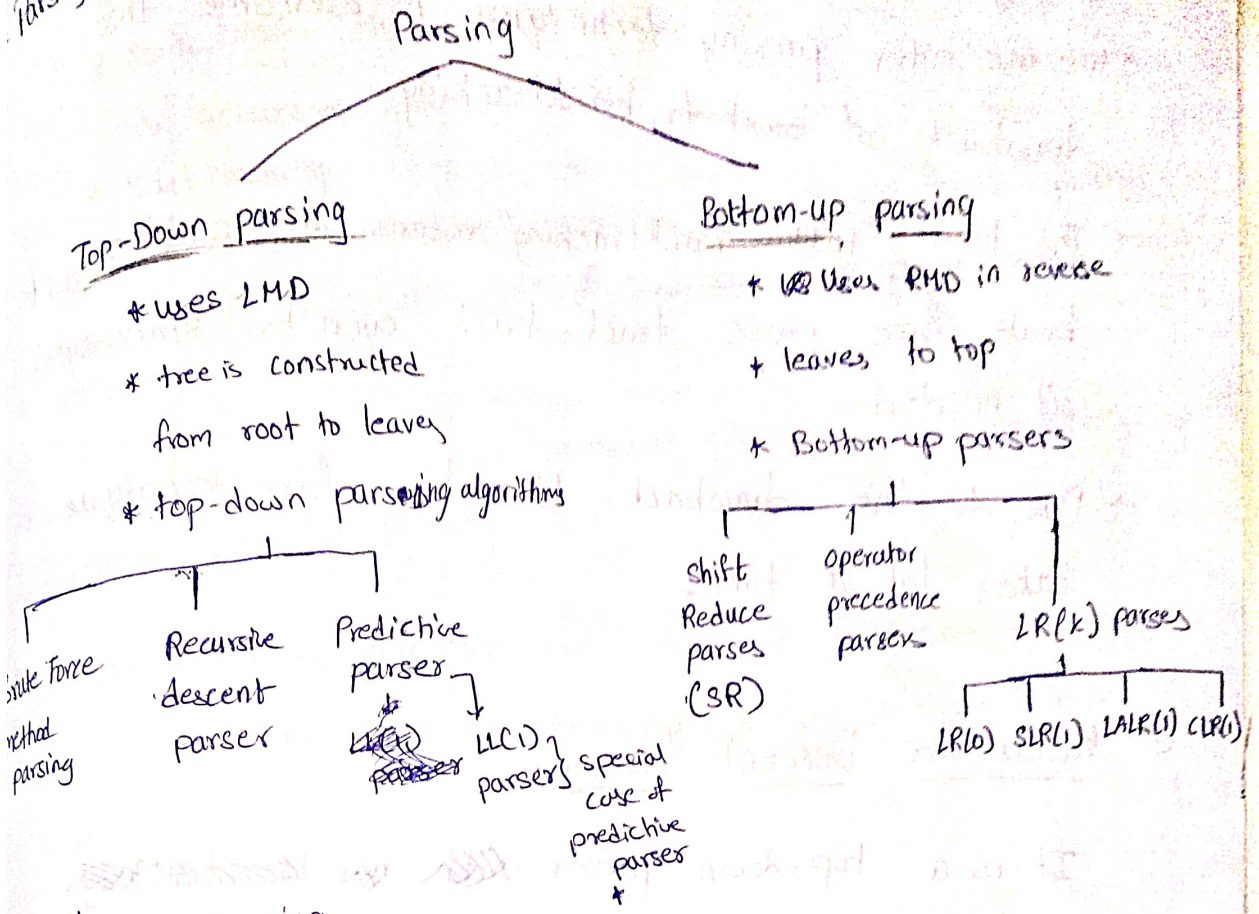
parser: Parser is program for a given CFG 'G' which take a string ω as input and outputs

(i) parse tree if $\omega \in L(G)$

(ii) error message if $\omega \notin L(G)$

This process is known as parsing.

Parsing techniques are of two types:



Top-down parsing:

A top-down parser constructs a parse tree starting from starting non-terminal symbol (root) to input string (leaves)

→ During this process Leftmost derivation is used.

Brute Force Method: It is a topdown parser with full backtracking

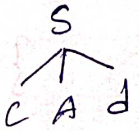
Eg: Consider the grammar

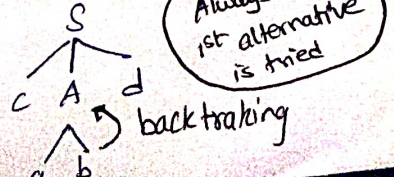
$S \rightarrow cAd$

$A \rightarrow ab \mid a$

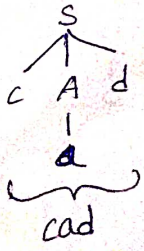
Below is topdown parsing of string "cad"

steps: (i) S

(ii)


(iii)


(iv)


(v)


drawbacks: ① Backtracking

→ We use other parsing techniques to overcome this drawback of ~~back-tr~~ backtracking.

→ The term "full backtracking" means if needed the brute force parse tracks back even to start symbol if needed.

→ Due to this drawback, the brute force technique takes lot of time.

② Recursive Descent Parser:

It is a top-down parser ~~with no backtracking~~.

* It is done by representing each production rule by a function call (with if else or switch case ~~stmts~~)

Eg: for the below grammar

$S \rightarrow aA$

$A \rightarrow b$

we write functions as

```
S()
{
    if (ip[i] == 'a')
    {
        i++;
        A();
    }
    else
        error();
}
```

```
A()
{
    if(ip[i] == b)
        accept();
    else
        error();
}
```

```
main()
{
    S();
}
```

Drawback: Recursive Function Calls

* Sometimes there is a chance for infinite loop due to recursive function calls.
* Backtracking.

→ Also Recursive descent parser and brute force parsers only a small class of grammars.
parse

→ A simple form of recursive descent parsing without backtracking is called predictive parsing.

→ A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop.