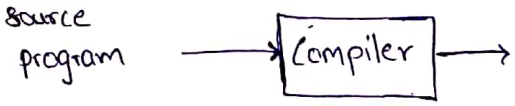
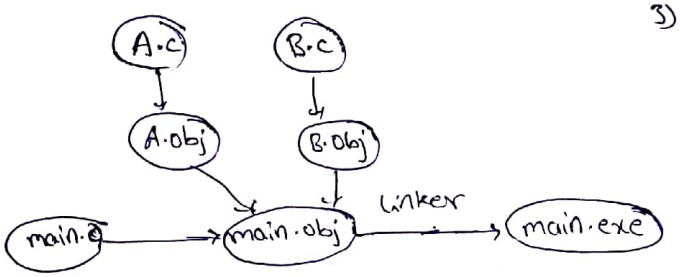
16/06/20

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

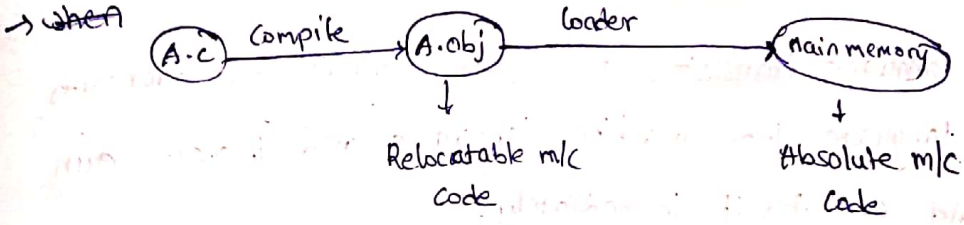
Introduction:

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

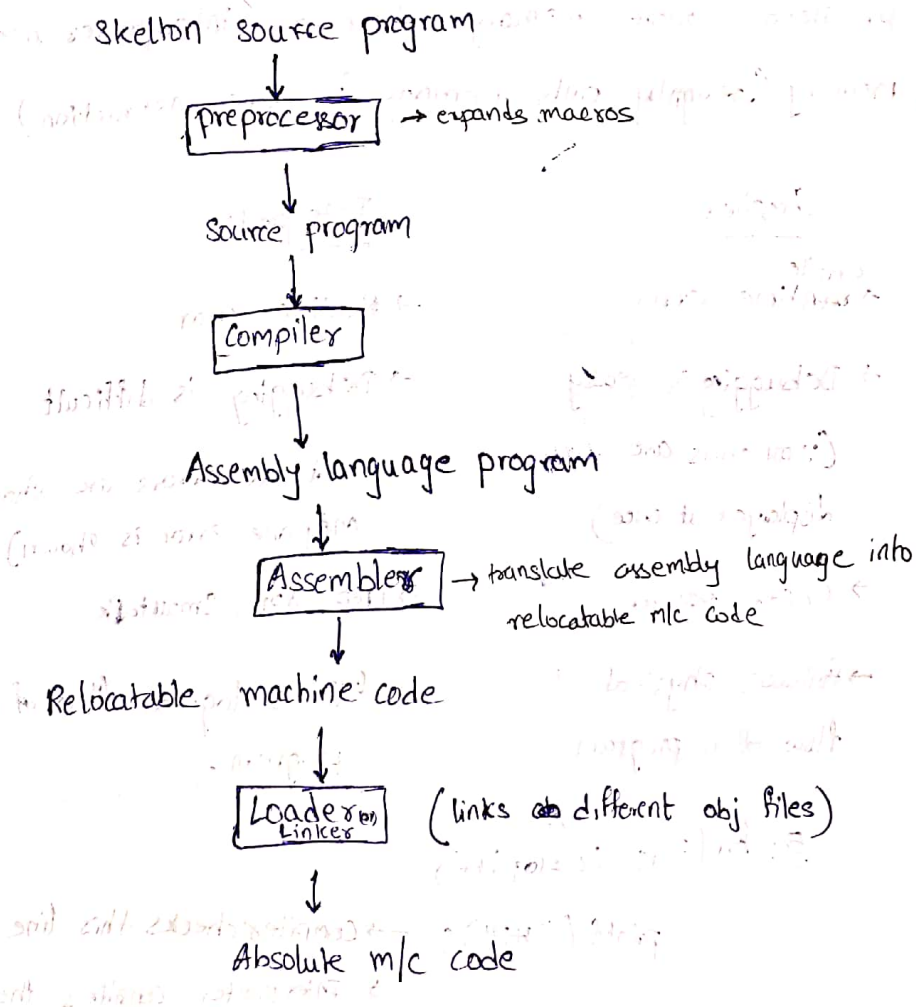
Object program may be any of below 3:

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

Scanned with CamScanner



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.

Scanned with CamScanner

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.

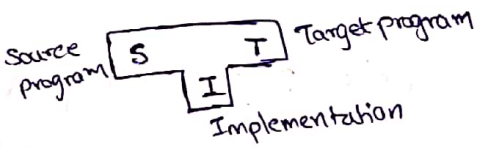
Eg: for(i=1; i<=10; i++)
printf("Hi");
→ Compiler checks this line only once
→ Interpreter considers the line 10 times

→ 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.

Scanned with CamScanner

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

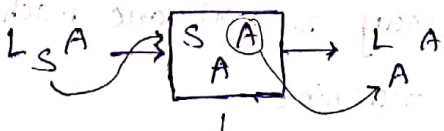


Existing compiler: $S_A A$ → compiler which takes 'S' program and generates target program for m/c A. which runs on A.

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

Required: Consider we need a compiler for java

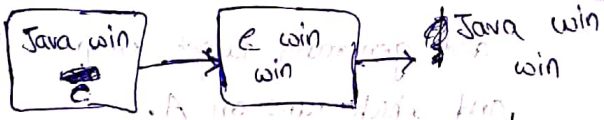
i.e., java win
win

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

java win
C

Step2: Compile

Java win using C win
C win



Java compiler targeting windows and running on windows

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)

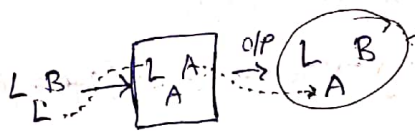
Scanned with CamScanner

Step 1 : write compiler program for ~~[illegible]~~ ~~[illegible]~~ 'L',
targeted for machine B using L.

ie., L_L^B

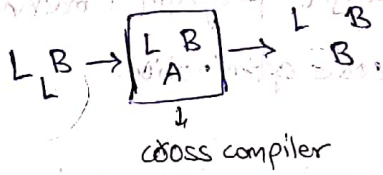
Step 2 : Compiler

L_L^B using L_A^A



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

~~set~~ Step 3 : Compile L_L^B into L_A^B



This above entire process is known as cross compilation.

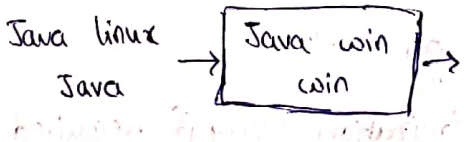
Eg:

Existing : Java win / win

Req : Java linux / linux

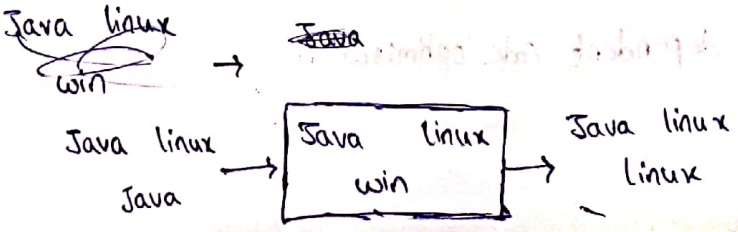
Step 1 : write a compiler program · Java linux / Java

Step 2 :



↳ cross compiler

Step 3 :



Scanned with CamScanner

17/06/20

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 (~~syntic~~ 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)

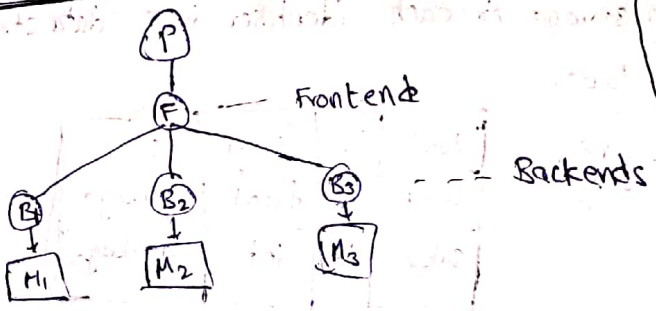
} front end

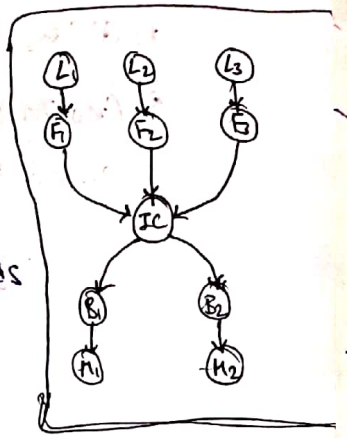
6) Code generation
7) M/c dependent code optimization

} back end

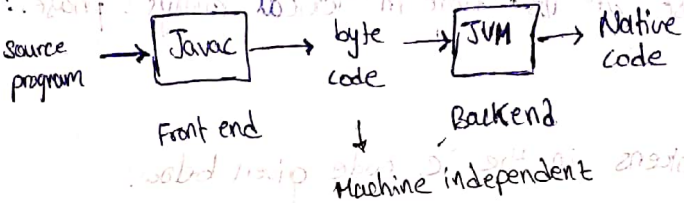
Scanned with CamScanner

* 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.

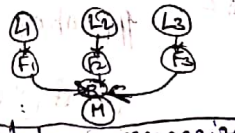




Eg:



Similiary 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.



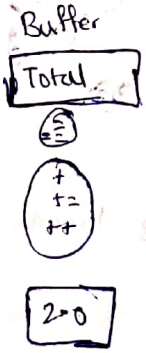
1) Lexical Analysis:

* 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



Scanned with CamScanner

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 groupes 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.

Scanned with CamScanner

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 G-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

> For lexeme =, the token is <=>. It does not need any attribute value. token of form <assign> can also be generated

> Difference blw 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>

Scanned with CamScanner

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

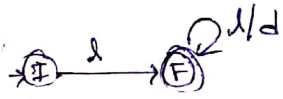
represented by ↓

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

represented by ↓

Reg. Exp ⟶ $l(l|d)^*$

represented by ↓

Finite Automata ⟶ 

represented by ↓

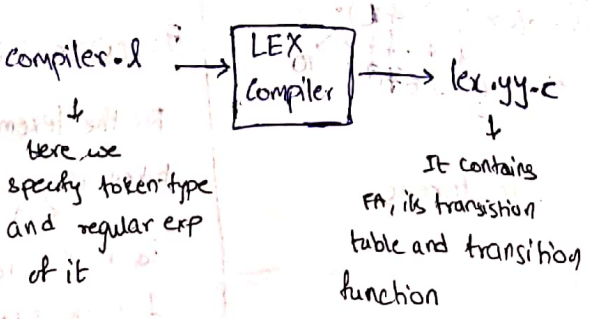
Transition table ⟶

	l	d
→ I	F	∅
(F)	F	F

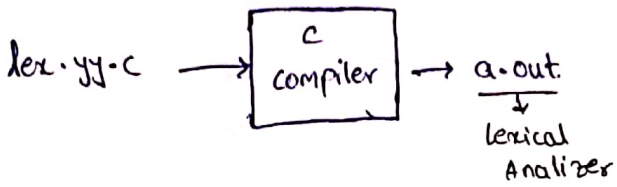
Implemented by ↓

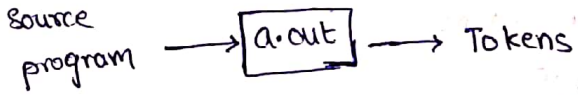
Transition function ⟶ ID() → for this we write a C function.

Designing with tool (not needed for gate)



Scanned with CamScanner





Types of Lexical errors:

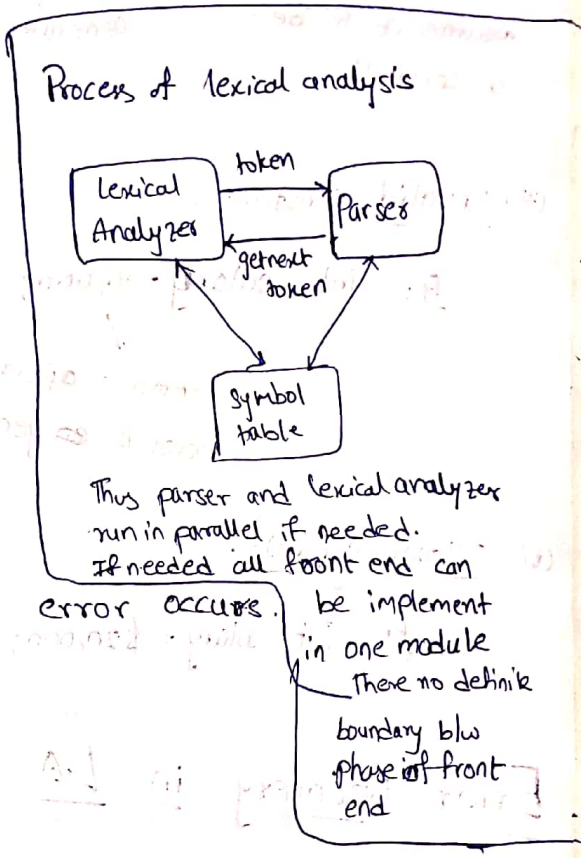
(i) Unterminated Comments error:

Eg:

```
/* Line1
   Line2
   Line3
```

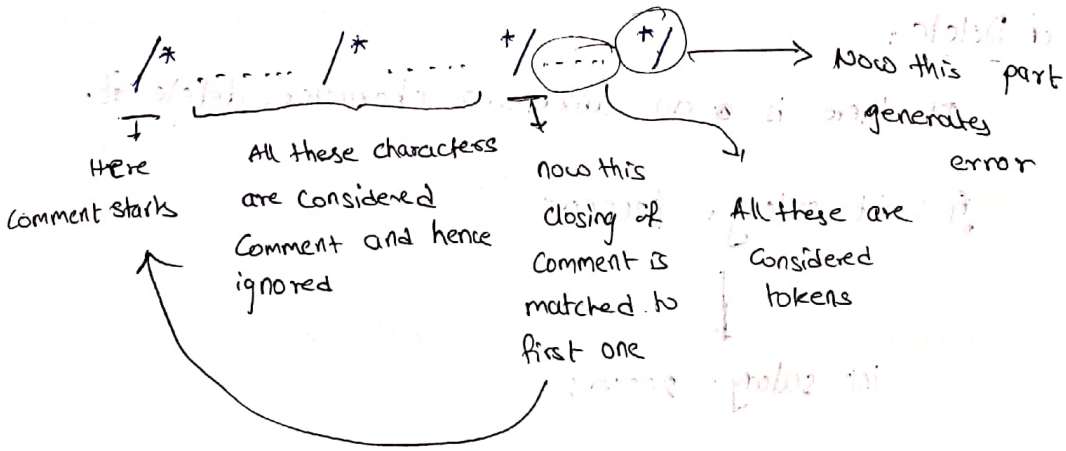
Here comment is not closed

Hence unterminated comment error occurs.



(ii) Nested Comments:

Eg: /* ... */



(iii) Length of longest identifier:

Eg:

```
int interest_on_principle_amount_and_compound_interest;
```

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

Scanned with CamScanner

(iv) Invalid Identifier

Eg: int 1stsalary;

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 an 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.

Scanned with CamScanner

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):

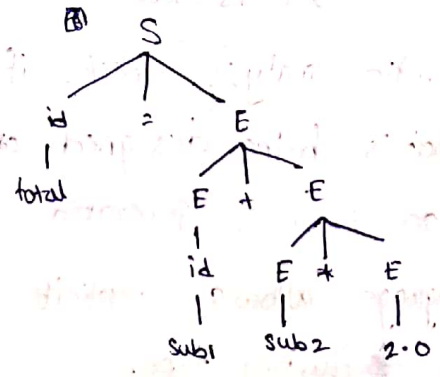
* To define the syntax of a language, CFG is used
* Consider below grammar to recognize certain mathematical assignment statements.

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.

Scanned with CamScanner

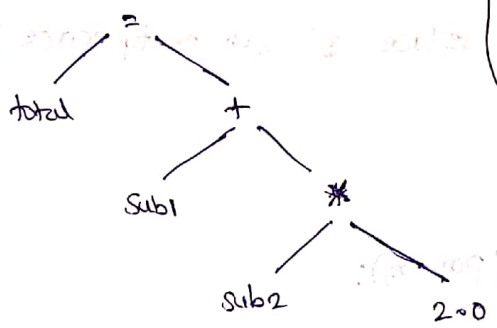
* Parse tree and syntax tree are different from each other.

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.

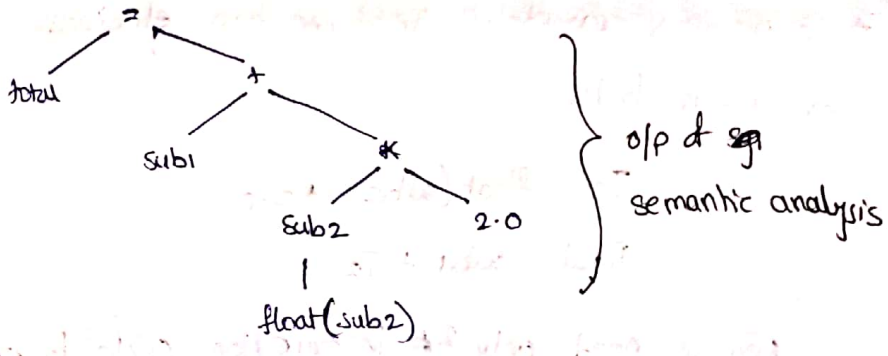
→ Now semantic analysis checks if this is permitted by the compiler that is being designed or not.

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.

Scanned with CamScanner



→ o/p of semantic analysis is called optimized syntax tree. or 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.

Scanned with CamScanner

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

T_2 = 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.

Scanned with CamScanner

18/06/20

Q6 G-18

A lexical analyzer uses the following patterns to recognize 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

Scanned with CamScanner

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 analyser 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 → AB~~

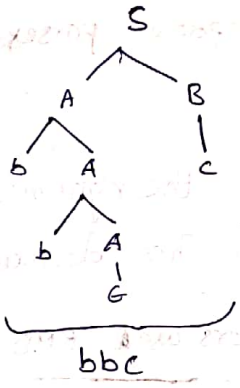
Scanned with CamScanner

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

} Derivation

Here derived string bbc is called sentence.

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



} Derivation tree (or) parse tree

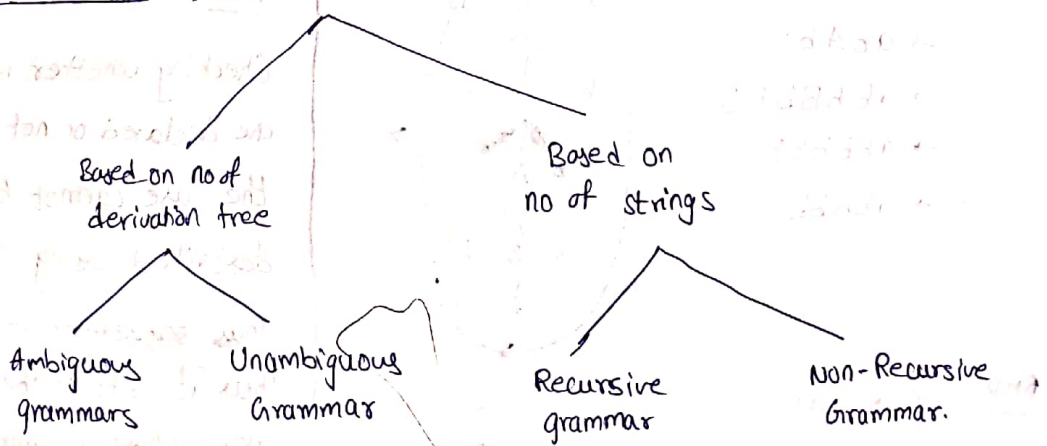
Derivation:

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 leftmost derivation or more than one right most derivation for some sentence of a language.

Scanned with CamScanner

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"

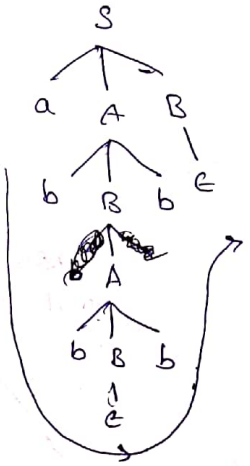
Eg: $S \rightarrow aAB$
$A \rightarrow bBb$
$B \rightarrow A|\epsilon$

i/p: abbbb

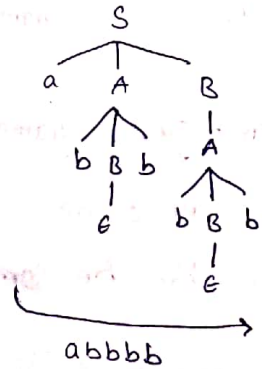
Construct LMD & RMD for above string.

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.

Scanned with CamScanner

(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 :

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

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 :

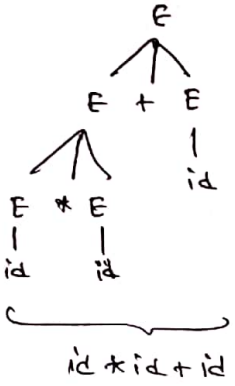
$E \to E+E$	$E \to E*E$
$\to id+E$	$\to E+E*E$
$\to id+E*E$	$\to id+E*E$
$\to id+id*E$	$\to id+id*E$
$\to id+id*E$	$\to id+id*id$

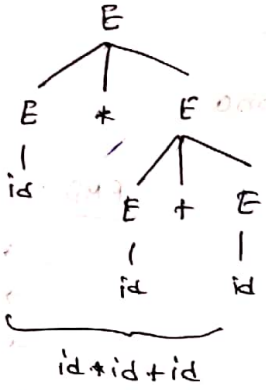
we have 2 left most derivations for above string

∴ Ambiguous.

Scanned with CamScanner

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





Observation:

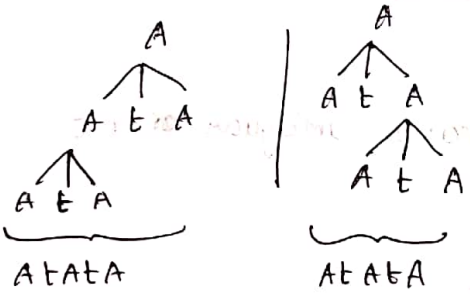
If we have a production of form

$A \rightarrow AtA$, 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.



Q10 (I) 2M 4M Which of the following terminal strings has more than one parse tree when parsed according to the grammar below.

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

a) id + id + id + id

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

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

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

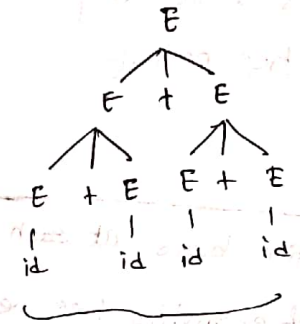
Scanned with CamScanner

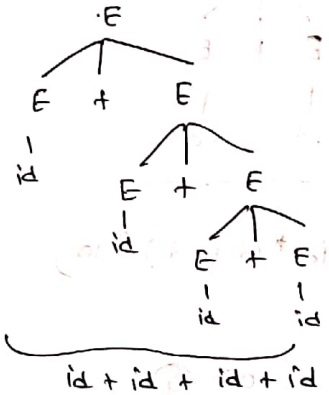
2M

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

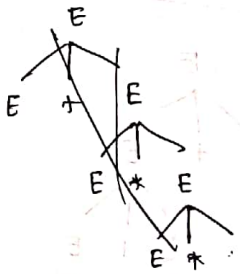
Sol:

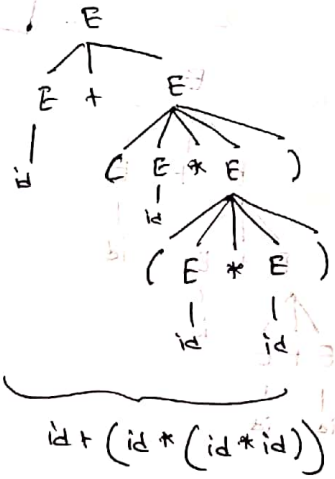
I) a) id+id+id+id





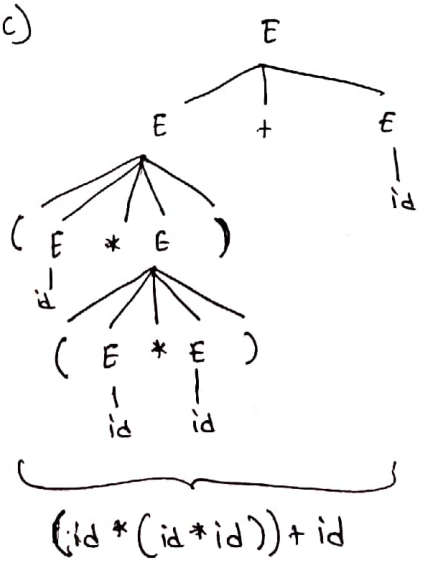
b)





This is the only derivation tree possible

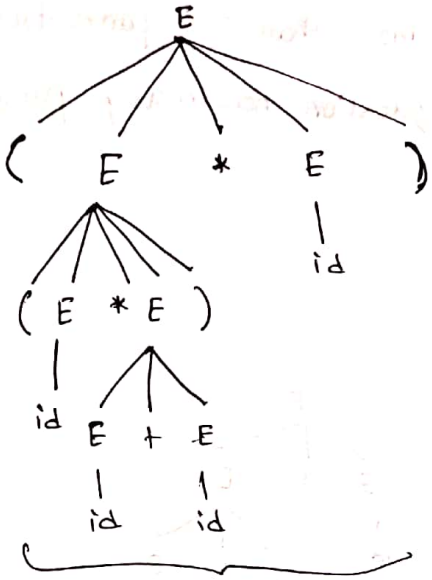
c)



This is the only tree possible

Scanned with CamScanner

d)



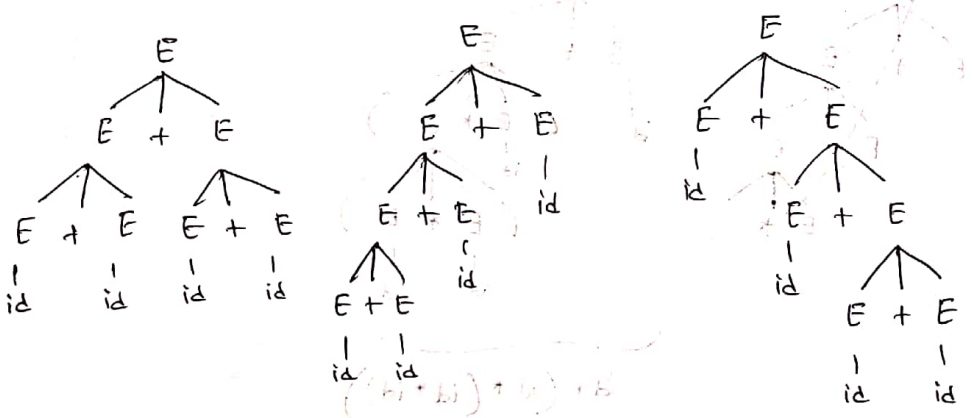
((id * id + id) * id)

This is the only tree possible

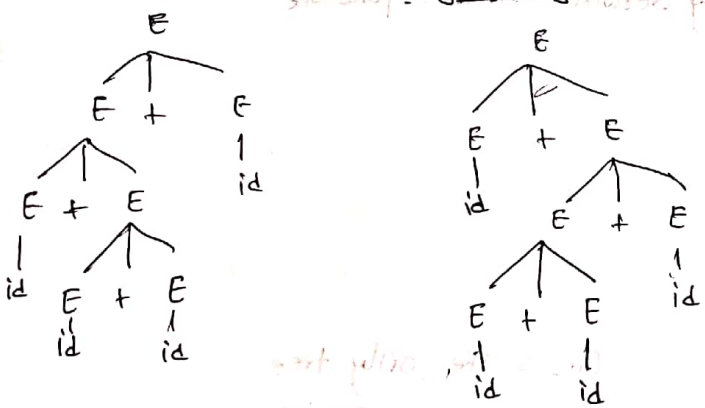
∴ opt (a)

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~~



Scanned with CamScanner

(Q11) Consider the following grammer

$S \rightarrow aS/A$

$A \rightarrow aAb/bAa/\epsilon$

(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) ~~a) aabbaba~~

observing grammar we can say it accepts any no of a's followed by ~~aAb~~ aAb / bAa

only opt (d) is of this form

∴ opt-(d)

(ii) ~~S→aS~~ d: aabbaa

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

} 6 steps

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

∴ only 1 tree

∴ 6 & 1

Scanned with CamScanner

19/08/20

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

$S \to aB$

$\to aaBB$

$\to aabB$

$\to aabbS$

~~$\to aabbbA$~~

$\to aabbaBB$

$\to aabbab$

α

c) aabbab

$S \to aB$

$\to aaBB$

$\to aabB$

$\to aabbS$

$\to aabbaB$

$\to aabbab$

$\therefore$ opt (c)

Scanned with CamScanner

(d) Now we check for different possibilities of derivation

$S \to aB$

$\to aaBB$

$\to aabB$ — $aabSB$ — $aabbAB$ — $aabbAAB$

$\to aabbS$ — $aabbaB$ — $aabbaAB$

$\to aabbaB$ — $aabbab$

$\to aabbab$

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

$T \to id$

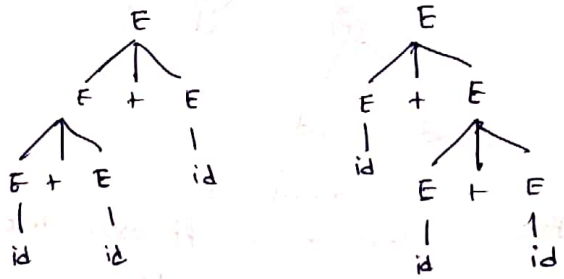
(or)

③ $E \to id + E \mid id$

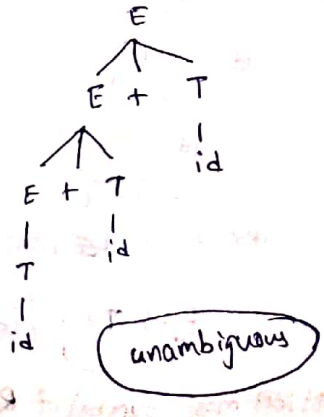
} unambiguous grammar

Consider $id + id + id$

deriving with grammar ①



deriving with ②



unambiguous

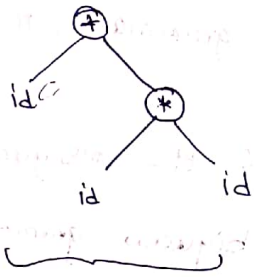
Scanned with CamScanner

→ By redefining 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.

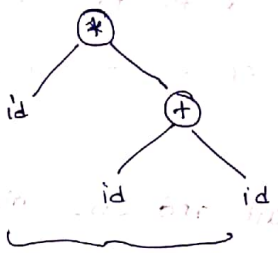
Converting ambiguous grammars into unambiguous using precedence and associativity.

→



id + id * id

P(*) > P(+)



id * id + id

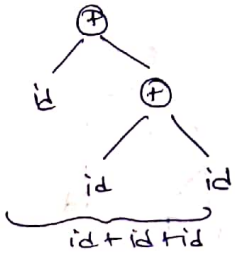
P(+) > P(*)

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

→

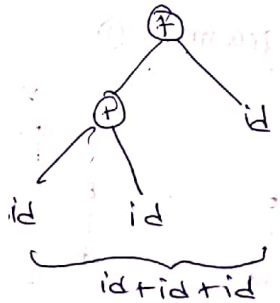


id + id + id

Here + is ~~left~~ right associative

E → T + E | id

right ~~left~~ most symbol of RHS is equal to 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.

Scanned with CamScanner

(Q13) 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

$F \to id$

Scanned with CamScanner

∴ The modified grammar is

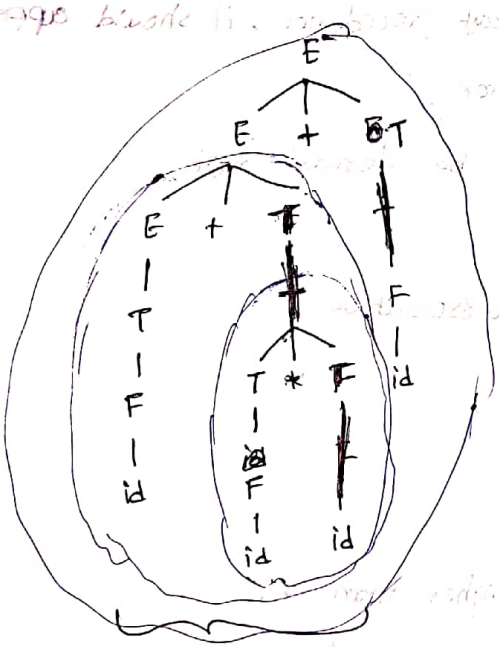
$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id$

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

Scanned with CamScanner

+, - 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) $E \to E*F \mid F$
$F \to F \wedge T \mid T$
$T \to P+T \mid P$
$P = id$

Scanned with CamScanner

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
– right
^ – right
$, + – left
-, * – right

Q.15 / GATE

E → E * F | F + E | F

F → F - F | id

Choose the correct answer based on precedence

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

Sol:

opt (b)

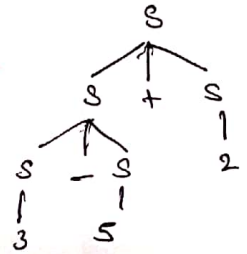
P(*) = P(+)

P(-) > P(*)

∴ opt (b)

Q.16 / GATE ★

Consider the following parse tree



which of the following based on Precedence

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

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

Scanned with CamScanner

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:

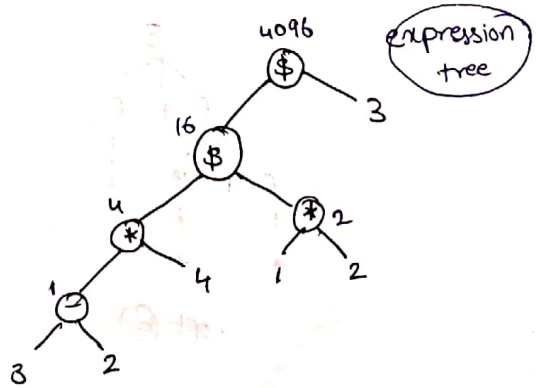
$3-2*4\$1*2\$3$

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

$4\$1*2\$3$

$4\$2\$3$

$16\$3 = 4096$



expression tree

Scanned with CamScanner

Q19 / Q-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:

$2-5+1-7*3$

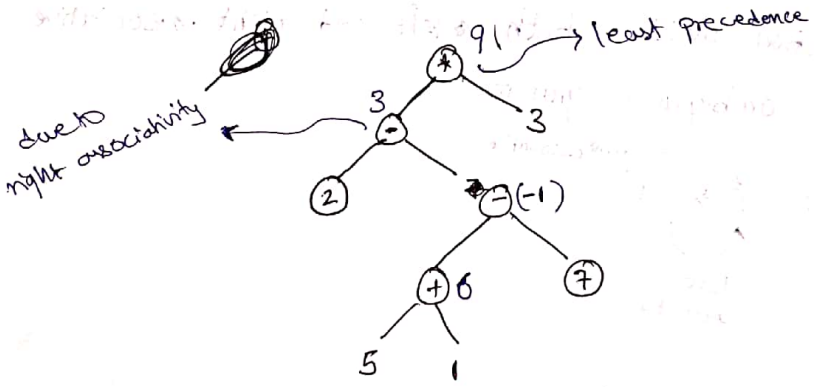
$2-6-7*3$

$2-(-1)*3$ (∵ right associative)

$3*3$

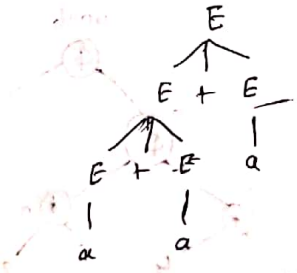
9

Solving by expression tree



Material problems on parsing techniques:

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



∴ opt (b)

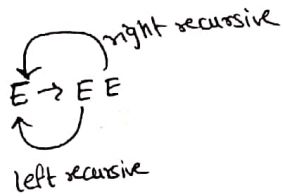
Scanned with CamScanner

P/2 since we have 'ε' we have infinite derivations

S → SS	S → SS
→ SSS	→ SSS
→ aSS	→ SSSS
→ aaS	→ aSSS
→ aaa	→ aaSS
	→ aaaS
	→ aaaa

∴ ∞

P/3

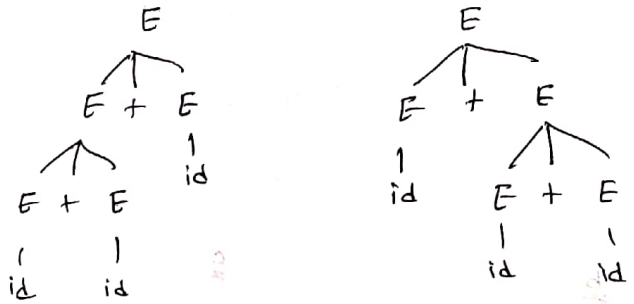


∴ ambiguous

P/4 observing grammar we can say for 'a' at right end we have 'b' at left end and viceversa. And in middle we have b.

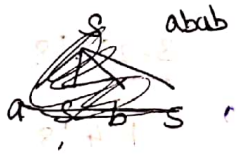
This satisfied by opt (d)

P/5



∴ 2 trees.

P/6



S → aSbS

abS → abaSbS → ababS → abab

abSaSbS → abaSbS → ababS → abab

∴ 2 derivations
∴ 2 parse trees.

Scanned with CamScanner

P/9

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

$T \to F + T \mid F$

$F \to i$

precedence

low
↑ - left
+ - right
high

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

∴ $c < a < b$

Scanned with CamScanner

Q20 / 6-07 Consider the grammar below with two operators * and +

$S \to T * P$

$T \to U \mid T * U$

$P \to Q + P \mid Q$

$Q \to id$

$U \to id$

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:

$P(+) > P(*)$

$T \to T * U \mid U$

$\therefore$ * is left associative

$P \to Q + P \mid Q$

$\therefore$ + is right associative

$\therefore$ opt (b)

Q21 / 4 If G is a grammar with productions

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

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

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

Scanned with CamScanner

a) abab

S → SS
→ asbS
→ abS
→ abasb
→ abab

b) aaab

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

∴ opt (d)

c) abbaa

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

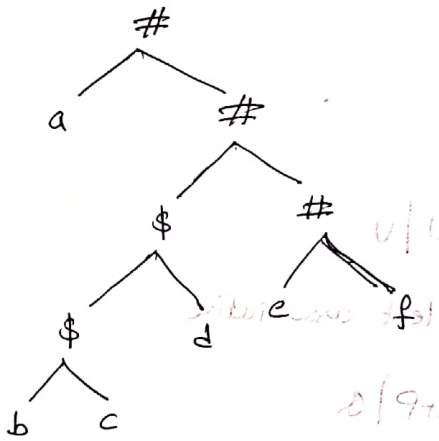
d) babba

S → bSa
→ ⊘

This has more b's than a's

∴ not possible

(Q22 / Gr) 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 $a\#\left[\left((b\$c)\$d\right)\#(e\#f)\right]$

Scanned with CamScanner

$ 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.

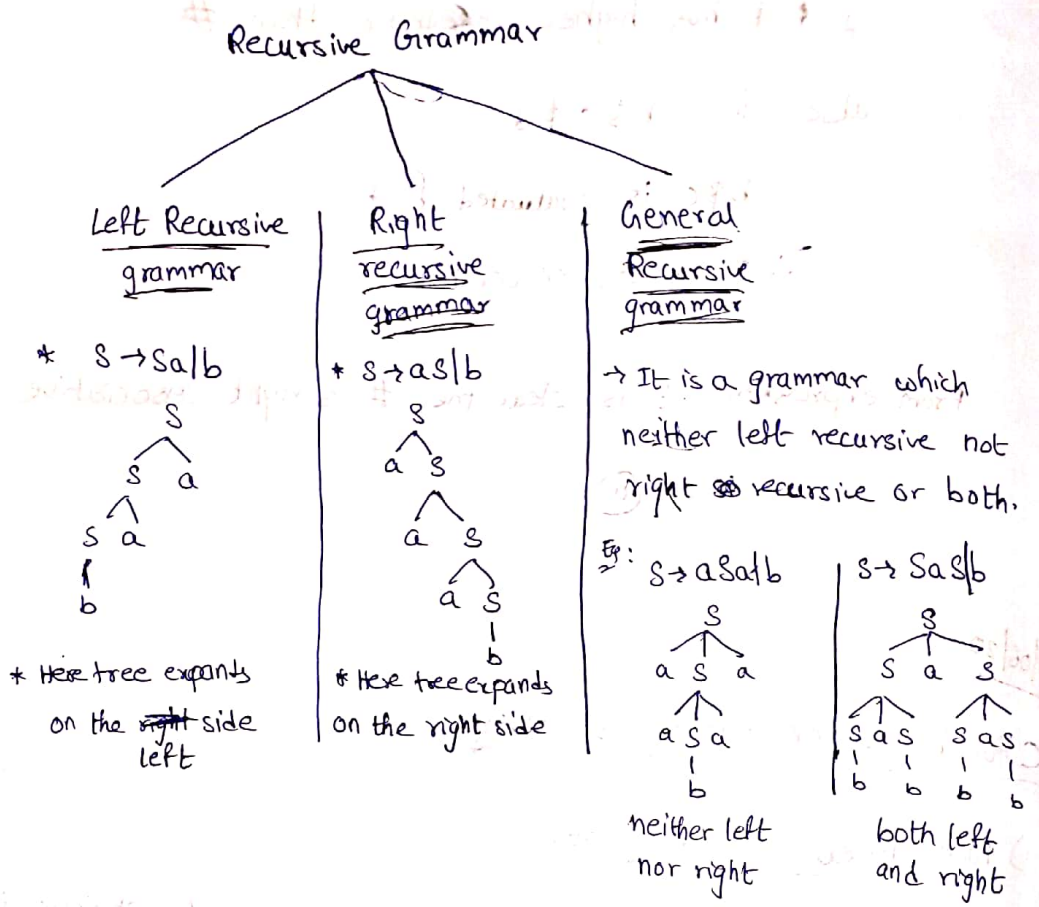
Eg: S → Sa | b

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

∴ Recursive grammar.

Scanned with CamScanner

Classification of Recursive grammars :



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.

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

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

Scanned with CamScanner

~~Modify~~

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

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

$A \to A\alpha$
$\to A\alpha\alpha$
$\to A\alpha\alpha\alpha$
$\to A\alpha\alpha\alpha \cdots \alpha$
$\to \beta\alpha\alpha\alpha \cdots \alpha$
$\to \beta A'$

Here A' generat $\alpha^n, n \geq 0$

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

Scanned with CamScanner

∴ Required grammar is

$E \to TE'$

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

$T \to FT'$

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

$F \to id$

(Q23) Eliminate left recursion from the below grammar

$S \to Sa / b / d$

$S \to bS' / dS'$

$S' \to aS' / \epsilon$

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

⇓ Eliminating left recursion

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

$A' \to \alpha_1 A' / \epsilon$

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

⇓

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

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

(Q24) Eliminate left recursion from below grammar

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

⇓ eliminating left recursion

$A \to cA' / dA'$

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

Scanned with CamScanner

Indirect Left recursion:

Consider the grammar

S → Aa | b
A → Sd | e

Consider derivation

S → Aa → Sda

Indirect left recursion.

Eg 2 : A → Ba
B → CAb | x.
C → d | ∈
Also has indirect left recursion
A → Ba → CAba → Aba

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

→ Direct left recursion is also called immediate left recursion.

→ Indirect left recursion should also be eliminated for top down 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

Scanned with CamScanner

$S \to Aa \mid b$

$A \to bdA' \mid eA'$

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

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:

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

∴ S has indirect left recursion

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

∴ A has indirect left recursion

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

∴ 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.

$S \to Aa \mid b$

$A \to Bd \mid e$

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

Scanned with CamScanner

Now we need to eliminate direct left recursion of B

$S \to Aa \mid b$

$A \to Bd \mid e$

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

$B' \to daB' \mid \epsilon$

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

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

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

Q.26 / G-16 Which of the following grammers is free from left recursion.

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

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

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

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

Sol:

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

∴ left recursive

b) $S \to Ab \longrightarrow Bdb$

$Bb \to eb$

∴ no left recursion

c) $S \to Aa \to Sca$

∴ left recursive

d) ~~$S \to Aa \to Bda \to Ae$~~

$A \to Bd \to Aed$

∴ left recursive

∴ opt (b)

Scanned with CamScanner

Q27 / G-17 Consider the following expression grammar G_1:

$E \to E-T \mid T$

$T \to T+F \mid F$

$F \to (E) \mid id$

which of the following grammars is not left recursive, but is equivalent to G_1?

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'

$\therefore$ opt (c) is correct

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

Scanned with CamScanner

Material problems

(P17)

$A \to Sa \mid b$

$S \to Sc \mid Ad$

$\Downarrow$

$A \to Sa \mid b$

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

$\Downarrow$

$A \to Sa \mid b$

$S \to bdS'$

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

opt (a)

(P18) 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.

Scanned with CamScanner

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)

$A \to aA'$

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

$A \to aA'$

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

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

$A \to abA' \mid ad$

$A' \to B \mid C$

$A \to aA''$

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

$A' \to B \mid C$

Parsing:

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

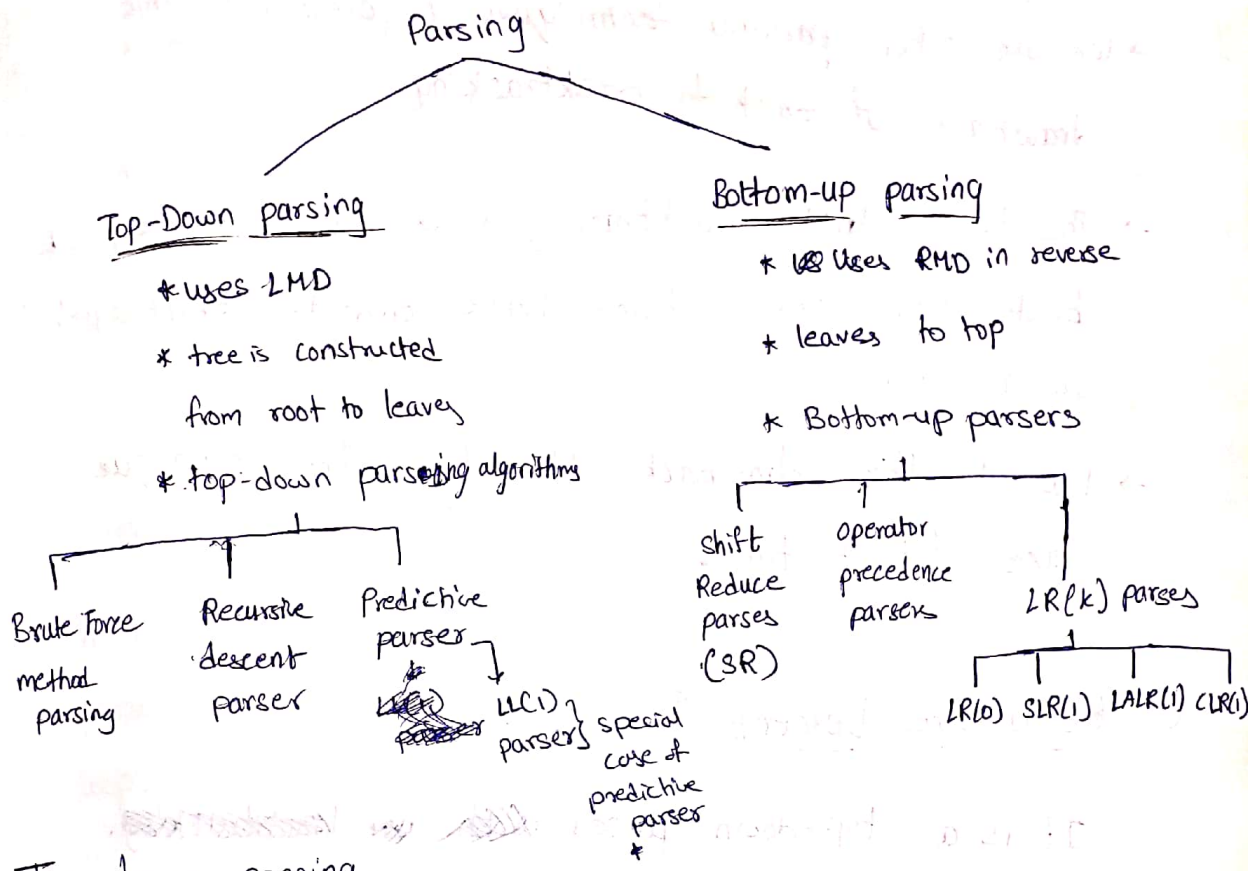
(i) parse tree if $w \in L(G)$

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

This process is known as parsing.

Scanned with CamScanner

* 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.

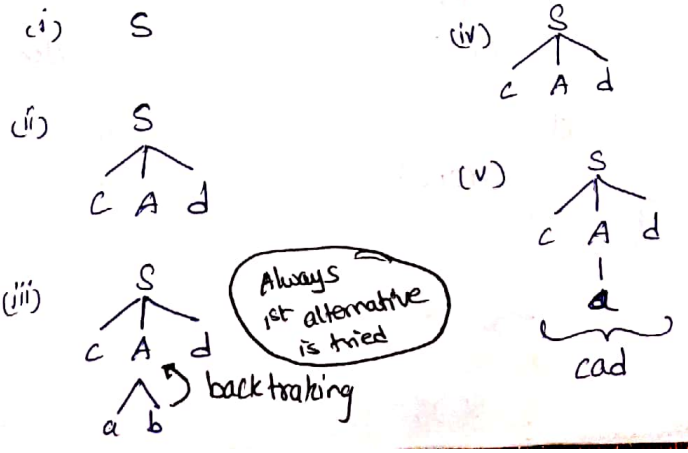
① Brute ~~Method~~ Force Method: It is a topdown parser with full backtracking

Eg: Consider the grammar

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

Below is topdown parsing of string "cad"



Scanned with CamScanner

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();
}
```

Scanned with CamScanner

Drawbacks 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 ~~accept~~ parse only a small class of ~~lang~~ ~~lang~~ grammars.

→ 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.

Scanned with CamScanner

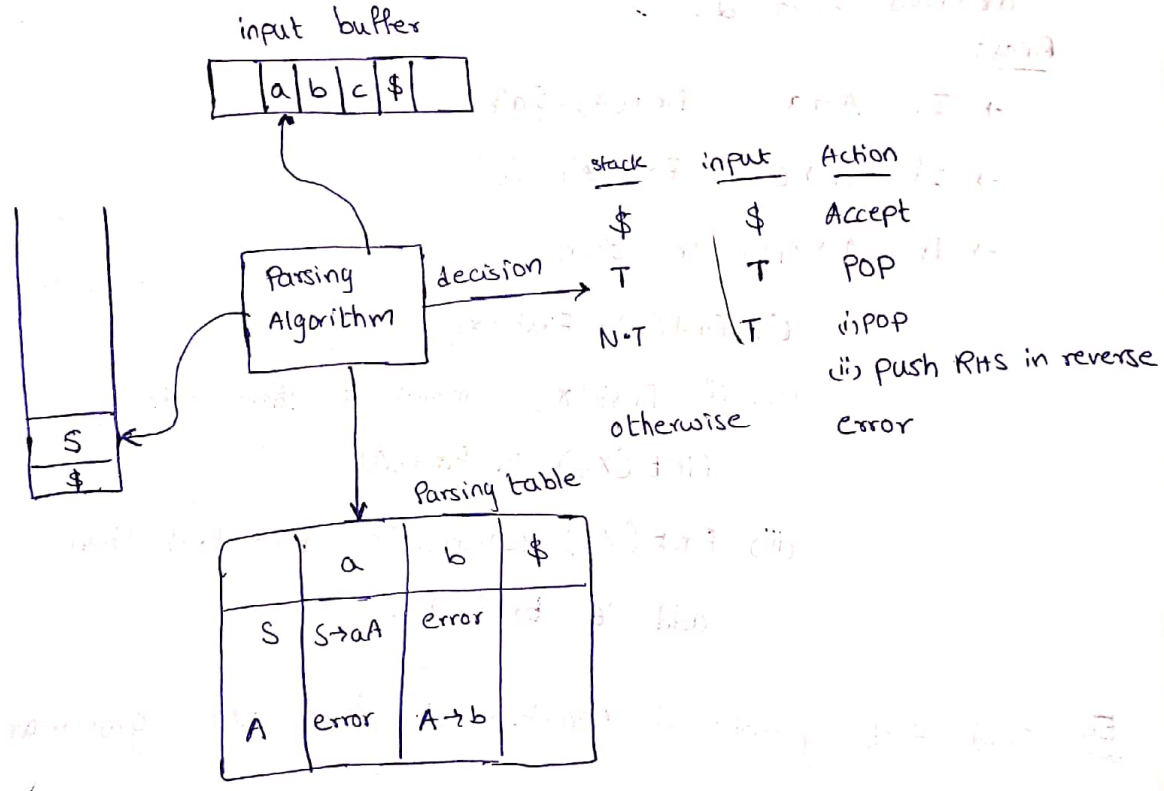
21/06/20

③ Predictive Parser:

It is a topdown parser which is a tabular representation of recursive descent parser.

* Using this table the parser predicts correct production for the derivation.

Structure:



initial configuration: stack → $ S (↓ Start symbol); input → w$ (↳ input string)

→ The i/p that is to be scanned is placed in the input buffer. '$' is used as end marker.

→ The stack contains '$' at the bottom denoting bottom of the stack and on top of that is S (start symbol)

Scanned with CamScanner

→ The parsing algorithm constructs a parse table using which it parses the input.

First and Follow:

→ The construction of a predictive parse table is aided by two functions called first and follow.

First set:

First(α) is the set of terminals that begin the string derived from α.

Rules:

→ If $A \to a$ First(A) = {a}

→ If $A \to \epsilon$ First(A) = {ϵ}

→ If $A \to X_1 X_2 \cdots X_n$ then

(i) First(A) = First(X_1)

(ii) if First(X_1) contains 'ϵ' then add First(X_2) to First(A)

(iii) First(X_i) contains 'ϵ' ∀ i = 1 to n then add 'ϵ' to First(A)

Eg: Find First symbols of non-terminal of following grammar

$S \to Aa$
$A \to BC$
$B \to d \mid \epsilon$
$C \to e \mid \epsilon$

First(A) = First(B) = {d, ϵ}
since First(B) contain ϵ
add First(c)
∴ F(A) = {d, e, ϵ}

First(S) = First(A) = {d, e, ϵ}
since First(A) contains ϵ
add First(a)
∴ First(S) = {d, e, a}

	First	
S	d, e, a	
A	d, e, ϵ	
B	d, ϵ	
C	e, ϵ	

Scanned with CamScanner

Eg: Find First sets for below grammar

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow d$

$B \rightarrow e \mid \epsilon$

	First
S	d, e, b
A	d
B	e, ϵ

First(S) = First(A) ∪ First(B)

= {d} ∪ {e} ∪ First(b)

= {d} ∪ {e} ∪ {b}

= {b, d, e}

Eg: $S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

S	i, a
S'	e, ϵ
E	b

Eg: $S \rightarrow ACB \mid Cbb \mid Ba$

$A \rightarrow da \mid BC$

$B \rightarrow g \mid \epsilon$

$C \rightarrow h \mid \epsilon$

S	d, g, h, b, a, ϵ
A	d, g, h, ϵ
B	g, ϵ
C	h, ϵ

First(S) = First(A) ∪ First(C) ∪ First(B)

↳ contains ϵ ↳ contains ϵ

$= \left[\{d\} \cup \{First(B) \cup First(C)\}\right] \cup \left[\{h\} \cup \{b\}\right] \cup \left[\{g\} \cup \{a\}\right]$

$= \{d, g, h, \epsilon\} \cup \{h, b\} \cup \{g, a\}$

$= \{a, b, d, g, h, \epsilon\}$

Scanned with CamScanner

Follow Set:

Consider the below grammar

$S \rightarrow Aa$

$A \rightarrow b \mid \epsilon$

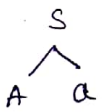
Let "a" be input string to be derived

Now $First(S) = \{a, b\}$

$First(A) = \{b, \epsilon\}$

To derive "a"

parser checks first(S) and 'a' is present



Now from A need to finish derivation

checking first(A) parser finds only b, ϵ

So we need to check symbols after certain non-terminal and this done by calculating follow.

→ Follow(A) is defined as set of terminals than can appear immediately to the right of A in some sentential form.

(A is non-terminal)

Rules:

(i) Follow(S) = $, where S is a start symbol.

(ii) If $A \rightarrow \alpha B \beta$, then

$Follow(B) = First(\beta)$

If First(β) contains ε

add Follow(A) to Follow(B)

Eg: Calculate follow sets for following grammars

(a) $S \rightarrow Aa$

$A \rightarrow BC$

$B \rightarrow d \mid \epsilon$

$C \rightarrow e \mid \epsilon$

Scanned with CamScanner

	First	Follow
S	d, e, a	$
A	d, e, ε	a
B	d, ε	e, a
C	e, ε	a

To find follow of a some non-terminal,

we need to ~~seed~~ see if the non-terminal is present in any R.H.S side

Follow(A):

$S \to Aa$

Follow(A) = First(a)

= a

Follow(S):

since S is start symbol

Follow(S) = $

Also is not present on the RHS of any production

Follow(B):

$A \to BC$

Follow(B) = First(C)

But First of C contains ε

∴ we add Follow(A) to Follow(B)

$\{e\} \cup \{a\} = \{a, e\}$

Follow(C):

$A \to BC$

Follow(C) = Follow(A) (∵ we have ε on the right to the C)

$= \{a\}$

Eg: $S \to AaAb \mid BbBa$

$A \to d$

$B \to e \mid \epsilon$

	First	Follow
S	d, e, b	$
A	d	a, b
B	e, ε	a, b

Follow(A):

$S \to AaAb$

∴ Follow(A) = {a, b}

Follow(B):

$S \to BbBa$

Follow(B) = {a, b}

Scanned with CamScanner

Eg: $S \to iEtSS' | a$

$S' \to eS | \epsilon$

$E \to b$

	First	Follow
S	i, a	$, e
S'	e, ϵ	$, e
E	b	t

Follow(S):

$S \to iEtSS'$

Follow(S) = First(S') → ②

First contains ϵ

∴ Add Follow(S) to follow(S)

∴ Follow(S) = {e}

when we encounter this kind of recursive things, we ignore it

Follow(S'):

$S \to iEtSS'$

Follow(S') = Follow(S)

$S' \to eS$

Follow(S) = Follow(S')

∴ Follow(S) = {$, e}

Follow(S') = {$, e}

Follow(E):

$S \to iEtSS'$

Follow(E) = t

Eg: $S \to ACB | Cbb | Ba$

$A \to da | BC$

$B \to g | \epsilon$

$C \to h | \epsilon$.

	First	Follow
S	d, g, h, ϵ, b, a	$, h, g
A	d, g, h, ϵ	h, g, $
B	g, ϵ	a, h, g, $
C	h, ϵ	b, h, g, $

Follow(A):

$S \to ACB$

Follow(A) = First(C)

First (C) contain ϵ

Add Follow(B) to Follow(A)

First of B contains ϵ

∴ Add Follow(S) to Follow(A)

Follow(C)

$S \to Cbb$

add 'b' to follow(C)

$A \to BC$

add follow(A) to follow(C)

$S \to ACB$

add first(B) to follow(C)

first(B) contains ϵ

∴ add follow(A) to follow(C)

Follow(B):

$S \to Ba$

add 'a' to follow B

$A \to BC$

Add first(C) to follow(B)

First(C) contain ϵ.

∴ Add follow(A) to follow(B)

Scanned with CamScanner

(Q28) Find first & follow sets for below grammar

$S \to aA$

$A \to BC \mid d$

$B \to Sb \mid \epsilon$

$C \to Aa \mid \epsilon$

Sol :

First(S) = {a}

~~Follow (B) =~~

~~Follow~~ First(B) :

Add first(S), ε

First (C) :

Add first (A) to first(C)

and ε

First (A) :

Add first (B) to first(A) & d

first (B) contains ε

∴ Add first (C) to first(A)

	First	Follow
S	a	$, b
A	a, d, ε	$, b, a
B	a, ε	$, a, d, b
C	a, d, ε	$, a, b

Follow (S) :

add $

$B \to Sb$

add first (b)

Follow (A) :

$S \to aA$

add follow(S)

$C \to Aa$

add first(a)

Follow (C) :

$A \to BC$

add follow(A)

Follow (B) :

~~A~~ $A \to BC$

add first(C)

first (C) contains ε

∴ add follow(A)

For this kind of question in which ~~need~~ first of a non-terminal is dependent on first of other non-terminal, calculate iteratively. i.e. Run through the table until you get no ~~more~~ more new modifications

(Q29) Find first & follow sets for below grammar

$A \to bAa \mid Ad \mid \epsilon$

Scanned with CamScanner

Sol:

First(A) :

$A \to bAa$

add 'b' to first(A)

$A \to \in$

add '∈' to first(A)

$A \to Ad$

since first(A) contains ∈

add 'd' to first(A)

$\therefore$ First(A) = {b, d, ∈}

Follow(A):

add $ to follow(A)

$A \to bAa$

~~follow(A)~~ =

add 'a' to follow(A)

$A \to Ad$

add 'd' to follow(A)

$\therefore$ follow(A) = {$, a, d}

Q30 / G-17 Consider the following grammar

$P \to xQRS$

$Q \to yz \mid z$

$R \to w \mid \in$

$S \to y$

what is FOLLOW(Q)?

a) {R} b) {w} c) {w, y} d) {w, $}

Sol: $P \to xQRS$

add first(R)

first(R) contain ∈

$\therefore$ add first(S)

w, y

$\therefore$ opt (c)

Scanned with CamScanner

Q3) G-19 Consider the grammar given below:

$S \to Aa$

$A \to BD$

$B \to b|\epsilon$

$D \to d|\epsilon$

let a, b, d and $ be indexed as follows

a	b	d	$
3	2	1	0

Compute the FOLLOW set of the non-terminal B and write the index values for the symbols in the follow set in the descending order.

Sol:

$A \to BD$

Add first(D) i.e., d

d, a

first(D) contains ϵ

∴ Add follow(A)

follow(A)

$S \to Aa$

follow(A) = {a}

∴ add 'a' to follow(B)

∴ Ans : 31

Construction of Predictive Parse table

Consider the grammar

$S \to as|b$

The predictive parse table can be directly constructed as shown in the figure since we have only 2 productions

i/p	a	b	$
S	S→aS	S→b	error

Scanned with CamScanner

From this way of construction we can generalize the below rule

> **Rule 1:** If $A \to \alpha$ is a production
>
> then add $A \to \alpha$ to $M[A, \text{first}(\alpha)]$

Ex:

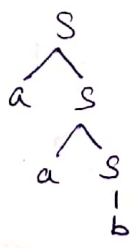
Stack	i/p	Action
$S	~~aab~~$ aab$	POP Push RHS of (S→aS) in reverse
$Sa	aab$	POP (∵ terminals matched) and i/p is advance to next symbol
	↙ Here i/p is not remove - Read head just moves.	
$S	ab$	POP Add RHS of production in (M[S,a]) in reverse
$Sa	ab $	POP
$S	b$	POP S Add RHS of production in (M[S,b]) in reverse
$b	b $	POP b
$	$	Accept.

Parse tree is constructed in the order in which productions are applied.

In above example we have applied

$S \to aS$, $S \to aS$, $S \to b$

∴ Parse tree constructed is



> **Parsing:**
>
> If top of stack is a non-terminal, i/p ~~string~~ then production from parse table in M[N.T, T] is applied

Scanned with CamScanner

→ Consider the grammar

$S \to aSb \mid \epsilon$

According to Rule 1

we need to add $S \to \epsilon$ in $M[S, \epsilon]$ but we don't have ϵ column in the parse table. So here we use Rule 2 with follow information

Rule 2: If $A \to \alpha$ is a production, and

if first(α) contains ϵ; then

- add $A \to \alpha$ to $M[A, \text{follow}(A)]$

Here Follow(S) = {$, b}

Thus the construction of parse table is as shown below

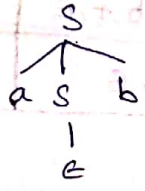
i/p	a	b	$
i/s	$S \to aSb$	$S \to \epsilon$	$S \to \epsilon$

The question in GATE are mainly asked on contruction of predictive parsing but not on the parsing algorithm or parsing a string

Eg:

Stack	input	Action
$ S	ab $	~~S → aSb~~ POP push RHS of (S → aSb) in reverse
$ bSa	ab $	POP
$ bS	b $	POP push RHS of (S → ε) in reverse
$ b	b $	POP
$	$	Accept

Here the parse is constructed is



Scanned with CamScanner

~~Q&S 2~~

Material Problems

P/16

S	d, f, a, e, b
A	d
B	e, ϵ
C	f, ϵ

∴ FIRST(S) = {a, b, d, e, f}

P/17

	FIRST	FOLLOW
exp	atom, +, *, ϵ, (	$,)
add	+, *, ϵ	$,

P/18

S → ABC
A → aA | ϵ
B → b | ϵ
C → c

	FIRST	FOLLOW
S	a, c	~~b~~
A	a, c	b, c
B	b, ϵ	
C	c	

FIRST(A) ∩ FOLLOW(A) = {c}

P/19

A → BCD
B → w | Bx
C → yCz | m
D → DB | a

	FIRST	FOLLOW
A		$
B	w,	y, m, x, w, $
C	y, m	
D		w, $

∴ {x, y, m, w, $}

P/20

	FIRST	FOLLOW
S	[, a	$, +, -, b,], c
X	-, b, +, ϵ	], c
Y	-, ϵ	

∴ opt (a)

Scanned with CamScanner

P/21

	FIRST	FOLLOW
S	a, +, ε	$
A	+, ε	
T	a, ε	+, $

$\{+,\$\} \cap \{a,+,\epsilon\} = \{+\}$

P/22

	FIRST	FOLLOW
S		$
A	e, d, ε	e, $, c, d
B	e,	e, d, $, c

P/23

	FIRST	FOLLOW
S	a, b	$, a, d
A	b, a	a
B	b, a	d, a
C	a	a, d, $

$\therefore \{A, B, C, S\}$

24/06/20

Construct predictive parse table for below grammar

(a) $S \to aA \mid bB$

$A \to a \mid \epsilon$

$B \to b \mid \epsilon$

	a	b	$
S	S→aA	S→bB	error
A	A→a	error	A→ε
B	error	B→b	B→ε

Scanned with CamScanner

→ Construct predictive parse table

$S \to aA \mid AB$

$A \to b \mid \epsilon$

$B \to c \mid \epsilon$

	a	b	e	$
S	S→aA	S→AB	S→AB	S→AB
A	error	A→b	A→bE	A→E
B	error	error	B→C	B→E

S→AB is added in first(AB)

first(AB) = {b, E, ∈}

∵ first(AB) contains ∈

∴ add S→AB ~~follow(S)~~ M[A, follow(S)]

→ Construct predictive parse table for below grammar

$A \to bAa \mid Ad \mid \epsilon$

Sol:

first(A) = {b, ∈, d}

follow(A) = {a, d, $}

	a	b	d	$
A	A→∈	A→bAa A→Ad	A→Ad A→∈	A→∈

A→Ad is added in first(Ad)

first(Ad) = {b, d}

Here M[A, b] and M[A, d] has multiple entries.

so the parser can't choose appropriate production.

Hence this grammar is not suitable for predictive parse

one reason for this is the grammar has left recursion.

Note:

If the given grammar is ambiguous or left recursive then atleast one multiply defined entry is present in parse table.

Some grammars which are not left recursive and not ambiguous also have multiply defined entries

Scanned with CamScanner

→ Construct predictive parse table for below grammar

$S \to AB \mid d$

$A \to Sb \mid \epsilon$

$B \to Aa \mid \epsilon$

	FIRST	FOLLOW
S	d, ε, b, a	$, b
A	ε, b, a, d	a, b, d, $
B	ε, a, b, d	$, b

	a	b	d	$
S	S→AB	S→AB	S→AB S→d	S→AB
A	A→Sb A→ε	A→Sb A→ε	A→Sb A→ε	A→ε
B	B→Aa	B→Aa B→ε	B→Aa	B→ε

Since the table has multiple entries for some cells, the grammar is not suitable for predictive parsing.

Note:

* Every left-recursive grammar is not suitable for predictive parsing. Thus left-recursive grammar will produce multiple entries.

However we do have other grammars which are not left recursive and still not suitable for predictive parsing.

Eg: Consider

$S \to aA \mid Ab$

$A \to a$

	a	b	$
S	S→aA S→Ab	err	err
A	A→a	err	err

For some grammars even if we eliminate left recursion and left factoring, and no matter how many modifications you do you can never have a predictive parser.

Eg: $S \to iEtSS' \mid a$

$S' \to eS \mid \epsilon$

$E \to b$

has no equivalent LL(1) grammar the above grammar is ambiguous

Observation: Here the language of grammar is

$\{aa, ab\}$

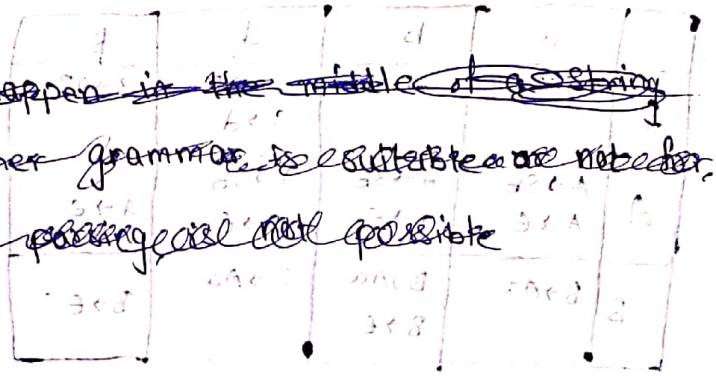
Scanned with CamScanner

Here aa is derived from using S → aA
ab is derived from using S → Ab

which means the multiple entries here are due to two string aa, ab starting with same symbols but they are derived from different productions.

This type of grammars are also not suitable for predictive parsing.

~~These can ever happen in the middle of a string for telling whether grammar is suitable are not for left predictive parsing case not possible~~



Note:

The grammars that are suitable for predictive parsers are called LL(1) grammar.

> A Grammar is suitable for predictive parsing if it is LL(k) grammar

LL(1) grammars:

1st L stands for Left-to right scanning of i/p
2nd L stands for Left most derivation
1 stands for no of look ahead symbols.
i.e., it look at one symbol and moves ahead

→ Consider a production

S → aA/bB

Seeing i/p string, S → aA (or) S → bB can be choosen accurately since starting symbol of grammar. i.e., 1 look ahead symbol

Thus if grammar contains all such productions we call it LL(1) gramimar.

→ Consider a production

S → abA/adB

Here seeing 'a' on the i/p table we can't decide which production to choose S → abA (or) S → adB

Scanned with CamScanner

∴ If a grammar has such productions is not LL(1)

However if looking ahead 2 symbols can help us choose accurate decision.

So we can call it LL(2) grammar.

→ Consider

$S \to abcA \mid abdB$

The above production belongs to LL(3) grammar.

→ In general

LL(k) grammar means look ahead symbols are k number to determine accurate productions.

Note:

* → No ambiguous grammar is LL(1). → Thus if a grammar is suitable for predictive parsing then it is not ambiguous.
* → No left recursive grammar is LL(1).

→ All the entries in the parsing table of a LL(1) grammar are unique.

* → A grammar G is LL(1) iff the following conditions hold for distinct production of the form

$A \to \alpha \mid \beta$

(i) $First(\alpha) \cap First(\beta) = \phi$

(ii) If $First(\alpha)$ contains ϵ then

$Follow(A) \cap First(\beta) = \phi$

(iii) If $First(\beta)$ contains ϵ then

$Follow(A) \cap First(\alpha) = \phi$

If any of above conditions is unsatisfied we can conclude the grammar is not LL(1).

Scanned with CamScanner

Eg: Check whether the below grammar is LL(1) or not.

$S \to iEtSS' \mid a$

$S' \to eS \mid \epsilon$

$E \to b$

Consider

$S \to iEtSS' \mid a$

$First(iEtSS') \cap First(a)$

$\{i\} \cap \{a\} = \phi$

and neither of the two first sets containing ϵ.

Now

$S' \to eS \mid \epsilon$

$First(eS) \cap First(\epsilon)$

$\{e\} \cap \{\epsilon\} = \phi$

But $First(\epsilon)$ has ϵ

$\therefore Follow(S') = \{\$, e\}$

Now $First(eS) \cap Follow(S')$

$\{e\} \cap \{\$, e\} = \{e\} \neq \phi$

$\therefore$ The grammar is not LL(1).

The parse table has multiple entries for $M[S', e]$

(Q32) Check whether the given grammar is LL(1) or not.

$S \to aA \mid AB$

$A \to b \mid \epsilon$

$B \to c \mid \epsilon$

Sol:

$S \to aA \mid AB$

$First(aA) \cap First(AB)$

$\{a\} \cap \{b, c, \epsilon\} = \phi$

$First(aA) \cap Follow(S)$

$\{a\} \cap \{\$\} = \phi$

Scanned with CamScanner

$A \to b \mid \epsilon$

$First(b) \cap First(\epsilon) = \phi$

$First(b) \cap Follow(A)$

$\{b\} \cap \{\$, c\} = \phi$

$B \to c \mid \epsilon$

$First(c) \cap First(\epsilon) = \phi$

~~$First(c) \cap First(B)$~~

$First(c) \cap Follow(B)$

$\{c\} \cap \{\$\} = \phi$

$\therefore$ LL(1)

(Q33) Check whether below grammars are LL(1) or not

(i) $S \to AA$

$A \to aA \mid \epsilon$

$S \to AA$

~~*~~ when we have only one production we will never get multiple entries for S.

$A \to aA \mid \epsilon$

$First(aA) \cap First(\epsilon)$

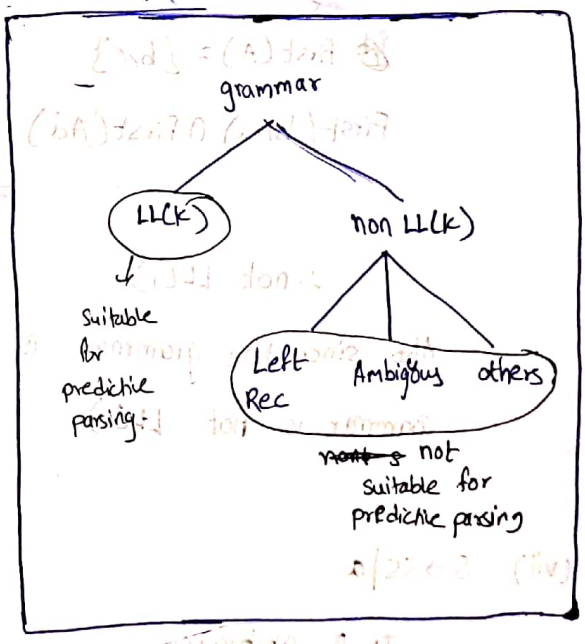
$\{a\} \cap \{\epsilon\} = \phi$

~~Follow~~ ~~First(aA) ∩ First~~

$First(aA) \cap Follow(A)$

$= \{a\} \cap \{a, \$\} = a$

$\therefore$ not LL(1)



(ii) $S \to aS \mid Ab$

$A \to a \mid b$

eg:

$S \to aS \mid Ab$

$First(aS) \cap First(Ab)$

$\{a\} \cap \{a, b\} \neq \phi$

$\therefore$ not LL(1)

Scanned with CamScanner

(iii) $A \to ab \mid ac \mid bd$

$First(ab) \cap First(ac) = \{a\} \cap \{a\} = \{a\} \neq \phi$

$\therefore$ not LL(1)

(iv) $S \to aB \mid \epsilon$
$B \to bC \mid \epsilon$
$C \to cS \mid \epsilon$

$\{ab\} \cap \{\epsilon\} = \phi$
$\{aB\} \cap \{\$\} = \phi$
$\{b\} \cap \{\epsilon\} = \phi$
$\{b\} \cap \{\$\} = \phi$
$\{c\} \cap \{\epsilon\} = \phi$
$\{c\} \cap \{\$\} = \phi$

$\therefore$ LL(1)

(v) $S \to aAa \mid \epsilon$
$A \to abS \mid \epsilon$

$\{a\} \cap \{\epsilon\} = \phi$
$\{a\} \cap \{\$, a\} \neq \phi$

$\therefore$ not LL(1)

(vi) $A \to bAa \mid Ad \mid c$ → Left recursive, hence we can directly say it is not LL(1)

$First(A) = \{b, c\}$

$First(bAa) \cap First(Ad) = \{b\} \cap \{b, c\}$
$= \{b\} \neq \phi$

$\therefore$ not LL(1)

Also since the grammar is left recursive we can conclude that grammar is not LL(1)

(vii) $S \to SS \mid a$

It is ambiguous

$\therefore$ not LL(1).

Scanned with CamScanner

Material Problems

(P/24)

$E \to FR$

$R \to *E \mid \epsilon$

$F \to id$

$E \to FR$ is added to first(FR)

i.e., {id}

$\therefore M[E, id]$ is $E \to FR$

to find $M[R, *]$

$R \to *E$ is added to $M[R, First(*E)]$

i.e., $M[R, *]$

Now consider $R \to \epsilon$

$R \to \epsilon$ is added to follow(R)

follow(R) = follow(E) = {$}

$\therefore R \to \epsilon$ is added to $M[R, \$]$

$\therefore$ opt (c)

(P/25) $S \to a \mid ab \mid abc$

even if we look ahead 2 symbols

we can't choose b/w $S \to ab$ | $S \to abc$

$\therefore$ the grammar is LL(3)

(P/26) we need to look ahead 6 symbols

$\therefore$ LL(6)

(P/27)

G_1 :

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

↓

left & right recursive

$\therefore$ ambiguous

G_2 :

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

↓

precedence : (*) > (+)

associative : + => L→R

* => R→L

Scanned with CamScanner

Since G2 is left recursive, it is not LL(1)

~~also G1 is conve~~ ∴ opt (c)

P/28 a) Every left recursive is not LL(1)

b) right recursive can be LL(1)

Eg: $A \to aA/b$

is LL(1)

c) No ambiguous grammar is LL(1)

d) parsing is only for CFGs

LL(1) is only for CFGs

∴ opt (a)

P/29 $A \to \epsilon$ is added to Follow(A)

Follow(A) = {a}

∴ 'a' column

P/30 ~~S→aSbS~~ $S \to aSbS$ is added to $M[S, a]$

$S \to \epsilon$ is added to $M[S, \text{follow}(S)]$

i.e., $M[S, a]$ & $M[S, b]$

∴ opt (d)

P/31 $S \to Aa/Bb$

First(Aa) ∩ First(Bb)

$\{b\} \cap \{b\} = \{b\} \neq \phi$

∴ not LL(1)

$S \to Aa/Bb$ is equivalent to : $S \to ba/bb$

∴ LL(2)

It means if string to be derived is ba the LL(2) parser uses $S \to Aa$. If the string is bb the production $S \to Bb$ is used

Scanned with CamScanner

Q34 / 9-03 Consider the grammar shown below

$S \to iEtSS' | a$

$S' \to eS | \epsilon$

$E \to b$

In the predictive parse table M, of this grammar, the entries M[S', e] and M[S', $] respectively are

a) $\{S' \to eS\}$ and $\{S' \to \epsilon\}$

b) $\{S' \to eS\}$ and $\{\}$

c) $\{S' \to \epsilon\}$ and $\{S' \to \epsilon\}$

d) $\{S' \to eS, S' \to \epsilon\}$ and $\{S' \to \epsilon\}$

Sol:

$S' \to eS$ is added to first(eS)

i.e., to M[S', e]

$S' \to \epsilon$ is added to column follow(S')

follow(S') = {$, e}

	e	$
S'	$S' \to eS$ $S' \to \epsilon$	$S' \to \epsilon$

opt (d)

~~Q35 / 9-06 Same as Q34~~

Q35 / 9-06 Consider the below grammar

$S \to FR$

$R \to *S | \epsilon$

$F \to id$

In the predictive parse table M, of the grammar entries M[S, id] and M[R, $] respectively.

a) $\{S \to FR\}$ and $\{R \to \epsilon\}$ b) $\{S \to FR\}$ and $\{\}$

c) $\{S \to FR\}$ and $\{R \to *S\}$ d) $\{F \to id\}$ and $\{R \to \epsilon\}$

Scanned with CamScanner

	id	*	$
S	S→FR		
R		R→*S	R→ϵ

∴ opt (a)

Q.36 / G-07 Consider the grammar with non-terminals $N=\{S, C, S_1\}$ terminals $T=\{a, b, i, t, e\}$ with S as the start symbol and the fo

$S \rightarrow iCtSS_1 \mid a$

$S_1 \rightarrow eS \mid \epsilon$

$C \rightarrow b$

The grammar is not LL(1) because:

a) it is left recursive

b) it is right recursive

c) It is ambiguous

d) it is not context-free

sol:

a) It is not left recursive

b) right recursive in not related to LL(1)

d) LL(1) is defined only for CFG

also the grammar given is CFG

c) It is ambiguous

The grammar is ambiguous.

The grammar represents dangling else and ambiguity arises in matching else. with then.

Eg: iCtSS$_1$ | iCtSS$_1$

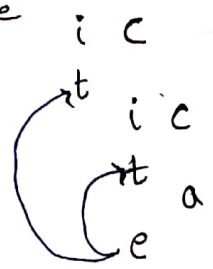
iCtiCtS$_1$ | iCtiCt

ibtibtS$_1$

ibtibteS

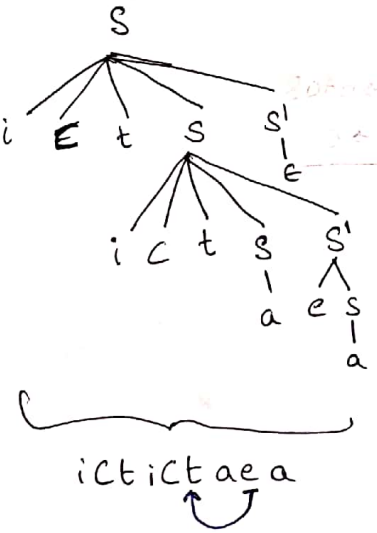
ibtibtea

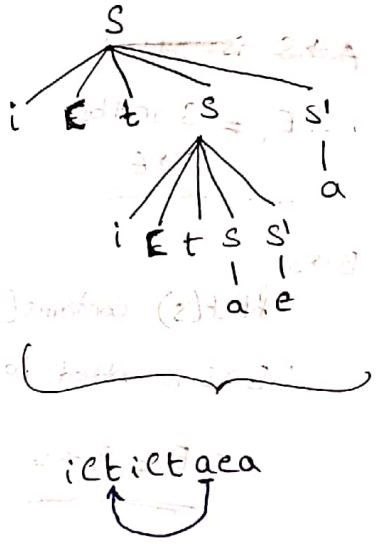
For code



Ambiguity is to which t we should match e i.e., matching else to then

> However while designing a compiler this problem is rectified by matching else to closest previous if.





Q37 / G-12 (4M) For the grammar below, a partial LL(1) parsing table is also presented along with the grammar. Entries that need to be filled are indicated as E_1, E_2 and E_3. ϵ is the empty string, $ indicates end of input and, | seperates alternate right hand sides of productions.

$S \rightarrow aAbB \mid bAaB \mid \epsilon$

$A \rightarrow S$

$B \rightarrow S$

	a	b	$
S	E_1	E_2	$S \rightarrow \epsilon$
A	$A \rightarrow S$	$A \rightarrow S$	error
B	$B \rightarrow S$	$B \rightarrow S$	E_3

(i) The FIRST and FOLLOW sets for A and B are _____

(ii) The appropriate entries for E_1, E_2 and E_3 are _____

Scanned with CamScanner

(i)

	FIRST	FOLLOW
S	a, b, ε	$, a, b
A	a, b, ε	b, a
B	a, b, ε	$, a, b

} Answer

(ii) S → aAbB is added to M[S, a]

~~S → bb~~

S → bAaB is added to M[S, b]

S → ε is added to M[S, a] M[S, b] M[S, $]

~~A → S is added~~

∴ E_1 = S → aAbB, S → ε

E_2 = S → bAaB, S → ε

B → S

∵ first(S) contains (ε)

∴ B → S is added to follow(B)

∴ E_3 = B → S

Error Recovery in Predictive parsing:

(i) Panic mode Recovery

(ii) Phrase recovery: error entries contains pointers to errors ~~routine~~ routines

Not needed for gate

25/06/20

Bottom-up Parsing:

In this type of parsing the parse tree is constructed from leaves (input string) and reduce it towards the root (starting non-terminal)

* It uses right most derivation in reverse.

Scanned with CamScanner

Eg: Consider the below grammar

$S \rightarrow aAcBe$

$A \rightarrow Ab \mid b$

$B \rightarrow d$

i/p : abbcde

The parser scans from left to right searching for a handle.

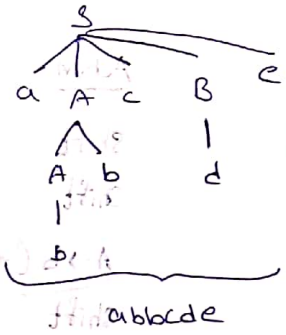
Ex. After finding a handle, handle pruning is done.

abbcde
↳ Handle found

aAbcde	$A \rightarrow b$
aAcde	$A \rightarrow Ab$
aAcBe	$B \rightarrow d$
S	$S \rightarrow aAcBe$

Here derivation is clearly reverse
Also we can see it RMD
∴ RMD in reverse

Parse tree construction



Handle: It is RHS of any production which present as a substring of input string.

Handle pruning: It is a process of substituting handle with the L.H.S of a production.

Shift Reduce Parser

It is implemented using a stack to hold the grammar symbols and uses input buffer for holding input string.

Scanned with CamScanner

Initial Configuration

Stack	input
$	w$

Accept Configuration:

Stack	input
$S	$

> The contents of stack during SR parsing are actually prefixes of right sentential forms But not all prefixes are found on the stack. The prefixes of right sentential forms that can appear on the stack of a shift reduce parser are called viable prefixes

Shift reduce parser performs any of below 4 actions:

(i) Shift: Moving i/p. symbols onto the stack

(ii) Reduce: Pop R.H.S (handle) and push appropriate non-terminal

(iii) Accept: If the stack contains start symbol and i/p buffer contains $.

(iv) Error: Unsuccessful parsing.

Eg: For the previous example below is the process of shift reduce parsing.

Stack	input	Action
$	abbcde$	shift
$a	bbcde$	shift
$ab	bcde$	A→b (reduce)
$aA	bcde$	shift
$aAb	cde$	A→Ab (reduce)
$aA	cde$	shift
$aAc	de$	shift
$aAcd	e$	B→d (reduce)
$aAcB	e$	shift
$aAcBe	$	S→aAcBe (reduce)
$S	$	Accept

Construction of parse tree

→ All bottom up parsers use shift reduce techniques. So all bottom-up parsers are called shift Reduce (SR) parsers. Other parsers use it more effective and accurately.

Scanned with CamScanner

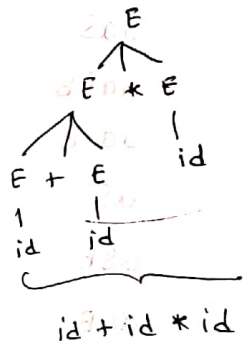
Eg: Consider below grammar and trace the steps of shift reduce parser for given i/p string.

Handle appears only on top of the stack

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

i/p: id+id*id

stack	input	Actions
$	id+id*id$	shift
$id	+id*id$	E→id
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	E→id
$E+E	*id$	E→E+E
$E	*id$	shift
$E*	id$	shift
$E*id	$	E→id
$E*E	$	E→E*E
$E	$	Accept



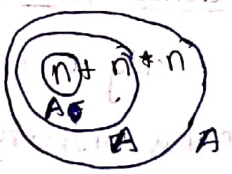
Given grammar is ambiguous

∴ Shift Reduce parsers are also possible for some of ambiguous grammar.

P/33 $A \to A+n \mid A*n \mid n$

i/p: n+n*n

$	n+n*n$
$n	+n*n$
$A	+n*n$
$A+n	*n$
$A	*n$
$A*n	$
$A	$



∴ n, A+n, A*n

Scanned with CamScanner

Q38 / G-20 Consider the following grammar:

$S \to aSB \mid d$

$B \to b$.

The number of reduction steps by a bottom up parser while accepting the string aaadbbb

Sol:

aaad	bbb	S→d
aaaS	bbb	
aaaSb	bb	B→b
aaaSB	bb	S→aSB
aaS	bb	
aaSb	b	B→b
aaSB	b	S→aSB
aS	b	
aSb	$	B→b
aSB	$	S→aSB
S	$	Accept

∴ 7 steps

aaadbbb	S→d
aaaSbbb	B→b
aaaSBbb	S→aSB
aaSbb	B→b
aaSBb	S→aSB
aSb	B→b
aSB	S→aSB
S	∴ accept

Also since the grammar is unambiguous we can check for no of derivations (steps) directly

$S \to aSB$
$\to aaSBB$
$\to aaaSBBB$
$\to aaadBBB$
$\to aaadbBB$
$\to aaadbbB$
$\to aaadbbb$

} 7 steps

Operator Precedence Parsers:

It accepts only a specific form of gramamars called operator grammars which has property that it is a null free grammar and no two non-terminal are adjacent.

Eg: $S \to \epsilon$

It is not an operator grammar because it is not null free.

Scanned with CamScanner

* $S \to AB$

The above is also not an operator grammar.

* $S \to \varepsilon$

The above is an operator grammar

∵ it is null free

and no two non terminals are adjacent.

* $S \to AaB$

It is operator grammar.

* $S \to aAb$

It is an operator grammar

* $S \to AB$

$A \to aA$

$B \to bS|d$

The above grammar is not an operator grammar (∵ $S \to AB$)

Now we modify the grammar and make it operator grammar

$S \to AbS \mid Ad$

$A \to aA$

$B \to bS|d$

Now the above grammar is an operator grammar.

→ The operator precedence parser uses three precedence relation b/w terminal (operators). These precedence relations guide the selection of handles.

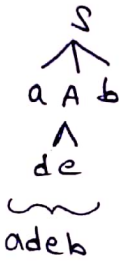
Relation	Meaning
$a < b$	a gives precedence to b
$a = b$	a has same precedence as b
$a > b$	a takes precedence over b

→ Precedence relation is used to delimit the handles with $<$ marking left end of the handle and $>$ marking the right end of the handle and $=$ may appear at the interior of the handle.

Scanned with CamScanner

Eg:

S → aAb
A → de



parsing:

$ a <d = e> b $

$ < a A b > $

$ S $

Now we can write

$ < a <d = e> b > $

→ Everything present b/w < and > is considered handle.

Eg: Construct an operator precedence parse table for the following grammar using precedence and associativity (* > +)

E → E+E / E*E / id

Now we create precedence rules

	id	+	*	$
id	error	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	accept

id id : we never get two id's consecutively ∴ precedence is not defined
id + : id > +
id * : id > *
id $: id > $

Scanned with CamScanner

for id + +

consider id + id

Here we 1st reduce id to E

E + id
E + E
E

∴ id has more precedence than +

Shy for id + *, id + $

Now + + id

id + id
id is reduced first
i.e. + gives precedence to id
id + <id>
id + E

Now + + +

consider id + id + id

Like in the case of id + id we never get + + + adjacent but still it is valid because consider
id + id + id ≡ E + id + id
= E + E + id
= <E + E> + id

it is considered <id + id> + id (Here we need to consider associativity)

+ + *

id + id * id
≡ id + <id * id>

Here we check precedence rule b/w terminal only.
Since we have + E + ≡ +E> +
we check b/ precedence b/w to + operators.

+ + $

$<id + id> $

$ + id

$<id> $

$ + +

$ <id + id> $

$ + $

we cannot have two $'s adjacent ∴

If two $'s are adjacent it means there is nothing to be parsed and hence considered accepted.

Scanned with CamScanner

Now consider input string id + id * id

Now we write the ip string with precedence relations.

$$\$ < id > + < id > * < id > \$$$

Now parser scans the above string from left to right.

Now < > is considered handle (first appeared one from left to right)

$$\$ < id > + < id > * < id > \$ \quad (E \to id)$$

$$\$ < E + < id > * < id > \$ \quad (E \to id)$$

↳ Here E is written only for understanding

No terminal is present in precedence relation

$\therefore \$ < E +$ ↓ says $\$ <$ +

E is actually not present there

The scanning is done from left to right searching for >. After finding >, now it scans backward and searches of <. The handle is found b/w <+>

Now scanning above string the first appearance of < ... > will undergo handle pruning.

$$\$ < E + < E * < id > \$ \quad (E \to id)$$

↳ This actually seen as

$$\$ < + < * < id > \$$$

$$\$ < E + < E * E > \$ \quad (E \to E * E)$$

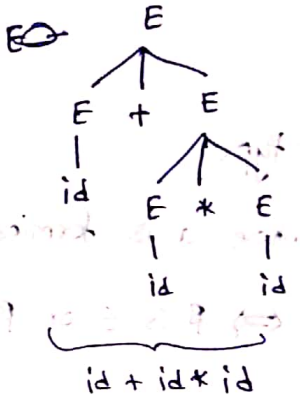
$$\$ < E + E > \$ \quad (E \to E + E)$$

$$\$ E \$$$

The above actually means $ $

$\therefore$ Accepted

Now construction of parse tree is done in the reverse of generated production



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

Note:

In operator grammar it is not neccessary that we need to have mathematical operators. operator just means a terminal.

Lead and Trail sets:

Construction of operator precedence table is aided by lead and trail sets.

Lead set:

It means leading symbols of the grammar.

① For a production

$A \rightarrow \alpha a \beta$, where a is terminal

$Lead(A) = \{a\}$ iff α is 'ϵ' or single non terminal

Eg: For $S \rightarrow aAbB$

$Lead(S) = \{a\}$

~~For A → BcD~~

~~Lead(A) = {c}~~

~~For B → d~~

~~Lead(B) = {d}~~

Scanned with CamScanner

② For a production of type

$A \to B\alpha$, $Lead(A) = Lead(B)$

Trail set :

① For a production of type

$A \to \alpha a \beta$, where a is terminal

$Trail(A) = \{a\} \iff \beta$ is ϵ or β is single non-terminal

② For a production of type

~~$A \to \beta$~~ $A \to \alpha B$

$Trail(A) = Trail(B)$

Eg: For below grammar, construct lead and trail sets.

$S \to aAbB$
$A \to BcD$
$B \to d$
$D \to e$

	Lead	Trail
S	a	b, d
A	c, d	c, e
B	d	c
D	e	e

Eg: Find lead & trail sets for below grammar

$S \to AaBb$
$A \to BcS$
$B \to dA$

	Lead	Trail
S	a, c, d	b
A	c, d	c, b
B	d	d, c, b

Computation of precedence relations:

Consider production

$A \to x_1 x_2 \cdots x_n$

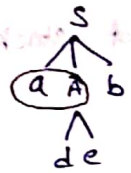
(i) If x_i and x_{i+1} are terminals $x_i = x_{i+1}$ $(S \to ab \Rightarrow a = b)$

(ii) If x_i and x_{i+2} are terminals $x_i = x_{i+2}$ $(S \to aAb \Rightarrow a = b)$
and x_{i+1} is single non-terminal

Scanned with CamScanner

(iii) If X_i is terminal and X_{i+1} is non-terminal

then $X_i < lead(X_{i+1})$



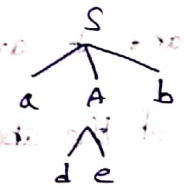
$S \to aAb$

$ a < d e > b $

$a < lead(A)$

(iv) If X_i is non-terminal and X_{i+1} is terminal

then $Trail(X_i) \doteq X_{i+1}$



$S \to aAb$

$ a < d e > b $

(v) $ < Lead(S)

(vi) $Trail(S) > $

} where S is start symbol

$ < a < d e > b > $

a is lead of S

b is trail of S

$S \overset{*}{\to} aAb$

Using these rules if we construct OP prec parse for (Eg: Pg:34) the table is

	id	+	*	$
id	err	>	>	>
+	<	>	>	>
*	<	<	>	>
$	<	<	<	Acc

This is due to ambiguity of grammar. In the previous example we have assume associativity & precedence to eliminate this problem

26/06/20

Eg: Construct operator precedence parse table for below grammar

$S \to aAd$

$A \to bc$

	a	b	c	d	$
a		<		=	
b			=		
c				>	
d					>
$	<				Accept

	Lead	trail
S	a	d
A	b	c

(a=d is not same as d=a
a<b is not same as b>a
and silly any other rules)

$S \to aAd$
$A \to bc$
$b \doteq c$

$S \to aAd$
$a < Lead(A)$

$S \to aAd$
$trail(A) > d$

$ < Lead(S)$
$Trail(S) > $

All the unfilled entries are considered error entries

Scanned with CamScanner

Operator Precedence Parsing Algorithm

If 'a' is on top of stack and 'b' is symbol pointed by input pointed

① If $a < b$ or $a = b$, then push b onto the stack

② If $a > b$, the pop the stack until the top of stack is related by $<$ to the most recently popped symbol.

③ If a and b are \$; then successful completion

④ If none of the above steps are possible, then error.

Eg: Consider abcd for previous grammar

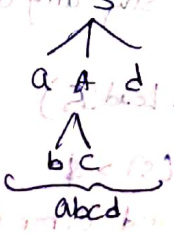
Stack	Input	Action
\$	abcd \$	Push (∵ \$ < a)
\$a	bcd \$	Push (∵ a < b)
\$ab	cd \$	Push (∵ b = c)
\$abc	d \$	pop until ⓐ rule ① satisfies (∵ c > d)
\$ab	d \$	pop (∵ b = c)
\$a	d \$	Stop popping (∵ a < b)
\$a	d \$	Push (∵ a = d)
\$ad	\$	start popping (∵ d > \$)
\$a	\$	pop (∵ a > b)
\$	\$	Accept

bc → Handle

A → bc

S → aAd

Parse tree Construction



Scanned with CamScanner

Q39) In the operator grammar given below which of the following operator precedence relations are correct.

a) ~~* > +~~ b) ~~- > *~~ c) ~~+ > -~~ d) ~~* < -~~

$E \to E * F$
$E \to F + E$
$E \to F - E$
$E \to F$
$F \to id$

a) $* > +$ b) $- > *$ c) $+ \geq -$ d) $* < -$

Sol:

Here the question is asked on operator grammar.

So we need to consider operator precedence relations.

$>$ is possible with only Trail(N.T) > Terminal

$<$ is possible only with Terminal < Trail(N.T)

a) $* > +$

Trail(N.T) > +

Thus we need to have some non-terminal A containing * such that they are of form A+

The only production with + is

$E \to F + E$

Trail(F) = id

id > +

∴ opt (a) is not the answer

	lead	Trail
E	*, +, -, id	*, id, +, -
F	id	id

b) $- > *$

Trail(N.T) > *

$E \to F + E$ ∴

Scanned with CamScanner

$\{-, +, *, id\} > *$

$-> *$

∴ opt (b) is correct

c) $+ > -$

$Trail(N.T) > -$

$E \to F - E$

$id > -$

$\therefore \alpha$

d) $* < -$

$terminal < lead(N.T)$

$* < lead(N.T)$

$E \to E * F$

$* < lead(F)$

$* < id$

$\therefore \alpha$

∴ opt (b)

Que 0 / G-04 Which of the following grammar rules violate the requirements of an operator grammars? P, Q, R are non-terminals and q, s, t are terminals.

(i) $P \to QR$ (ii) $P \to QsR$ (iii) $P \to \epsilon$ (iv) $P \to QtRq$

a) i, b) (i) & (iii) c) (ii) and (iii) d) (iii) and (iv)

Sol:

(i) $P \to QR$

two N.Ts adjacent

∴ not op grammar

(ii) ✓

(iii) Operator grammar should be null free

∴ not op grammar

(iv) ✓

∴ opt (b)

Scanned with CamScanner

Material Problems:

P/33 $S \to AB$ ✗ (two N.Ts adjacent)

$S \to AaB$ ✓

$S \to a$ ✓

$S \to \epsilon$ ✗ (not null free)

∴ I & III

∴ opt (d)

P/34 c) $S \to AaB$

$A \to aA \mid b$

$B \to bB \mid \epsilon \Rightarrow B \to \epsilon$

∴ opt (c)

P/35 $S \to AB$

$A \to c \mid d$

$B \to aAB \mid d$

a) $S \to Aa \mid Ab$

$A \to c \mid d$

It finite grammar where as grammar in the question is infinite

∴ ✗

b) $S \to AaAB$

not operator grammar

c) $S \to AaS \mid Ab$

The terminal 'b' is not even present in the question

∴ none

opt (d)

one of the ways to convert given grammar into operator grammar is

$S \to AB$

$A \to c \mid d$ $\equiv$

$B \to aS \mid d$

$\downarrow$

$\because S \to AB$

$S \to AaS \mid Ad$ (∵ substituting B)

$A \to c \mid d$

$B \to aS \mid d$

Now this grammar is an operator grammar.

Scanned with CamScanner

P/36

E → E + E
E → E * E
E → E - E
E → E / E
E → E ^ E
E → (E)
E → id

^ is right associative

In, a^b^c b^c is evaluated first

∴ ^ < ^

-, +, * are left associative

∴ - > - , + > + , * > *

∴ opt (d)

P/37

Lead(B) = {e}

Lead(A) = Lead(B) ∪ {c}

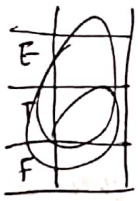
= {e, c, d}

Lead(S) = Lead(A) ∪ {a}

= {a, c, e, d}

∴ opt (B)

P/38



E	+, *, id,)
T	*, id,)
F	id,)

Lead(F) = {id,)}

Lead(T) = {*} ∪ Lead(F)

Lead(E) = {+} ∪ Lead(T)

= {+, *,), id}

∴ opt (B)

Scanned with CamScanner

P/39 It is mentioned operator grammar. So we should consider operator precedence relations.

	Lead	Trail
E	+, ↑, id	+, ↑, id
T	↑, id	↑, id
F	id	id

a) ↑ ≥ ↑

Trail(N·T) ≥ ↑

T → F ↑ T

Trail(F) ≥ ↑

~~id ≥ id~~ id ≥ ↑

b) ↑ > +

Trail(N·T) > +

E → E + T

Trail(E) > +

↑ > + ✓

c) + ≥ ↑

Trail(N·T) ≥ ↑

T → F ↑ T

Trail(F) ≥ ↑

id ≥ ↑

∴ α

d) + < +

T < lead(N·T)

E → E + T

+ < lead(T)

+ < ↑ and + < id

∴ α

P/40 S → aSb | Ac

A → d

	Lead	Trail
S	a, d, c	c, b
A	d	d

S → aSb

a = b ✓

a < lead(S)

∴ a < a, a < d, a < c ✓

Trail(S) > b

c > b, b > b

S → Ac

trail(A) > c

d > c ✓

A → d

↳ nothing can be concluded

only c < b is incorrect

∴ opt (d)

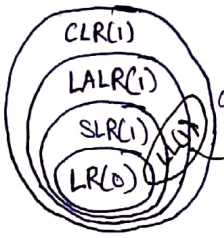
Scanned with CamScanner

LR(k) Parser:

* L stands for Left to Right scanning of i/p
* R for Right most derivations in reverse
* k is no of look ahead symbols.

LR - Family:

$$LR(0) \subseteq SLR(1) \subseteq LALR(1) \subseteq CLR(1)\ [LR(1)]$$



grammars → LL(1), but not true for LL(k) $\forall k \geq 2$

From the figure

CLR(1) is the most powerful parser

Every LR(0) is SLR(1), LALR(1), CLR(1) (LR(1))

Every SLR(1) need not to be LR(0)

Every SLR(1) is LALR(1), LR(1)

Note:

Every LL(1) is also CLR(1) but the reverse need not to be true

No Ambiguous grammar is LR(k)

P/47 d) LL(1) can also be considered as LL(2)

∴ There exists a grammar which LL(1) & LR(0)

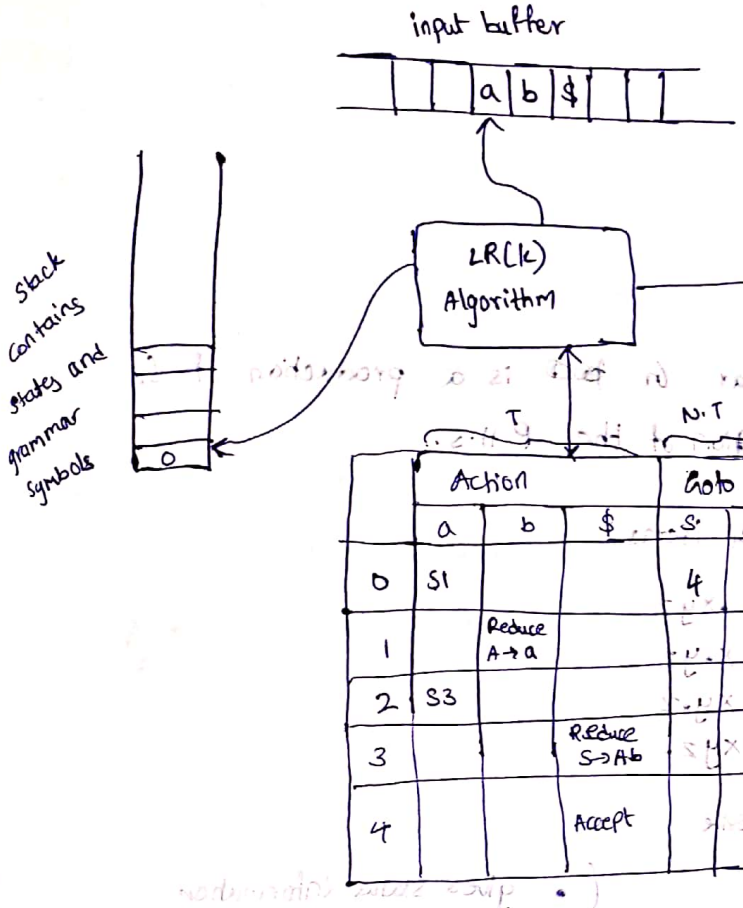
∴ There exists a grammar which LL(2) & LR(0)

∴ d is false

Scanned with CamScanner

LR(k) Parser Structure:

The passing can be implemented by DFA. Any state containing a conflict is called inadequate state



	Action			Goto	
	a	b	$	S	A
0	S1			4	2
1		Reduce A→a			
2	S3				
3			Reduce S→Ab		
4			Accept		

Of all parsers LR parsers detect syntax errors as soon as possible

Actions

① Shift j : push i/p onto the stack and push state j

② Reduce $A \to \alpha$: (i) POP $2 * |\alpha|$ symbols

(ii) Push L.H.S of the production

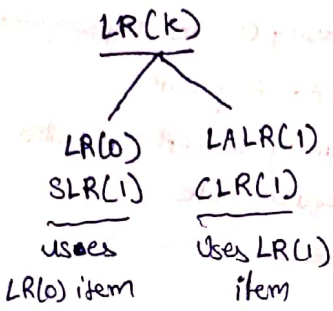
(iii) Push [Goto (current state, N.T (A))]

③ Accept : Successful Completion of parsing

④ Error : Unsuccessful Parse.

→ All the LR parsers have same structure and uses same algorithm for parsing. The only difference is the way they construct the parse table.

Scanned with CamScanner



LR(0) item:

An LR(0) item of a grammar G is a production of G with a 'dot' at some position of the R.H.S.

Eg: $A \to xyz$ generates 4 items.

$A \to \cdot xyz$

$A \to x \cdot yz$

$A \to xy \cdot z$

$A \to xyz \cdot$

derived ← • → expect to derive

$x \cdot yz$

derived using x — expecting to derive using yz

(• gives status information telling till which part of the production is derived)

Note:

$A \to \epsilon$ generates only one item

i.e., $A \to \cdot$ (It means derivation is over)

Finding LR(0) items:

1) Augument the grammar with production $S' \to S$

$S' \to \cdot S$ (Start state)

$S' \to S \cdot$ (end state)

This augument production is used to known when the production is started and when it is ended.

Scanned with CamScanner

2) Find the closure(I):

If $A \to \alpha . B\beta$ is in the item I

- then add $B \to .\gamma$ to I

Eg:

$S' \to .S$
$S \to .Aa$
$A \to .Bb$
$B \to .d$

This is considered one state (one closure set)

3) Find Goto(I,X):

Goto(I,X) is defined to be the closure set of all items $A \to \alpha X . \beta$ such that $A \to \alpha . X \beta$ is in I

Here X can be either terminal or non-terminal

$A \to \alpha . X \beta \longrightarrow A \to \alpha X . \beta$

Eg: $E \to E+T$
$E \to T$
$T \to id$

Q) Find LR(0) items for above grammar

① Augument the grammar

i.e., add $E' \to E$

② Find closure of $\{E' \to E\}$

$E' \to .E$
$E \to .E+T$
$E \to .T$
$T \to .id$

③ Finding goto

kernel items:

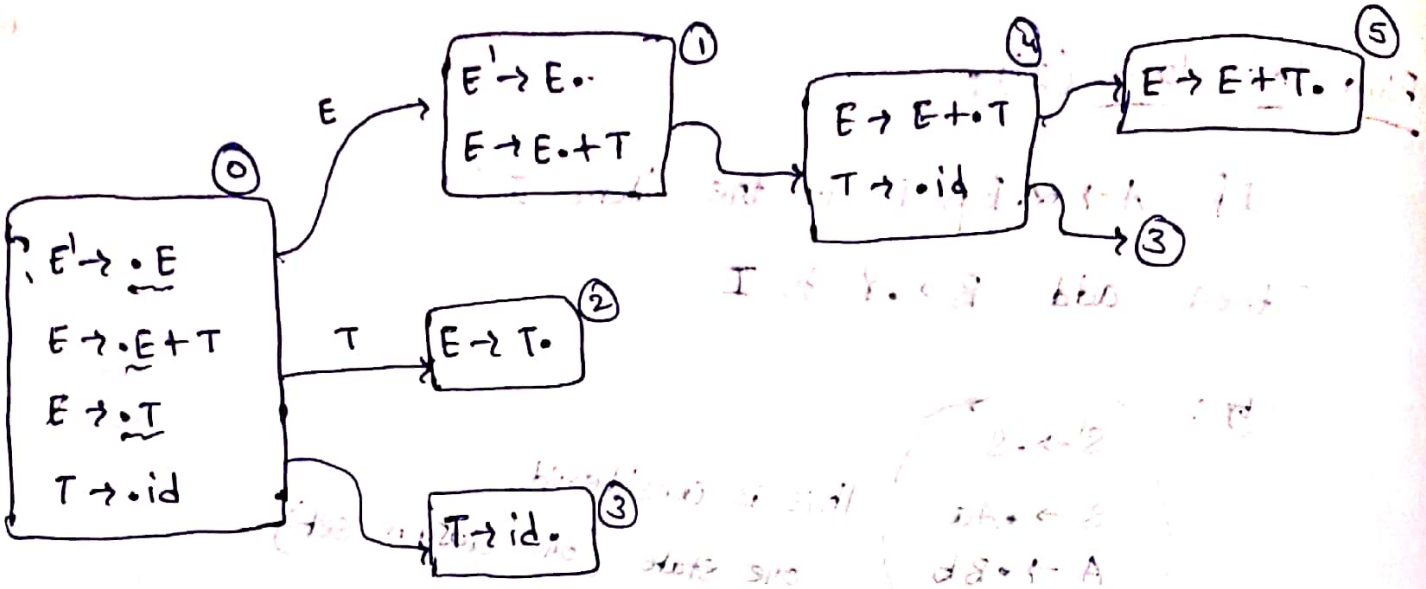
$S' \to .S$

and all items whose dots are not at the left end

Non-kernel items:

all items with their dots at the left end except $S' \to .S$

Scanned with CamScanner



∴ The parser has 6 states (0 to 5)

Rules for constructing LR(0) parsing table:

① If $S' \to S.$ is in state I then (. at end of augmented production)

$$\boxed{\text{Action}[I, \$] = \text{Accept}}$$

② If $A \to \alpha.a\beta$ is in state I then

$$\boxed{\text{Action}[I, a] = \text{shift } j}$$ (a is terminal)

(. terminal)

③ If $A \to \alpha.$ is in state I then

↓ Except for augmented production

$$\boxed{\text{Action}[I, \text{all terminals}] = \text{Reduce } A \to \alpha}$$ (. at end of production)

④ If $A \to \alpha.B\beta$ is in state I then

$$\boxed{\text{Goto}[I, B] = j}$$ (. non-terminal)

All the unfilled entries are considered error entries.

Scanned with CamScanner

LR(0) Parse table :

	Action			Goto	
	id	+	$	E	T
0	S3			1	2
1		S4	Accept		
2	E→T	E→T	E→T		
3	T→id	T→id	T→id		
4	S3				5
5	E→E+T	E→E+T	E→E+T		

The parse table has no multiply defined entries.

∴ The grammar is said to be LR(0)

∴ It is also SLR(1), LALR(1) & CLR(1)

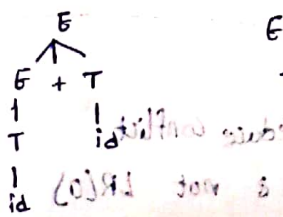
Parsing :

Consider input id+id

stack	input	Actions
0	id+id$	S3
0id3	+id$	Reduce by T→id
0T2	+id$	Reduce by E→T
0E1	+id$	S4
0E1+4	id$	S3
0E1+4id3	$	Reduce by T→id
0E1+4T5	$	Reduce by E→E+T
0E1	$	Accept

RMD in reverse

Parse is



E → E+T
→ E+id
→ T+id
→ id+id

Scanned with CamScanner

LR(0) Parser Conflicts:

1) Shift - Reduce Conflict:

If any state of LR(0) parse contain item of the form

$A \to \alpha . a\beta$ (I)
$B \to \alpha .$

For $A \to \alpha . a\beta$

Action[I, a] = shift j

For $\beta \to \alpha .$

Action[I, all terminals] = Reduce $\beta \to \alpha$

∴ Action[I, a] contains both shift & Reduce and parse can't determine which action to perform.

2) Reduce - Reduce Conflict:

If any state of LR(0) parser contains item of form

$A \to \alpha .$ (I)
$B \to \alpha .$

	all terminal
I	Reduce $A \to \alpha$
	Reduce $B \to \alpha$

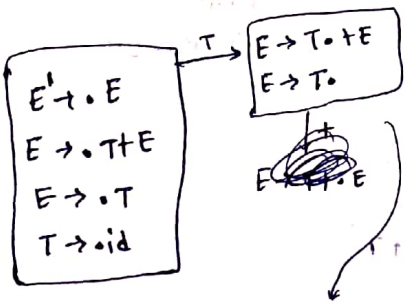
} Conflict

Eg: Check whether the following grammar is LR(0) or not.

$E \to T + E$

$E \to T$

$T \to id$



In this I state

$E \to T . + E$ gives shift action

$E \to T .$ give reduce

∴ shift - Reduce Conflict

∴ The grammar is not LR(0)

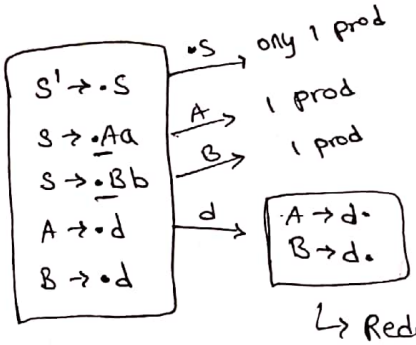
Shortcut:

Here we don't construct ~~table~~ the parse table. we just build states and determine whether it is LR(0) or not. Also we don't need to generate all the states (states with only one production). we generate states with multiple ~~entries~~ productions only

Scanned with CamScanner

Qui) Check whether the below grammar is LR(0) or not

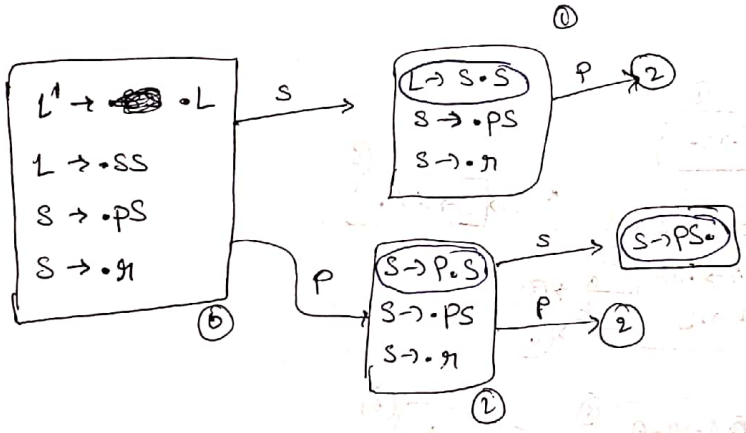
$S \to Aa$
$S \to Bb$
$A \to d$
$B \to d$



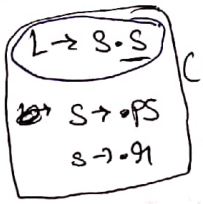
↳ Reduce-Reduce conflict

Material Problems:

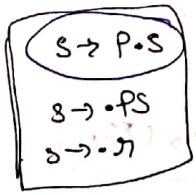
Plug) $L \to SS$
$S \to pS \mid r$



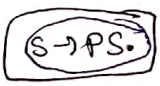
Method 2: given $L \to S \cdot S$, $S \to P \cdot S$ $S \to pS \cdot$



(∵ we have •S we must have other S productions also)



(∵ •S is present)



∴ opt ⓐ

Scanned with CamScanner

P/50

$E \to FR$

$R \to *E$

$R \to \epsilon$

$F \to id$

(Question is mistake which two)

(i) $E \to .FR$

(ii) $E \to .*R$

(iii) $F \to .id$

(i) $E \to .FR$ (ii) $E \to .*R$ (iii) $F \to .id$

$F \to .id$

(iii)

$\therefore$ (i) & (iii) $\therefore$ opt (b)

P/51

$S' \to .S$

$S \to .SB$

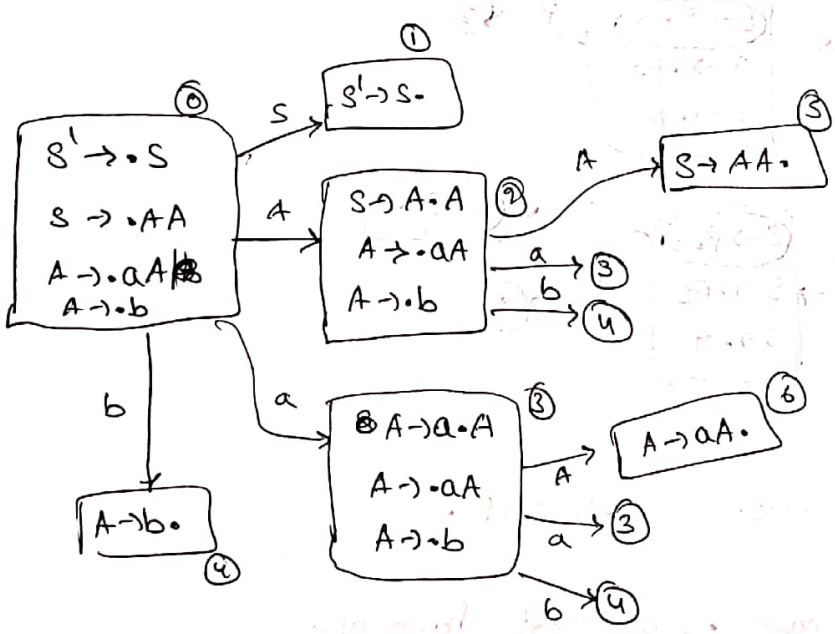
$S \to .A$

$A \to .a$

($\because$ we have $.S$ add 'S' productions)

(we don't add ~~B since~~ B-productions since we don't have $.B$)

P/52



$\therefore$ 0 to 6

i.e., 7 states

shift-reduce

$E \to E+.T$ — T → $E \to E+T.$

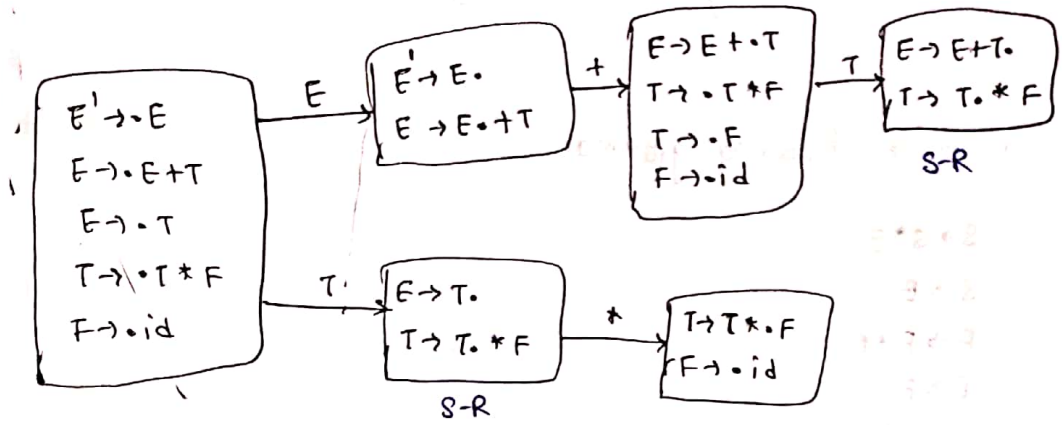
Scanned with CamScanner

P/s3 same as previous example

It is LR(0)

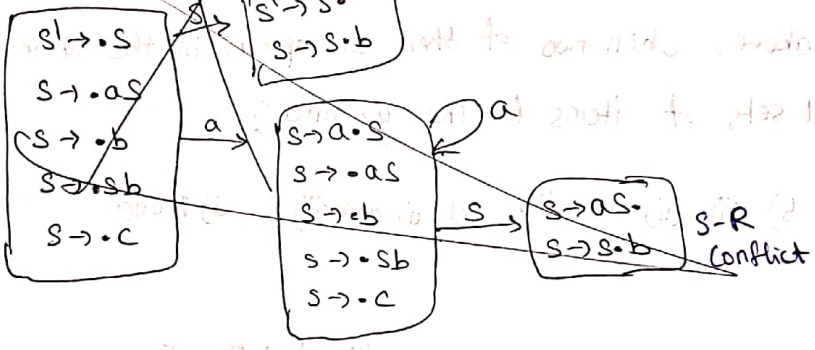
∴ CLR(1)

P/s4

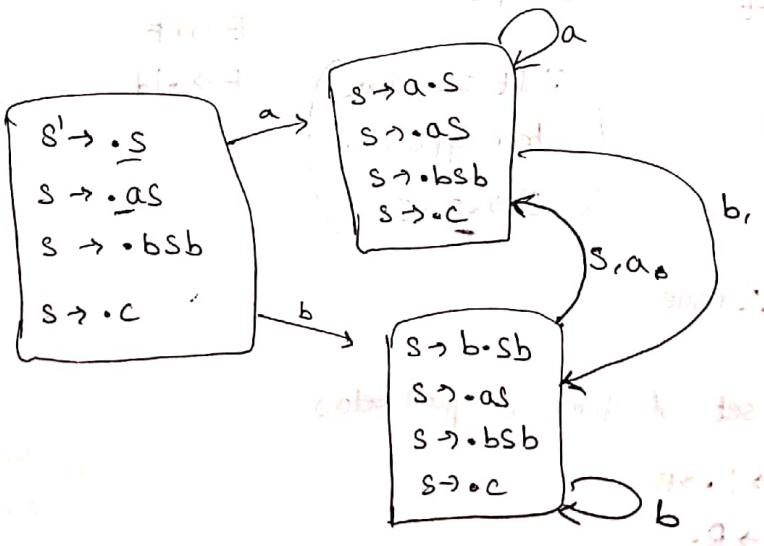


∴ 2 conflicts.

P/s5 S → aS | b | sb | c



P/s5



~~X~~ LR(0)

∴ not LR(0) is false

Scanned with CamScanner

(f/56) $A \to A+A \mid i$

ambiguous

∴ doesn't belong to LR(k) family

(Q42 / 9-06) Consider the following grammar

$S \to S*E$

$S \to E$

$E \to F+E$

$E \to F$

$F \to id$

Consider the following LR(0) items corresponding to the grammar above

(i) $S \to S* \cdot E$

(ii) $E \to F \cdot +E$

(iii) $E \to F+ \cdot E$

Given the items above, which two of them will appear in the same set in the canonical sets of items for the grammar?

a) (i) & (ii) b) (ii) and (iii) c) (i) and (iii) d) None

(i) $S \to S* \cdot E$

$E \to \cdot F+E$

$E \to \cdot F$

$F \to \cdot id$

(ii) $E \to F \cdot +E$

$E \to F \cdot$

(∵ It could have been applied on $E \to \cdot F+E$)

(iii) $E \to F+ \cdot E$

$E \to \cdot F+E$

$E \to \cdot F$

$F \to \cdot id$

∴ none

(Q43 / 9-14) A canonical set of items is given below

$S \to L \cdot > R$

$S \to R \cdot$

Scanned with CamScanner

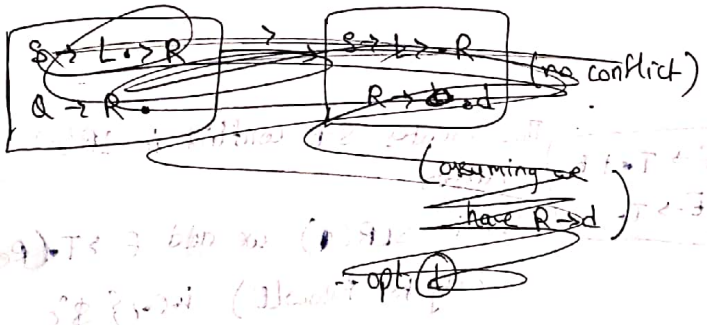
on the input symbol $ the set has

a) S-R and R-R conflict

b) S-R but not R-R

c) R-R but not S-R

d) neither S-R and nor R-R



we don't have < in given question

∴ neither S-R nor R-R (It generates error instead)
opt (d)

If the question is asked with '>' the answer would be

shift-reduce conflict (opt-(b))

27/06/20

Q4 / 9-19 Consider the ~~alphabet~~ augmented grammar given below:

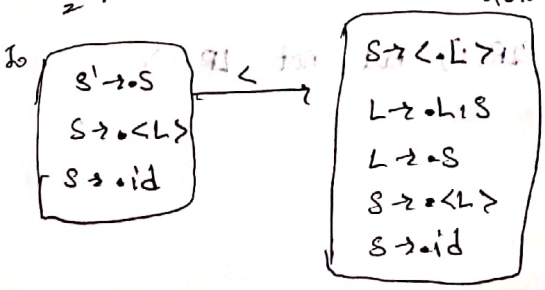
$S' \to S$

$S \to <L> \mid id$

~~$L \to L,S \mid S$~~ $L \to L,S \mid S$

Let I_0 = CLOSURE($\{[S' \to S]\}$). The number of items in the set GOTO(I_0, <) is ____

Sol:



∴ 5 items

Scanned with CamScanner

SLR(1) Parser: (Simple LR(1))

Constructing SLR(1) is same as LR(0), except for reduce entries

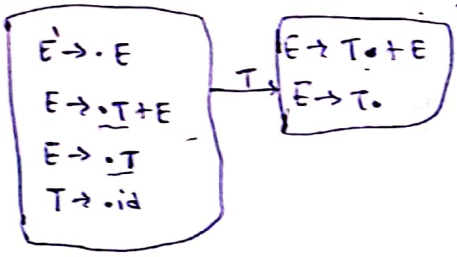
i.e., if $A \to \alpha\cdot$ is in state I then

$$\text{Action}(I, \text{FOLLOW}(A)) = \text{Reduce } A \to \alpha\cdot$$

Eg: $E \to T+E$

$E \to T$

$T \to id$



This creates S-R conflict in LR(0) parser

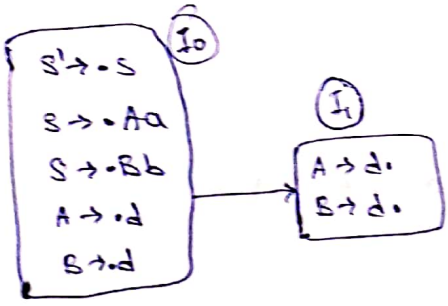
But in SLR(1) we add $E \to T\cdot$ (Reduce) only to follow(E) i.e., $\{\$\}$

∴ no conflict.

∴ The grammar is not LR(0)

But it is SLR(1)

Eg: Check whether the below grammar is SLR(1) or not.



In SLR(1) parse

add $A \to d$ to follow(A) = {a}

add $B \to d$ to follow(B) = {b}

∴ we don't have any conflict

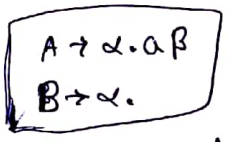
Thus above grammar is SLR(1) but not LR(0)

Scanned with CamScanner

Parser Conflict in SLR(1):

i) Shift-Reduce Conflict:

If any state has items of type

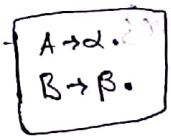


~~Action[I, a] = shift j~~

S-R conflict occurs if $\{a\} \cap \text{Follow}(B) \neq \phi$

ii) Reduce-Reduce Conflict:

If any state has items of type



R-R conflict occurs if $\text{Follow}(A) \cap \text{Follow}(B) \neq \phi$

Eg: Check whether the below grammar is SLR(1) or not.

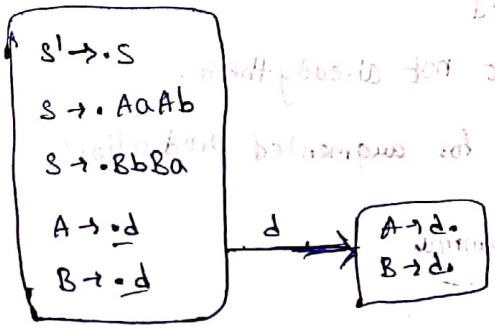
$S \to AaAb$

$S \to BbBa$

$A \to d$

$B \to d$

Sol:



$\text{Follow}(A) = \{a, b\}$

$\text{Follow}(B) = \{a, b\}$

$\text{Follow}(A) \cap \text{Follow}(B) = \{a, b\} \neq \phi$

∴ R-R conflict.

not SLR(1)

Scanned with CamScanner

5.

Check whether the below grammar is SLR(1) or not

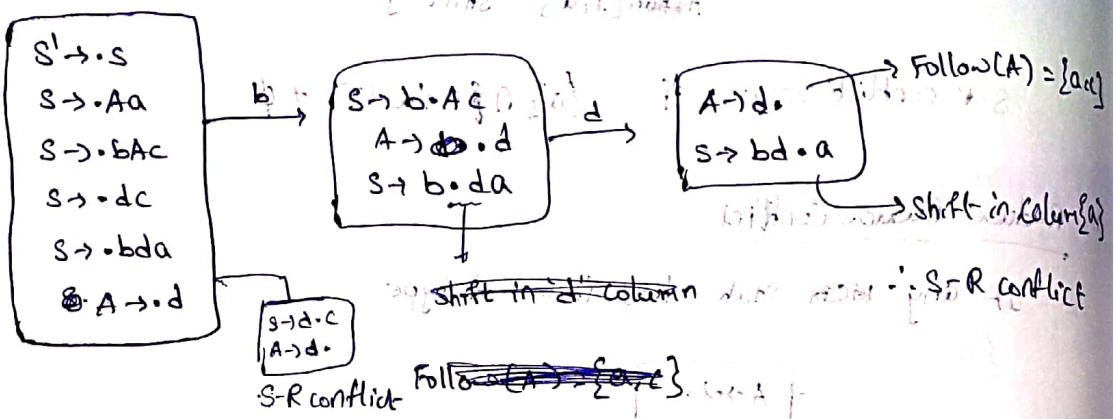
$S \to Aa$

$S \to bAc$

$S \to dc$

$S \to bda$

$A \to d$



∴ not SLR(1)

Note :

* CLR(1)
  LALR(1)
  } uses LR(1) items

i.e., LR(0) item + 1 Look ahead = LR(1) item.

LR(1) items:

Finding LR(1) items is same as LR(0) except for finding closure(I).

i.e., If $A \to \alpha.B\beta, a$ is in state I and

$B \to \gamma$ is a production then add

$B \to .\gamma$, First(βa) to I if it not already there.

Initially we add $ as lookahead for augmented production.

5: Construct LR(1) items for below grammar

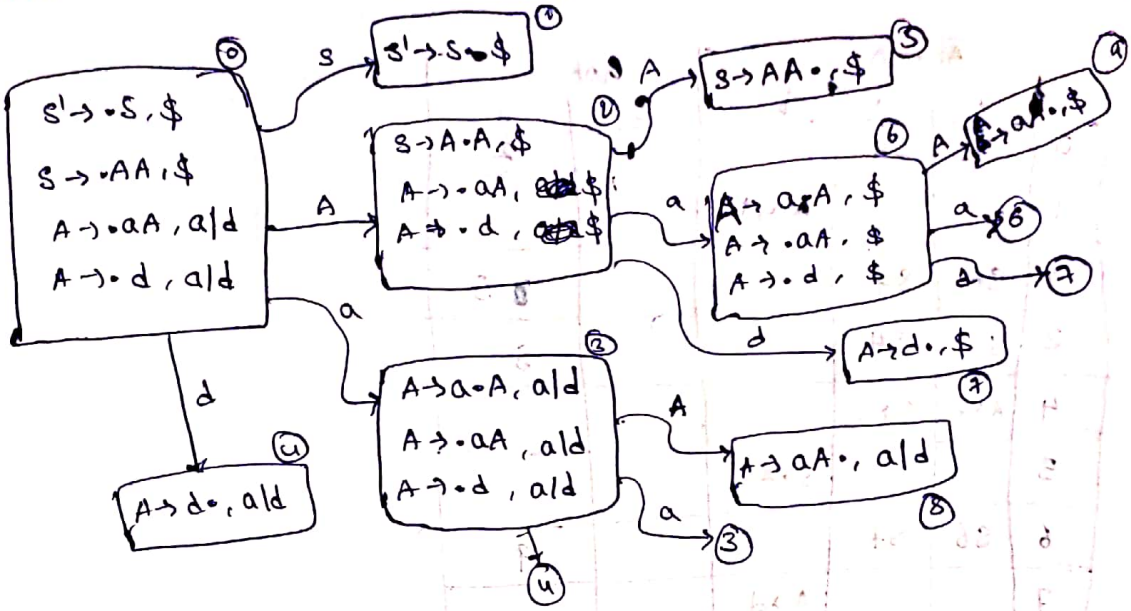
(2)

$S \to AA$

$A \to aA$

$A \to d$ First(A) = {a, d}

Scanned with CamScanner



Here we got 10 states for LR(1)

where as we got 7 states LR(0) for the same grammar.

Note:

→ LR(1) has more ↑ states than LR(0) for any given grammars.

CLR(1) Parse table (Canonical LR):

Construction of parse table is same as LR(0) except for reduction.

* If $A \to \alpha \cdot , a$ is in state I then

$$\text{Action}(I, a) = \text{Reduce } A \to \alpha$$

Rest of the construction is same as LR(0) parse table.

→ In CLR(1) & LALR(1), reduce entries should be filled only in lookahead terminals.

Scanned with CamScanner

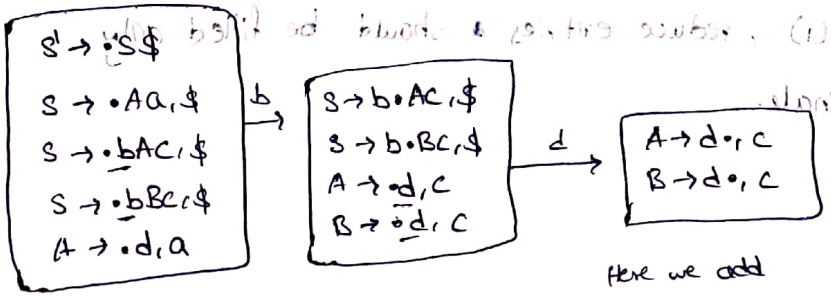
	Action			Goto	
	a	d	$	S	A
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	A→d	A→d			
5			S→AA		
6	S6	S7			9
7			A→d		
8	A→aA	A→aA			
9			A→aA		

As the CLR(1) parse table ~~denote~~ don't have multiply defined entries the grammar is CLR(1) & (~~LR~~ LALR(1))

Eg:

Check whether the below grammar is CLR(1) or not.

S→Aa
S→bAc
S→bBc
A→d
B→d



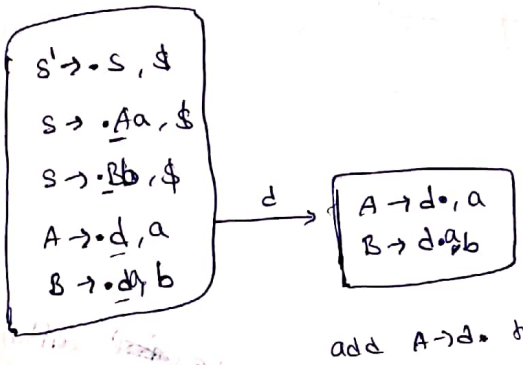
Here we add
A→d to Action[I, c]
B→d to Action[I, c]

∴ Reduce - Reduce Conflict.

Scanned with CamScanner

Ex: Check whether the below grammar is CLR(1) or not

$S \to Aa$
$S \to Bb$
$A \to d$
$B \to da$



add $A \to d\cdot$ to Action[I, a]
$B \to d\cdot a$ to Action [I, a]

∴ shift-reduce conflict

Ques Check whether the below grammar is CLR(1) or not.

$S \to aS \mid bS \mid c$

Sol:

The grammar is clearly LL(1)

First(aS) ∩ First(bS) = ϕ
First(bS) ∩ First(c) = ϕ
First(aS) ∩ First(c) = ϕ

Every LL(1) is CLR(1)

∴ above grammar is CLR(1)

Ques Check whether below grammar is CLR(1) or not

6

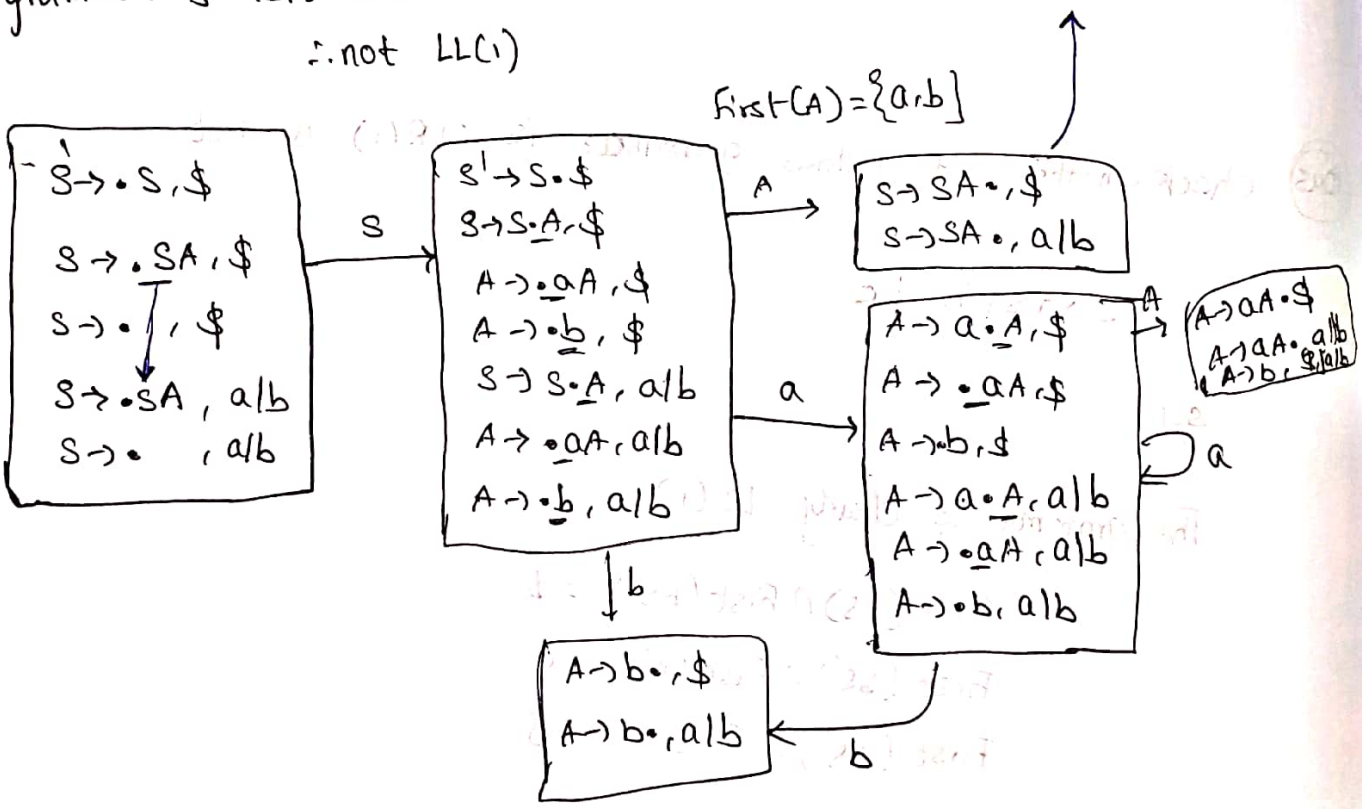
$S \to SA$
$S \to \epsilon$
$A \to aA$
$A \to b$

Scanned with CamScanner

Sol:

The grammar is left rec

∴ not LL(1)



no conflicts

∴ CLR(1)

Note:

CLR(1) has more states compared to LR(0). or This requires more space and computational time. This is a drawback for CLR(1).

Scanned with CamScanner

LALR(1) Parser (Look Ahead LR(1)):

→ The drawback of ↑ (CLR) having more states is overcome here.

→ In Eg (2) observe that states (3) & (6) differ only in look ahead symbols. So we merge them and make it one state.

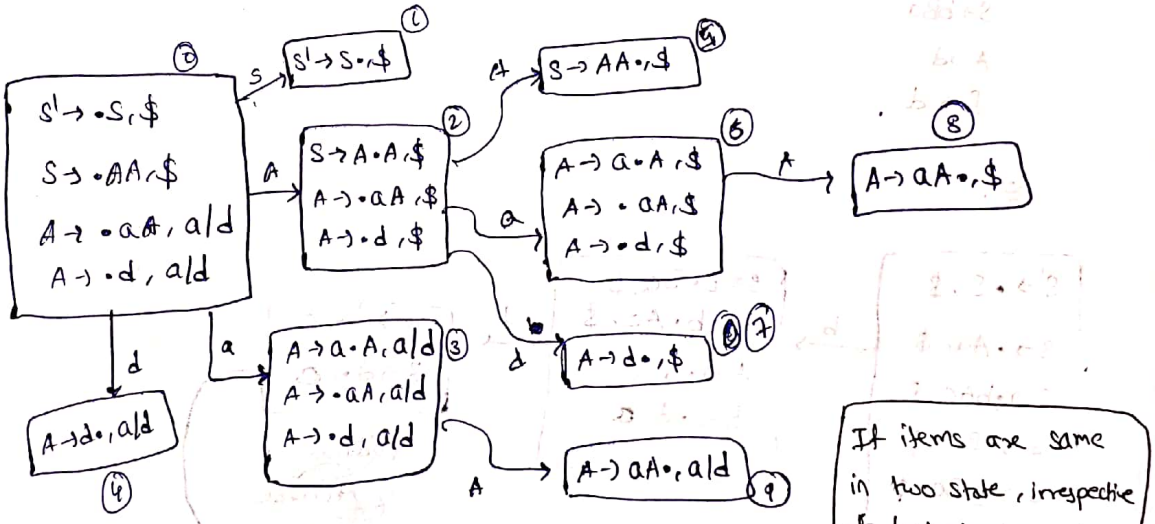
→ we can find LALR states by merging LR states that have the same set of first components and we can take the union of look ahead symbols. ~~do get LALR(1) states~~

Eg: Consider the same example as Eg (2)

S → AA

A → aA

A → d



Here (6) & (3) are merged

also (4) & (7) are merged

(8) & (9) are merged

(63) A → a•A, a/d/$
A → •aA, a/d/$
A → •d, a/d/$

(47) A → d•, $/a/d

(89) A → aA•, a/b/$

Scanned with CamScanner

Now for the above grammar

CLR(1) has 10 states

LALR(1) has 7 states

SLR(1) has 7 states

Note:

no of states in

LR(0) = [SLR(1) = LALR(1)] ≤ CLR(1)

Q47 Check whether the grammar is LALR(1) or not

S → Aa

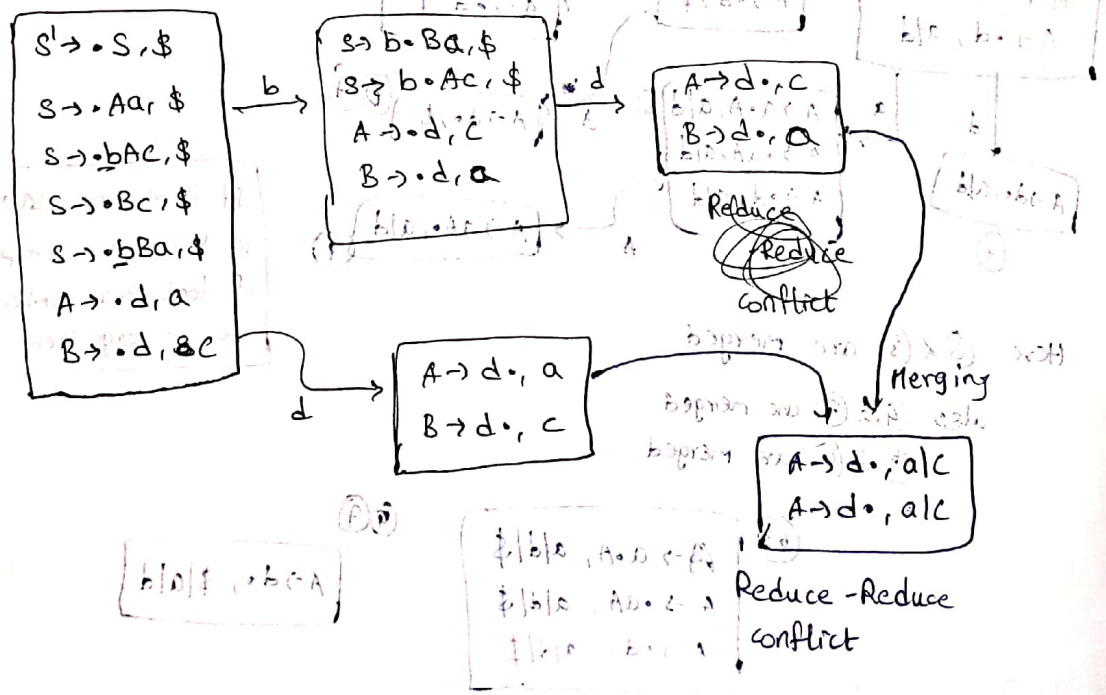
S → bAc

S → Bc

S → bBa

A → d

B → d



∴ Above grammar is not LALR(1)

But it is CLR(1)

Note:

* If there is no S-R conflict in CLR(1) state, it will never be reflected in the LALR(1) state obtained by combining CLR(1) states. However, this reduction process may introduce R-R conflicts

SR conflicts are same for CLR & LALR
The only benefit of CLR over LALR is in R-R conflicts

Note:

* CLR(1) is most powerful
* LALR(1) is most efficient (∵ it has less no of states)
* YACC is a parser generator tool which uses LALR(1) parsing algorithm

Q48 / q-03 Consider the grammar shown below

$S \to CC$
$C \to cC \mid d$

The grammar is

a) LL(1)
b) SLR(1) but not LL(1)
c) LALR(1) but not SLR(1)
d) LR(1) but not LALR(1)

Sol:

$S \to CC$ (one production ⇒ LL(1) production)

$S \to cC \mid d$

First(cC) ∩ First(d)
$\{c\} \cap \{d\} = \phi$
∴ LL(1)

Q49 / q-10 The grammar $S \to aSa \mid bS \mid c$ is

a) LL(1) but not LR(1)
b) LR(1) but not LL(1)
c) Both LL(1) and LR(1)
d) Neither LL(1) nor LR(1)

Scanned with CamScanner

Sol :

$S \to aSa \mid bS \mid c$

all first symbols are different

∴ LL(1)

Every LL(1) is LR(1)

∴ opt (c)

Q.50 / G-15 Consider below grammar G

$S \to F \mid H$

$F \to P \mid c$

$H \to d \mid c$

which is/are correct?

S1 : LL(1) ~~pass~~ can parse G

S2 : LR(1) can parse G

a) S1 b) S2 c) S1 & S2 d) none

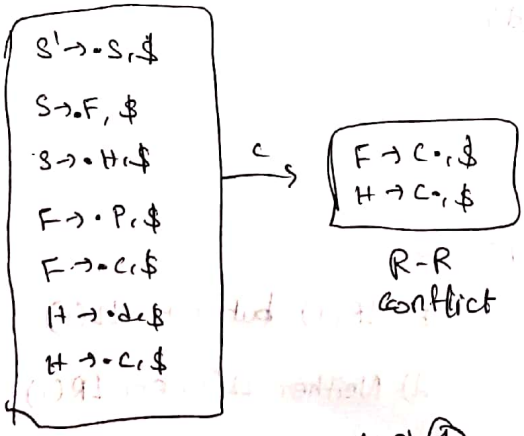
Sol :

$S \to F \mid H$

$First(F) \cap First(H)$

$= \{P, c\} \cap \{d, c\} = \{c\} \neq \phi$

∴ not LL(1)



∴ opt (d)

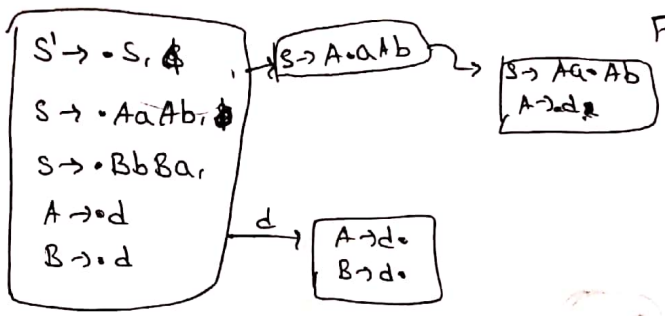
Scanned with CamScanner

Material Problems

P/57

Follow(A) = {a, b}

Follow(B) = {a, b}



Follow(A) ∩ Follow(B)

= {a, b}

∴ R-R conflict

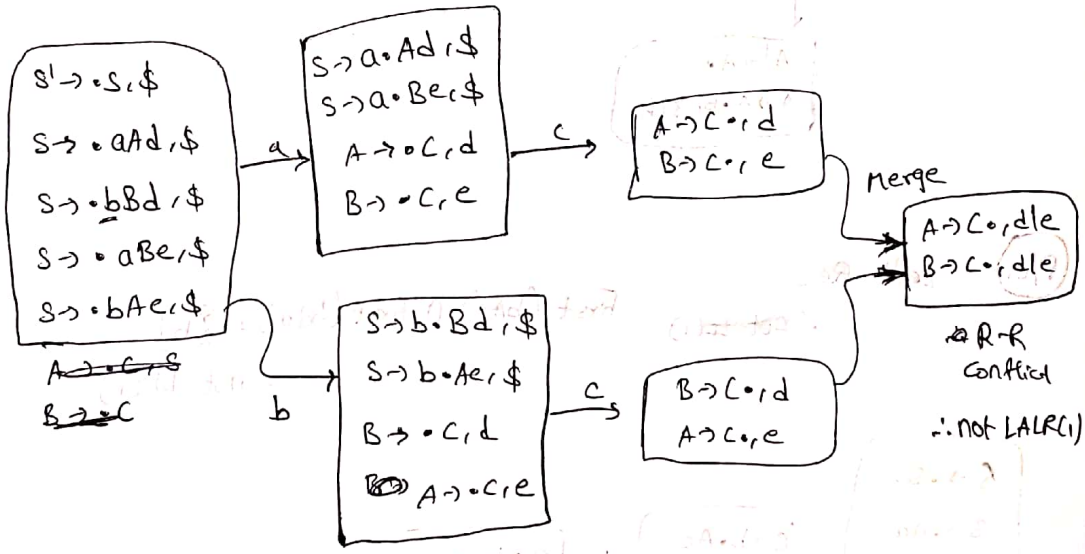
The question is asked no of conflicting entries

A → d •

B → d •

i.e., 2 entries.

P/58



R-R Conflict

∴ not LALR(1)

But it is clearly CLR(1)

∴ LR(1) but not LALR(1)

Scanned with CamScanner

P/59

$S' \to .S, \$$

$S \to .Aa, \$$

$S \to .bAc, \$$

$S \to .Bc, \$$

$S \to .bBa, \$$

$A \to .d, a$

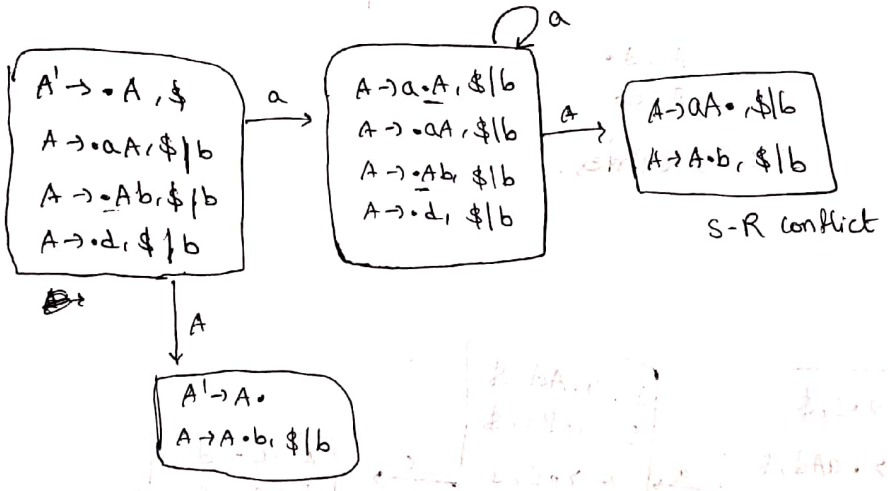
$B \to .d, c$

This is same as Q47

∴ LR(1) but not LALR(1)

28/06/20

P/60



S-R conflict

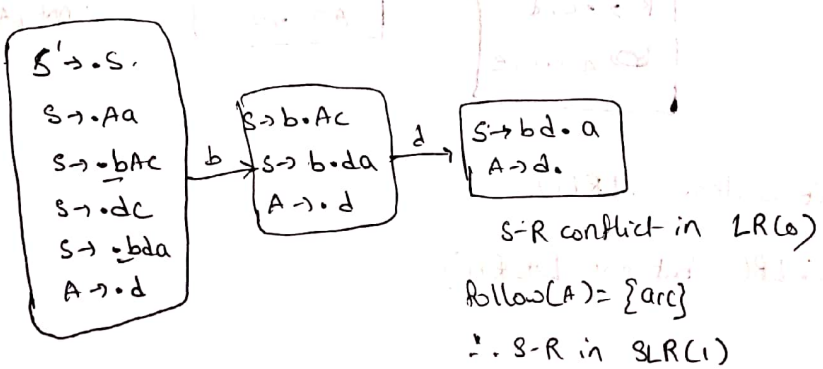
P/61

~~Left Rec~~

∴ ~~not LL(1)~~

First (bAc) ∩ first(bda) = {b}

∴ not LL(1)



S-R conflict in LR(0)

Follow(A) = {a, c}

∴ S-R in SLR(1)

∴ LALR(1)

Scanned with CamScanner

P/62

$S \to \cdot A, \$$

$A \to \cdot AB, \$$

$A \to \cdot\ , \$$

$A \to \cdot AB, b$

$A \to \cdot\ , b$

First(B) = {b}

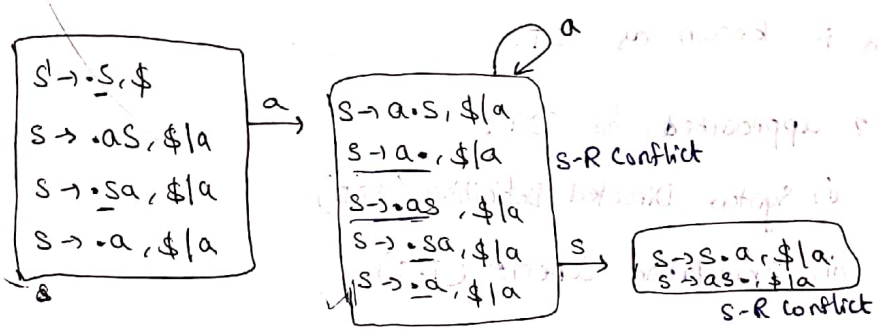
$\therefore S \to \cdot A, \$$

$A \to \cdot AB, \$/b$

$A \to \cdot\ , \$/b$

∴ opt (c)

P/63



∴ 2-conflicting states

P/64

$S' \to \cdot S, \$$

$S \to \cdot L = R, \$$

$S \to \cdot R, \$$

$L \to \cdot *R, =/\$$

$R \to \cdot L, \$$

$L \to \cdot id, =/\$$

∴ opt (c)

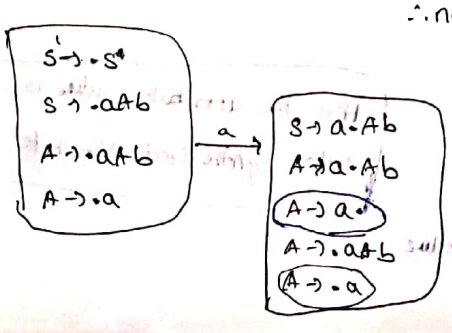
P/65

$S \to aAb$

$A \to aAb/a$

First(aAb) ∩ First(a) = {a}

∴ not LL(1)



SR-Conflict in LR(0) parse

follow (A) = {b}

∴ we dont have SR conflict in SLR(1)

∴ LR(1)

Scanned with CamScanner

(P/66) $S \to (S) \mid \epsilon$

$First((S)) \cap First(\epsilon) = \phi$

$First((S)) \cap Follow(S)$

$= \{(\} \cap \{)\} = \phi$

$\therefore$ LL(1) & LR(1)

Syntax Directed Translation:

It associates semantic rules with productions of a grammar.

This process is known as SDT.

There are 2 approaches for SDT:

(i) Syntax Directed Definition (SDD)

(ii) Translation Scheme (T.S)

Syntax Directed Definition (SDD)

It is an augumented CFG in which attributes are added to grammar symbols and with each production a set of semantic rules are added. This grammar and set of semantic rules together constitutes Syntax Directed Definition.

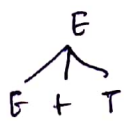
Attributed are of two types:

Synthesized attributes or Inherited attributes.

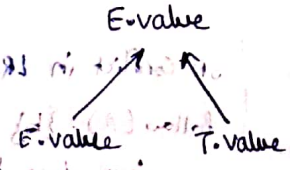
Synthesized attribute:

Production	Semantic rule
$E \to E + T$	$\{E.value = E.value + T.value\}$

Parse tree



Dependency graph



Here the attribute value is called synthesized attribute

Scanned with CamScanner

→ An attribute is said to be synthesized if its value, at a parse node, is defined from attribute value of child nodes.

→ Synthesized attributes are evaluated during a bottom-up traversal of the parse.

Inherited attributes:

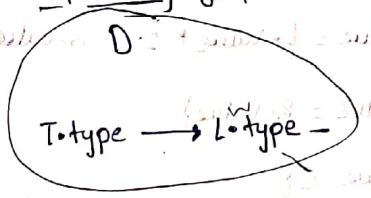
An inherited attribute is on whose value is defined in terms of attributes of parent and/or siblings of that node.

Eg:

Production	Semantic Rule
D → T L ; (declaration of variables)	{L.type = T.type}

Parse tree

Dependency graph



Here the attribute 'type' is called inherited attribute

~~→ An SDD that uses synthesized att~~

A synthesized attribute at node N can be defined in terms of inherited attribute values of at node N itself.

If A is a grammar symbol and let A.a is synthesized attribute and let A.b, A.c be inherited attributes

{A.a = A.b + A.c} is valid

Eg: Consider the following SDD

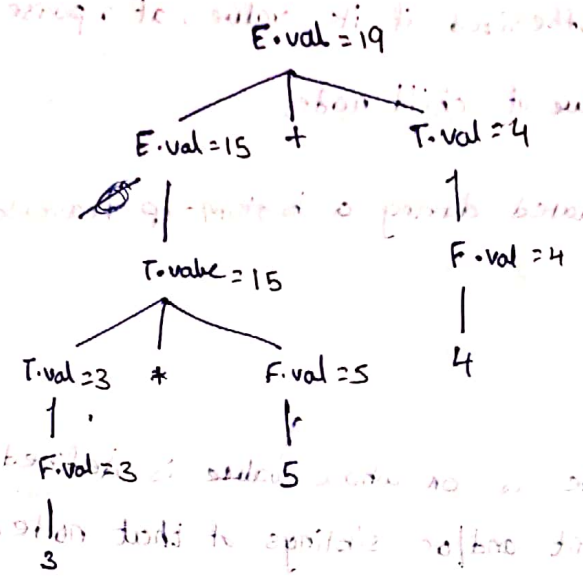
Production	Semantic Rules
E → E + T	{E.val = E.val + T.val}
E → T	{E.val = T.val}
T → T * F	{T.val = T.val * F.value}
T → F	{T.val = F.val}
F → num	{F.val = num}

Terminals have only synthesized attributes. And values of these attributes are defined as lexical values at the time of lexical analysis

Compute E.value for the root of the parse tree for the input string 3*5+4

There are no any semantic rules in SDD for determining attribute values of terminals

Scanned with CamScanner

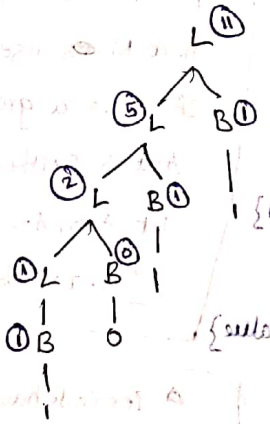


$\therefore$ E.value = 19

Q51) Consider the following SDD

Production	Semantic rule
$L \rightarrow LB$	$\{L.value = L.value * 2 + B.value\}$
$L \rightarrow B$	$\{L.value = B.value\}$
$B \rightarrow 0$	$\{B.value = 0\}$
$B \rightarrow 1$	$\{B.value = 1\}$

Compute L.value for the root of the parse tree for the input string 1011



$\therefore$ L.value = 11

Q52 / G-04

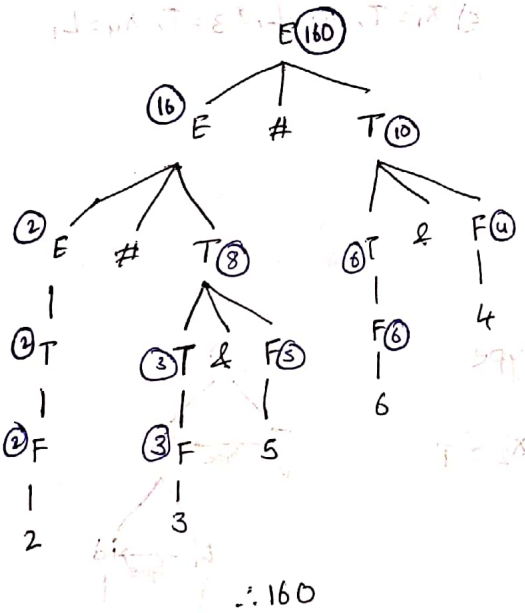
Consider the grammar

$E \to E_1 \# T$	$\{E.val = E_1.val * T.val\}$
$E \to T$	$\{E.val = T.val\}$
$T \to T_1 \& F$	$\{T.val = T.val + F.val\}$
$T \to F$	$\{T.val = F.val\}$
$F \to num$	$\{F.val = num.val\}$

Compute E.val for the root of the parse tree for the expression

2#3&5#6&4

a) 200 b) 180 c) 160 d) 40



∴ 160

Q53 / G-19

Consider the following grammar and the semantic actions to support the inherited declaration attributes. Let $x_1, x_2, x_3, x_4, x_5, x_6$ be the place holders for the non-terminals D, T, L or L_1 in the following table:

Which of the following are appropriate choices for x_1, x_2, x_3 and x_4?

Scanned with CamScanner

Production rule	Semantic action
$D \to TL$	$X_1.type = X_2.type$
$T \to int$	$T.type = int$
$T \to float$	$T.type = float$
$L \to L_1, id$	$X_3.type = X_4.type$ $addType(identry, X_5.type)$
$L \to id$	$addType(identry, X_6.type)$

* a) $X_1 = L \quad X_2 = T \quad X_3 = L_1, \quad X_4 = L$ b) $X_1 = L, \quad X_2 = L, \quad X_3 = L_1, \quad X_4 = T$

c) $X_1 = T, \quad X_2 = L, \quad X_3 = L_1, \quad X_4 = T$ d) $X_1 = T, \quad X_2 = L, \quad X_3 = T, \quad X_4 = L_1$

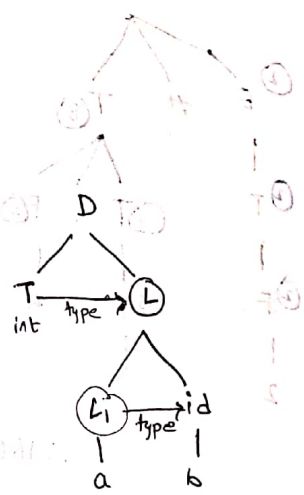
Sol :

$D \to TL$

T → type, L → list of variable

$\therefore L.type = T.type$

$\therefore X_1 = L \quad X_2 = T$



$L \to L_1, id$

This list of identifiers

$\therefore L_1.type = L.type$

$\therefore X_3 = L_1 \quad X_4 = L$

$\therefore$ opt (a)

Note :

→ An SDD that uses synthesized attributes exclusively is said to be S-attributed definition

Ex: SDD in (Q51)

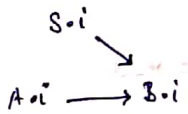
Scanned with CamScanner

→ ~~An SDD that uses either synthesized and/or inherited attributes is said to be L-attributed definition.~~
↳ evaluation is from left to right

→ Every S-attributed definition is also L-attributed, but need not to be vice-versa.

→ L in L-attributed stands for left to right evaluation

Eg: $S \to AB \quad \{B.i = f(A.i, S.i)\}$



(left to right)

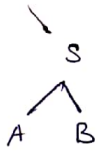
'L' attributed

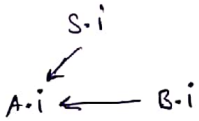
but not 'S' attributed

L-attributed

An SDT in which attribute is restricted to inherit either from parent or left sibling is called L-attributed Definition

- Eg: $S \to AB \quad \{A.i = f(S.i, B.i)\}$



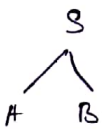


(no left to right evaluation)

∴ not L-attributed

and not 'S' attributed

Eg: $S \to AB \quad \{S.i = f(A.i, B.i)\}$



It means 'i' is synthesized

∴ 'S' attributed definition

∴ 'L' attributed definition.

Scanned with CamScanner

Q54 (G-20) Consider the productions $A \to PQ$ and $A \to XY$. Each of the five non-terminals A, P, Q, X and Y has two attributes: s is a synthesized attribute, and i is an inherited attribute.

Consider the following rules:

Rule 1: $P.i = A.i + 2$, $Q.i = P.i + A.i$ and $A.s = P.s + Q.s$

Rule 2: $X.i = A.i + Y.s$ and $Y.i = X.s + A.i$

which of the following is true?

~~a) none b) 1 & 2 are L-attributed~~

a) Neither ① nor ② are L-attributed

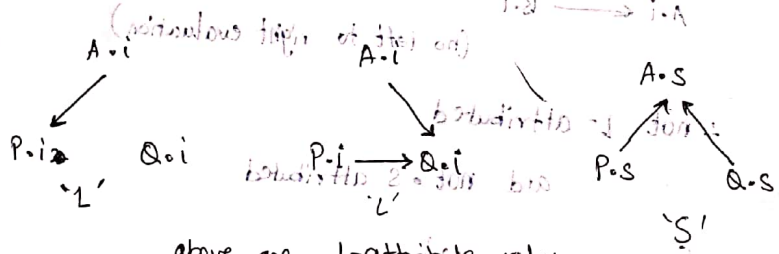
b) Both ① & ② are L-attributed

c) ① is L-attributed

d) ② is L-attributed

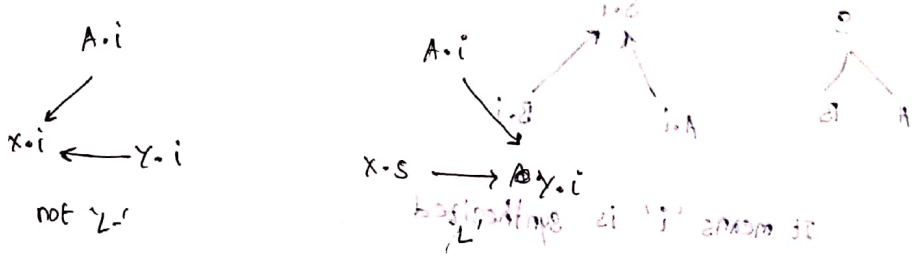
Sol:

Rule 1:



above are L-attribute rules

Rule 2:



∴ above rule is not L-attribute

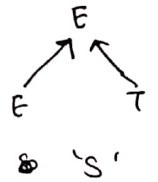
∴ opt ©

Scanned with CamScanner

Material Problems on SDT:

P/2

~~E → E + T~~



E → T { print '5' }

E	T
↑	↑
T	id
'5'	5

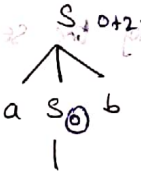
∴ S-attributed

∴ L-attributed

∴ opt (c)

P/3

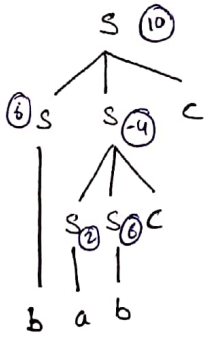
L = {$a^n b^n \mid n \geq 1$}



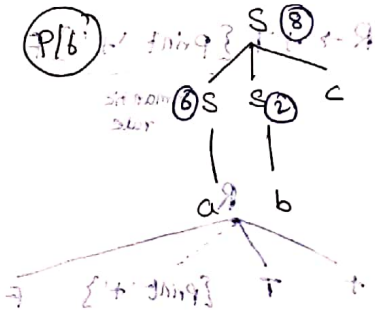
∴ length of string

i.e., no of a's & b's

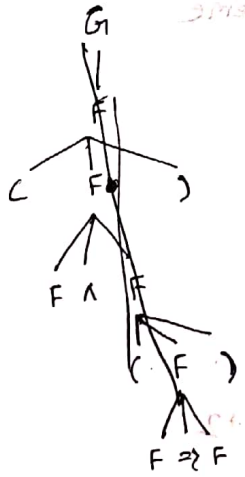
P/5

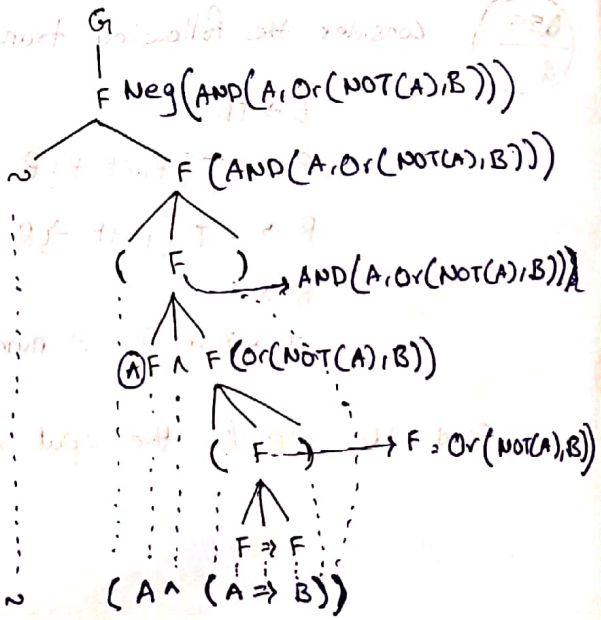


P/6



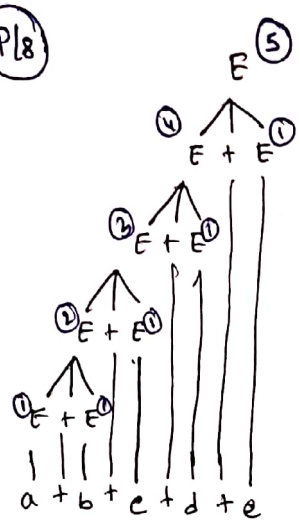
P/7





Scanned with CamScanner

P18

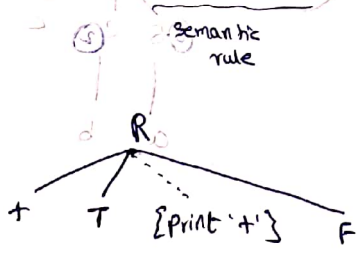


Translation Scheme:

It is like SDD except that the order of evaluation of the semantic rules is explicitly shown by entering the semantics within the R.H.S of a production.

→ Semantics are evaluated using "depth first traversal" of a parse tree.

5 : $R \rightarrow +T \{print '+'\} F$



Q55 / Q-

Consider the following translation scheme

$E \rightarrow TR$

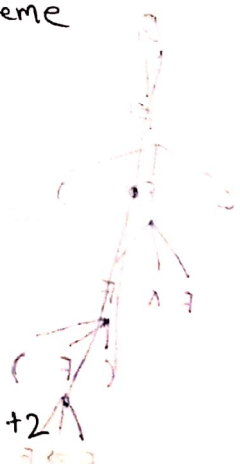
$R \rightarrow +T \{print +\} R$

$R \rightarrow -T \{print -\} R$

$R \rightarrow \epsilon$

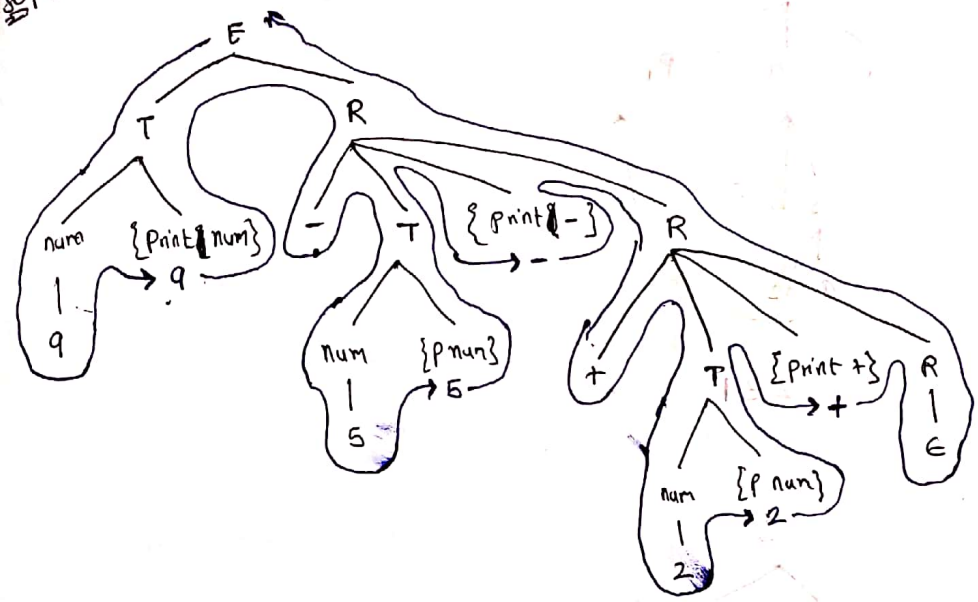
$T \rightarrow num \{print\ num\}$

Find the o/p for the input string : 9-5+2



Scanned with CamScanner

Sol:-

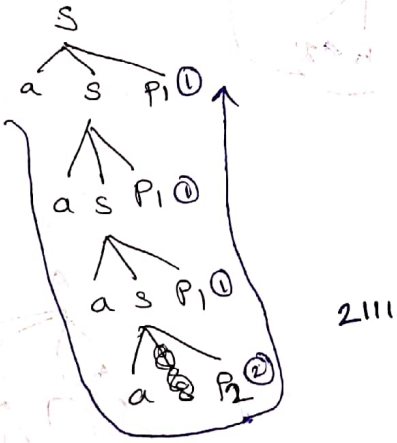


we apply DFS on above tree to get o/p.

∴ 95-2+

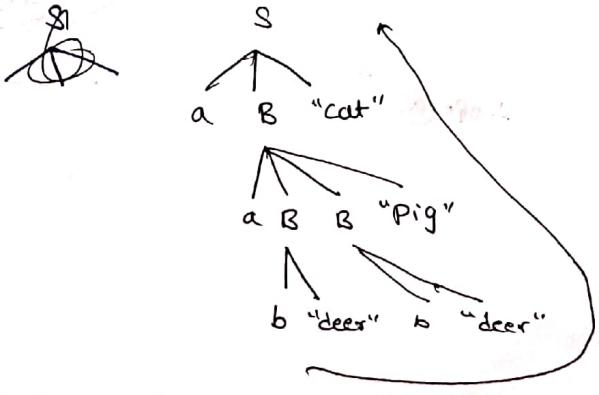
Material Problems

P/9



2111

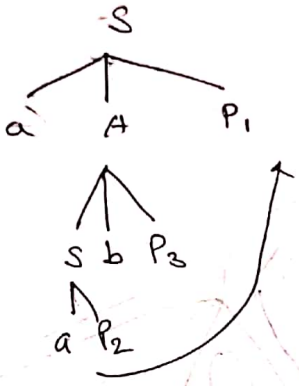
P/10



deer, deer, pig, cat

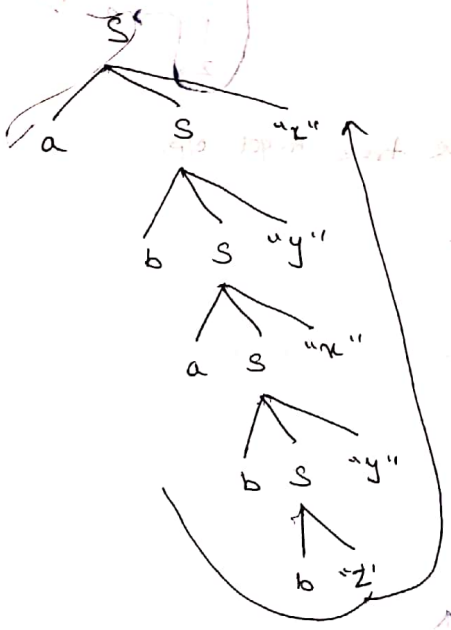
Scanned with CamScanner

P/11



∴ "231"

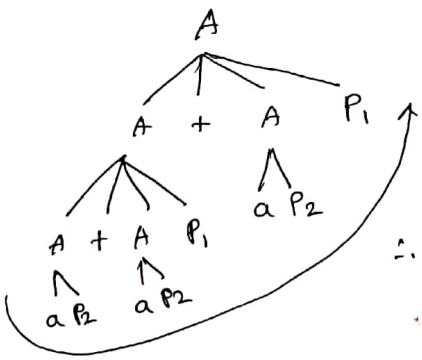
P/12



zyxyx

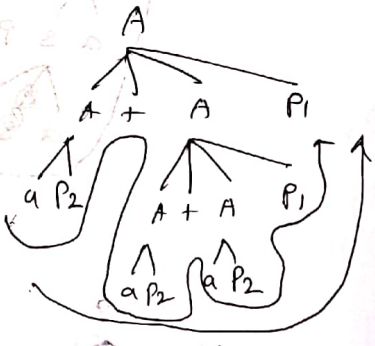
∴ opt (c)

P/13



∴ 22121

∴ opt (a)



∴ 22211

Scanned with CamScanner

(P/14) a+b * c

SR parser scans from left to right

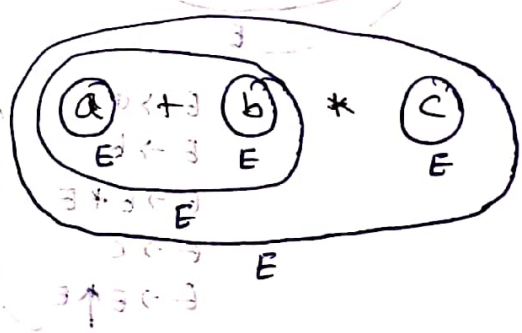
∴ The 1st handle found is a

later the handle is b

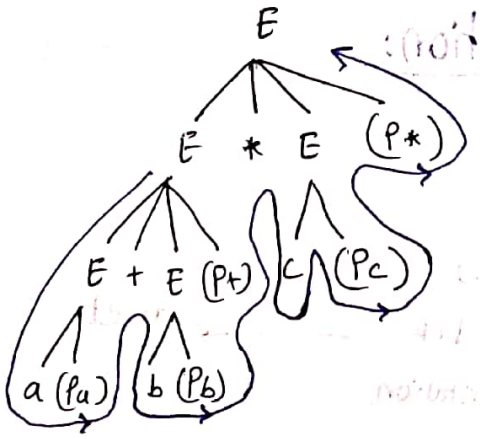
next it is E+E

then c

then E*E



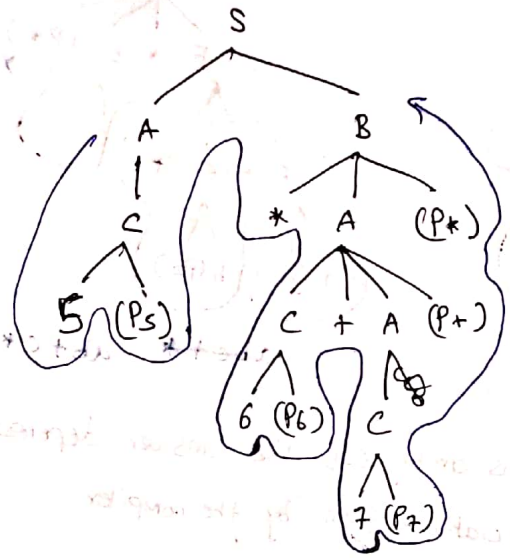
∴ The parse tree is



ab+c*

∴ opt (b)

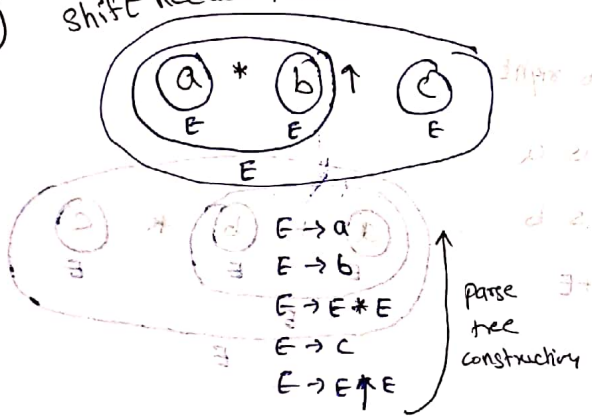
P/15

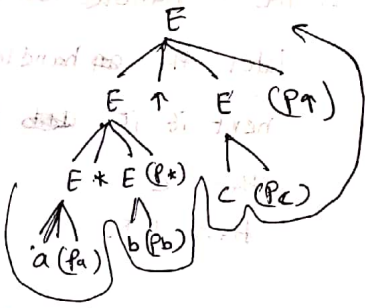


5 6 7 + *

P/16

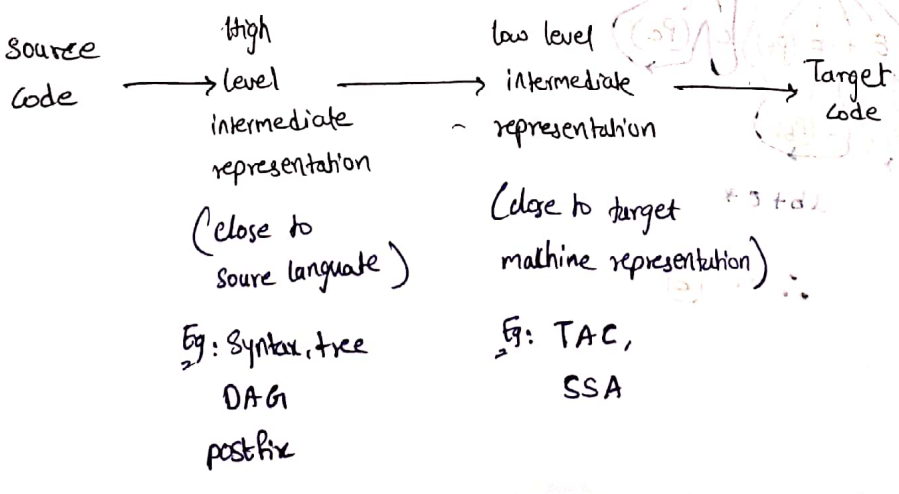
Shift Reduce Parser





a b * c ↑

Intermediate Code Generation:



Scanned with CamScanner

Three Address Code (TAC):

It is a stmt with atmost three memory references.

Eg: a = b + c (3 ref)
a = b (2 ref)
goto a (1 ref)
return; (0 ref)
If a > b then goto c (3 ref)
a[i] = b (3 ref)
a = b[i] (3 ref)
a = &x; (2 ref)
a = *p. (2 ref)

Note:
a = b+c+d is also considered TAC stmt
But while implementation it is broke down.

Eg: a = b * c / d + e * f - g

Represent above expression in TAC form

Sol:
we need check precedence & associativity

$T_1 = b * c$
$T_2 = T_1 / d$
~~$T_3 = T_2 + e$~~
$T_3 = e * f$
$T_4 = T_2 + T_2$
a = $T_4 - g$

In above example each assignment is done to distinct variables.
Such kind of TAC is called Static Single Assignment (SSA)

For the same the TAC can be

Same ∴ not SSA but TAC

$T_1 = b * c$
$T_2 = T_1 / d$
$T_3 = e * f$
$T_3 = T_1 + T_3$
$a = T_3 - g$

Scanned with CamScanner

Static Single Assigment:

- A TAC in which all assignment are done to distinct variables.

Q56 / G-14 One purpose of using intermediate code in compilers is to

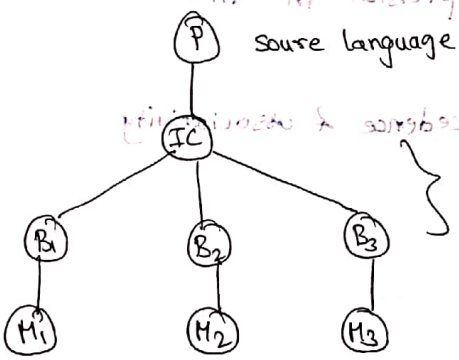
a) make parsing and semantic analysis simpler.

b) improve error recovery and error reporting.

c) increase the chances of reusing the machine independent code optimizer in other compilers

d) improve the register allocation

Sol:



Thus we can use the same M/c independent code optimizer for M_1, M_2, M_3 as we have same IC.

∴ opt (c)

Q57 / G-16 Consider the following code segment

```
x = a - t;
y = x * v;
x = y + w;
y = t - 2;
y = x * y;
```

The minimum no of total variables required to convert the above code segment into static single assignment form is _____

Scanned with CamScanner

$T_1 = u - t$

$T_2 = T_1 * y;$

$x = T_2 * w;$ (for the last occurance of x we use x)

$T_3 = t - z;$

$y = x * T_3;$ (for the last occurance of y we use y)

So to convert the given segment into SSA we need

3 more variables

(The question is asked variable required for conversion process)

$\therefore$ Ans : 3

Q58 / 9-19

Consider the following intermediate program in TAC.

$p = a - b$

$q = p * c$

$p = u * v$

$q = p + q$

which one of the following corresponds to SSA form of the above code.

a) $p_1 = a - b$

$q_1 = p_1 * c$

$p_1 = u * v$

$q_1 = p_1 + q_1$

b) $p_3 = a - b$

$q_4 = p_3 * c$

$p_4 = u * v$

$q_5 = p_4 + q_4$

c) $p_1 = a - b$

$q_1 = p_2 * c$

$p_3 = u * v$

$q_2 = p_4 + q_3$

d) $p_1 = a - b$

$q_1 = p * c$

$q_2 = u * v$

$q_2 = p + q$

Scanned with CamScanner

Sol :

a) It doesn't have assigments to distinct variables

∴ not SSA

Silly (d) is not SSA

(b) & (c) have distinct assignments

but (b) preserves meaning equivalent to the given code segment

∴ opt (b)

Implementation of TAC

Eg : x = a+b*c+d

$T_1 = b*c$

$T_2 = a+T_1$

$x = T_2 + d$

TAC can be implemented in 3 ways :

① Quadraple : It is a record structure with 4 fields called operator, argument 1, argument 2, result.

For above example quadraples are represented as

Op	arg1	arg2	result
*	b	c	T_1
+	a	T_1	T_2
+	T_2	d	x

② Triple : It is a record structure with 3 fields. called operator, argument 1, argument 2

For previous example the triple is as shown below

Scanned with CamScanner

	op	arg1	arg2
(1)	*	b	c
(2)	+	a	(1)
(3)	+	(2)	d
(4)	assign	x	(3)

positions

In triple assignment is represented as show below

a=b

assign	a	b

③ Indirect Triple:

Here the positions in triple are replaced with pointer.

	op	arg1	arg2
	*	b	c
	+	a	(100) → address of triple 1
	+	(200)	d → address of triple 2
	assign	x	(300) → address of triple 3

Eg: Convert the following C code into TAC

```
while (a<b)
{
    if (c>d)
        x=y+z;
    else
        x=y-z;
}
```

Sol:

1) If a<b goto (3)
2) goto (8)
3) if c>d goto (6)
4) x=y-z
5) goto (1)
6) x=y+z
7) goto (1)
8)

Scanned with CamScanner

Eg: Write TAC for below C code

```
for(i=0; i<10; i++)
    z = y+z;
```

TAC:

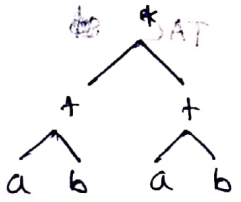
1. i=0
2. if i<10 goto 4
3. goto 7
4. x=y+z
5. i=i+1
6. goto 2
7. ---- (remaining code)

These goto stmts addresses are filled by a process called backpatching

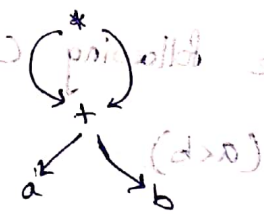
Directed Acyclic Graph (DAG)

DAG is a varient of syntax tree.

Eg: (a+b) * (a+b)



Syntax tree



DAG

In syntax tree "a+b" is evaluated twice.

But Dag evaluates 'a+b' only once and uses twice

TAC

$T_1 = a+b$

$T_2 = a+b$

$T_3 = T_1 * T_2$

Syntax tree

$\Leftrightarrow$

$T_1 = a+b$

$T_3 = T_1 * T_1$

DAG

Scanned with CamScanner

Note:

DAG is used to find common sub expression

↓

expressions that return same value.

Eg: Consider below code segment

```
a = b - c
b = a + c
e = b - c
f = a - c
```

Here both b-c given diff values since 'b' is modified b/w the two expressions.

both a-c have same values in both cases

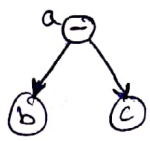
∴ a-c is common sub expression.

Construction of DAG for above stmt:

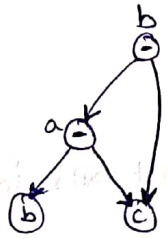
- For given code from top to bottom create a node for every new memory reference.

  If reference already exists used existing one (latest one)

a = b - c

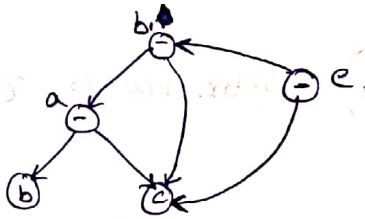


b = a - c

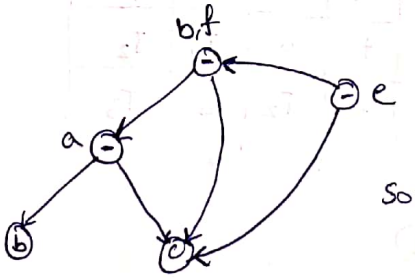


e = b - c

Here we consider latest modified b.



f = a - c



So Here

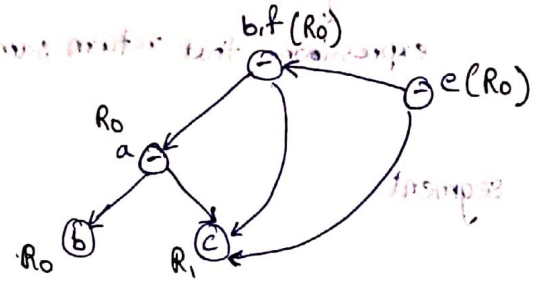
b = a - c

f = a - c

are identified as common sub expressions.

Scanned with CamScanner

Finding minimum number of registers required from a DAG (Not needed for gate)



Finding min no of reg comes under code optimization and now it is not there in syllabus.

Initial we store b & c in (bottom ones) in R0 & R1

For 'a' we store in R0

After that b,f in R0

and later e in R0

Material Problems on IC

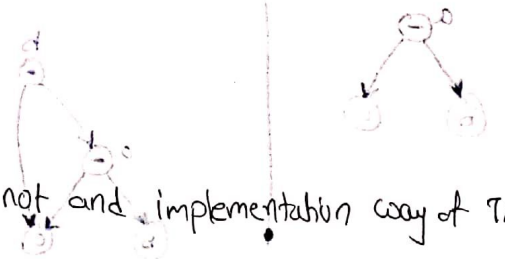
(P/1) same as (Q56)

(P/3) return 0 (0 ref)

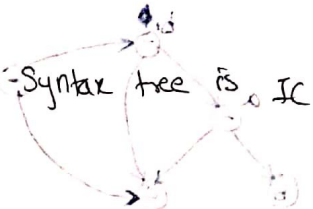
if a>b goto c (3 ref)

a[i] = b (3 ref)

∴ none



(P/4) Syntax tree is IC but not and implementation way of TAC



(P/5) a+b*c-d

$T_1 = b * c$

$T_2 = a + T_1$

$T_3 = T_2 - d$

op	arg 1	arg 2	Res
*	b	c	T_1
+	a	T_1	T_2
-	T_2	d	T_3

∴ opt (b)

Scanned with CamScanner

P/6 a or (b and c) or d

(1)	and	b	c	T_1
(2)	or	a	T_1	T_2
(3)	or	T_2	d	T_3

∴ opt (c)

P/7 a* - (b+c)
↳ unary minus
Paralysis is evaluated first

(1)	+	b	c
(2)	Neg	(1)	
(3)	*	a	(2)

∴ opt (a)

P/8 Same as Q57

P/9 Same as Q58

P/10 a) true

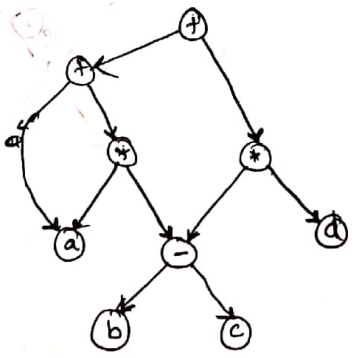
b) dead code is code which has no use

Eg: a = a
This can be eliminated (not needed for gate optimization technique)

c) helps with using min of registers

P/11 ∴ none

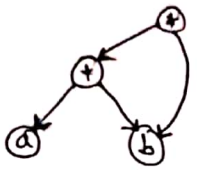
P/12



(b-c) is evaluated first

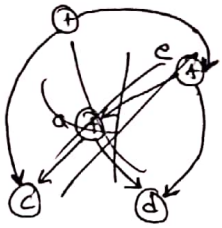
Scanned with CamScanner

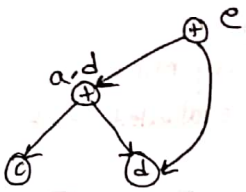
P/11 $a+b*b$



∴ opt (b)

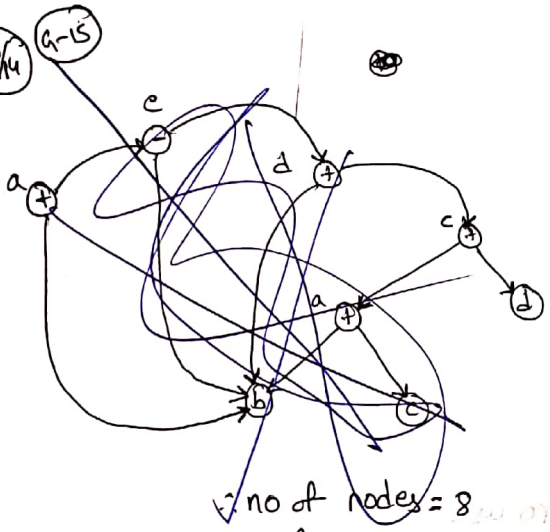
P/13





∴ 4 nodes

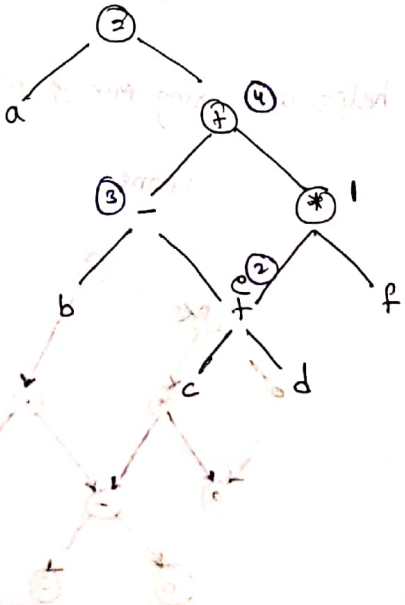
P/14 G-15



∴ no of nodes = 8
no of edges = 10

P/16

	op	arg1	arg2
(1)	*	e	f
(2)	+	c	d
(3)	-	b	(2)
(4)	+	(3)	(1)
(5)	=	a	(4)



Scanned with CamScanner

P/15 Q-15

Since it is asked to find minimum number of nodes and edges we need to optimize the code

(Now we don't have optimization in the syllabus)

```
a = b+c
c = a+d
d = b+c
e = d-c
a = d
```

(From substitution
a = e+b
= d-b+b
= d)

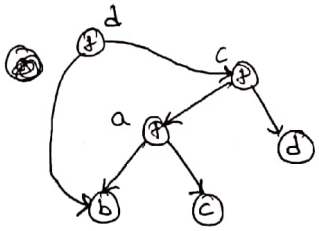
```
a = b+c
c = a+d
d = b+c
e = b+c-c = b   (substitution)
a = d
```

⇒

```
a = b+c
c = a+d
d = b+c
e = c
a = d
```

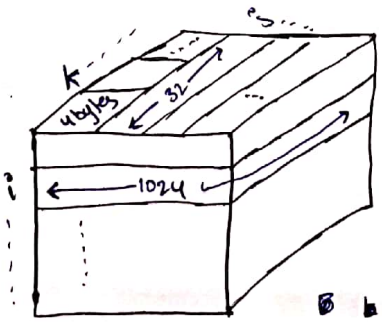
} Dead code because they have no further references

∴
```
a = b+c
c = a+d
d = b+c
```
is optimized code



∴ 6 nodes & 6 edges

P/15 Q-07



j = no of vertical slices

$= \frac{1024}{32} = 32$

$k = \frac{32}{4} = 8$

value can't be determined

Each k is 4 bytes ⇒ int array

int a[ ][32][8] (from this opt ⑨)

Scanned with CamScanner