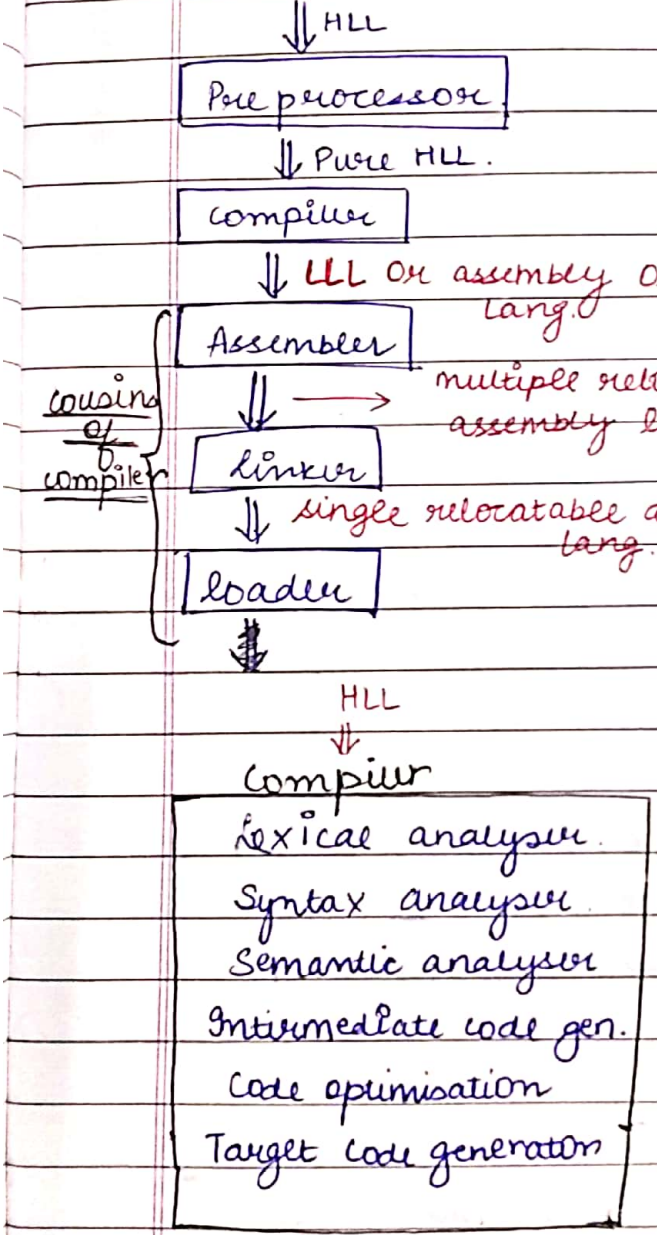
Date: / /
Page No. 21

12/11/2019

CH I INTRODUCTION TO COMPILER



* Preprocessor reads #

```
# define square(x) x*x
main()
{ int x=5, y;
  y = 125/ square(x);
  pf(y);
}
```

Here during compilation, preprocessor reads # statement & finds square(x) in program & replaces it by x*x.

125/x*x = 125/5*5 = 125

Once preprocessor replaces text it then removes # statement.

* m/c lang means instruction set of that particular processor
* High level lang consists of many shortcuts which are replaced by preprocessor & hence converted to pure HLL.
* compiler o/p is assembly lang because it is i/p to assembler
* The prog. after compilation is still present in hard disk.
* To execute, we need to bring prog. from hard disk to main mem. First prog. is converted to relocatable prog. by assembler & then it is sent to main memory.

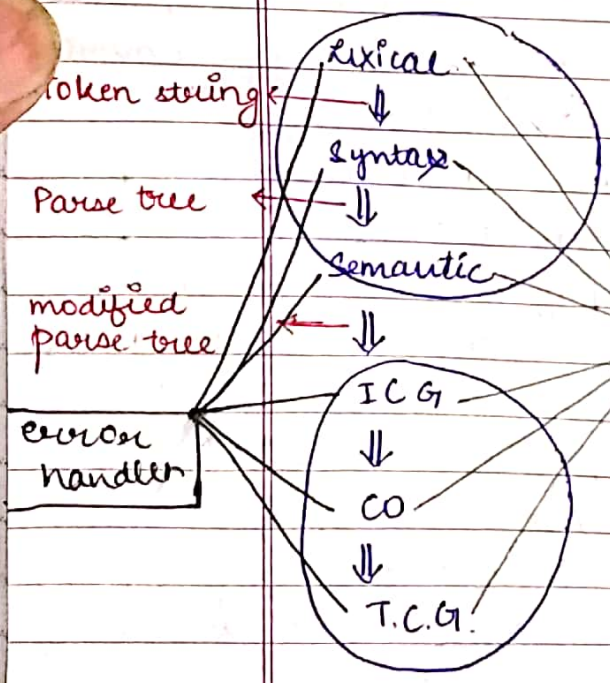
Date: / /
Page No. 22

* linker o/p :- multiple relocatable assembly language. They are linked since they are related to each other. If files are not linked & only 1 is sent to main memory then it will cause trouble in execution

* Loader loads the single relocatable A.L file to main mem. Here prog. waits in ready queue to get its chance to run. It is in ready state. This file is called .exe file.

* Compiler is a software/program which may contain a large amt. of code. Here code is divided into various modules.

* Code Optimisation is where unnecessary code is removed. This phase is not present in some compilers because they optimise code in ICG phase itself.



Verifies whether compiler can actually convert the given prog. to m/c or not. checks if code is valid

Store anything in Symbol table

Conversion begins here.

Date: / /
Page No. 23

- Lexical analyser basically introduces / identifies everyone in the program. It o/p a to token string.

int c = a * b

keyword / operator / operator → identifier

identifier identifier

It divides program to tokens.

- Once tokens are created, syntax analyser creates a parse tree for each token. eg. a token is created for "for". There is a grammar associated to each token. which gives rules to that token. It creates a tree to check if it matches the grammar defined. If no, then syntax error.

eg Once turbo C is installed, we get rules / grammar for entire C lang. wherein grammar is defined for each possible token.

- Semantic analysis makes modified parse tree & it performs meaningful checking of tokens.

eg. undeclared variable, type mismatch, multiple declaration

- If any problem occurs in any phase, it is given to error handler. If error handler has a solution to it, it will return the sol^n to that phase.

eg. type mismatch error sent to error handler. It returns a sol^n :- "type conversion" to semantic analyser.

- User defined variables are not present in compiler. They are added to Symbol Table.

Whenever declaration occurs, semantic analyser adds these entries to symbol table.

float x, a, b;

Date: / /
Page No. 24

eg. $x = a + b * 60$

L.A.

x - identifier - (id, 1)
= assignment op.
a - identifier - (id, 2)
+ addition op.
b - identifier - (id, 3)
* - multiplication op.
60 integer const.

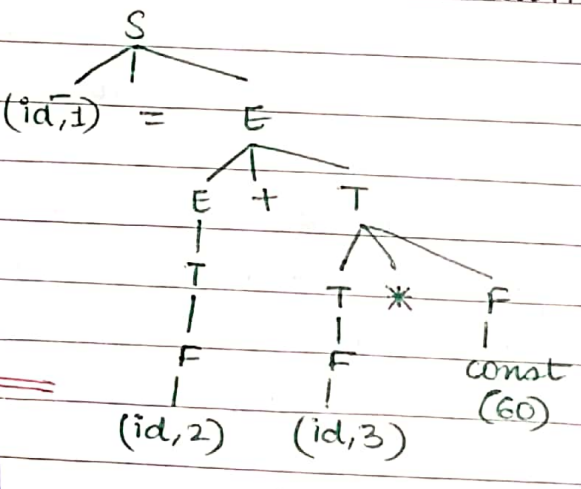
S.No	V.name	V.Type
1	x	float
2	a	float
3	b	float

Symbol Table.

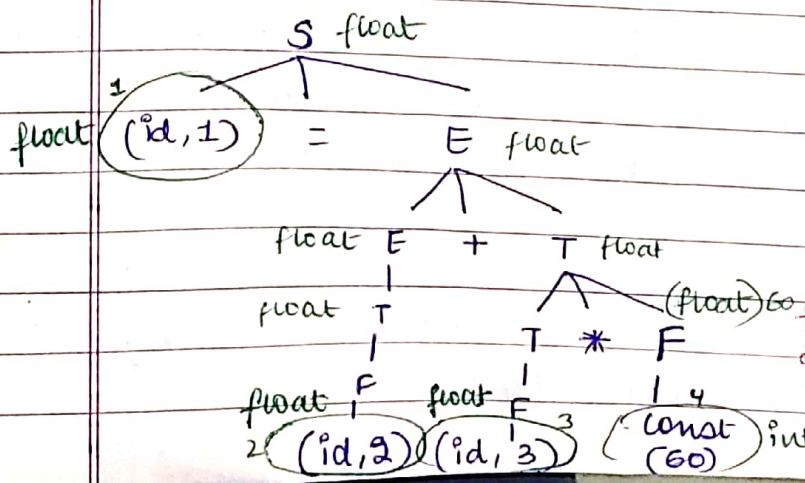
string ⟸ (id, 1) = (id, 2) + (id, 3) * 60

Syntax analyser → complex phase because of tree construction

grammar:
S → id = E
E → E + T | T
T → T * F | F
F → id | const



semantic analyser



Steps
① Symbol table created by semantic analyser for declarations
② tokens created by lexical analyser
③ Once tree is constructed by syntax analyser, semantic analyser picks up a token & checks it in symbol table to identify its type. It makes use of token no. to directly find token in symbol table.

→

If in lexical analysis phase, correct token nos. are assigned to all identifiers then it helps all the other phases. It increases overall speed of compiler.

Date: / /
Page No. 25

⇓

Intermediate code generation

⇓

$t_1 = b * 60.0$
$t_2 = a + t_1$
$x = t_2$

⇓

code optimisation

⇓

Not in syllabus

$t_1 = b * 60.0$
$x = a + t_1$

⇓

Target code generation

⇓

MUL t1, b, 60
ADD x, a, t1

It is possible that when semantic analyser checks symbol table for 'type' of "x", type may not be mentioned even if x entry is present. This is called declaration missing.

∴ Semantic analyser does type checking as well as finds undeclared variables.

x = a+b * 60

↓ i/p.

LA

x - (id,1)
a - (id,2)
b - (id,3)

Here x, a, b are not declared. So symbol table is empty. When this is sent to LA, LA has to give id to x. It checks in symbol table & finds that it is not present. So instead of sending an error, it adds x to symbol table without giving it a TYPE.

Now when semantic analyser wants to know type of identifier, it checks symbol table but type missing. So it sends declaration missing error.

* Who can write in symbol table → All
  mainly who can write ——→ semantic
  If not semantic then ——→ Lexical

* If we write float x, a, b then semantic analyser comes before LA, stores these variables in symbol table.

* If semantic analyser has already written the entries in symbol table and in the prog. float x, a, b come again, SA comes into picture again to add them to symbol table. But since they are present already, Sem A. gives Multiple Declaration Error.

* After compilation, we get Machine Dependent Code.

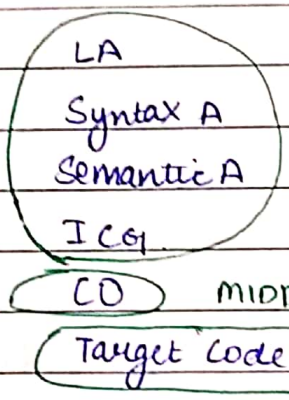
* Some compilers stop at ICG phase. The o/p of ICG is still machine independent code. Java implements this and hence we get Byte code. ∴ Java is platform inapendent / highly portable.

* compiler is a s/w which converts a language to another language.
  If C compiler :- HLL to LLL
  If Java compiler :- HLL to Intermediate code

Date: / /
Page No. 27

Comparing C, Java :- Compilation : Java faster (has only 4 phases)
Running : C faster (we already have LLL)

There are some languages that donot require any compilation (like python) These are interpreted lang & they are very slow. Here the interpreter takes in line by line, converts the line into LL & then executes it.

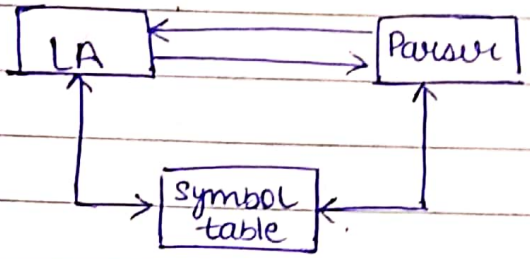


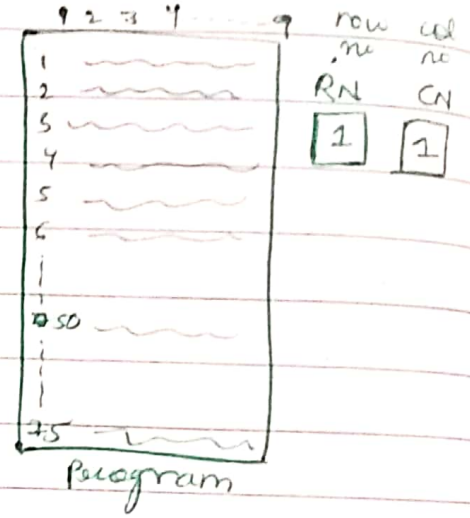
Compiler has a large amt. of code when a prog. has to be compiled, the compiler's code has to be executed. To execute compilers code, it should be present in main memory. So we run compiler's code phase by phase.
So
If total compilation is done at a time then it is known as singee pass compiler otherwise multipass compiler.

	adv.	disadv.
Singee pass compilation :-	less time	more space
multi pass compilation :-	less space	more time

Date: / /
Page No. 28

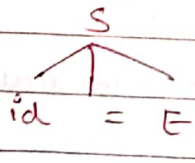
CH-2 LEXICAL ANALYSER :-





(1) Parser begins the process by making parse tree
S → fe | we | AE | BE | id = E
But it has confusion, which one to take so it asks LA to give token()

(2) LA is now present at 1st row 1st col and starts reading. It gets to know that first 3 characters of 1st row give 1st token. It then returns that token (id) to parser.

(3)


} made by parser

(4) Now again parser sends give token() to LA. Now LA reads next TOKEN as "id" but parser was expecting "=". So syntax analyser informs error handler that syntax error occured.

(5) Error handler asks LA where it is right now & it makes an entry of syntax error along with LA location.

(6) Just when id comes into parse tree, semantic analysis begins its working by checking its type in symbol table.

∴ All 3 phases LA, Syntax A. and Semantic A. are working simultaneously.

→ Lexical analyser is first phase of compiler which is also known as SCANNER. It will divide the given program into meaningful strings which are known as tokens.

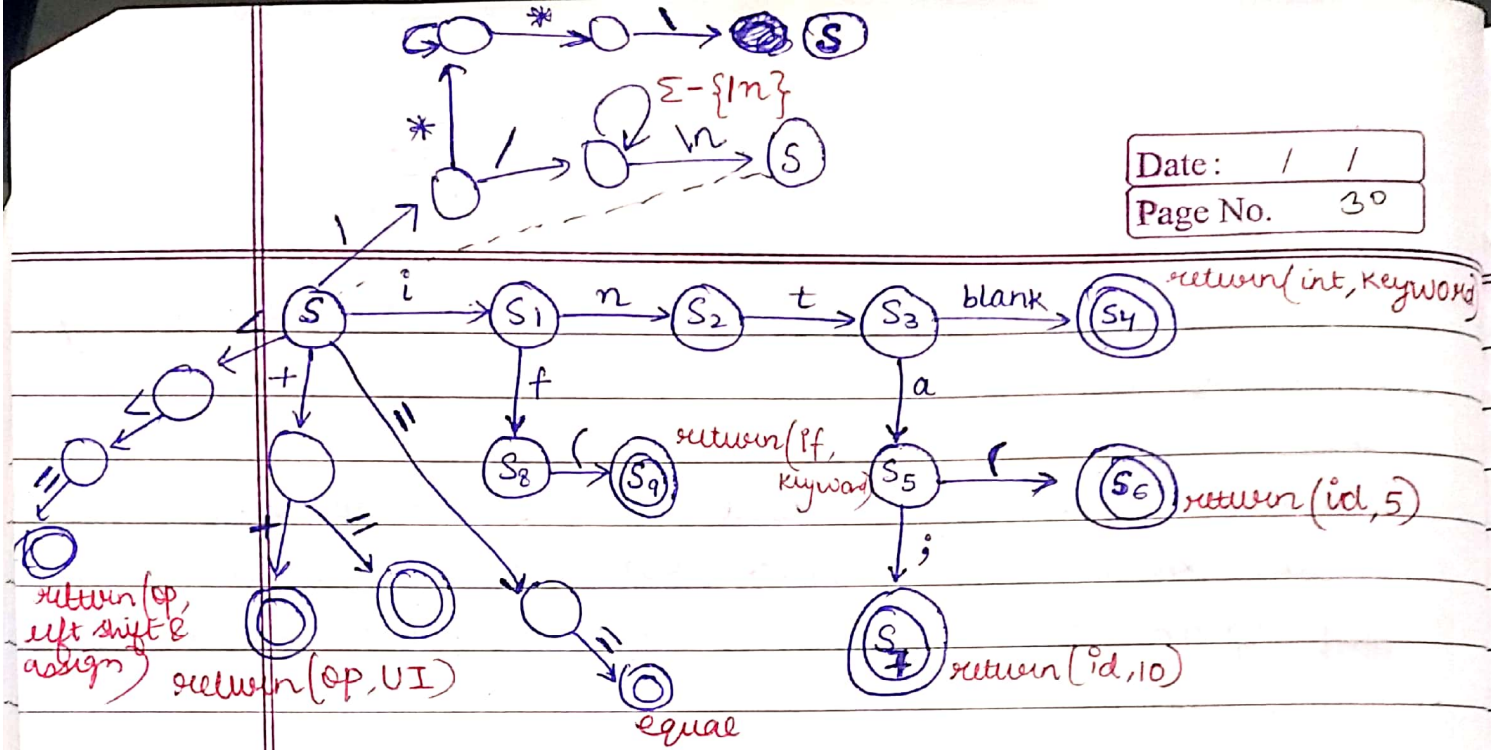
→ Different types of tokens are :-
1. identifier
2. operator
3. keyword
4. constant
5. Special symbols.

→ Responsibility of lexical analyser :-
1. Dividing the given program into tokens.
2. It will eliminate white space characters
3. It will ignore / eliminate comment lines.
4. It will help in giving error messages by providing row no. & column no.

Ex Find no. of tokens present in the following C program :-

```
int main()
{
  /* Finding max of a & b */
  int a = 10, b = 20, max;
  if (a<b)
    max = b;
  else
    max = a;
} return (max);
```

ans = 37



a is token
b is token
c is token
} abc is token Hence take the longest matching token.

If at start symbol we get + then go further
if blank → + is token
if + → ++ is token.

+=
-=
*=
/=
} all present in C op. list

&&= ⇒ not present
%= } present in C
&= } operation list

eg3
```
int main()
{
    x = a + b * c;
    int x, a, b;
    printf("you and your family got to hell %d %d", a, b);
{
```

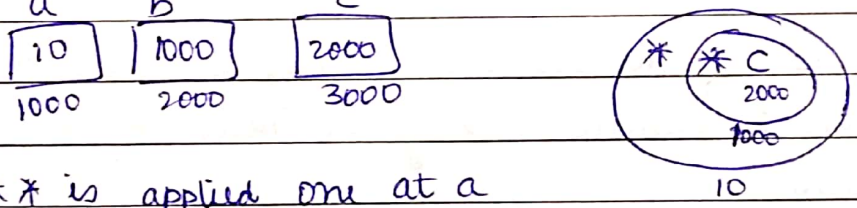
ans 13

We will not go inside " ". because 123.4 is 1 token & not 5. "a" is 1 token. Similarly anything inside " " is one string.

ans 32

ex3
```
main()
{
  x = a+b-c;
  y = ++++---===*/*++++++-----=-=-
      ***+++===***/+-+-;
  printf("%d %d %d", a, *a, **a, ***a, && &a);
}
```

Ans



** is applied one at a time so they are 2 tokens whereas ++a here ++ is applied on a to get 11.

∴ ans 49

ex
```
main()
{
  x = a+b;
  y = xab-123abcdef***ghi**/***
  x = ab+cd-gh abcd/*****/abcd
}
```

Ans 24 Variable name cannot start with integer. If the comment never ends & finally entire program is over then it gives token error. Also if "entire program ends within " then also token error.

```
y = xab-...../*
x = ---
}
```
- here comment didn't close

```
"y = xab---
x = ----
}
```
here " didn't close

Q
```
main()
{
   int x = 10.;
   int x = 10;
   i /* comment */ nt x = 10;
}
```

ans 22

Q Which one of the following string is said to be a token without looking at next i/p character.

a) return
b) if
c) main
d) +

a) cannot be because return followed by (or blank can be a keyword.
b) if cannot be because it can be keyword.
(c) main can be keyword or main can be variable
(d) Operator
(e) ; can be a string or [, ( ) ++]

ans will be where what follows the op does not impacts its meaning.

Q
T_1 : a? (b/c)* a
T_2 : b? (a/c)* b
T_3 : c? (b/a)* c

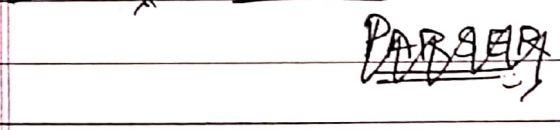
Note:- x? means 0 or 1 occurence of x
i/p :- bbaacabc

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

a)

T_1 :- bba	T_1 :-	T_3 :- bbaac
T_2 :- acab	T_1 :-	T_3 : abc
T_3 :- c	T_3 :-	longest matching

If total string came from just 1 token then ans would be that TOKEN.



PARSER

CH-3 PARSER

* The process of driving the string from the given grammar is known as derivation (Parsing).

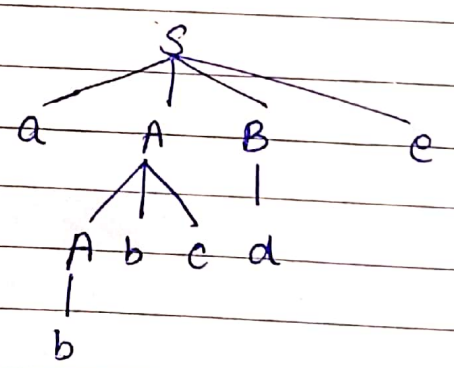
* Depending upon how derivation is performed, we have 2 types of parser
  ① Top down parser ② Bottom up parser

* **WORKING OF TOP DOWN PARSER :-**

S → aABe
A → Abc | b
B → d
i/p :- abbcde $
↳ lookahead symbol

Some T-D parser also apply backtrack to decide which possibility to take
TD with backtrack : recursive descent
TD without backtrack : LL(1)

top-down begins from start symbol & proceeds to string.



top down
Every parser follows leftmost derivation only because string is read from left to right

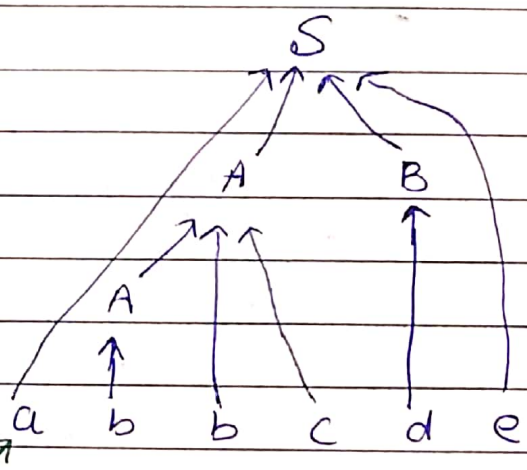
At variable A we have dilemma which one to choose out of Abc or b. This is the difficulty with top down parser. as a variable as more than one possibility
Top down parsers are generally called LL parser.

Date: / /
Page No. 35

WORKING OF BOTTOM UP PARSER :-

→ It starts from string & goes to start symbol.
→ Bottom up parser also scans the string from left to right.
→ These are generally called LR parser.

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$
i/p :- abbcde



④ $S \rightarrow aABe$
③ $B \rightarrow d$
② $A \rightarrow Abc$
① $A \rightarrow b$

No reduction when it was read first time. So skip

* Bottom up parser difficulty is which string to compress which one shouldn't be compressed.
* Handling substring is difficult here.
* Bottom up parser follows no derivation because no variables.
* Bottom up looks like ~~top down parser~~ rightmost parser in reverse. (more apt)

NO Parser follows Rightmost Derivation

→ It follows reverse of rightmost derivation.
→ Identifying correct handle (substring) is always difficult in bottom up parser.

16/11/2019

Date: / /
Page No. 36

TOP DOWN PARSER

- With backtrack (Recursive descent Parser)
- Without backtrack (LL(1) parser or Non recursive descent or predictive parsing)

RECURSIVE DESCENT PARSER.

start symbol.

```
S()
{
  select any production of S
  (S → x1, x2 x3 ..... xk)
  for (i = 1 ; i ≤ k ; i++)
  {   if (xi is variable)
          xi();
      else
          if (xi == lookAheadSymbol)
              incr. pointer
      else
          error ⟹ try another possibility
                   ∴ Backtrack
  }
}
```

eg.

S → ABC | DEF | GHI
A → ab | gh | mn
B → cd | ij | op
C → ef | kl | qr
D → d G → g
E → e H → h
F → f I → i

I/P :- abcdqr $

lookahead symbol.

lex. analysis : dfa
Parse analysis: stack } dfa + stack = PDA

func. call :- push
func. over :- pop

Push (S)
Push (A)
Pop (A)
Push (B)
Pop (B)
→ Push (C)
Pop (C)

Date: / /
Page No. 37

Ans Note that every parser requires an i/p string & grammar as its input.

① Recursive begins from start symbol with backtracking
② we have to derive a (1^{st} i/p symbol ⇒ lookahead symbol)
③ Select any production of S S → ABC
④ Now since K = 3, loop goes 3 times

i = 1	i = 2	i = 3
A()	B()	C()
A → ab (x_1, x_2)	B → cd	BC → ef
i = 1 i = 2	i = 1 i = 2	i = 1 i = 2
a b	c d	e f
check if a == LAS Yes! ∴ incr. i/p ptr	check if b == LAS Yes. incr i/p ptr.	check if c == LAS Yes! incr. i/p ptr	check if d == LAS Yes! incr. i/p ptr.	check if e == LAS NO! ERROR. ⇒ Backtrack

If C failed in identifying i/p string, then entire work done by A, B, C has to be cancelled and another production of S has to be taken.

If all productions of S fail to identify i/p string then compiler generates parsing error or syntax error.

C → Kl.	C → qr
i = 1 i = 2	i = 1 i = 2
K l	q r
check if K == LAS NO! Error ⇒ Backtrack.	check if q == LAS yes! incr. i/p pointer	check if r == LAS yes! incr i/p pointer

disadv: Time wasted in backtracking

eg2 $S \rightarrow aS|b$

i/p : aaab $

S()

$S \rightarrow aS$

i = 1

a

a == LAS

Yes.

incr ptr.

i = 2

S()

$S \rightarrow aS$

i = 1

a

a == LAS

Yes

incr i/p ptr

i = 2

S()

$S \rightarrow aS$

a

i = 1

a == LAS

Yes

incr i/p ptr

S

i = 2

$S \rightarrow aS$

i = 1 i = 2

a

a == LAS

No

error ⇒ Backtrack

$S \rightarrow b$

i = 1

b == LAS

Yes.

Incr. ptr.

Push(S)
Push (S)
Push (S)
Pop (S)
Pop (S)
Pop (S)
Pop (S)

eg3 $S \rightarrow Sa|b$

i/p : baaa $

Lft recursion!!

S()

$S \rightarrow Sa$

i=1 i=2

S() a

$S \rightarrow Sa$

S i

$S \rightarrow Sa$

- stack overflow msg.
- infinite loop.

eg $S \to ABC \mid DEF \mid GHI$.
$A \to ab \mid abc \mid abcd$
$B \to ef \mid gh \mid ij$
$C \to kl \mid mn \mid op$
$D \to d$ $G \to g$
$E \to e$ $H \to h$
$F \to f$ $I \to i$
i/p :- abcdghOP $

S()
S → ABC
i=1 i=2 i=3
A B C
A → ab | B → ef
a b | e f
Yes Yes | NO NO.
B → gh
g h
NO NO
B → ij
i j
NO NO
∴ S → ABC is wrong.

S()
S → DEF
i=1 i=2 i=3
D E F
D → d
d
NO.
∴ S → DEF is wrong

S()
S → GHI
i=1 i=2 i=3
G H I
G → g
NO.
S → GHI is wrong!

Push S.
Push A
Pop A
Push B
Pop B
Push D
Pop D
Push G.
Pop G.
Pop S

∴ Parsing Error.
string not found by compiler even if actually it is possible.
S → ABC
A → abcd
B → gh
C → op.

- In recursive descent parser, every variable is having recursive func. (same code).
- If grammar contain left recursion, recursive descent parser may go to infinite loop.
- If grammar contain left factoring, recursive descent parser may give parser error.
- In recursive descent parser, lot of time is wasted in form of backtracking (exponential time)

Imp :- • LL(1) PARSER :-

→ It has same algo as that of recursive descent. Only diff. is that instead of selecting any production, LL(1) parser selects the correct production

→ Without backtracking by using parsing table.

→ To construct LL(1) parsing table, we need 2 functions :-

1. First ( )
2. Follow ( )

FIRST( ) :-

→ found for variables

S → abc | def | ghi

First (S) = First(abc) + First(def) + First(ghi)

= First(a) + First(d) + First(g)

= {a, d, g}

First (Terminal) ⇒ Terminal
First (ϵ) ⇒ ϵ

First (A) :- contains set of all terminals present in 1st position of every string derived by A.

eg
S → ABC
A → a | d | e | f | ϵ
B → b | g | h | i | ϵ
C → c | j | K | ϵ

First (S) = First (ABC)
= First (A)
= Fi(a), Fi(d), Fi(e), Fi(f), Fi(ϵ)
= a, d, e, f, ~~ϵ~~ Fi(BC)
⇓
Fi(B)
⇓
b, g, h, i, ~~ϵ~~ Fi(C)
⇓
c

S is generating 12 kinds of strings where each string begins from a, d, e, f, b, g, h, i, c, j, K, ϵ.

ex
S → aBDh — First (S) = {a}
B → bC | ϵ — First (B) = {b, ϵ}
C → cC | ϵ — First (C) = {c, ϵ}
D → FE — First (D) = {g, f, ϵ}
F → g | ϵ — First (F) = {g, ϵ}
E → f | ϵ — First (E) = {f, ϵ}

First(S) - go LHS of production & find S
Follow(S) : go RHS of production & find S

Date: / /
Page No. 42

eg $S \rightarrow aSbS / bSaS / \epsilon$

First(S) = {a, b, ∈}

eg $S \rightarrow AaAb / BbBa$ First(S) = {a, b}
$A \rightarrow \epsilon$ First(A) = {∈}
$B \rightarrow \epsilon$ First(B) = {∈}

First(S) = First(AaAb) , First(BbBa)
= First(A) , First(B)
= ~~∈~~ First(aAb) , ~~∈~~ First(bBA)
a b
= {a, b}

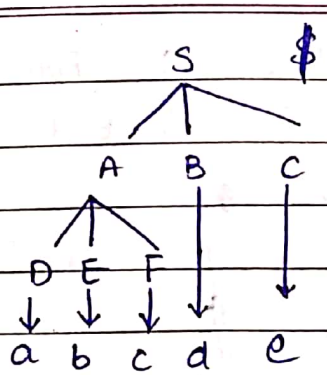
FOLLOW() :-

Follow(S) contains set of terminals present in 1st position of every string immediate right of S.

{a} = First(S) means it generates a string which begins with a

{b} = Follow(S) means it generates a string where after S whatever string comes has its first letter as b.

eg. $S \rightarrow ABC$
$A \rightarrow DEF$
$D \rightarrow a$ $B \rightarrow d$
$E \rightarrow b$ $C \rightarrow e$
$F \rightarrow c$ i/p :- abcde$



FOLLOW (D) = First (EF) = First (E) = b

FOLLOW (F) = FOLLOW (A) = First (BC) = First (B) = d

FOLLOW (C) = FOLLOW (S) = First ($) = $

A → DEF ; follow (F) means after F who is there. But there is nothing means A → DEF production is over. So we need to find follow (A)

eg2 S → ABC

A → DEF

D → a

E → ∈

F → c

B → ∈

C → e

i/p : ace $

FOLLOW (D) = First (EF) = {First (E) = {∈} First (F) = {c}

FOLLOW (F) = FOLLOW (A) = First (BC) = {e}

FOLLOW (C) = FOLLOW (S) = First ($) = {$}

eg3 S → ABC

A → DEF

D → ∈ B → ∈

E → ∈ C → ∈

F → ∈ i/p :- $

Follow (B) = First (C) = {ϵ} = Follow (S) = First ($) = $
Follow (D) = First (EF) = First (E) = {ϵ} First (F) = {ϵ}
= Follow (A)
= First (BC) = First (B) = {ϵ} = First {C} = {ϵ}
= Follow (S) = First ($) = $

Follow() Never Gives {ϵ}. It will give $ if no letter comes

Rules to find Follow:-
assume A is variable

① If A is start symbol then Follow (A) = {$}

② B → CADG } Follow (A) = First (DG) = First (D) = {b}
D → b

③ B → CA or B → CAD
D → ϵ
Follow (A) = Follow (B)

After A no one

Q
E → TE'
E' → ϵ | +TE'
T → FT'
T' → ϵ | *FT'
F → id | (E)

	First()	Follow()
E :-	{id, (} ✓	{), $} ✓
E' :-	{ϵ, +} ✓	{), $} ✓
T :-	{id, (} ✓	{+,), $} ✓
T' :-	{ϵ, *} ✓	{+,), $} ✓
F :-	{id, (} ✓	{*, +,), $} ✓

Q $S \to aSbS \mid bSaS \mid \epsilon$

first(S) = {a, b, ϵ}
follow(S) = {b, a, $}

Q $S \to AbAb \mid BaBa$
$A \to \epsilon$
$B \to \epsilon$

	First()	Follow()
S	{b, a}	{$}
A	ϵ	{b}
B	ϵ	{a}

Q $S \to (L) \mid a$
$L \to SL'$
$L' \to \epsilon \mid ,SL'$

	First()	Follow()
S	(a	$,)
L	(a	)
L'	ϵ ,	)

Q $S \to aBDh$ $\quad D \to FE$
$B \to bB \mid \epsilon$ $\quad E \to f \mid \epsilon$
$C \to cC \mid \epsilon$ $\quad F \to g \mid \epsilon$

	First	Follow.
S	a	$
B	b, ϵ	g, f, h
C	c, ϵ	
D	g, f, ϵ	h
E	f, ϵ	h
F	g, ϵ	f, h

LL(1) PARSING TABLE CONSTRUCTION ALGO :-

For each production $A \to \alpha$ repeat the following

(1) Add $A \to \alpha$ under $M[A, b]$ $\forall b \in First(\alpha)$

(2) If $first(\alpha)$ contains $\in$ then add $A \to \alpha$ under $M[A, c]$ $\forall c \in follow(A)$

Construct LL(1) parsing table for following grammar

$S \to (L) \mid a$

$L \to SL'$

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

No. of rows in parsing table = No. of variables

No. of cols. in parsing table = No. of terminals + $

	,	a	(	)	$
S		$S \to a$	$S \to (L)$		
L		$L \to SL'$	$L \to SL'$		
L'	$L' \to ,SL'$			$L' \to)$	

* If a grammar has 20 variables & 25 terminals then size of parsing table = 20×26.
* Only 1 entry should be present in each cell.
* $L' \to ,SL'$ => Parser should be at L' row when LAS = , then take this prod. to $M[L', ,]$ ← First(,SL')

$L \rightarrow SL' \Rightarrow M[L, First(SL')]$

$\Downarrow$ First (S)

$\Downarrow$ (, a

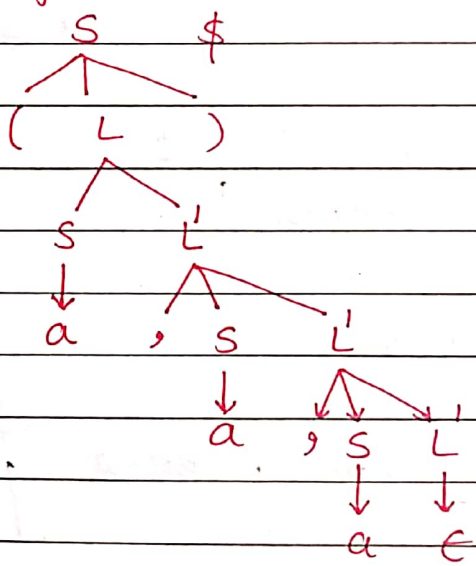
$L' \rightarrow \epsilon \Rightarrow M(L', first(\epsilon))$

ϵ but ϵ col not present

Follow $(L') \Rightarrow)$

$M[L',)]$

For the given grammar, i/p : (a, a, a) $



* If while deriving a string, we try to access a block which is empty (no production present) then it is an invalid string & parse error will be generated.

* No. of empty blocks in LL(1) = error entries.

* If while writing productions in table, one block contains more than 1 entry. Then given grammar is not LL(1).

* In the above LL(1) parsing table, no entry contains

multiple productions. So given grammar is LL(1).

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

$S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

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

	a	b	$
S	$S \to AaAb$	$S \to BbBA$	
A	$A \to \epsilon$	$A \to \epsilon$	
B	$B \to \epsilon$	$B \to \epsilon$	

Q $S \to aSbS \mid bSaS \mid \epsilon$ LL(1) ??

First (S) = { a, b, ϵ }

Follow (S) = { b, a, $ }

	a	b	$
S	$S \to aSbS$ $S \to \epsilon$	$S \to bSaS$ $S \to \epsilon$	$S \to \epsilon$

Not in LL(1)

Q $E \to {}^{(1)}TE'$

$E' \to {}^{(2)}\epsilon \mid {}^{(3)} * TE'$

$T \to {}^{(4)}FT'$

$T' \to {}^{(5)}\epsilon \mid + FT' {}^{(6)}$

$F \to {}^{(7)} id \mid (E) {}^{(8)}$

	First	Follow
E	id, (	$,)
E'	ϵ, *	$,)
T	id, (	*, $,)
T'	ϵ, +	*, $,)
F	id, (	+, *, $,)

	id	(	)	+	*	$
E	$E \to TE'$ (1)	$E \to TE'$ (1)				
E'			$E' \to \epsilon$ (2)		$E' \to * TE'$ (3)	$E' \to \epsilon$ (2)
T	$T \to FT'$ (4)	$T \to FT'$ (4)				
T'			$T' \to \epsilon$ (5)	$T' \to + FT'$ (6)	$T' \to \epsilon$ (5)	$T' \to \epsilon$ (5)
F	$F \to id$ (7)	$F \to (E)$ (8)				

Q $S \to {}^{(1)}aBDh$

$B \to {}^{(2)}bC \mid {}^{(3)}\epsilon$

$C \to {}^{(4)}cB \mid {}^{(5)}\epsilon$

$D \to {}^{(6)}FE$

$E \to {}^{(7)}gB \mid {}^{(8)}\epsilon$

$F \to {}^{(9)}bC \mid {}^{(10)}\epsilon$

Date: / /
Page No. 50

	a	b	c	h	$
S	S → aBDh (1)				
B		B → ∈ (3), B → bC (2)	B → ∈ (3)	B → ∈ (3)	
C		C → ∈ (5)	C → cB (4), C → ∈ (5)	C → ∈ (5)	
D				D → FE (6)	
E			E → cB (7)	E → ∈ (8)	
F		F → bC (9)	F → ∈ (10)	F → ∈ (10)	

	First	Follow
S	a	$
B	b, ∈	b, c, h, [illegible]
C	c, ∈	b, c, h, [illegible]
D	b, c, ∈	[illegible] h
F	c, ∈	h
E	b, ∈	c, h

∴ Not LL(1)

SHORTCUT TO CHECK GRAMMAR LL(1) OR NOT.

NOTE:- ① If G has no null productions

$S \to \alpha_1 | \alpha_2 | \alpha_3$

First(α_1) First(α_2) First(α_3)

a, b c, d e, f

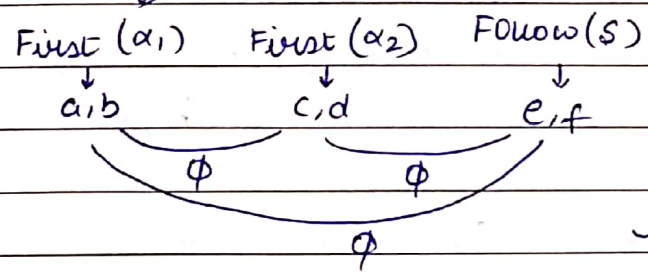
ϕ ϕ

ϕ

} each pair is disjoint. Then it is LL(1).

② If G contains NULL productions

$S \to \alpha_1 | \alpha_2 | \epsilon$



Every pair is disjoint Then it is LL(1)

Q Check if the grammar is LL(1) or not ?

$S \to E | b$

$E \to b$

$S \to E \Rightarrow$ First (E) = b
$S \to b \Rightarrow$ First (b) = b

} Not LL(1).

Q $S \to aS | a$

$S \to aS \Rightarrow$ First (aS) = a
$S \to a \Rightarrow$ First (a) = a

} Not LL(1).

* If grammar contains left factoring then it cannot be LL(1).

Q $S \to Sa | b$

$S \to Sa$ First (Sa) = First (S) = First (SA), b
$S \to b$ First (b) = b

Not disjoint
$\Rightarrow$ Non LL(1)

* If grammar contains left recursion then it cannot be LL(1).

$S \to aSbS \mid bSaS \mid \in$

$S \to aSbS \Rightarrow First(aSbS) = a$
$S \to bSaS \Rightarrow First(bSaS) = b$
$S \to \in \Rightarrow FOLLOW(S) = \$, b, a$

} Not disjoint

Not LL(1) as it is ambigous also.

* Ambigous grammar can never be LL(1).

Note:- Every regular grammar need not be LL(1).
eg. $S \to aS \mid a$ (Not LL(1))

This is because that reg. grammar may contain left recursion or right recursion or ambiguity

* If given grammar is LL(1) then it cannot contain
  - left recursion
  - left factoring
  - ambiguity

$S \to ab \mid ac \mid ad \Rightarrow$ left factoring exists
LL(1) fail
LL(2) means take 2 symbols as lookahead
ab, ac, ad.
$\therefore$ LL(2) ✓

In terms of power

$LL(1) \subseteq LL(2) \subseteq LL(3) \subseteq \cdots \subseteq LL(K)$

L → L → leftmost derivation

scan i/p from left → L → R → reverse of rightmost derivation

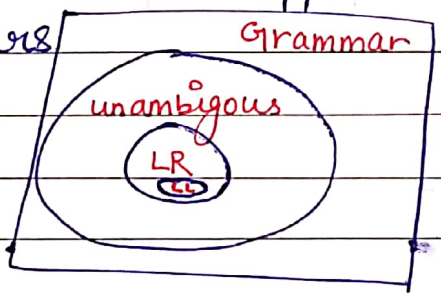
Date: / /
Page No. 53

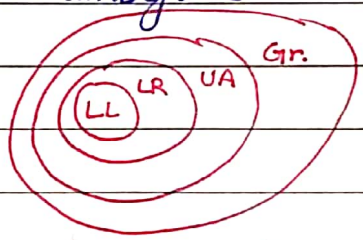
BOTTOM UP PARSER:

LR- Parser. | Operator Precedance Parser.

LR PARSER :-

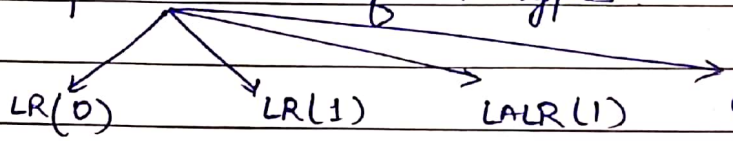
(1) LR parsers will be applicable only for unambiguous grammars





- Every LR is unambigous but very unambigous need not be LR.
- Every LL is LR. but every LR need not be LL.

LR parsers are of 4 types



For all LR parsers, parsing algorithm is same. but parsing table is different.

Imp LR PARSING ALGORITHM :-

Let X be the state number on top of stack & "a" is lookahead symbol.

1) If action [X, a] = S_i then shift a && i and incr. i/p pointer.
2) If action [X, a] = r_j and j^{th} production is $\alpha \to \beta$ then pop $2 * |\beta|$ symbols & replace by α.

If S_{m-1} state number below α then push goto $[S_{m-1}, \alpha]$.

3) If action $[X, a]$ = blank then parsing Error
4) If action $[X, a]$ = accept then successful completion of Parsing

ex
$S \rightarrow AA$
$A \rightarrow aA/b$
i/p :- abab $

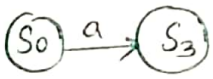
AA } 3 handles
aA
b

	action			goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			acc		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

LR(0) parsing table.

No. of rows = no. of states in DFA
No. of col = action + goto
terminal + $ variables.

→ LR parse table is almost double to that of LL parse table.
→ LR parser are more powerful than LL as they can accept more languages.



Date: / /
Page No. 55

SANCHITA

i/p	Stack	Reduction
abab$	0 ⇒ 0a3	
bab$	0a3 ⇒ 0a3b4	
ab$	0a3b4 ⇒ 0a3A6 ⇒ 0A2	A → b, A → aA
b$	0A2 ⇒ 0A2a3	
$	0A2a3 ⇒ 0A2a3b4	

starting from state 0

TOS

LAS

go to A-3 = 6 to remember A in stack

b replaced by A. Remove 2x1 = 2 symbols from stack.

2x2 = 4. Pop 4 sym a3A6 & replace by A. Then goto 0A ⇒ 2 .. Push 2.

when read goes on, i/p doesn't incr.

NOW $-4 ⇒ r_3 ⇒ A → b Pop b4, replace by A then Push 3A = (6)

0A2a3A6

NOW $-6 ⇒ r_2 ⇒ A → aA Pop a3A6, replace by A then Push 2A = (5)

0A2A5

NOW $-5 ⇒ r_1 ⇒ S → AA Pop A2A5, replace by S then Push 0S = 1

0S1

NOW $-1 ⇒ acc Stop Successful completion

S → AA (1)

A → aA (2) | b (3)

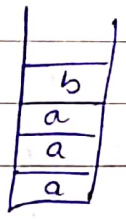
} i/P = aabb$

i/P	Stack	Reduction
aabb$	0 ⇒ 0a3	
abb$	0a3 ⇒ 0a3a3	
bb$	0a3a3 ⇒ 0a3a3b4	
b$	0a3a3b4	A → b Pop b4, replace by A & push 6
	0a3a3A6	A → aA Pop a3A6 replace by A & push 6.
	0a3A6	
b$	0A2 ⇒ 0A2b4	
$	0A2 ⇒ 0A2b4	A → aA
	0A2A5	A → b
	0S1	S → AA

acc

VIABLE PREFIXES :- while doing bottom up parsing, the prefixes present in stack are called viable prefixes.

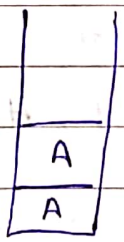
eg.



a, aa, aaa, aaab are viable prefixes.

For above grammar

Handle. NO other symbol can be pushed to BUP stack unless this handle is popped.



AA → viable prefix
AAA → Not viable prefix
b → viable prefix

ex3 i/p :- aaab$

i/p	stack	Reduction
aaab$	0a ⇒ 0a2	
aab$	0a2 ⇒ 0a2a2	
ab$	0a2a2 ⇒ 0a2a2a2	
b$	0a2a2a2 ⇒ 0a2a2a2b4	
$	0a2a2a2b4 ⇒	A → b
	0a2a2a2A6	A → aA
	0a2a2A6	A → aA
	0a2A6	A → aA
	0A2	
	Parsing error as $-2 is empty	

Lecture 5

17/11/2019

Date: / /
Page No. 57

LR(0) PARSING TABLE CONSTRUCTION :-

To construct LR(0) parsing table, we need 2 functions
i) closure()
ii) goto()

G: S → AA
A → aA | b

G': S' → S
S → AA
A → aA | b

Both G & G' language are same
Only a unit production S' → S is added.
} Augmented grammar

S → .abc , S → a.bc , S → ab.c , S → abc. (as they have •)
↑ LR(0) items

• Indicates how much production is over.
S → .abc - production not started
S → abc. - entire production over.
↳ Final LR(0) item or
complete LR(0) item or
reduced LR(0) item

CLOSURE() :-

closure() will be computed only for LR(0) items

eg1. closure(S → .AA) = ① S → .AA → (write eqn as it is)
② A → .aA → [S → .AA means parser is at A. So we can take A's productions Add . to A's production.
→ .b

eg2. closure(S' → .S) = ① S' → .S → (as it is. Now include S prodn)
S → .AA → (we can write only Variable's prod)
A → .aA } [as . is before terminals, hence we cannot write further
.b }

eg 3 closure (A → .aA) :- ① A → .aA

Definition :- closure (I) :- ① add I
② if A → B.CDE is I and C is a variable which generates C → EFG is in grammar then add C → EFG to closure of I.
③ Repeat ② for every newly added LR(0) item until there is a terminal just after .

GOTO() :- requires 2 i/p
(i) LR(0) item
(ii) some input.
after . whatever is there, give it as i/p

ex① Goto (S → .AA , A) = ① S → A.A (move the dot after the given i/p)
② A → .aA
→ .b
[whatever ans we get in ①, just do closure of it]

ex② Goto (A → .aA , a) = ① A → a.A
② A → .aA
→ .b

Definition :- Goto(I, x) :- ① add I by moving dot after x.
② Apply closure to first step.

LR(0) PARSING TABLE CONSTRUCTION ALGORITHM :-

① I_0 = closure (augmented LR(0) item)
② using I_0 construct DFA.
③ convert DFA to LR(0) Parsing table.

Date: / /
Page No. 59

Q1 Construct LR(0) parsing table for following grammar

G: $S \to AA$

$A \to aA \mid b$

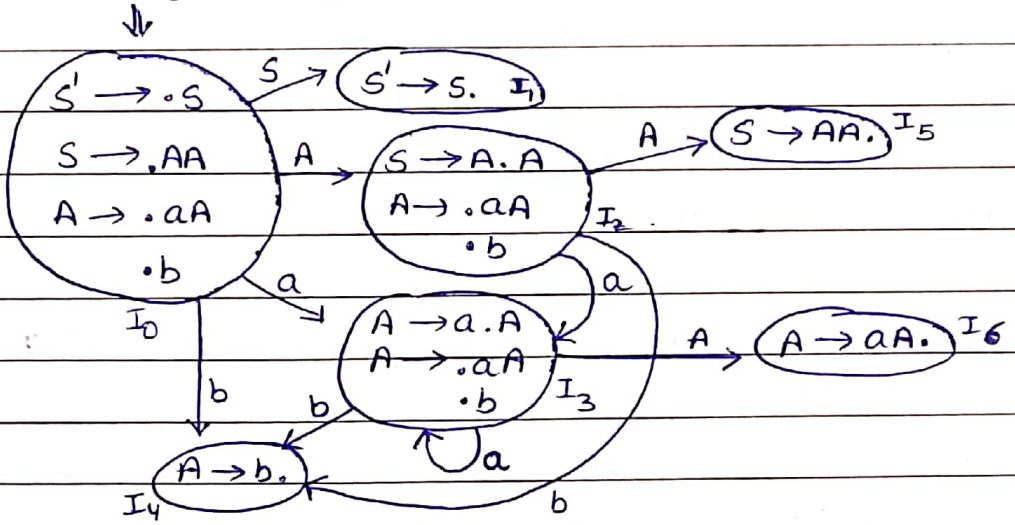
①

G': $S' \to S$

$S \to AA$

$A \to aA \mid b$

② closure $(S' \to \cdot S)$



	a (Action)	b	$	S (goto)	A
I_0	S_3	S_4		1	2
I_1			ACC		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

since LR(0) is blind, we write reduced prod^n in entire action row

Diff b/w LR(0) & SLR(1) parsing table is that in LR(0) we write reduced production blindly everywhere in action part. But in SLR(1) before writing r, we do some calculation to decide where reduced LR(0) item is written.

B/w LR(0) & SLR(1) :-

(1)	Shift entries remain same	True
(2)	Reduced entries may differ	True
(3)	error entries may differ	True
(4)	No. of states may differ	False

Imp
- Shift entries come because of non completed LR(0) item.
- Reduced entries come because of completed LR(0) item
- Not every incomplete LR(0) item generates shift entry (Here see $I_0 \to A$, $I_0 \to S$)

***
- At a particular state, when a terminal is present just after dot then a shift entry is generated eg. at I_2, 3 incomplete productions but only 2 shift entries as $A \to .aA$ & $A \to .b$ Only these 2 have terminal after.

Augmented production is used to check whether S over or not because S occurs no where on RHS of given grammar.

If in a particular state, 2 productions are reduced then LR(0) Parser table will have 2 - reduced entries for that particular

state. This is called RR conflict. Parser doesnot understand which one to pop from stack. r_1/r_3 | r_1/r_3

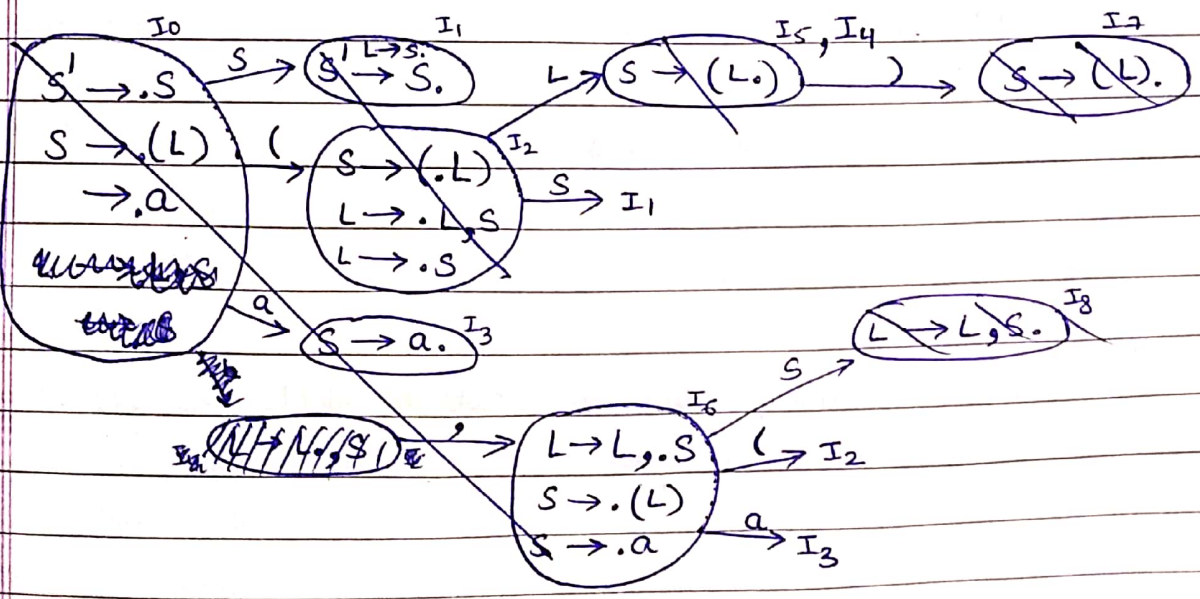
If in a particular state, there are shift as well as reduced productions then LR(0) parser table will have S/R entry for that row. This is called SR conflict. Parser doesnot know whether it needs to push or pop.

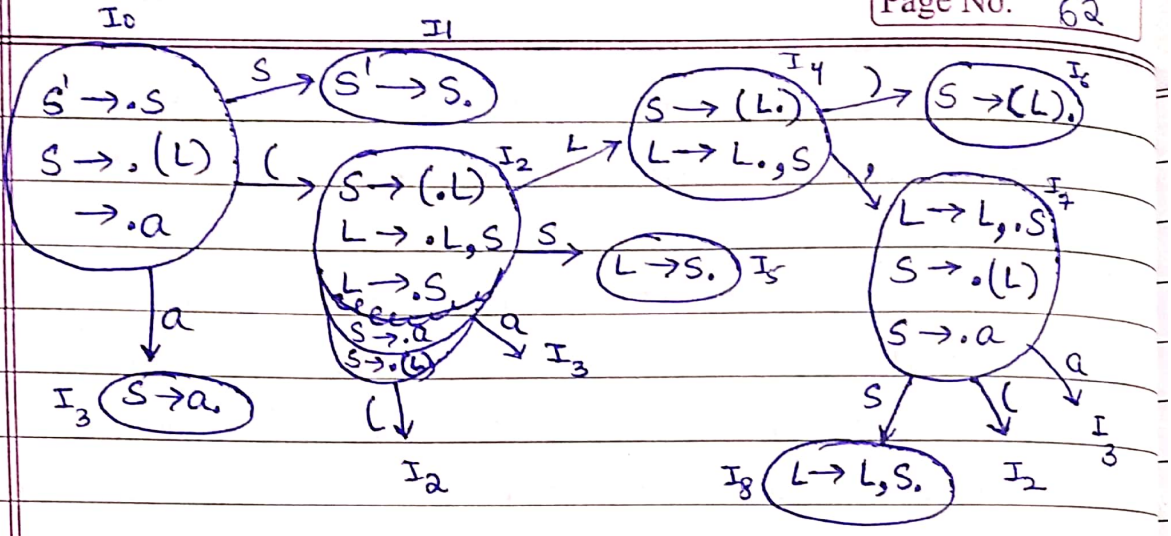
R R
S R
] To get a conflict, minimum 2 productions are required & one of them should be reduced.

To check if grammar is LR(0) or not, draw DFA & LR(0) parsing table. If there is no conflict then given grammar is LR(0) grammar else no.

Q Construct LR(0) parsing table for following grammar:-

G: $S \to (L) \mid a$
$L \to L,S \mid S$





	Action					goto	
	a	,	(	)	$	S	L
I_0	S_3		S_2			1	
I_1					acc		
I_2	S_3		S_2			5	4
I_3	R_2	R_2	R_2	R_2	R_2		
I_4		S_7		S_6			
I_5	R_4	R_4	R_4	R_4	R_4		
I_6	R_1	R_1	R_1	R_1	R_1		
I_7	S_3		S_2			8	
I_8	R_3	R_3	R_3	R_3	R_3		

NO conflict

⇒ LR(0)
SLR(1)
CLR(1)
LALR(1)
} all exists

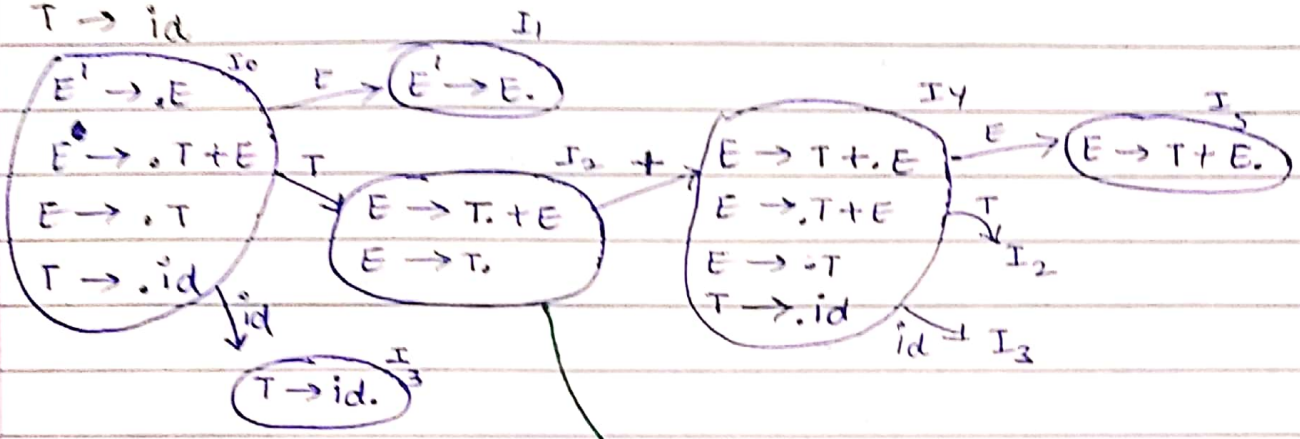
This grammar does not exist in LL(1) as left recursion is present.

Date: / /
Page No. 63

Q Check following grammar is SLR(1) or not ?

$E \to T+E \mid T$

$T \to id$



Not in SLR(0)
May be in SLR(1)

This state has problem because 2 part present & one of them is getting reduced. So IT MAYBE A PROBLEM. But it doesnot guarantee that ~~problem~~ it is a conflict

S-R

S-R occuring on + so draw table for state 2

+	id	$
r_2 / S4	r_2	r_2

Problem would not have occured even because of Shift & reduce ~~be~~ when $E \to T.E$. Here after dot there is a variable whose entry would go in goto part ~~&~~ and there is no conflict in goto part

NOTE:- A state have a problem is called Inadequative state
Here there is 1 inadequative state.

NOW CHECK FOR SLR(1) :-

Here the reduced entry goes to the columns which are part of follow(LHS).

Here $E \to T.$ goes to follow(E) = $

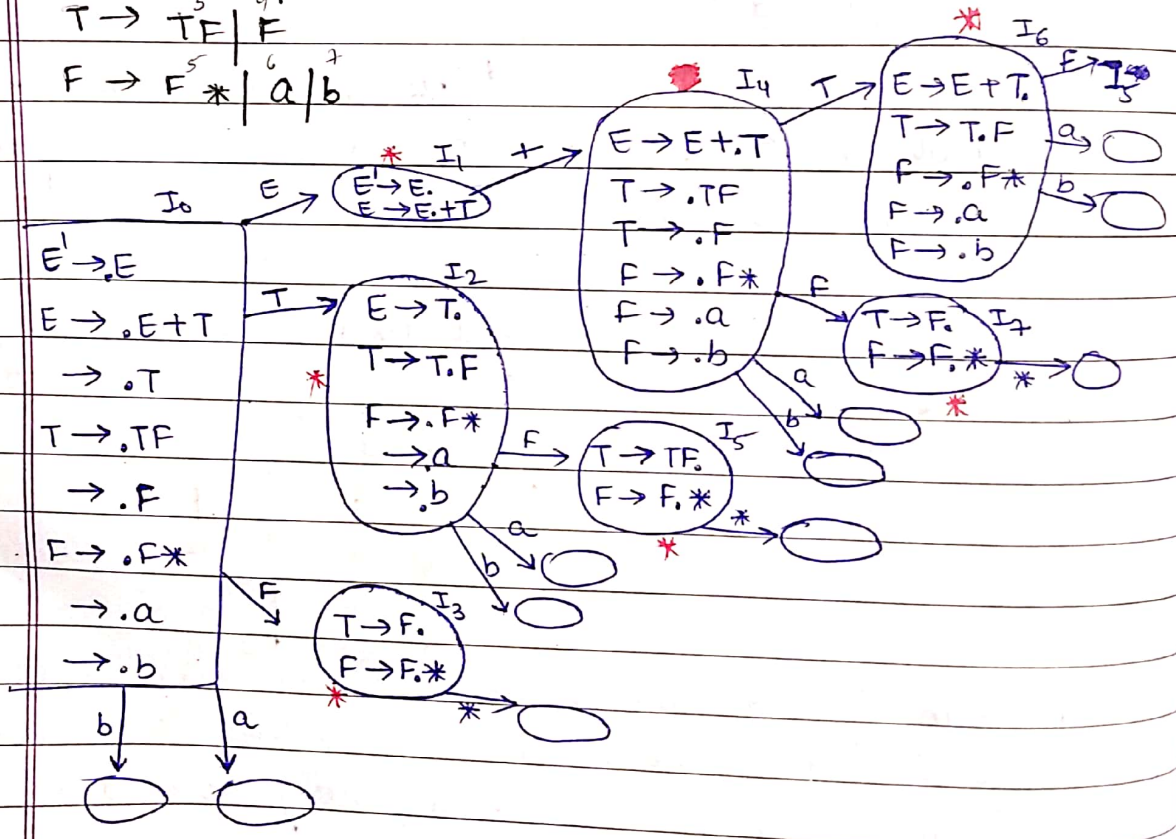
Date: / /
Page No. 64

+	id	$
S_3		r_2

Given grammar is SLR(1)
LALR(1)
CLR(1)

Q Check the given grammar is SLR(1) or not?

$E \to E+T \;(1) \mid T \;(2)$
$T \to TF \;(3) \mid F \;(4)$
$F \to F* \;(5) \mid a \;(6) \mid b \;(7)$



		+	*	a	b	$
	I_0					
X	I_1					
$E \to T.$	I_2	r_2		S	S	r_2
$T \to F$	I_3	r_4	S	r_4	r_4	r_4
$T \to TF.$	I_5	r_3	S	r_3	r_3	r_3
$E \to E+T$	I_6	r_1		S	S	r_1

IF LR(0)
I_1 : not a problem because $E' \to E.$ is not a part of our grammar. So no reduction prod^n
I_2 : RR not possible as only 1 prod^n is being reduced.

IF (LR(0))

follow(E) = {$, +}
follow(T) = {$, +, a, b}
~~follow(F) = {}~~

∴ grammar is in SLR(1).

LL(1) ✗
LR(0) ✗
SLR(1) ✓
⇓
LALR(1), CLR(1) ✓

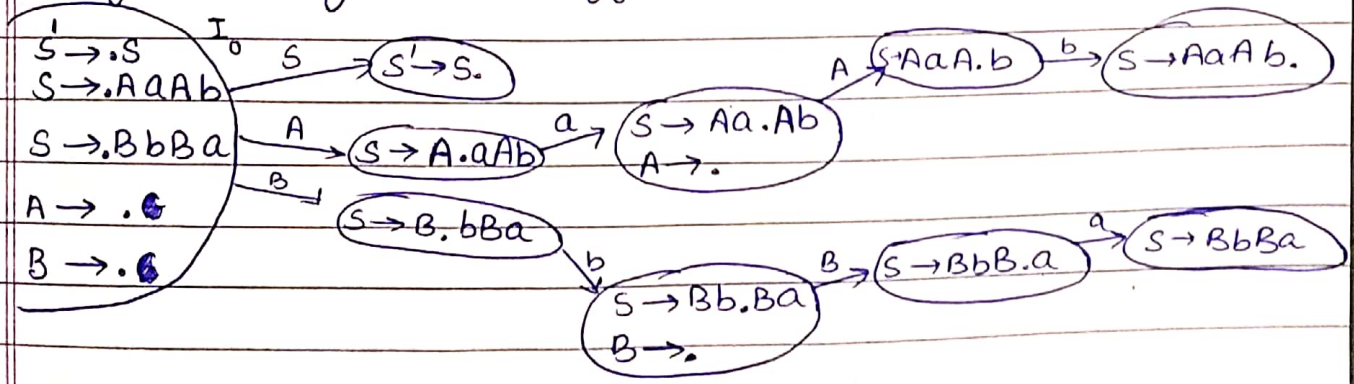
For I_2 on this state by reading terminal you are going outside
∴ 2 shift entries present
But reduced entry everywhere
~~as follow(T) = {$, +, a, b}~~
∴ 2-SR conflicts.

For I_3 :- only 1 is getting reduced.
No R-R.
But SR possible. Reduce occur everywhere. ⇒ (SR-1)

For I_5 :- 1-SR.
For I_6 :- 2-SR.
Total = 6-SR conflicts.
Total 4 inadequative states (2,3,5,6)

Q $S \to AaAb / BbBa$ check if SLR(1) or not ?
$A \to \epsilon$
$B \to \epsilon$

This is in LL(1) ⇒ Top down pass ⇒ since bottom up parser is more powerful hence CLR(1) will also satisfy because LL(1) is most powerful in Top down. and CLR(1) is most poweful in bottom up. LR(0), SLR(1), LALR(1) may or may not satisfy it.



There are 2 reductions. May be RR conflict?

3 RR conflict ← For LR(0) :- r_4, r_5 present in entire row of I_0. ⇒ Not LR(0).

2 RR conflict ← For SLR(1) :- r_3, r_4 present in a, b of I_0.
⇒ Not SLR(1).

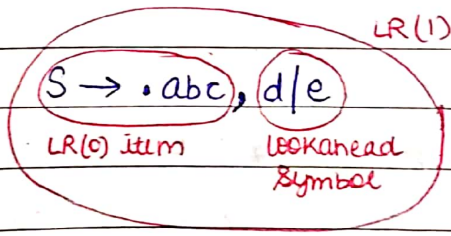
1 inadequative state in LR(0) & SLR(1).

HOW TO CHECK FOR CLR(1) & LALR(1) ?

using LR(0) items → LR(0) parser
↘ SLR(1) parser

using LR(1) items → CLR(1) parser
↘ LALR(1) parser.

LR(1) item = LR(0) item + lookahead symbol.



CLOSURE() & GOTO() FOR LR(1) ITEMS :-

G: S → AA
A → aA | b

G': S' → S
S → AA
A → aA | b

Closure ( ) :

augmented LR(0) item

Ex 1 closure ($S' \to .S$, \$)

⇓ copied as it is from above

① $S' \to .S$, \$ — $.S$ → whatever after S, take its First ( ). Here nothing after S, so \$

② $S \to .AA$, \$

③ $A \to .aA$, a/b
$.b$, a/b

} For lookahead symbol, go to prev. prod and take First (A \$) = a/b

Ex 2 closure ($S \to .AA$, c/d)

⇓

① $S \to .AA$, c/d

② $A \to .aA$, a/b
$\to .b$, a/b

} For lookahead symbol, go to ① & find first (A c d) = a/b.

Definition :- closure (I) =

① add I

② $A \to B.CDE$, a/b) if I and $C \to EFG$ is in Grammar then add [$C \to .EFG$, First (DE, a/b)] to closure (I).

③ repeat step ②

GOTO ( )

goto ($S \to .AA$, c/d , a)

LR(1) item — i/p (after . , a is present)

⇓

$S \to a.A$, c/d

$A \to .aA$, c/d
$.b$, c/d

} taking First (c/d)

No. of states in LR(0), SLR(1), CLR(1)
$n_1 \quad n_2 \quad n_3$

$\Rightarrow (n_1 = n_2) \leq n_3$

Date: / /
Page No. 68

Def :- Goto (I, x) =

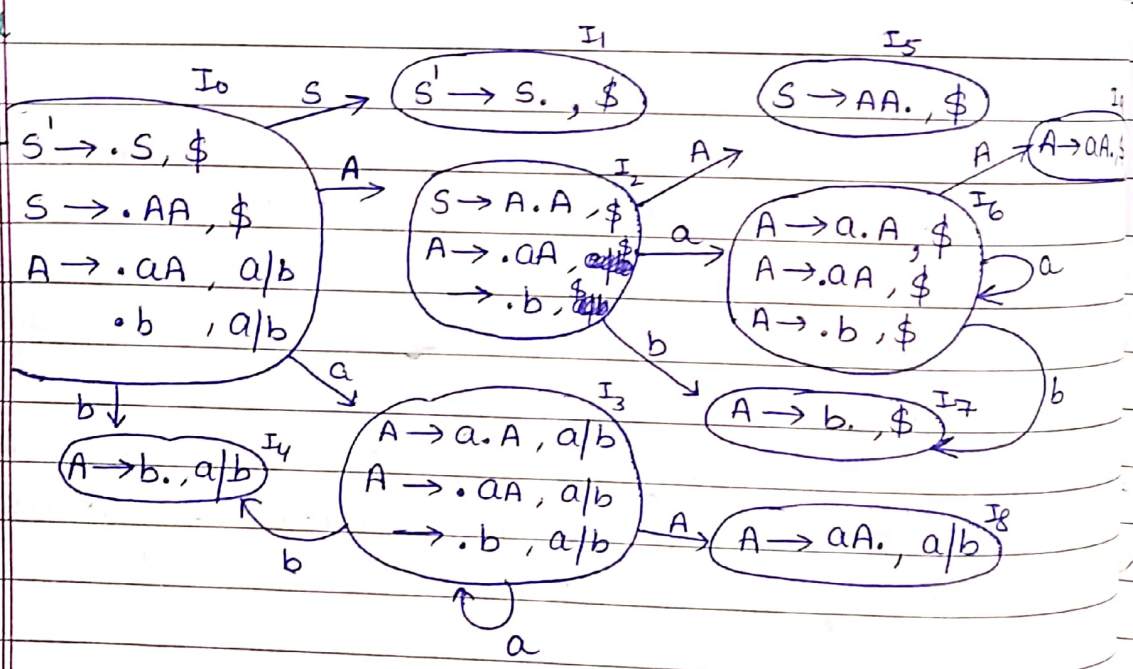
1. add I by moving . after x
2. closure (1^{st} step)

Q Construct CLR(1) & LALR(1) parsing table for following grammar :-

G: $S \to AA$ ①
$A \to aA$ ② $| \; b$ ③

G' :- $S' \to S$
$S \to AA$
$A \to aA \mid b$

For augmented production, LAS = $



In LR(0) DFA, from I_2 on i/p = a we go to I_3. But here we go to diff. state.

A state having a meanings then seprate final if any problem present with any path.

Date: / /
Page No. 69

	ACTIONS			GOTO	
	a	b	$	S	A
0	S_3	S_4		1	2
1			acc.		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

CLR(1) parsing table

To fill reduce entries, write them only in LAS.
For I_4 :- r_3 goes in a, b.
For I_5 :- r_1 goes in $
For I_7 :- r_3 gees in $
For I_8 :- r_2 goes in a, b
For I_9 :- r_2 goes in $

Only drawback of CLR(1) is that its cost is high. To reduce the cost, we minimize the state by using LALR(1).

- In the above CLR(1) table, no conflict. Hence given grammar is CLR(1)
- CLR(1) parser is more powerful comparing to other parsers because it takes care everytime. The only problem is that states more so cost also increases.

Date: / /
Page No. 70

To decrease cost of CLR(1), we apply minimisation algo. [If 2 states differ only by lookahead then make it as single state]

- Minimised CLR(1) is also known as LALR(1).

$I_3, I_6 \equiv I_{36}$ a|b|$ $\equiv$ A → a.A, a|b|$; A → .aA, a|b|$; → .b, a|b|$

$I_4, I_7 \equiv I_{47}$ a|b|$ → A → b., a|b|$

$I_8, I_9 \equiv I_{89}$ a|b|$ → A → aA., a|b|$

	Action			goto	
	a	b	$	S	A
0	S_{36}	S_{47}		1	2
1			acc.		
2	S_{36}	S_{47}			5
3	S_3	S_4			89
4	r_3	r_3	r_3		
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2	r_2		
9			r_2		

LALR(1) parsing table.

no. of states in LALR(1) = n_4
CLR(1) = n_3
SLR(1) = n_2
LR(0) = n_1

$n_1 = n_2 = n_4 \leq n_3$

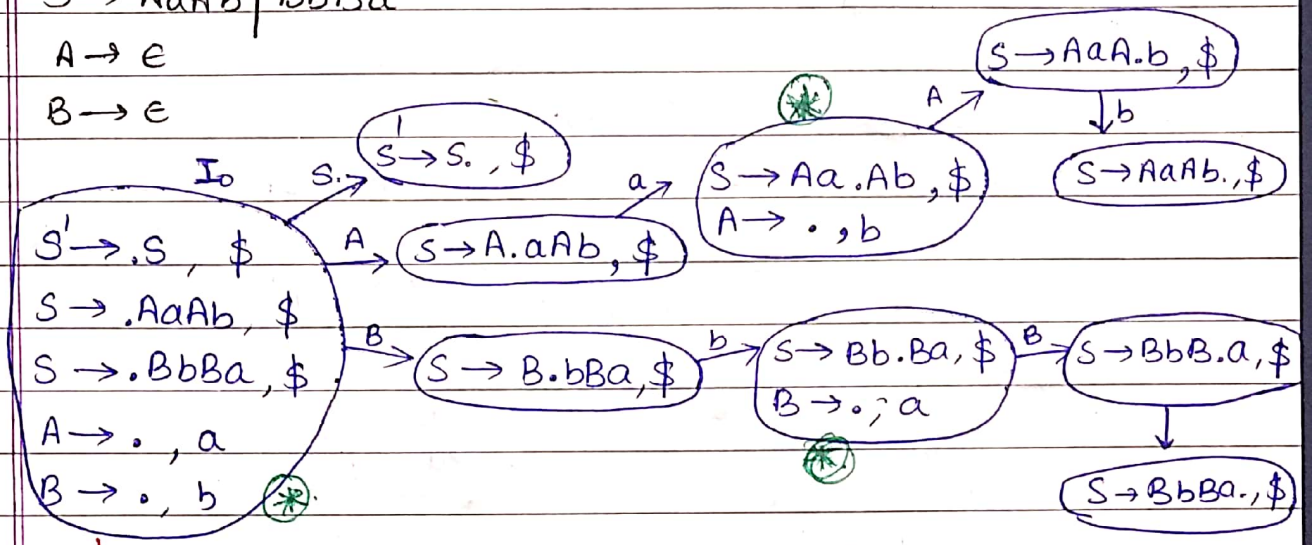
Date: / /
Page No. 71

Q Check if the following grammar is LALR(1) or not?

$S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$



In I0, RR conflict may be possible

But lookaheads are diff. so no problem.

∴ In CLR(1) ✓

		a	b	$	
LR(0)	⇒	r3 | r4	r3 | r4	r3 | r4	⇒ 3RR ⇒ Not LR
SLR(1)	⇒	r3 | r4	r3 | r4		⇒ 2RR ⇒ Not SLR
CLR(1)	⇒	r3	r4		⇒ ✓ ⇒ CLR ✓

No minimised state so CLR ≡ LALR.

∴ LALR(1) ✓

Q Check if following grammar is LALR(1) or not?

$S \to Aa \mid Bc \mid bBa \mid bAc$

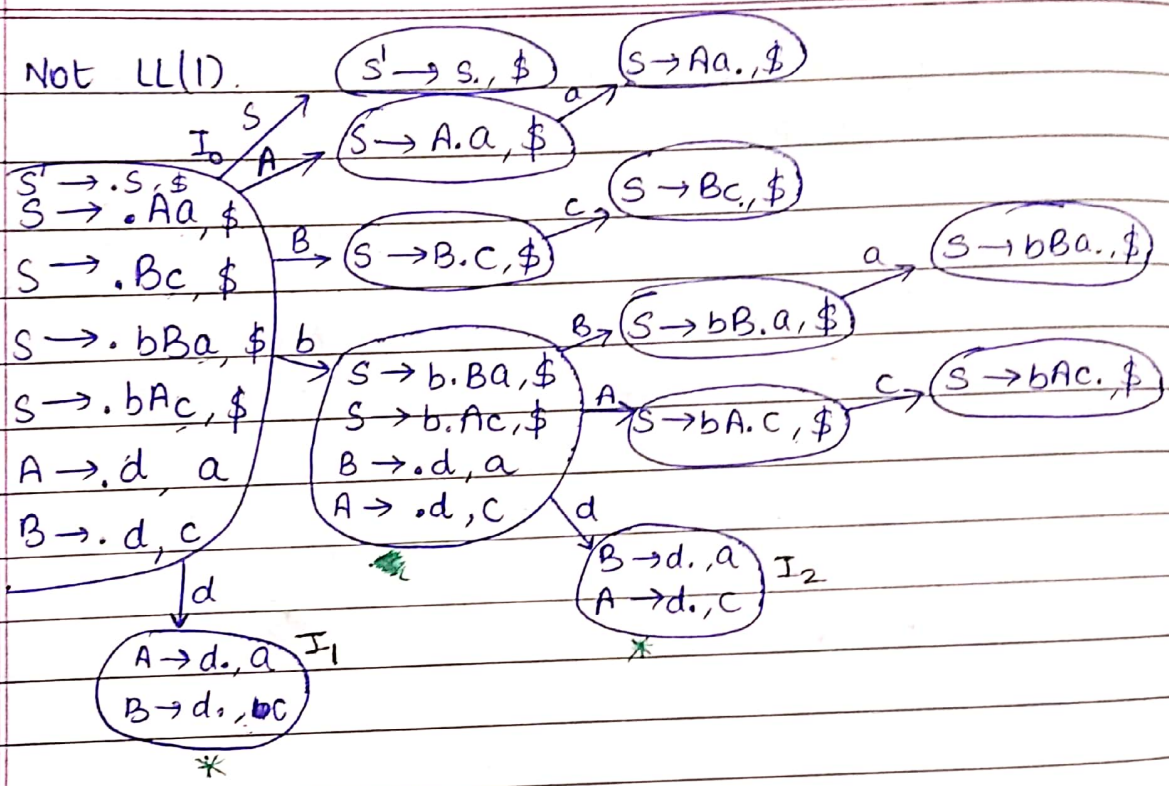
$A \to d$

$B \to d$

Date: / /
Page No. 72

(i) Not LL(1).

(ii)



	a	b	c	d	$	
LR(0) ⇒ I_1	r_5/r_6	r_5/r_6	r_5/r_6	r_5/r_6	r_5/r_6	5RR } Not LR(1)
I_2	"	"	"	"	"	5RR } [10 RR]
SLR(1) ⇒ I_1	r_5/r_6		r_5/r_6			2RR } Not SLR(1)
I_2	r_5/r_6		r_5/r_6			2RR } (4RR)
CLR(1) I_1	r_5		$r_6/$			} NO RR
I_2	r_6		r_5			

I_1 & I_2 states are minimised. Hence CLR ≠ LALR

∴ exists in LALR(1)

	a	b	c	d	$
LALR(1)					
I_{12}	r_5/r_6		r_5/r_6		

2 RR conflict
Not in LALR(1)

Imp
* If CLR(1) don't have RR conflict then LALR(1) may or may not contain RR conflict.
* If CLR(1) don't have SR conflict then LALR(1) also does not have SR conflict.

- If G is CLR(1) but not LALR(1) because RR conflict occured.
- If G is not LALR(1) because of SR conflict then definitely G is not CLR(1) because of SR conflict.

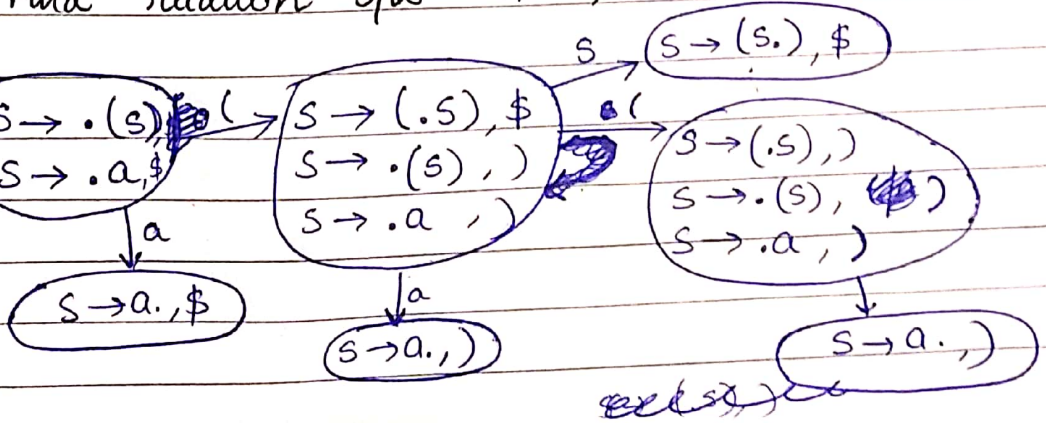
Q Given grammar which does not have RR conflict in any parser. This grammar is CLR(1). What about LALR(1)?

Ans It is also LALR(1) because RR not possible & since it is in CLR(1) ⇒ NO SR conflict

Q. Consider the following grammar :-
$S \rightarrow (S) \mid a$

LR(0) ⇒ no. of states = n1
SLR(1) ⇒ no. of states = n2
LALR(1) ⇒ no. of states = n3
CLR(1) ⇒ no. of states = n4.

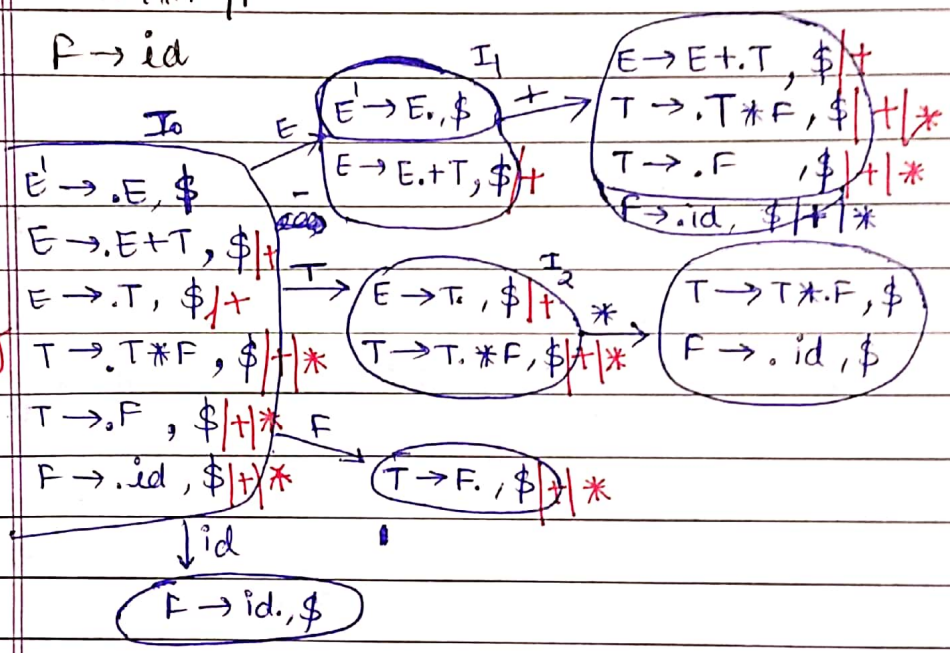
Find relation b/w n1, n2, n3, n4 ?



Date: / /
Page No. 74

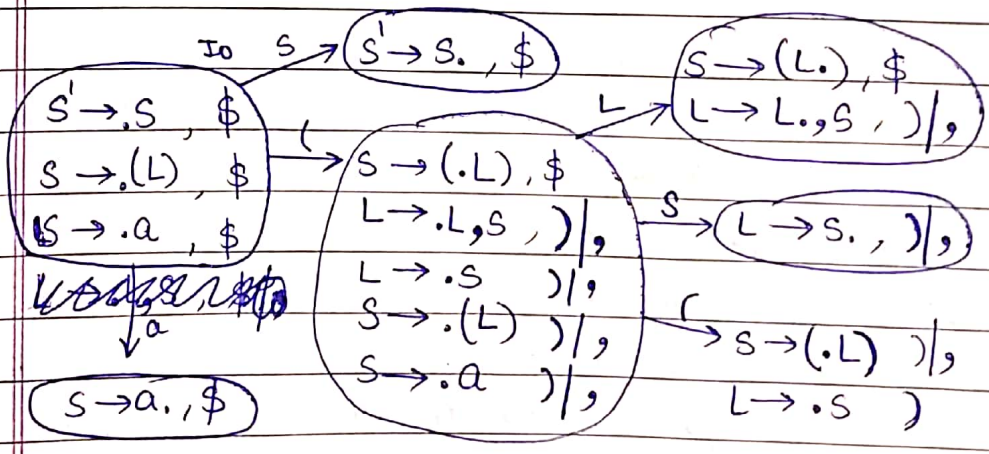
$n_1 = n_2 = n_3 < n_4$

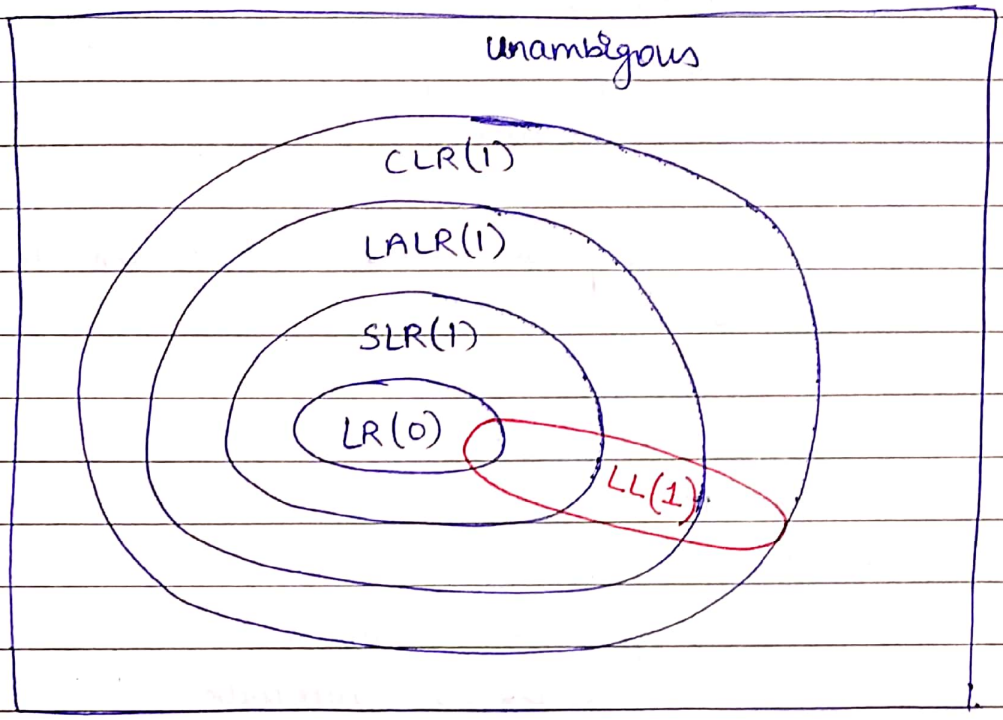
Q $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id$



Q $S \to (L) \mid a$
$L \to L,S \mid S$

Find CLR(1) ~~parsing~~ DFA only 4 - states.





Every LL(1) is LR(1) but every LR(1) need not be LL(1).

$LL(1) \subseteq LR(1)$

$LL(2) \subseteq LR(2)$

$\vdots$

$LL(K) \subseteq LR(K)$

OPERATOR PRECEDENCE PARSER

① It is applicable only for operator grammar.

② A grammar G is said to be operator grammar if & only if:

(i) G does not have null productions

(ii) G does not have 2 adjacent variables on RHS of production

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

$\Rightarrow$ This is operator grammar.
ambigous grammar.

This is the only parser which works on ambigous grammar as well.

eg $E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow id$

$\Rightarrow$ operator grammar
$\Rightarrow$ unambigous grammar.

eg $S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$

unambigous grammar.
but not operator grammar.

eg $S \rightarrow E+E \mid E*E \mid id \mid \epsilon$.
ambigous grammar.
not operator grammar.

Lecture 5

Date: / /
Page No. 77

23/11/2019

SYNTAX DIRECTED TRANSLATION :-

SDT = Grammar + Semantic action

eg1 $S \rightarrow ABC$ { printf ("Hi") } → semantic action

eg2 $E \rightarrow E_1 + E_2$ { if ($E_1.type == E_2.type$) $E.type = E_1.type$ else printy ("Type mismatch") } ← Semantic action for type checking

Parent E getting value by child E_1.
⇒ Synthesised attr.

SDT can be written for any phase.

Attributes (Every variable has its attributes)

Synthesised attribute

$S \rightarrow ABC$ { $S.val = f(A.val, B.val, C.val)$ }

S S.val = A.val + B.val + C.val

A B C
A.val B.val C.val

At any point, value of S is calculated using only its children's attributes.

Inherited attribute

$S \rightarrow ABC$ { $B.val = f(S.val, A.val, C.val)$ }

S

A B C
A.val B.val C.val

B.val = S.val + A.val + C.val

At any point, val of B depends on attributes of parent or/and attributes of sibling.

TYPES OF SDT :-

① S-attributed definition :-

→ If attributes are required then only synthesised attributes are used.

→ It uses bottom up approach because children are req. first to compute parent's attributes. Bottom up also

→ RHS of production at rightmost position

called as post order evaluation

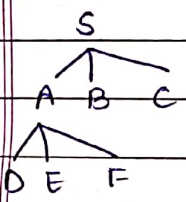
→ Semantic action has to be written at the end of production S → ABC { }.

→ It is more restricted and is subset of L-attributed. Every S is L also.

L - attributed

→ It used both synthesised and inherited attribute but with a condition that only parent & left child's attributes can be used but not right child.

→ It follows depth first order Left to right evaluation.



DFO: SADEFBC

→ Semantic action can be placed anywhere on the RHS of production.

→ It is less restricted and is superset of S-attributed. Every L-attribute is not S-attribute.

APPLICATIONS OF SDT :-

eg1 Construct SDT to execute given arithmetic expression:

i/p: 2+3*4

o/p: 14

For i/p : write grammar.
For o/p : write semantic action.

Date: / /
Page No. 79

Ans ① Write grammar for i/p.

$E \rightarrow E_1 + T$ { E.val = E_1.val + T.val, printf(E.val) }

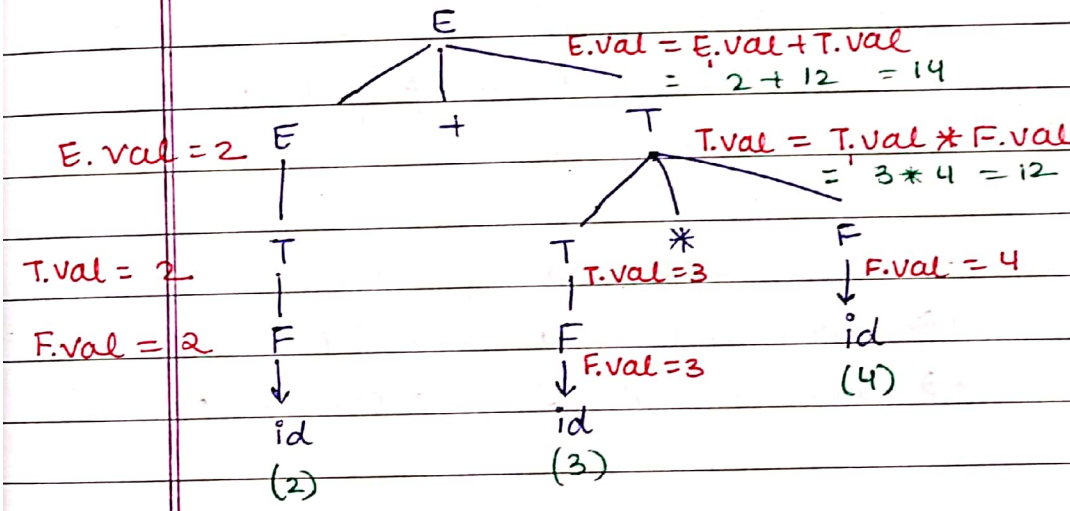
$E \rightarrow T$ { E.val = T.val }

$T \rightarrow T_1 * F$ { T.val = T_1.val * F.val }

$T \rightarrow F$ { F.val = F.val }

$F \rightarrow id$ { F.val = id }

② Write its tree



③ Write semantic action for each production

o/p: screen | 14

This tree is called Annonated Parse Tree or Decorated parse tree. Here at every level, each variable has its attributes mentioned.

Here val is a synthesised attribute.

⇒ S-attributed distribution

⇒ L-attributed distribution

To decide priority of operators :- consider the grammar.

To decide what action has to be performed :- consider the semantic actions

eg
$E \rightarrow E_1 * T$ (E.val = E_1.val + T.val)
$E_1 \rightarrow T$ (E_1.val = T.val)
$T \rightarrow T_1 + F$ (T.val = T_1.val * F.val)
$T_1 \rightarrow F$ (T_1.val = F.val)
$F \rightarrow id$ (F.val = id)

Here * has less priority than +, so + operator executed first. But carefully look at actions. * actually means perform + and + actually means perform *.

For gate, ques → Given SDT & i/p, find o/p
→ Given SDT & o/p, find i/p
→ Given SDT, i/p, o/p, find missing statements

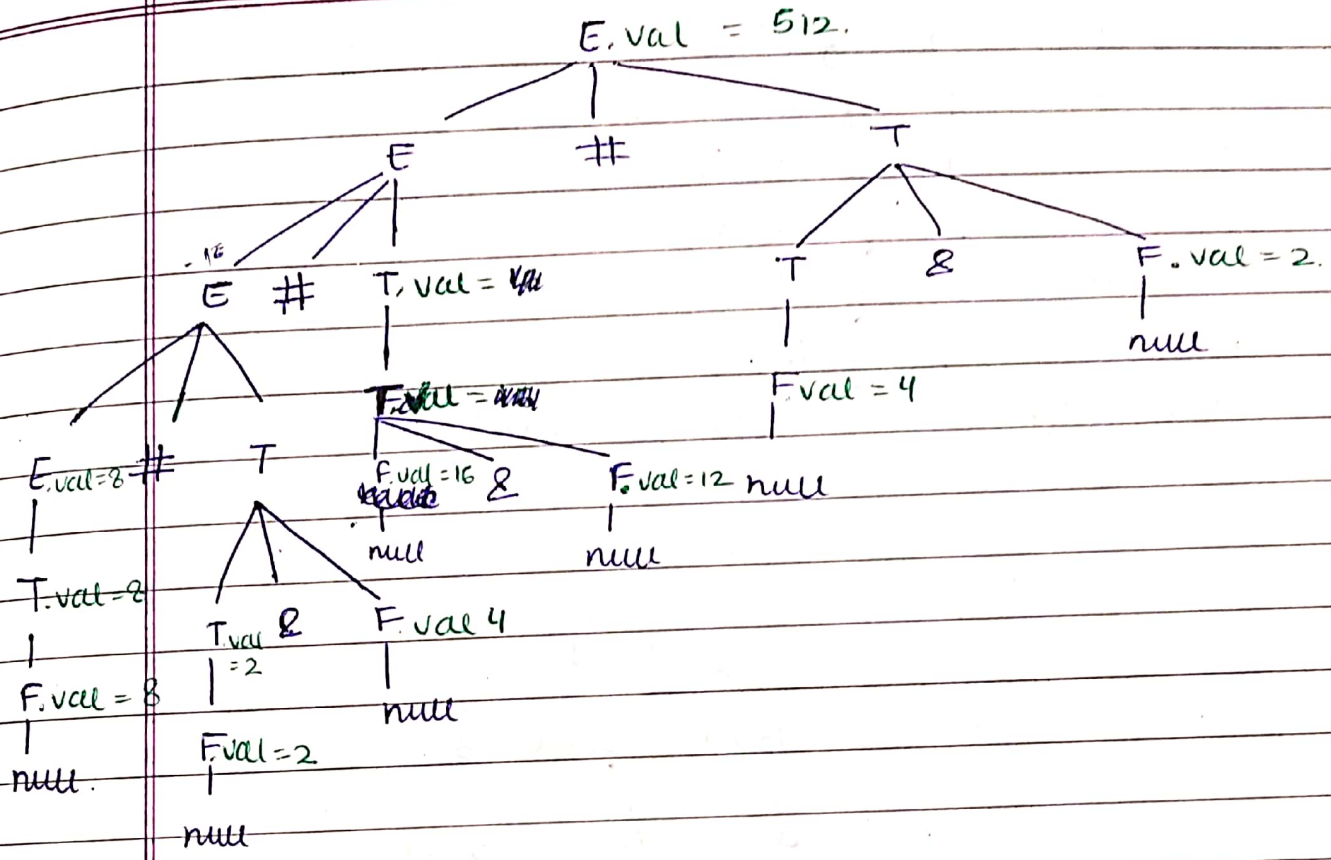
Q Consider the input

$E \rightarrow E \# T$ { E.val = E_1.val * T.val }
| T { E.val = T.val }
$T \rightarrow T \& F$ { _____ }
| F { T.val = F.val }
$F \rightarrow$ null { F.val = num }

i/p :- 8 # 2 & 4 # 16 & 12 # 4 & 2

o/p : 512

(a) T.val = T.val − F.val
(b) T.val = T.val + F.val
(c) T.val = T.val * F.val
(d) None



if & ≡ − ans E.val = −128. No.

if & ≡ + ans E.val = (48×28) * (6) > 512 No.

if & ≡ * ans. > 512.

(d) [But in gate O/P = −128 ∴ ans (a)]

(ii) I/P 10 # 8 & 6 # 9 & 4 # 5 & 2

Here & = −

= *

10 * 8 − 6 * 9 − 4 * 5 − 2

10 * 2 * 5 * 3 = 300

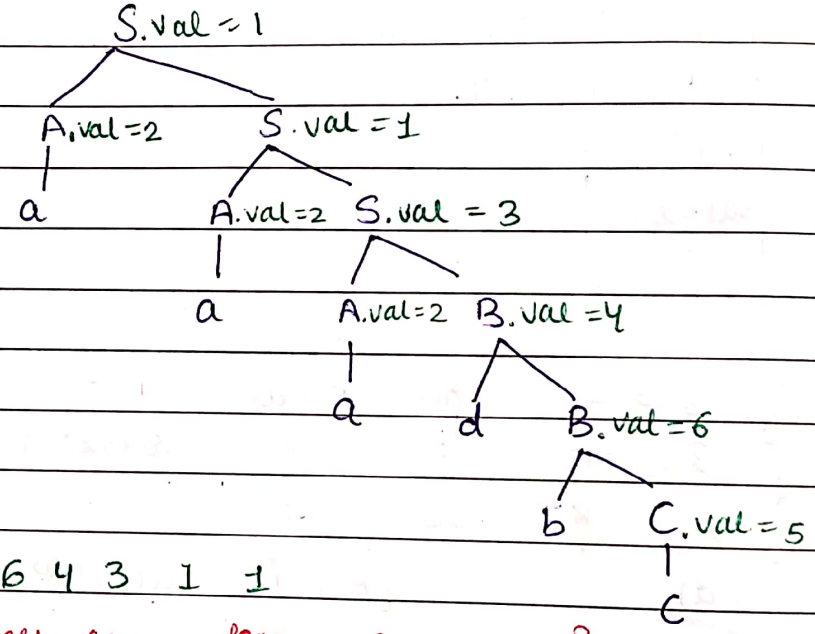
Q consider the following SDT :-

Since the production has action at the end means when S → AS is compiled only then perform action.

Date: / /
Page No. 82

S → AS { printf (1) }
S → AB { printf (3) }
A → a { printf (2) }
B → bC { printf (6) }
B → dB { printf (4) }
C → c { printf (5) }

Find o/p for i/p : aaadbc.



2 2 2 5 6 4 3 1 1

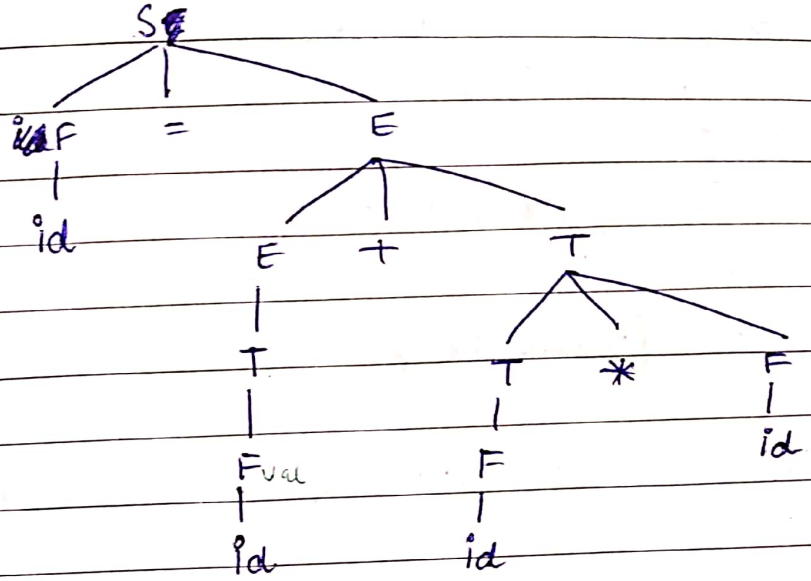
Every parser scans from left to right.

Q Construct SDT to convert the given infix expression to postfix.

i/p :- x = a + b * c

o/p : x a b c + * =

S → F = E
E → E + T
E → T
T → T * F
T → F
F → id



Ans.

$S \rightarrow F = E$ { print (=) }

$E \rightarrow E + T$ { print (+) }

$E \rightarrow T$ { print (" ") } / nothing

$T \rightarrow T * F$ { print (*) }

$T \rightarrow F$ { print (" ") } / nothing

$F \rightarrow id$ {print(id)}

No attribute used ⇒ S-attributed ⇒ L-attributed

Q Construct SDT to convert infix to prefix :-

I/P :- $x = a + b * c$.

O/P :- $= x + a * bc$

$S \rightarrow F = E$

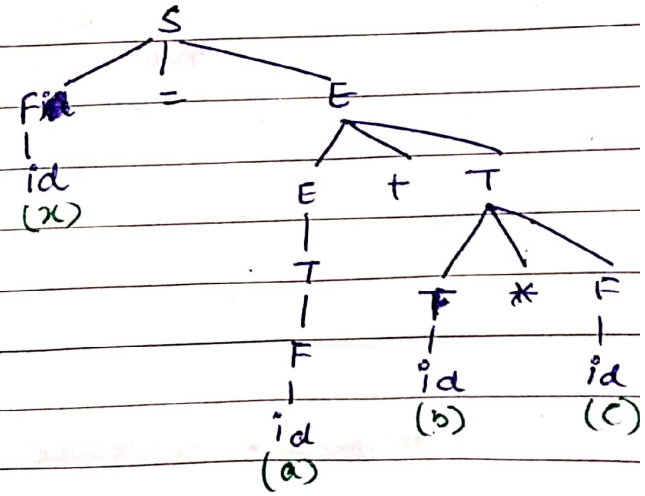
$E \rightarrow E + T$

$\rightarrow T$

$T \rightarrow T * F$

$\rightarrow F$

$F \rightarrow id.$



$S \rightarrow \{printf(=)\}\ F = E$

$E \rightarrow \{printf(+)\}\ E + T$

$E \rightarrow T$

$T \rightarrow \{printf(*)\}\ T * F$

$T \rightarrow F$

$F \rightarrow id\ \{printf(id)\}$

This is L-attributed definition but not S-attributed

Had it been infix to infix then semantic action in between. $S \rightarrow F\{\} = E$ or $S \rightarrow F = \{\} E$

Q Consider the following SDT :-

$S \rightarrow T\ \{pf(+)\}\ R$

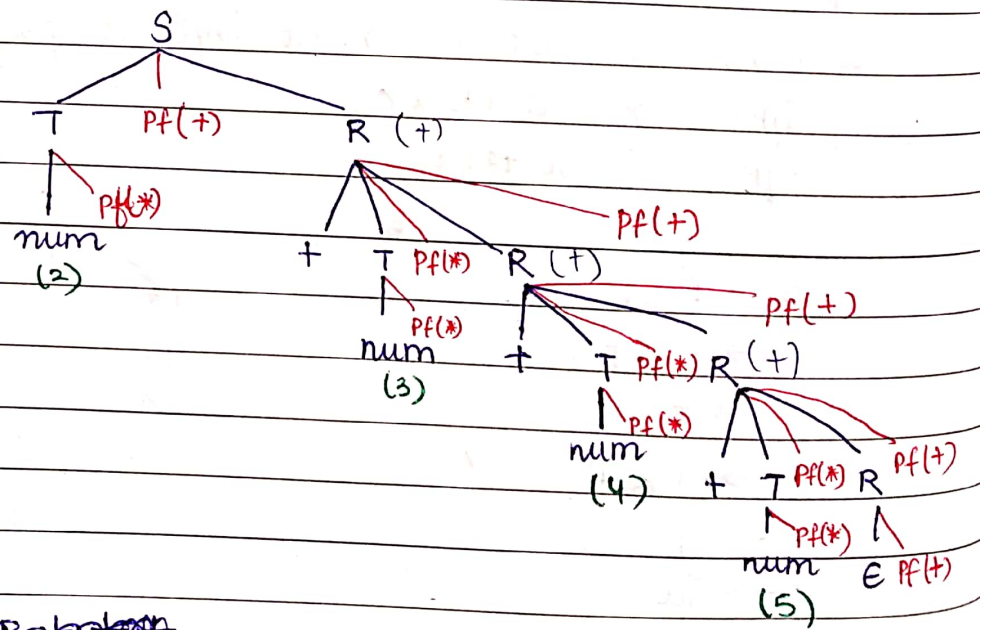
$R \rightarrow +T\ \{pf(*)\}\ R\ \{pf(+)\}$

$R \rightarrow \epsilon\ \{pf(\epsilon)\}$

$T \rightarrow num\ \{pf(*)\}$

i/p : 2 + 3 + 4 + 5

o/p :



* + * * * * * * + + + +

Q Construct SDT to convert binary to decimal

i/p :-	101.111	101
o/p :	85.875	5

0.5
0.25
0.075
1-50

$S \rightarrow L.L \quad \{ S.dv = L_1.dv + \frac{L_2.dv}{2^{L_2.nb}} \}$

$| L \quad \{ S.dv = L.dv \}$

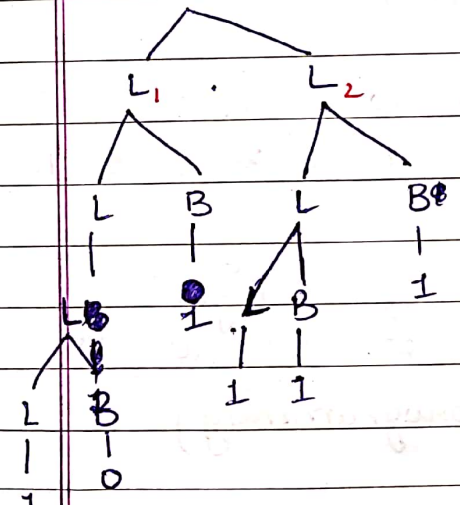
$L \rightarrow LB \quad \{ L.dv = 2L_1.dv + B.dv, \; L.nb = L_1.nb + B.nb \}$

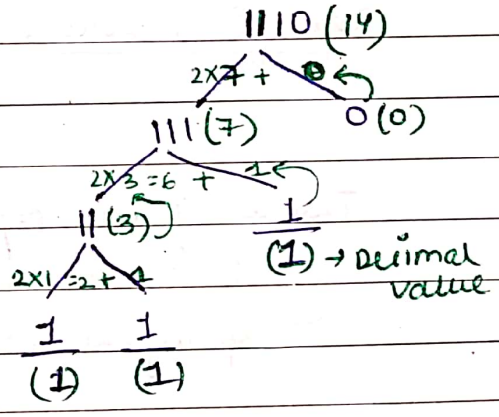
$| B \quad \{ L.dv = B.dv, \; L.nb = B.nb \}$

$B \rightarrow 1 \quad \{ B.dv = 1, \; B.nb = 1 \}$

$| 0 \quad \{ B.dv = 0, \; B.nb = 1 \}$

$S \rightarrow L_1.dv + \frac{L_2.dv}{2^{L_2.nb}}$

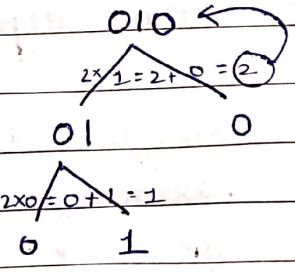




Multiply LHS by 2 & add RHS

To get LHS of decimal

2 X LHS + RHS



To get RHS of decimal

check no. of bits of RHS.

then $\frac{L_2.dv}{2^{L_2.nb}}$

Date: / /
Page No. 85

* + * * * * * * + + + +

Q Construct SDT to convert binary to decimal

i/p :-	101.111	101
o/p :	85.875	5

0.5
0.25
0.875
1.750

$S \to L.L \quad \{ S.dv = L_1.dv + \frac{L_2.dv}{2^{L_2.nb}} \}$

$| L \quad \{ S.dv = L.dv \}$

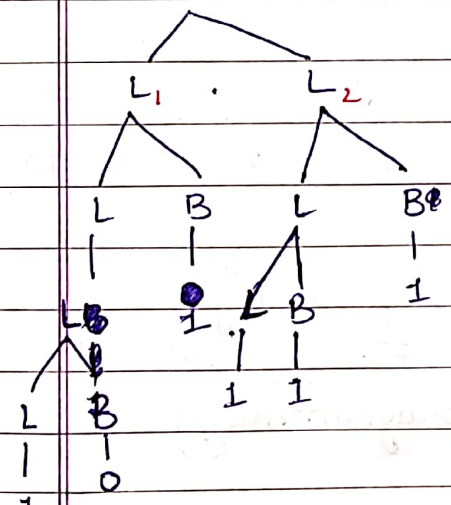
$L \to LB \quad \{ L.dv = 2L_1.dv + B.dv, \; L.nb = L_1.nb + B.nb \}$

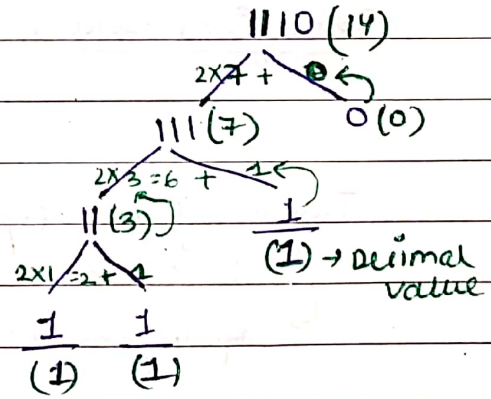
$| B \quad \{ L.dv = B.dv, \; L.nb = B.nb \}$

$B \to 1 \quad \{ B.dv = 1, \; B.nb = 1 \}$

$| 0 \quad \{ B.dv = 0, \; B.nb = 1 \}$

$S \longrightarrow L_1.dv + \frac{L_2.dv}{2^{L_2.nb}}$

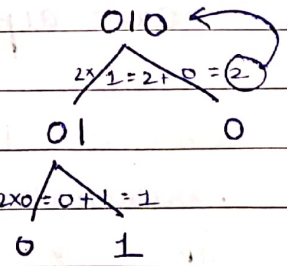




Multiply LHS by 2 & add RHS

To get LHS of decimal

2 X LHS + RHS

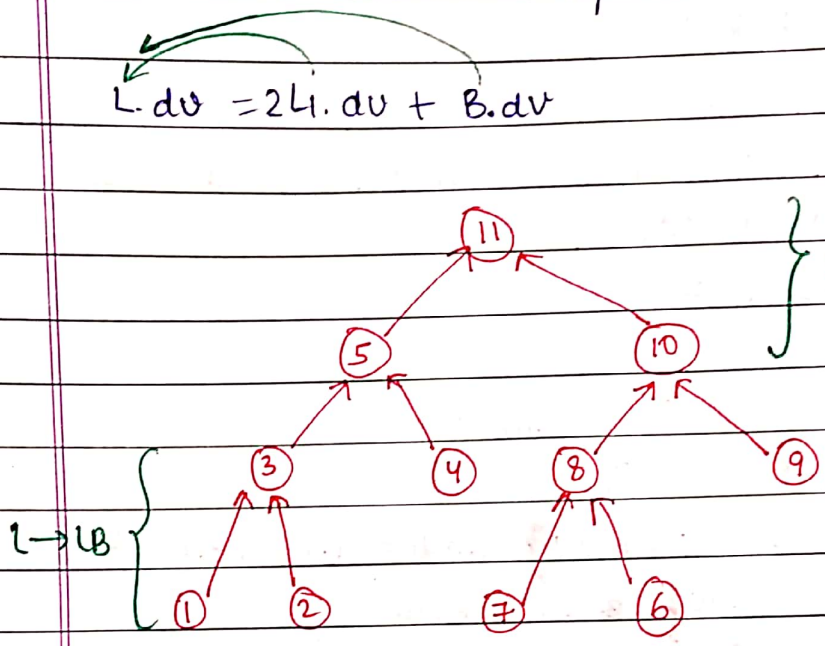


To get RHS of decimal

check no. of bits of RHS.

then $\frac{L_2.dv}{2^{L_2.nb}}$

a = b means a depends on b. $a \leftarrow b$



This graph is called dependency graph (Directed Acyclic graph)

Semantic action gives dependency graph.

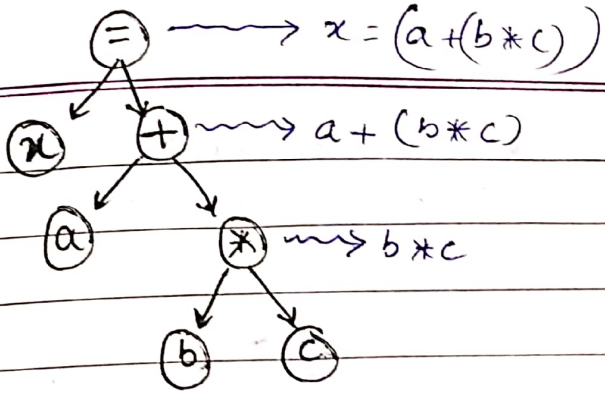
Topological sort :- 9 6 7 8 10 4 1 2 3 5 11
6 9 7 8 4 1 2 3 5 10 11

Try to follow Depth First order left to Right.
1 2 3 4 5 6 7 6 8 9 10 11
↳ Topological sort (close to programming)

NOTE: If ques asked which semantic action done first?
ans Do topological sort close to programming on dependency graph.

Q Construct SDT to create syntax tree for given arithmatic expression :-
i/p :- x = a + b * c
o/p :-

Date: / /
Page No. 87

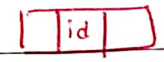


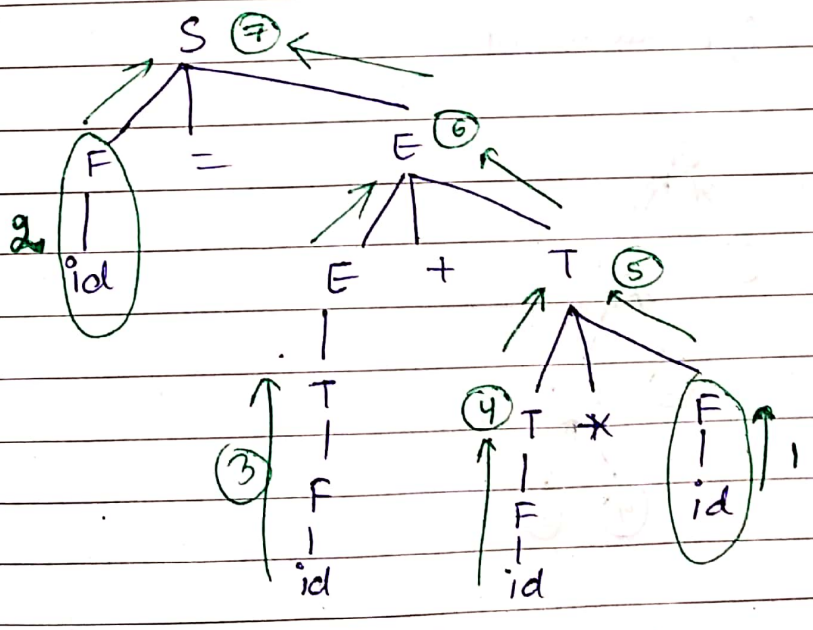
This is a syntax tree

Different b/w syntax tree & parse tree is that in syntax tree, all intermediate nodes are operators & leaf nodes are operands.

Parse tree :- All intermediate nodes are variables & all leaf nodes are terminals.

$S \rightarrow F = E$	{ S.Ptr = mknode (F.ptr, =, E.ptr) } (7) (2) (6) { return (S.Ptr) }
$E \rightarrow E + T$	{ E.ptr = mknode (E_1.ptr, +, T.ptr) } (6) (3) (5)
$\rightarrow \mid T$	{ E. ptr = T. ptr }.
$T \rightarrow T * F$	{ T. ptr = mknode (T_1.ptr, *, F.ptr) } (5) (4) (1)
$\mid F$	{ T. ptr = F. ptr }
$F \rightarrow$ id	{ F.ptr = mknode (Null, id, Null)

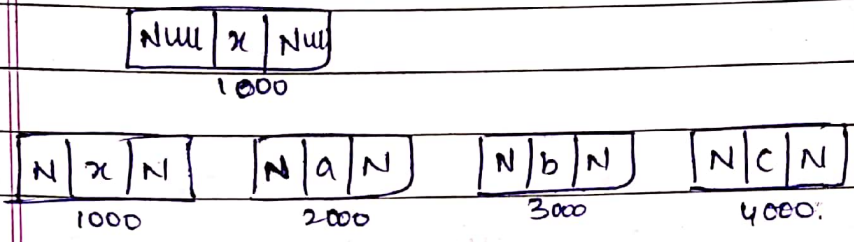




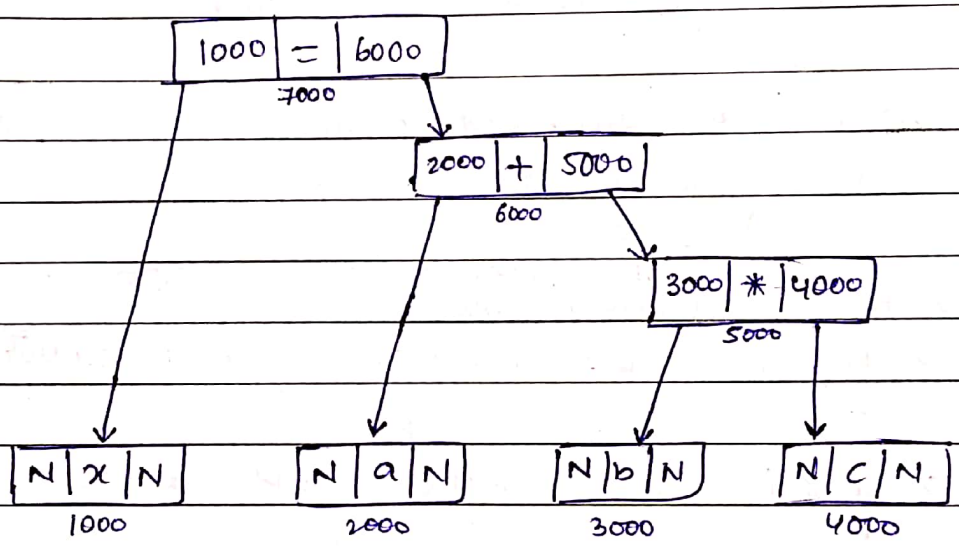
Parse Tree

To convert parse tree to syntax tree:

① create a node for operand only. ⇒ leaf node

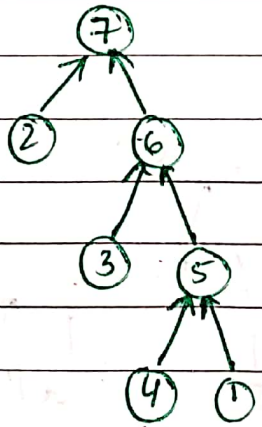


②



ptr is a synthesized attribute

⇒. S, L attributed



Topological sort ② ③ ④ ① ⑤ ⑥ ⑦

Q Construct SDT to generate intermediate code for given arithmetic expression

i/p :- x = a + b * c

o/p :- $t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

(1)

$S \rightarrow F = E$ { gen (F.val = E.val) }

$E \rightarrow E + T$ { E.val = new temp (); gen (E.val = E_1.val + T.val) }

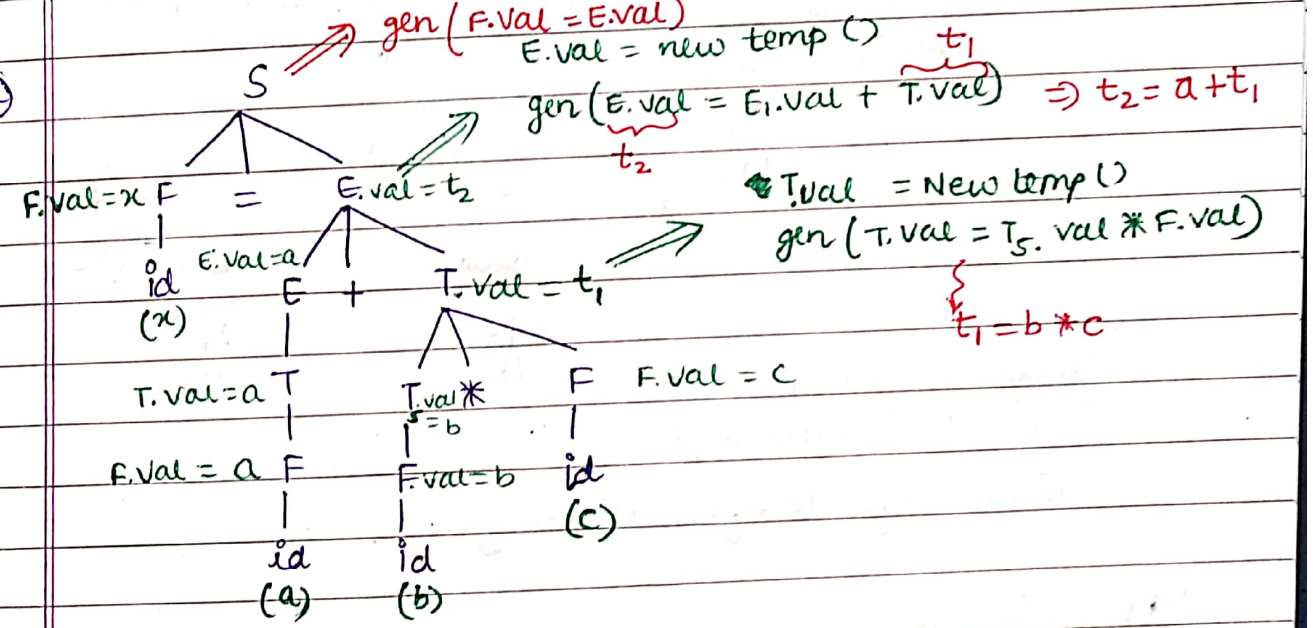
$\quad | T$ { E.val = T.val }

$T \rightarrow T * F$ { T.val = new temp (); gen (T.val = T_1.val * F.val) }

$\quad | F$ { T.val = F.val }

$F \rightarrow id$ { F.val = id }

(2)



(3) gen() function generates o/p code ie. it does not do any calculation. It prints as it is.

new temp() is a function which creates a new variable whenever it is called.

Q Consider the following SDT

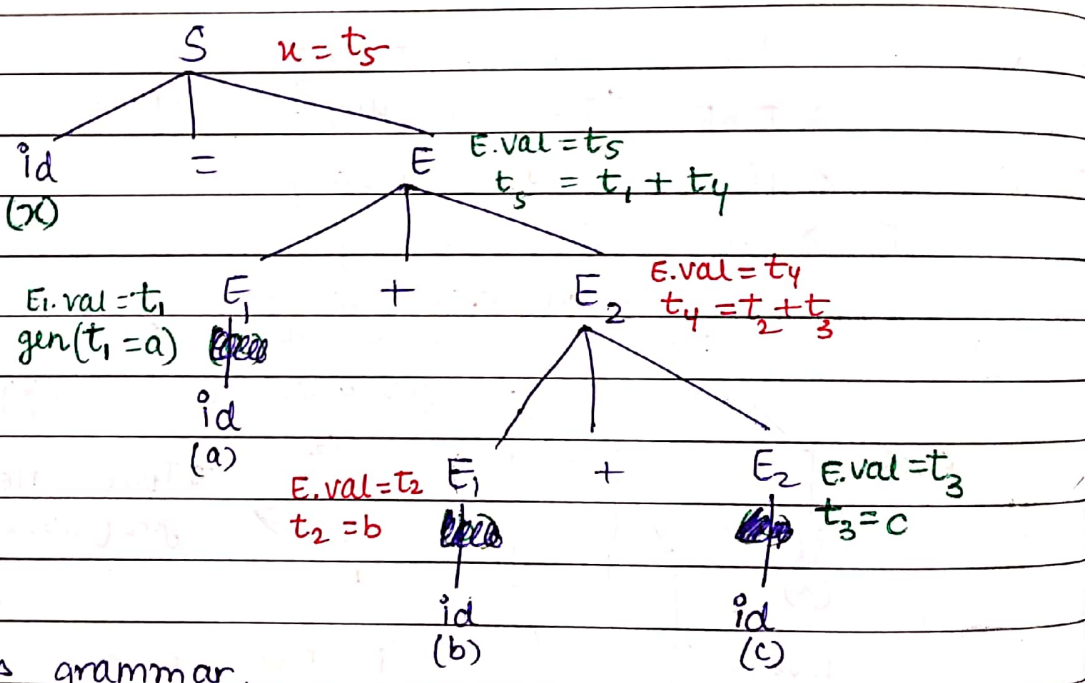
$S \rightarrow id = E$ { gen(id = E.val) }

$E \rightarrow E_1 + E_2$ { E.val = newTemp() ; gen(E.val = E_1.val + E_2.val) }

| id { E.val = New temp() ; gen(E.val = id) }

I/P :- x = a + b + c

O/P :- ?



ambigous grammar.
Cannot be expanded.
∴ Assume right to left associativity

O/P:
$t_1 = a$
$t_2 = b$
$t_3 = c$
$t_4 = t_2 + t_3$
$t_5 = t_1 + t_4$
$x = t_5$

Q Construct SDT to store type information into symbol table.

i/p :- int x, y, z

o/p :-

S.NO	V.Name	V.Type
1	x	int
2	y	int
3	z	int

Symbol table

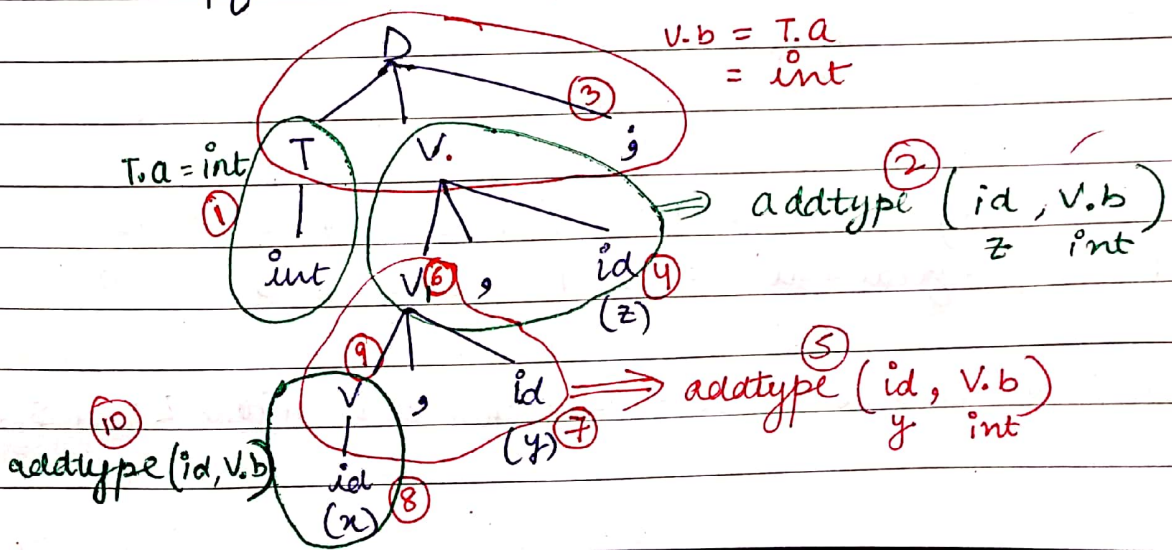
When we have to add an entry to symbol table, call :-
addtype (V.N , V.T)

Now write context free grammar to generate all C declarations.

$D \rightarrow TV;$ { V.b = T.a }

$V \rightarrow V_1, id$ { addtype (id, V.b), $V_1.b = V.b$ }
| id { addtype(id, V.b)}

$T \rightarrow$ int { T.a = int }
| char { T.a = char }
| float { T.a = float }

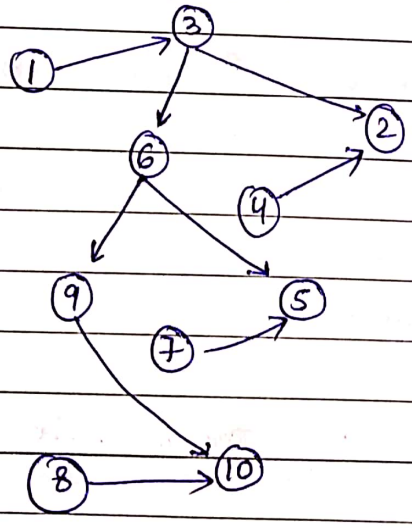


To store x in symbol table, "type" required.
It asks its parents & then to its siblings
∴ we used Inherited attribute.

'a' is synthesised attribute
'b' is inherited attribute
Parent gives variable type
child gives variable name.

(3) based on (1)
(2) " " (3) & (4)
(8) " " (6) & (7)
(9) " " (6)
10 " " (8) & (9)

Dependency graph.

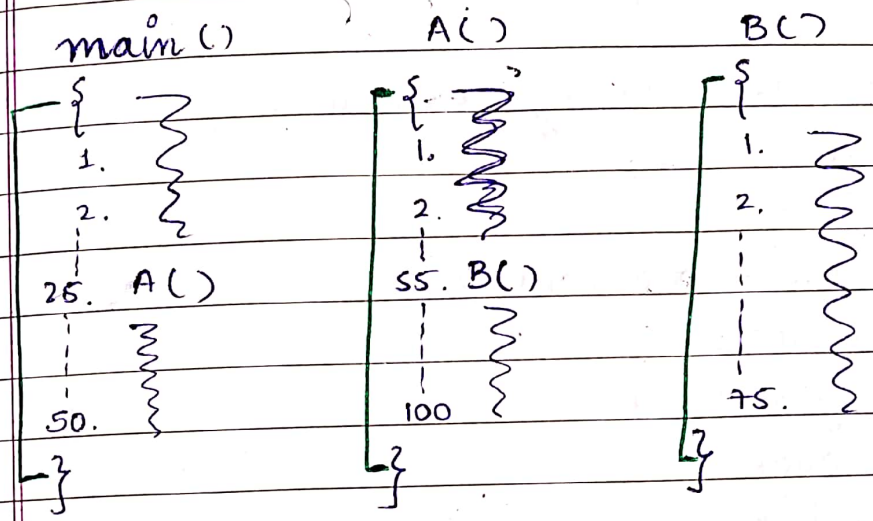


(1) Topological sort :- 1 3 6 4 2 9 7 5 8 10
(2) Topological sort :- 8 7 4 1 3 2 6 5 9 10

NOTE :- If $D \rightarrow TV$ $\{T.a = V.b\}$ then it is nitrous L rose S.

Date: / /
Page No. 93

RUNTIME ENVIRONMENT :-



* 24 lines of code executed in main(). Before going to execute A(), we need to store this work done. This is stored in a stack called ACTIVATION RECORD.
* It contains local variables, temp variables, return addr.
* At line 25, we leave to execute A(). When A is completed at its return statement, we return back to line 25.

main func. activation record holds :-
local variables, temporary variables, return addr., control link (addr of A()), actual parameters A(m,n), system status (RAM, harddisk size, processor etc.), access link (from which func. you access the code. eg for A(), access link is main())

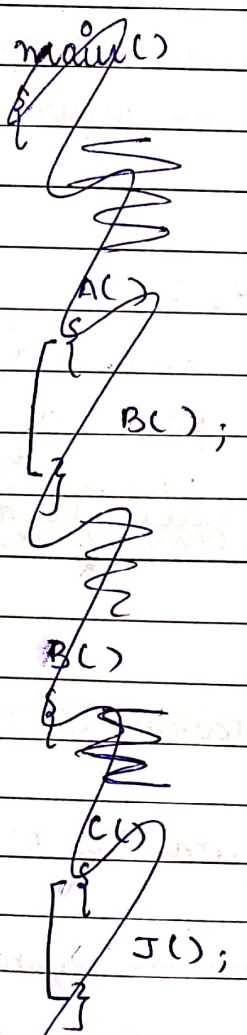
Creating main() activation record as soon as main() starts executing is better.
Main activation record initially contains Null.

Preparing activation record. is worst when no func. is called from A().

(1) Whenever A() is called, its activation record will be created & pushed inside stack

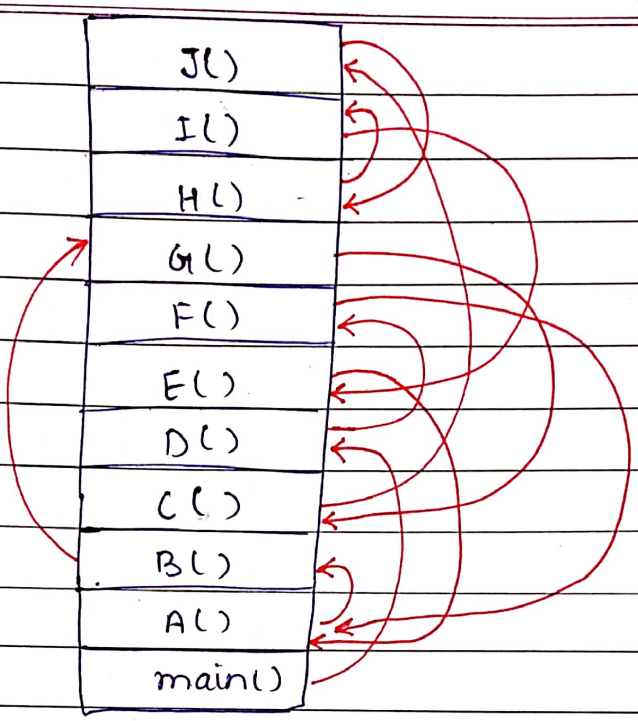
(2) Whenever A() is running, activation record gets updated

(3) A() activation record contain :-
(i) local variables
(ii) temporary variables
(iii) return addr.
(iv) control link
(v) access link
(vi) actual parameters.
(vii) machine states



```
main()
{
    A()
    {
        B();
    }
    B()
    {
        C()
        {
            J();
        }
        G();
        D()
        {
            F();
        }
    }
    E()
    {
        F()
        {
            A();
        }
    }
    G()
    {
        {
            C();
        }
    }
    A();
} // E() ended.
H()
{
    I()
    {
        E();
        J()
        {
            {
                H();
            }
        }
        I();
    }
    O();
} // main over.
```

Date: / /
Page No. 96



main control link = D()
A() " " = B()
B() " " = G()
C() " " = J()
D() " " = F()
E() " " = A()
F() " " = A()
G() " " = C()
H() " " = I()
I() " " = E()
J() " " = H()

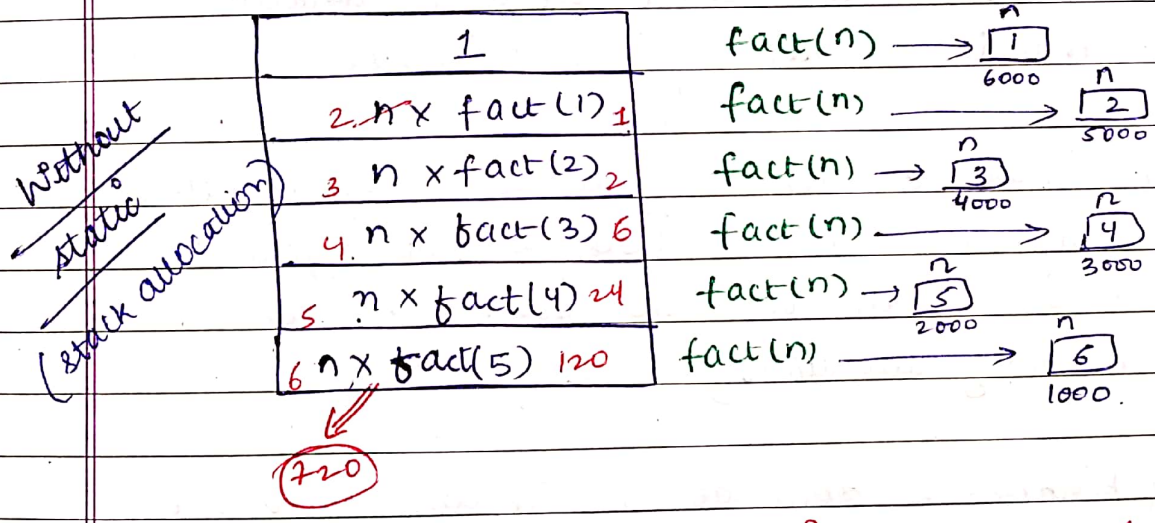
A() access link = main()
B() access link = main()
C() access link = B()
D() " " = B()
E() " " = main()
F() " " = E()
G() " " = E()
H() " " = main()
I() " " = H()
J() " " = I()

A() function's code is present in which function?

STORAGE ALLOCATION TECHNIQUES :-

1. static storage allocation
2. stack storage allocation
3. Heap storage allocation.

```
fact(n)
{
    if (n ≤ 1)
        return n;
    else
        return (n * fact(n-1))
}
```



Here memory created 6 times because 6 func. calls (int n) declaration.

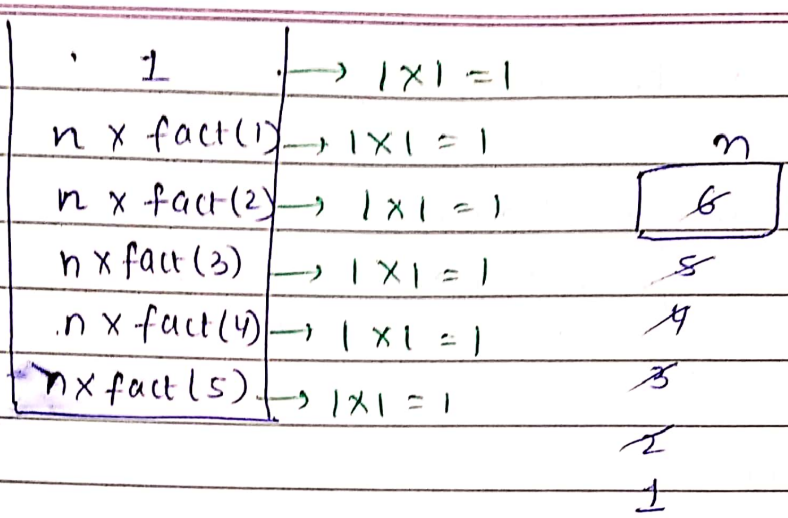
STATIC STORAGE ALLOCATION :-

But if fact(static int n) is used then mem for n is created only once.

Here we end up saving space but answer may be wrong as if recursion is no longer occuring

Date: / /
Page No. 98

With static



(1) for static variable, memory will allocate in static area. and memory is allocated only once. For static variable, it occurs only once at compile time.

(2) The drawback with static storage allocation is that it does not support recursion.

(3) One more drawback of static storage allocation is dynamic data structure is not supported. (i.e. in between user has no control over memory allocation)

(4) If binding is done at compile time, it cannot be changed at run-time.

STACK STORAGE ALLOCATION :-

(1) When a function is called, its activation record is created and is pushed inside the stack.

(2) Whenever function completes its execution, it is popped out of the stack.

(3) It supports recursion

(4) Drawback with stack storage allocation is that dynamic data structure is not supported. Ie. (whenever you want, you cannot insert/delete). Func. call → Push } it does not ask user. So not
Func. over → Pop } dynamic.

(5) If a function is over, it is popped. If this func. is required again, it has to be pushed & computed again. This problem is solved by dynamic programming where we store the result somewhere.

(6) In stack storage allocation, always local variables belong to new activation record only. This means if a stack contains 6 elements then only TOS is printed.

HEAP STORAGE ALLOCATION :-

(1) Memory allocation & deallocation can be done anytime based on user requirement.
C ~~calloc~~ malloc - allocation , free - deallocation
C++ new - allocation , delete - deallocation
Java new - allocation ; deallocation is automatic

(2) It support dynamic data structure

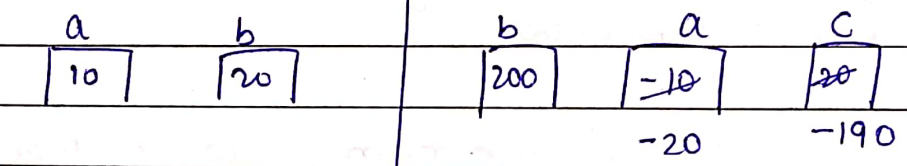
(3) It can implement recursion.

Date: / /
Page No. 100

```
main ()
{
    int a = 10 , b = 20 ;
    f (a*b , a-b , b);
    printf (a,b);
}
f (int b, int a, int c)
{
    c + = a - b ;
    a - = b + c ;
}
```

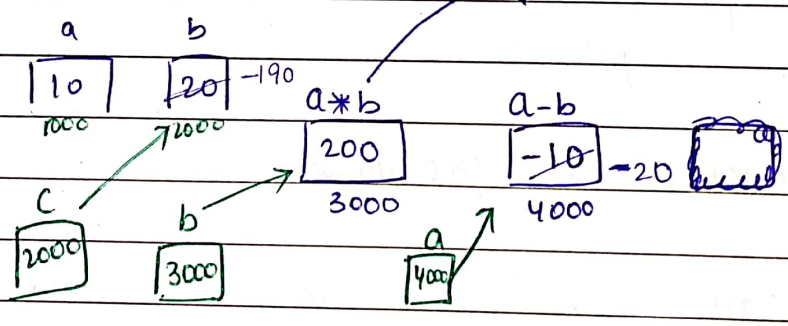
ans

By value.



O/P : 10 , 20.

By ref :-



O/P : 10 , -190.

Lecture 6

Date: / /
Page No. 101

24/11/2019

WORKBOOK

pg 57 Q2 A-3, C-4, B-2, D-1 (b).

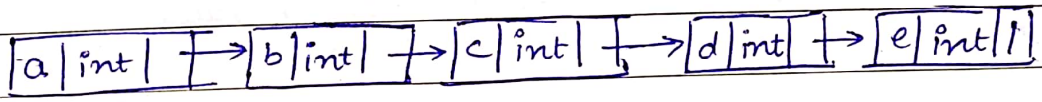
Q3 (a)

Q4 int a, b, c
float d, e

S.N	V.N	V.T
1	a	int
2	b	int
3	c	int
4	d	float
5	e	float

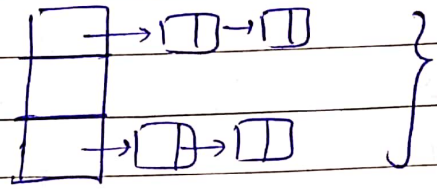
Both array & linked list can be used to implement symbol table.

Or.



But creation of symbol table using LL is better than array because in array we need to fix the size at the start when we don't know how many variables will be used in program.

But also LL can be modified by using chaining



} So it is easy to retrieve a symbol.

∴ ans: (c). Had he asked most efficient ⇒ (d)

Q5 (d)
Q6 (a)
Q7 (c)
Q8 (c)
Q9 (a)
Q10 (d).
Q11 (c)
Q12 (c).
Q13 (d).
Q14 (c).
Q15 (b)
Q16 (c)
Q17 (d).
Q18 (c).
Q19 (c)
Q20 (d)
Q21 (d)

Date: / /
Page No.

21 $\underbrace{L(L+D)^*}$ ⇒ identifier
regular exp or pattern matching
abc ⇒ lexeme.

22 S_1 ✓ (a)
S_2 ✓

23 (b).

24 (b). while a( );
while (a, b);
Note x = a, b ≡ It will take a value
x = (a, b) ≡ It will take b value.

25 (C). Here we need to check i/p – layer.
Syntax tree is o/p of syntax analyser but i/p to semantic analyser.

26 line 4 : ; missing x = 1 (i)
integer & variable cannot be side by side.
(b).

27 (d). 31 40
28 58 32 34
29 35 33 0
30 22. 34. (a).

CH-2

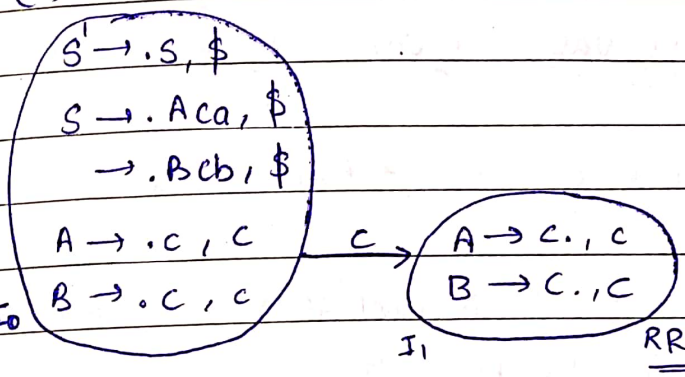
① (d). There maybe ambiguity
② (d).
③ (a)
④ (a) Both ambigous & left recursion but ambiguity is a larger problem.

Date: / /
Page No.

5 (d) Construct DFA

6 (a).

7



when grammar given is big then expand that i/p first which gives immediate reduction as reduction is req. for any kind of conflict (SR, RR).

LL(1) X LL(2) X

AC → cc
Bc → cc

(d)

8 ~~(a) (b) (c)~~ a ✓ b ✓ c ✗ d ✓ } (c).

9 (b).

10 (d).

11 (b)

12 a → ✓ b → ✓ C → X d → ✓ } ans (c) ✓

13 (b) Parser detects only syntax errors.

14 (c).

15 Shift reduce → LR parser. (c).

16 (c) (19) (a)

17 (b) (20) (a)

18 (c) (21). (b)

Date: / /
Page No.

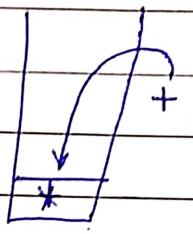
22 $E \rightarrow num.$ $\{E.val = number.val\}$

$| E+E$ $\{E_1.val = E_2.val + E_3.val\}$

$| E*E$ $\{E_1.val = E_2.val * E_3.val\}$

ambigous grammar as + & * have same priority

i/p: 3*2+1 result is ambigous as + & * have same priority.



We don't know whether to push or to pop.

This is called Shift Reduce Conflict

construct LALR(1) parsing table.

To this we pass this to YACC [Yet Another Compiler Compiler] tool.

Since this is ambigous grammar, diffinitely some conflict will come in LR parser table. But when YACC is used it convert SR conflict to a shift opration. It gives priority to shift opration

In YACC if $R_5 R_6$ conflict occurs then it gives priority to the reduction whose productions comes first in grammar.

ans (C). (b) also correct but it does not tell how YACC resolves.

=> Stronger ans. is (C).

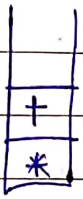
Date: / /
Page No.

23. $3 + 2 * 1$.

$3 + (2 * 1)$

(5)

$3 * 2 + 1$



$3 * (2 + 1)$

(9)

* Pushed by YACC becoz. when SR conflict occurs, priority is given to S.

⟹ whichever comes late is evaluated first as priority is equal. (Right to left asso.)

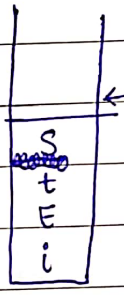
ans (b)

24. (b)

25. (d) Handle definition

26. (b). CLR(1) should also have SR conf.

S → i E t S | i E t S e S



Here already handle is present at TOS. We are in dilemma whether to pop & get iES or push & wait for iEtSeS.

We may push or pop ⇒ SR conflict

Since type of parser is not mentioned, we go with general ans: ⇒ (c).

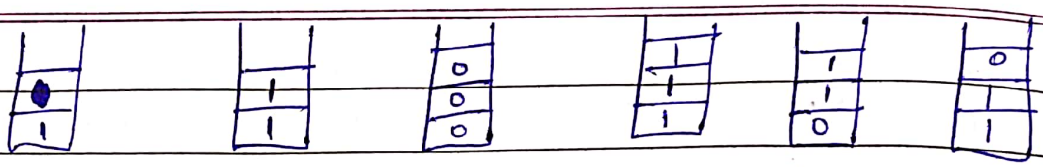
27. (c).

29. Handles :- XX, 0X, 1.

LR(1) item :- [X → 0.X, 0]

No use of lookahead symbol as it is not completed.

Date: / /
Page No.



viable prefix ✓ | Not a viable prefix ✗ | viable prefix ✓ | No. | No, 2 1's not possible | 2 1's can't be together

∴ ans (a)

When i/p is 0. & production $X \to 1., 0$

This means if 0 comes then don't push. Simply reduce

30 (a).

31 (C).

32

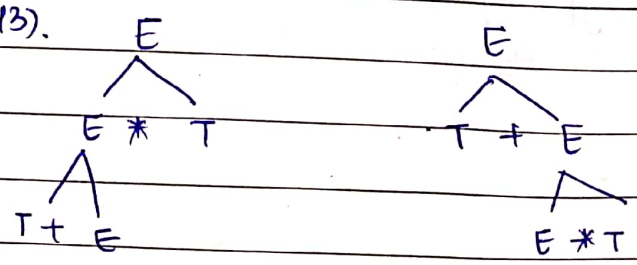
$S' \to .S, \$$
$S \to .X1, \$$
$X \to .A1B, 1$
$\quad .2, 1$
$A \to .2, 1$

2 →

$X \to 2., 1$
$A \to 2., 1$

Not CLR(1)

33	(C)	39	(C).
34	(a)	(40)	(C)
35	(b)	(41)	(a) Non empty ⇒ non ϵ
36	(C)	(42)	(C).
37	(a)	(43).	
38	(c)		



ambigous.

(C)

Date: / /
Page No.

44 $F \to idF'$ } option(a) ✗ (c) ✗
$F' \to \epsilon \mid +2F'$ } (b) ✗

⇓ after removing null production

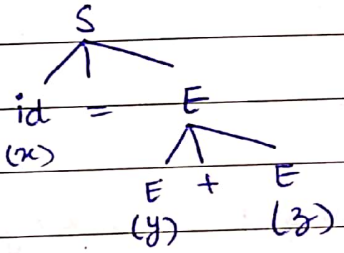
$F \to idF' \mid id$ } matches option(a) now
$F' \to +2F' \mid +2$

(a).

45 For ( ) eras ; inside and not ,
∴ (b).

46 b has higher priority than a ans (b).

47



(b). $t_1 = y + z$
$x = t_1$

48 (c).

49 (a).

50 SR parser ≡ LR parser ≡ Bottom up parser.

(a).

51 Every reg. set has atleast 1 grammar. can be satisfied by LR(1).
Reg. set ≡ Reg. lang.

Every Reg. lang is LR(1).
Every reg. lang is unambigous.
But reg. grammar can be ambigous.
P – False. some reg. grammar can be ambigous.
Q – True

(c)

(52) (c)

Date: / /
Page No.

(53) (a)
(54) (a)
(55) (d).
(56) (d).
(57) (c)
(58) (a)
(59) (a)
(60) (d).
(61) (c)
(62) (b)

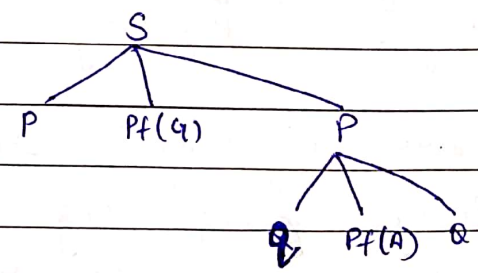
(63). (b).
(64) (d).
(65) (b).
(66). (c).
(67). Dynamic type checking is done at run time. Type checking at compile time is lesser than that of runtime.

```
a = 20
b = 30.
if (a<b)
{
}
else {
}
```

During runtime type checking it won't go in else part. But there may be error in else part.
∴ Dynamic TC won't find all errors.
∴ ans (c).

68 (c)
69 (a)
70 (c)
71 (c)
72
73
74
75
76
77
78
79
80
81
82 (d)
83 (b)
84
85 No option matches.
86 all are false. (d)
87 (d)

85. PQSQSR.



Q AAAAT

Date: / /
Page No.

83 $A \to SB \mid S$
$B \to ,SB \mid ;S$
$S \to a$

replace S by a

$A \to aB \mid a$
$B \to ,aB \mid ;a$

(c) is not an operator grammar.
b/w (a) & (b) check which is equivalent to given grammar.
ans (b).

T7 Left sentential form ≡ Leftmost Derivation Tree.

A B C D E F
↓
First(D) = d.

Follow(C) in LMD = d.
Follow(C) in RMD = d

Follow depends on what comes next. It does not depend upon LMD or RMD.
(C).

T9 Handles : aS, b
in stack we can have, aS, b, a
aaS ✓
a ✓
aa ✓
} Possible in stack.
bb ✗

Both (a) & (d) lead to reduction but bb cannot occur in stack. ∴ ans. (a)

T8 (a)

$S \to AB$
$A \to a$
$B \to b$
unambiguous as only 1 str. possible (ab)

LMD	RMD
S	S
AB	AB
aB	Ab.
ab	ab

Not same.

Date: / /
Page No.

Derivation Trees are same but derivations are not same.

CH-3

① (b)

② statically typed ≡ only 1 type lang.
untyped lang ≡ No type
(c)

③ ~~I~~ II X III X True when few L-attributes
ans (b).

④ 1 X 3 X (a) ✓

⑤ 1 ✓
2 ✓
3 ✓
} (d)

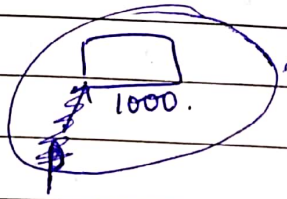
⑥ (c)

⑦ (d) Prog. counter ≅ return addr.
reg. values ≅ local variables

⑧ (a)

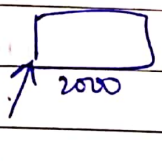
⑨ (c)

⑩ P = malloc



garbage memory

P = malloc



Garbage : allocated memory which cannot be reached.
(b)

Date: / /
Page No.

CH-4 INTERMEDIATE CODE GENERATOR

ex $x = a + b * c$

$\Downarrow$

$t_1 = b * c;$

$t_2 = a + t_1;$

$x = t_2;$

ex $x = (a+b) * (a-b) + c$

advantage with intermediate code generation is will get machine independent code.

REPRESENTATION OF IC:-

LINEAR CODE.

eg: solved 3 address code (max 3 addr.)

$t_1 = a + b$

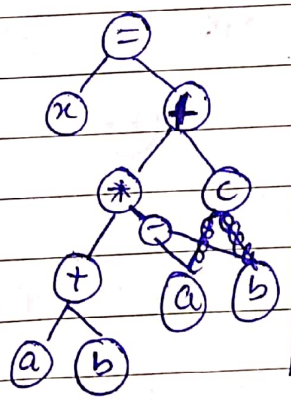
Postfix notation

$x\,ab+\,ab-*\,c+=$

$t_1\ t_2\ t_3\ t_4$

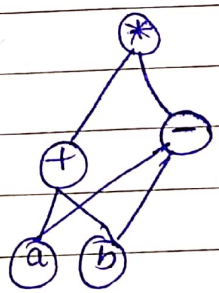
TREE FORM

Syntax Tree



DAG

common sub expression elimination



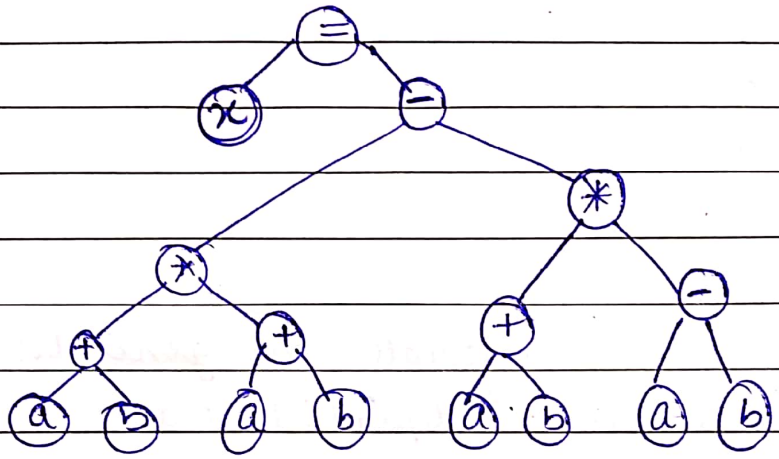
ex3 $x = ((a+b) * (a+b)) - ((a+b) * (a-b))$

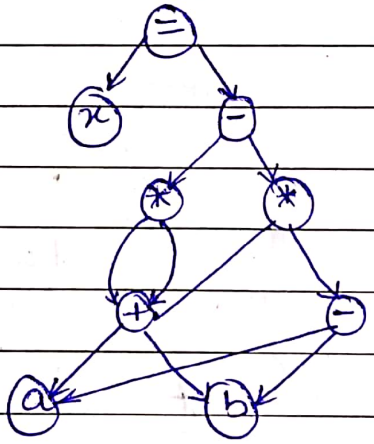
$t_1 = a + b$

$t_1 = a+b$	$t_2 = a+b$
$t_2 = t_1 * t_1$	$t_3 = t_1 * t_2$
$t_3 = a-b$	$t_4 = a+b$
$t_4 = t_1 * t_3$	$t_5 = a-b$
$t_5 = t_2 - t_4$	$t_6 = t_4 * t_5$
$x = t_5$	$t_7 = t_3 - t_6$
	$x = t_7$

Date: / /
Page No.

x ab+ ab+ * ab+ ab- * - = postfix Notation

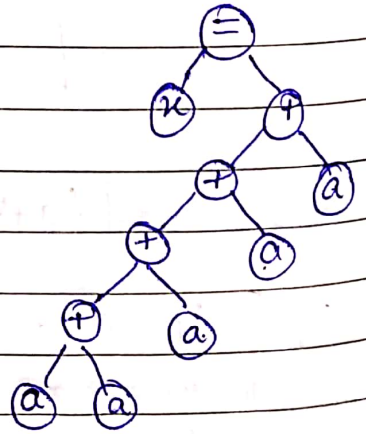




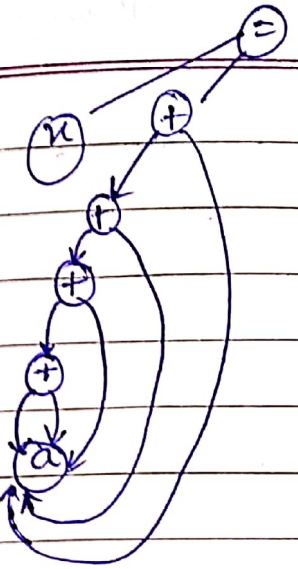
Q $x = a+a+a+a+a$

$t_1 = a+a$
$t_2 = t_1+a$
$t_3 = t_2+a$
$t_4 = t_3+a$
$x = t_4$

aa+a+a+a+



Date: / /
Page No.



THREE ADDRESS CODE :-

1. $x = y \ op. \ z$
2. $x = op. \ y.$
3. $x = y$
4. $x = a[i]$
5. $a[i] = x$
6. $x = a[i][j]$ No. three are 4 addr.
6. $x = f(a)$
7. if $(x < y)$ goto z
8. goto x.

Q Write Three Address code for following C program.

```
i = 9;
for(i = 25; i <= 93; i++)
{
    x = a + b * c;
}
```

Date: / /
Page No.

1000	i = 9	Block 1.
1001	i = 25	
1002	if i ≤ 93 goto 1004	B2
1003	goto 1009	B3.
1004	$t_1 = b * c$	B4
1005	$t_2 = a + t_1$	
1006	$x = t_2$	
1007	i = i + 1	
1008	goto 1002	
1009		B5

→ This shows for every recursion, non recursion is possible.

→ Inside comp., everything is ICG i.e. further changed to target code.

→ Filling addresses which are not available initially after completion of intermediate code. This process is called Backpatching.

How to find leaders:
- 1st statement is a leader.
- Target of goto is a leader.
- Next statement after goto is leader.

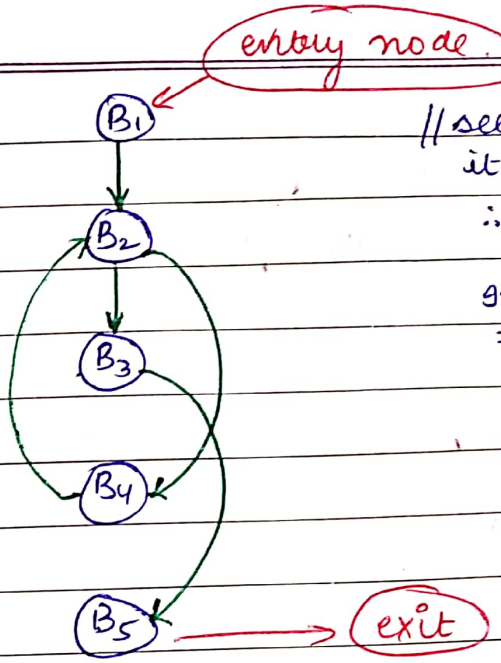
Leaders :- 1000, 1002, 1003, 1004, 1009.

One leader to next leader is one basic block.

No. of Basic Blocks = No. of leaders

Def. of Basic block :-
It is a set of 3 addr. statements where control enter at the beginning & leaves at the end. in between there is no halt or jump.

Date: / /
Page No.



// seeing last line of B1 (i=25) it is neither goto nor JMP.
∴ Goto B2.

on B_2, if cond^n present.
⇒ conditional jump.
∴ 2 edges will come i.e. for pass & fail case.

CONTROL FLOW GRAPH. (CFG)

No. of nodes in CFG = 7.
No. of edges in CFG = 7

DFT :- B_1 B_2 B_4 B_2
↑ again same. ⇒ cycle present

∴ Apply DFT on CFG to detect cycle or loop.

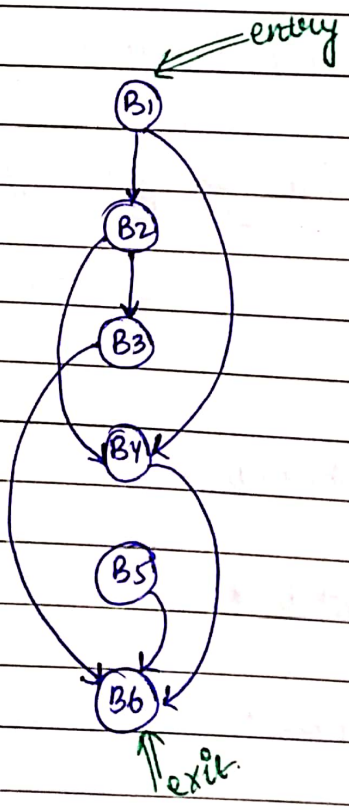
Q Write three address code for following C-program.

```
i = 93;
switch (i)
{
    case 1 : x1 = a1 + b1 * c1;
             break;
    case 90 : x2 = a2 + b2 *
             break;
    default : x3 = a3 + b3 * c3
}
```

Date: / /
Page No.

L1	1000	$i = 93$	B1
	1001	if $i == 1$ goto 1007	
L2	1002	if $i == 90$ goto 1011	B2
L3	1003	$t_5 = b_3 * c_3$	
	1004	$t_6 = a_3 + t_5$	B3
	1005	$x_3 = t_6$	
	1006	goto 1015	
L4	1007	$t_1 = b_1 * c$	
	1008	$t_2 = a_1 + t_1$	B4
	1009	$x_1 = t_2$	
	1010	goto 1015	
L5	1011	$t_3 = b_2 * c_2$	
	1012	$t_4 = a_2 + t_3$	B5
	1013	$x_2 = t_4$	
	1014	goto 1015	
L6	1015	HALT	B6

There are 6 leaders



Date: / /
Page No.

Q Write TAC for following statement in C language :-

$x = a[i][j]$ $\quad a[1....90, 1.....75]$

c [size of element]

row major order.

$$\text{LOC}(a[i][j]) = a + [(i-1) * 75 + (j-1)] \times c$$

1000	$t_1 = i - 1$
1001	$t_2 = t_1 * 75$
1002	$t_3 = j - 1$
1003	$t_4 = t_2 + t_3$
1004	$t_5 = t_4 \times c$
1005	$t_6 = a + t_5$
1006	$x = *t_6$ → unary operator. [$x = OP\ y$]

REPRESENTATION OF TAC :-

1. Quadruple
2. triple
3. Indirect tripple.

ex = $x = a + b * c - d / e + f$

$t_1 = b * c$

$t_2 = d / e$

$t_3 = a + t_1$

$t_4 = t_3 - t_2$

$t_5 = t_4 + f$

$x = t_5$

Date: / /
Page No.

Quadruple

	Operator	operand 1	operand 2	Result
1	*	b	c	t_1
2	/	d	e	t_2
3	+	a	t_1	t_3
4	−	t_3	t_2	t_4
5	+	t_4	f	t_5
6	=	t_5		x

Triple

S.No.	Operator.	op 1	op.2.
1	*	b	c
2	/	d	e
3	+	a	(1)
4	−	(3)	(2)
5	+	(4)	f
6	=	x	(5)

→ Result is stored on index itself

Saved space but we cannot move result wherever we want.

Indirect triple

S.No.	
1	
2.	
3.	⇒ 5000
4.	
5.	
6.	

Here we overcome disadv. of triple by shifting the result from index to a particular address.

Date: / /
Page No.

STATIC SINGLE ASSIGNMENT.

It is 3 addr. code where each address has single assignment.

$t_1 = z * a$
$t_2 = y + t_1$
$x = t_2$

} Every variable has single meaning. ⇒ SSA ✓

$a = z * a$
$z = y + a$

} Here 1 variable (a) is having 2 meanings. ⇒ NOT SSA.

$z = z * a$
$z = y + z$

} Here z has 2 meaning ⇒ SSA X

$t_1 = z * a$
$x = y + t_1$

} Every variable has just 1 meaning. ⇒ SSA ✓

Q $x = a + b * a + b$

$c = b * a$	$x = b * a$	We cannot store b*a into a or b because they have to be used afterwards. a & b are live variables so they cannot be replaced.
$a = a + c$	$a = a + x$	
$x = a + b$	$x = a + b$	
	3 var.	

SSA X ⇐

Total 4 variables
Total 1 temp. variable.

$c = b * a$
$d = a + c$
$x = d + b$

Total variables = 5
Total temp. variables = 2

Date: / /
Page No.

$x = (a*b) + (a*b) + (a*b)$

TAC.	SSA
$a = a*b$	$t_1 = a*b$
$b = a+a$	$t_2 = t_1 + t_1$
$x = b+a$	$x = t_2 + t_1$
Total var = 3	Total var = 5.
Total temp = 0	Total temp var = 2.

Q $x = a*b$

$y = x+c$

$z = a*b$

$w = y+z$

$w = y+z$.

$= x+c+z$ [expanded y]

$= x+c+a*b$ [expanded z]

$= a*b+c+a*b$

$= a*b+c+a*b$.

TAC	SSA
$a = a*b$	$t_1 = a*b$
$c = a+c$	$t_2 = t_1 + c$
$w = c+a$	$w = t_2 + t_1$
Total var = 4	Total var = 6
Temp var = 0.	Temp var = 2.