INDEX

NAME: K. Gowtham STD.: ______ SEC.: ______ ROLL NO.: ______ SUB.: ______

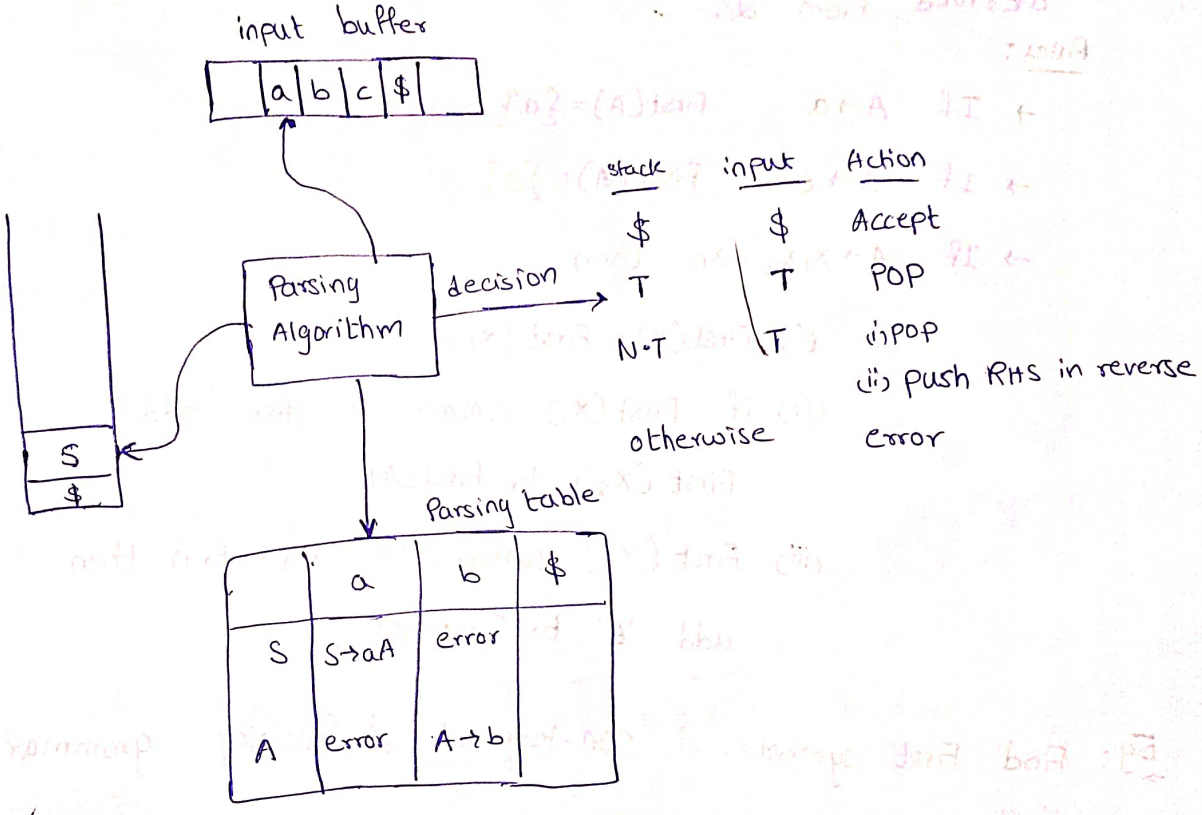
21/06/20

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	input
$ S ↓ Start symbol	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)

→ 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' $\forall i = 1$ 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
S	d, e, a
A	d, e, ϵ
B	d, ϵ
C	e, ϵ

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}

Eg: Find First sets for below grammar

$S \to AaAb \mid BbBa$

$A \to d$

$B \to e \mid \epsilon$

	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: $S \to iEtSS' \mid a$

$S' \to eS \mid \epsilon$

$E \to b$

S	i, a
S'	e, ϵ
E	b

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

$A \to da \mid BC$

$B \to g \mid \epsilon$

$C \to h \mid \epsilon$

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)

(First(A) ↳ contains ϵ; First(C) ↳ contains ϵ)

= [{d} ∪ {First(B) ∪ First(C)}] ∪ [{h} ∪ {b}] ∪ [{g} ∪ {a}]

= {d, g, h, ϵ} ∪ {h, b} ∪ {g, a}

= {a, b, d, g, h, ϵ}

Follow Set:

Consider the below grammar

$S \rightarrow Aa$

$A \rightarrow b \mid \epsilon$

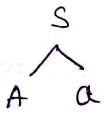
Let "a" be input string to be derived

Now First(S) = {a, b}

First(A) = {b, ∈}

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, ∈

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(β)

If First(β) contains ∈

add Follow(A) to Follow(B)

Eg: Calculate follow sets for following grammars

(a) $S \rightarrow Aa$

$A \rightarrow BC$

$B \rightarrow d \mid \epsilon$

$C \rightarrow e \mid \epsilon$

	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 → 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 → BC

Follow(B) = First(C)

But First of C contains ε

∴ we add Follow(A) to Follow(B)

{e} ∪ {a} = {a, e}

Follow(C):

A → BC

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

= {a}

Eg: S → AaAb | BbBa

A → d

B → e | ε

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

Follow(A):

S → AaAb

∴ Follow(A) = {a, b}

Follow(B):

S → BbBa

Follow(B) = {a, b}

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') → ϵ

First contains ϵ

∴ Add follow(S) to follow(S)

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

∴ Follow(S) = {e}

~~Follow~~ $S' \to eS$

Follow(s) = Follow(s')

Follow(E):

$S \to iEtSS'$

Follow(E) = t

Follow(S'):

$S \to iEtSS'$

Follow(s') = Follow(S)

∴ Follow(S) = {$, e}

Follow(s') = {$, e}

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 of (C) contain ϵ

~~Add Follow(B) to~~

Add First(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)

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)

$S \to ACB$

add first(B) to follow(C)

~~If~~ first(B) contains ϵ

∴ ~~Add~~ add follow(S) to follow(C)

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

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)

Follow (S) :

add $

B → Sb

add first (b)

Follow (A) :

S → aA

add follow(S)

C → Aa

add first(a)

Follow (C) :

A → BC

add follow(A)

Follow (B) :

~~add~~ A → 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

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

Sol :

First(A) :

$A \to bAa$

add 'b' to first(A)

$A \to \epsilon$

add 'ϵ' to first(A)

$A \to Ad$

Since first(A) contains ϵ

add 'd' to first(A)

$\therefore$ First(A) = {b, d, ϵ}

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

$P \to xQRS$

$Q \to yz \mid z$

$R \to w \mid \epsilon$

$S \to y$

what is FOLLOW(Q)?

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

Sol : $P \to xQRS$

add first(R)

first(R) contain ϵ

$\therefore$ add first(S)

w, y

$\therefore$ opt (c)

Q3) G-19 Consider the grammar given below:

$S \to Aa$

$A \to BD$

$B \to b \mid \epsilon$

$D \to d \mid \epsilon$

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

$S \to aS \mid b$

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

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 $M[A, First(\alpha)]$

5:

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
$S	ab$	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.

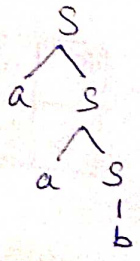
Here i/p is not removed. Read head just moves.

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, and i/p symbol then production from parse table in M[N.T, T] is applied

→ 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 in 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

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

Here Follow(S) = {$, b}

Thus the construction of parse table is as shown below

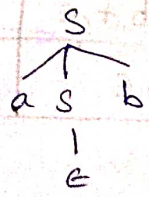
i/p	a	b	$
S	S→aSb	S→ε	S→ε

The question in GATE are mainly asked on contruction of predicative 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



~~Q2~~

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)

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

→ Construct predictive parse table

$S \to aA \mid AB$
$A \to b \mid \epsilon$
$B \to c \mid \epsilon$

	a	b	ϵ	$
S	S→aA	S→AB	S→AB	S→AB
A	error	A→b	A→ϵ	A→ϵ
B	error	error	B→c	B→ϵ

S→AB is added in first(AB)

first(AB) = {b, ϵ, c}

∵ first(AB) contains ϵ

∴ 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, ϵ, d}

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

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

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

→ 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 not have other grammars which are not left recursive and still not suitable for predictive parsing.

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: Consider

$S \to aA \mid Ab$

$A \to a$

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

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

$S' \to eS \mid \epsilon$

$E \to b$

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

Observation: Here the language of grammar is

{aa, ab}

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 even happen in the middle of a string~~
~~So telling whether grammar is suitable are not for left predictive parsing is 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

∴ 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).

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

$S \to iEtSS' / a$

$S' \to eS / \epsilon$

$E \to b$

Consider

$S \to iEtSS' / a$

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

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

and neither of the two first sets containing ϵ.

Now

$S' \to eS / \epsilon$

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

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

But First(ϵ) has ϵ

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

Now

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

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

∴ 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.

$S \to aA / AB$

$A \to b / \epsilon$

$B \to c / \epsilon$

Sol:

$S \to aA / AB$

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

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

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

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

$A \to b \mid \epsilon$

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

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

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

$B \to c \mid \epsilon$

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

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

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

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

$\therefore$ 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.

$A \to aA \mid \epsilon$

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

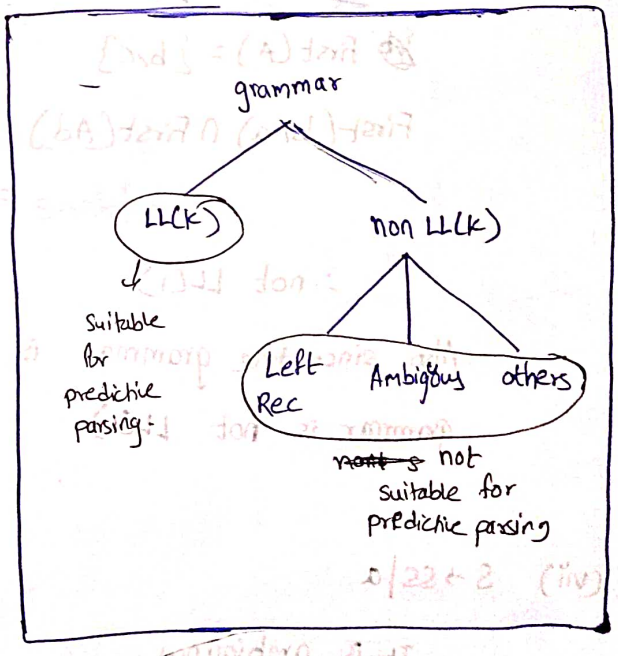
$\{a\} \cap \{\epsilon\} = \phi$

~~Follow~~ ~~First(A) ∩ 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)

(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).

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 $S \to ab \mid S \to abc$

$\therefore$ the grammar is LL(3)

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

$\therefore$ LL(6)

(P/27) G1:

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

↓

left & right recursive

$\therefore$ ambiguous

G2:

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

↓

precedence : (*) > (+)

associative : + ⇒ L→R

* ⇒ R→L

Since G_2 is left recursive, it is not LL(1)

~~also G_1 is same~~ ∴ 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 → ϵ~~ $S \to aSbS$ 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

Q34 / G-03 Consider the grammar shown below

$S \to iEtSS' \mid a$

$S' \to eS \mid \epsilon$

$E \to b$

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 / G-06 Same as Q34~~

Q35 / G-06 Consider the below grammar

$S \to FR$

$R \to *S \mid \epsilon$

$F \to id$

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) c) $\{S \to FR\}$ and $\{R \to *S\}$

d) $\{S \to FR\}$ and $\{\}$

e) $\{F \to id\}$ and $\{R \to \epsilon\}$

	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$_1$

iCtiCtS$_1$

ibtibtS$_1$

ibtibtes

ibtibtea

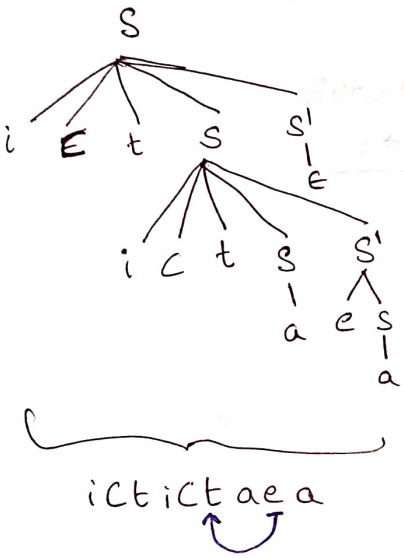
iCtSS$_1$

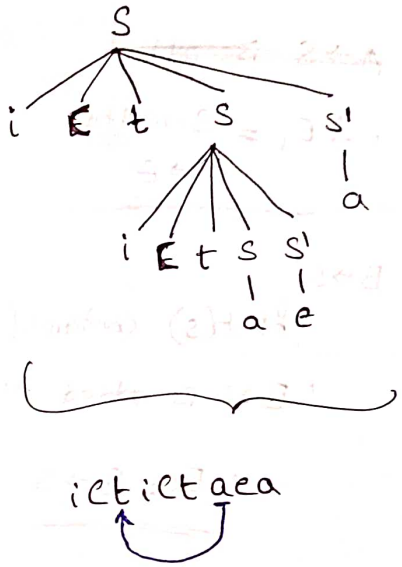
iCtbiCt

For code i C t i C t a e a

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 / 4-12 (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 _____

(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~~

$\therefore E_1 =$ S → aAbB, S → ε $\qquad E_2 =$ S → bAaB, S → ε

B → S

first(S) contains (ε)

∴ B → S is added to follow(B)

$\therefore E_3 =$ B → S

Error Recovery in Predictive parsing:

(i) Panic mode Recovery

(ii) Phrase recovery : error entries contains pointers to errors 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.

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.

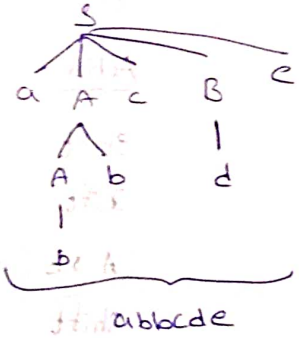
6. After finding a handle, handle pruning is done.

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: Its 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.

Initial Configuration

Stack	input
$	w$

Accept Configuration:

Stack	input
$S	$

Note: The contents of stack during SR parsing are actually prefixes of right sentential forms but not all prefixes are allowed 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 (reduce)

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.

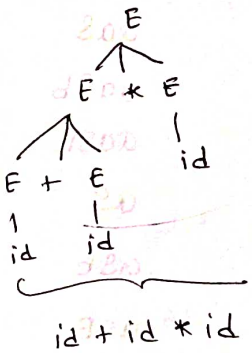
Eg: Consider below grammar and trace the steps of shift reduce parser for given i/p string.

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

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.

* $S \to AB$

The above is also not an operator grammar.

* $S \to ab$

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.

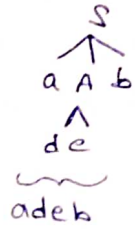
Relation	Meaning
$a < b$	a gives precedence to b
$a = b$	a has same precedence as b
$a > b$	a takes precedence over b

→ 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.

Eg:

$S \to aAb$

$A \to de$



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 (* > +)

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

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 > $

for id & +

Consider id + id

Here we 1st reduce id to E

E + id
E + E
E

∴ id has more precedence than +

Silly for id & * , id & $

Now + & id

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

Now + & +

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

consider id + id + id

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

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>

+ & $

$<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.

Now consider input string id + id * id

Now we write the i/p 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)

$\$ < id > + < id > * id < id > \$$ (E → id)

$\$ < E + < id > * < id > \$$ (E → 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 <+7

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

$\$ < E + < E * < id > \$$ (E → id)

↳ This actually seen as

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

$\$ < E + < E * E > \$$ (E → E * E)

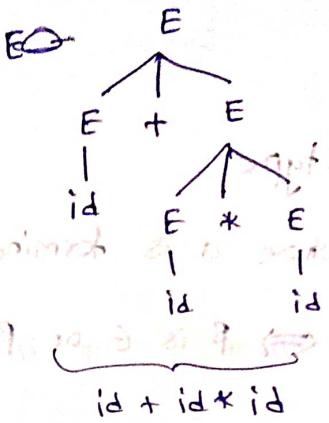
$\$ < E + E > \$$ (E → E + E)

$\$ E \$$

The above actually means $\$\$$

$\therefore$ Accept

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 → cd~~

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

② A 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

~~$A \to \epsilon$~~ $A \to \alpha B$

$Trail(A) = Trail(B)$

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	d
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

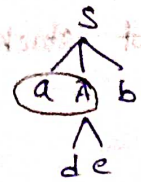
$A \to x_1 x_2 \cdots x_n$

(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

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

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



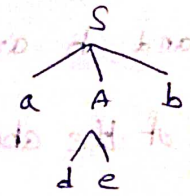
$S \to aAb$

$ a < d e > b $

$a < lead(A)$

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

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



$S \to aAb$

$ 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

$S \to aAb$

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)

$S \to aAd$

$A \to bc$

$b = c$

$S \to aAd$

$a < Lead(A)$

$S \to aAd$

$trail(A) > d$

$ < Lead(S)$

$Trail(S) > $

All the unfilled entries are considered error entries

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

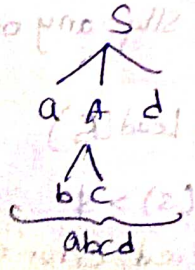
Stack	input	Action
\$	abcd \$	push ($\because \$ < a$)
\$a	bcd \$	push ($\because a < b$)
\$ab	cd \$	push ($\because b = c$)
\$abc	d \$	pop until (a rule ① sto satisfies ($\because c > d$)
\$ab	d \$	pop ($\because b = c$)
\$a	d \$	Stop popping ($\because b < b$)
\$a	d \$	Push ($\because a = d$)
\$ad	\$	Start popping ($\because d > \$$)
\$a	\$	pop ($\because a > b$)
\$	\$	Accept

bc Handle

A → bc

S → aAd

Parse Tree Construction



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

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

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

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 → F + E, id > +~~.

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

$\rightarrow *$

$\therefore$ opt (b) is correct

c) $+ > -$

Trail$(N \cdot T) > -$

$E \rightarrow F - E$

$id > -$

$\therefore \alpha$

d) $* < -$

terminal $<$ lead$(N \cdot T)$

$* <$ lead$(N \cdot T)$

$E \rightarrow E * F$

$* <$ lead(F)

$* < id$

$\therefore \alpha$

$\therefore$ opt (b)

GATE 2004 Which of the following grammar rules violate the requirements of an operator grammar? P, Q, R are non-terminals and q, s, t are terminals:

(i) $P \rightarrow QR$ (ii) $P \rightarrow QsR$ (iii) $P \rightarrow \epsilon$ (iv) $P \rightarrow Qt Rq$

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

Sol:

(i) $P \rightarrow QR$

two N.Ts adjacent

$\therefore$ not op grammar

(ii) ✓

(iii) Operator grammar should be null free

$\therefore$ not op grammar

(iv) ✓

$\therefore$ opt (b)

Material Problems:

P/33 $S \to AB$ ∝ (two N.Ts adjacent)

$S \to AaB$ ✓

$S \to a$ ✓

$S \to \epsilon$ ∝ (not null free)

∴ I & IV

∴ opt (d)

P/34 c) $S \to AaB$

$A \to aA|b$

$B \to bB|\epsilon \Rightarrow B \to \epsilon$

∴ opt (c)

P/35 $S \to AB$

$A \to c|d$

$B \to aAB|d$

a) $S \to Aa|Ab$

$A \to c|d$

It finite grammar where as grammar in the question is infinite

∴ ∝

b) $S \to AaAB$

not operator grammar

c) $S \to AaS|Ab$

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

$S \to AB$

$A \to c|d$ ≡

$B \to aS|d$

$S \to Aas|Ad$ (∵ substituting B)

$A \to c|d$

$B \to as|d$

Now this grammar is an operator grammar.

Q3 Ans

P/36

$E \to E + E$

$E \to E * E$

$E \to E - E$

$E \to E / E$

$E \to E \wedge E$

$E \to (E)$

$E \to id$

$\wedge$ is right associative

In, $a \wedge b \wedge c$ $b \wedge c$ is evaluated first

$\therefore \wedge < \wedge$

-, +, * are left associative

$\therefore - > -$, $+ > +$, $* > *$

$\therefore$ opt (d)

P/37

$Lead(B) = \{e\}$

$Lead(A) = Lead(B) \cup \{c\}$

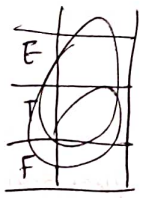
$= \{e, c, d\}$

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

$= \{a, c, e, d\}$

$\therefore$ opt (a)

P/38



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

$Lead(F) = \{id,)\}$

$Lead(T) = \{*\} \cup \{Lead(F)\}$

$Lead(E) = \{+\} \cup Lead(T)$

$= \{+, *,), id\}$

$\therefore$ opt (b)

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~~ id > ↑

b) ↑ > +

Trail(N·T) > +

E → E + T

Trail(E) > +

↑ > + ✓

c) + ≥ ↑

Trail(N·T) > ↑

T → F ↑ T

Trail(F) > ↑

id > ↑

∴ ✗

d) + < +

T < lead(N·T)

E → E + T

+ < lead(T)

+ < ↑ and + < id

∴ ✗

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

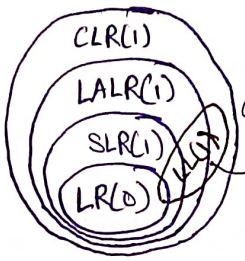
∴ opt (d)

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:

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



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)

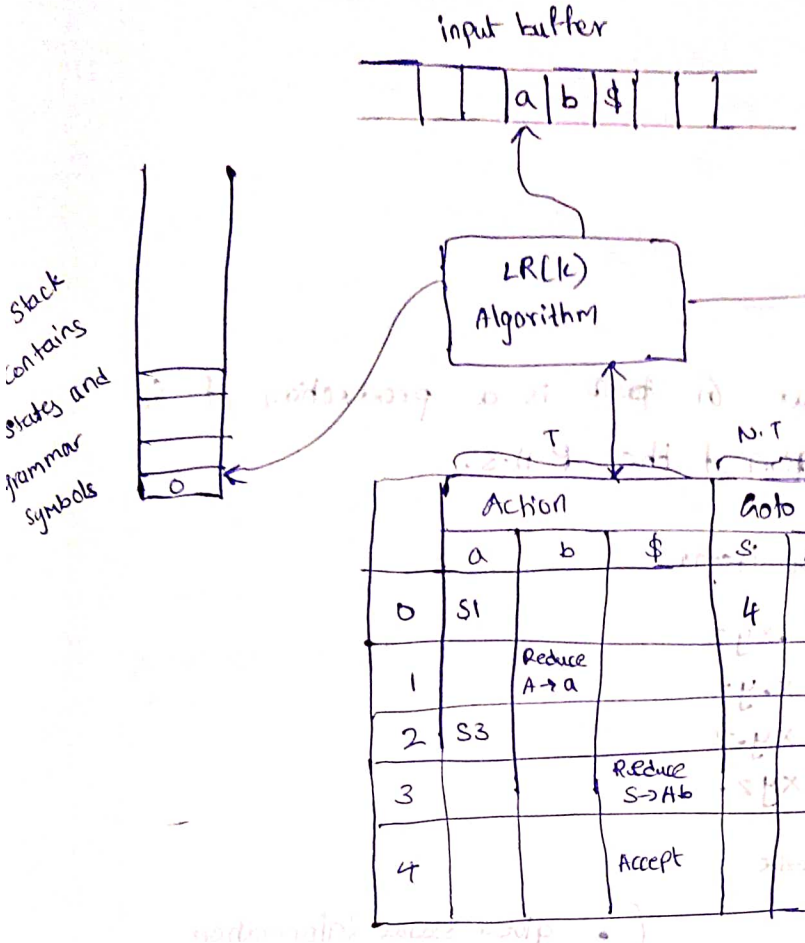
∴ There exists a grammar which LL(1) & LR(0)

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

∴ d is false

LR(k) Parser Structure :

The parsing 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

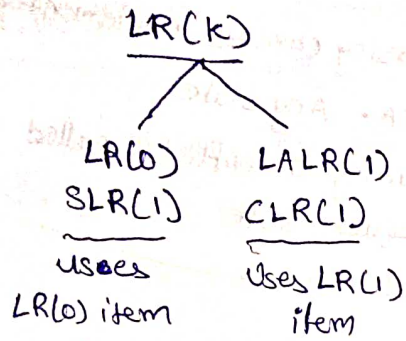
(ii) Push L.H.S of the production

(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.



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.

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

$A \to \alpha . X \beta$ (i.e. $A \to \alpha X . \beta$) 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: $E \to E + T$

Q $E \to T$

$T \to id$

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

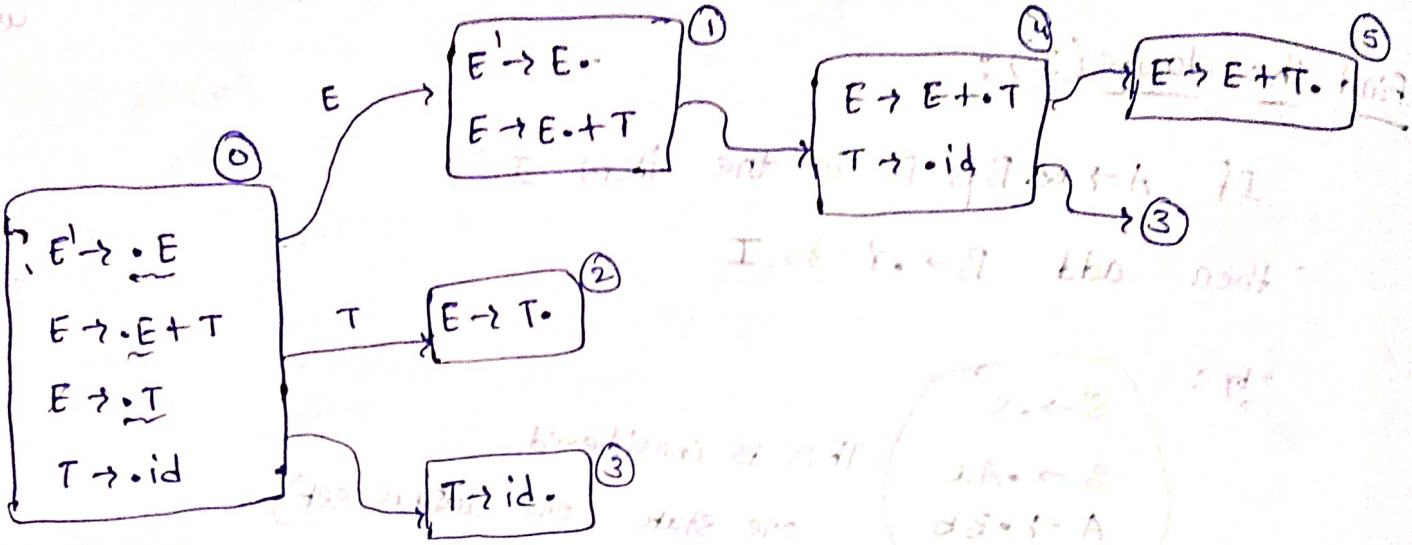
kernel items:

$S' \to .S$

and all items whose dots are not at the left end

Non-kernel items:

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



∴ The parser has 6 states (0 to 5)

Rules for constructing LR(0) parsing table:

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

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

② If $A \to \alpha \bullet a\beta$ is in state I then, (a is terminal)

$$\boxed{\text{Action}[I, a] = \text{shift } j}$$

(• terminal)

③ If $A \to \alpha\bullet$ 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 $A \to \alpha \bullet B\beta$ is in state I then

$$\boxed{\text{Goto}[I, B] = j}$$

(• non-terminal)

All the unfilled entries are considered error entries.

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)

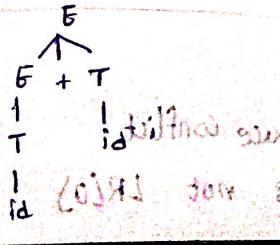
Parsing :

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

LR(0) Parser Conflicts:

1) Shift - Reduce Conflict:

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

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

(I)

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

Action[I, a] = shift j

For $\beta \to \alpha .$

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

$\therefore$ 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

$A \to \alpha .$
$B \to \alpha .$

(I)

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

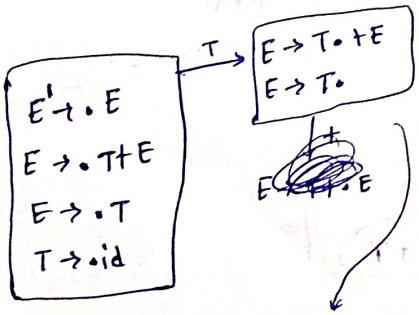
} Conflict

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

$\therefore$ shift - Reduce conflict

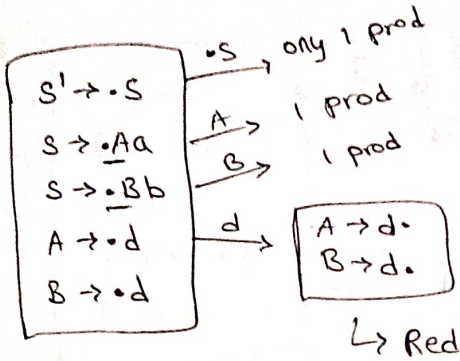
$\therefore$ The grammar is not LR(0)

Shortcut:

Here we don't construct 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 productions only

Qu1) Check whether the below grammar is LR(0) or not

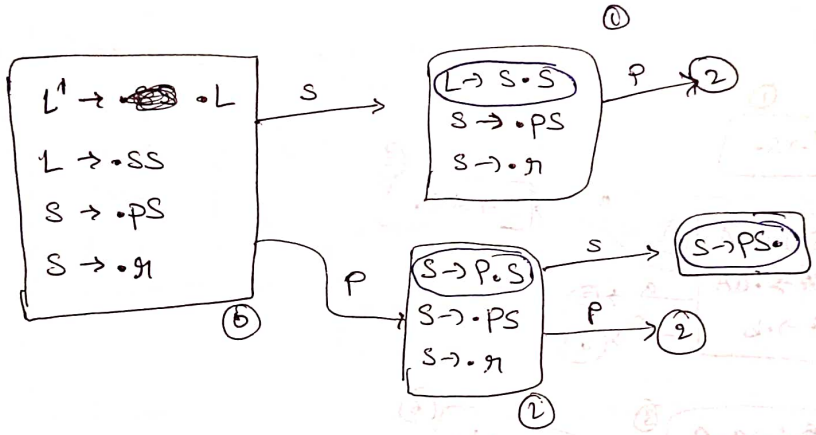
$S \to Aa$
$S \to Bb$
$A \to d$
$B \to d$



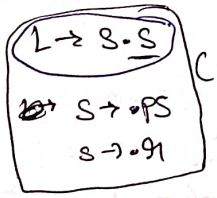
↳ Reduce - Reduce conflict

Material Problems:

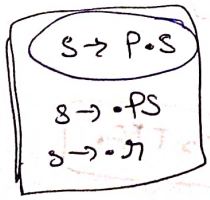
P/q9) $L \to SS$
$S \to pS \mid \eta$



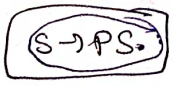
Method 2: Given $L \to S.S$, $S \to P.S$ $S \to PS.$



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



(∵ .S is present)



∴ opt (a)

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

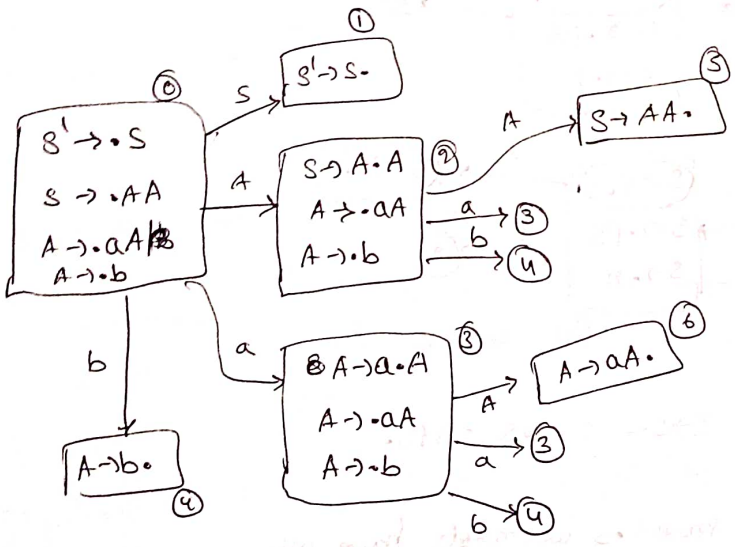
$S' \to .S$ ($\because$ we have $.S$ add 'S' productions)

$S \to .SB$

$S \to .A$ (we don't add B since B-productions since we don't have $.B$)

$A \to .a$

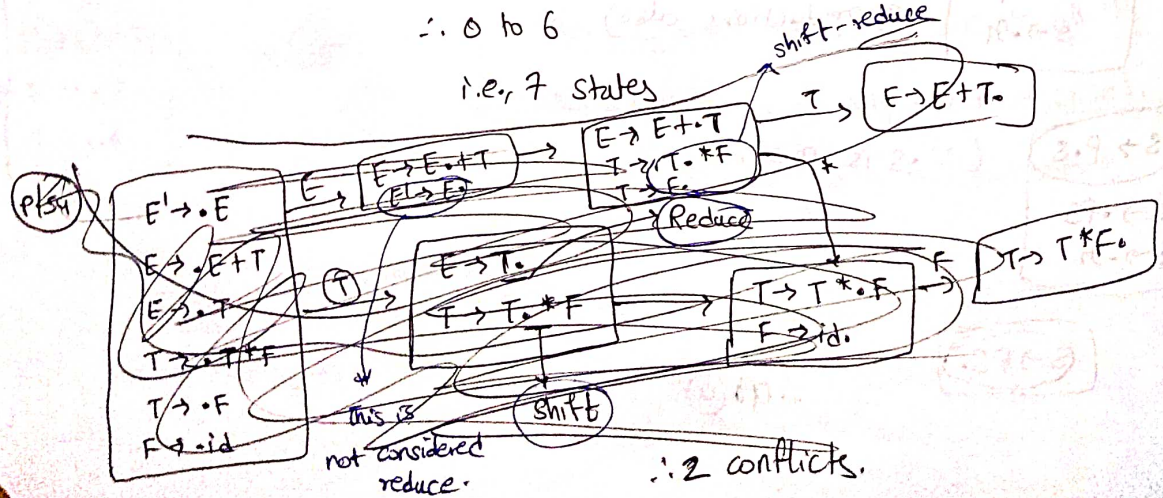
P/52



$\therefore$ 0 to 6

i.e., 7 states

P/54



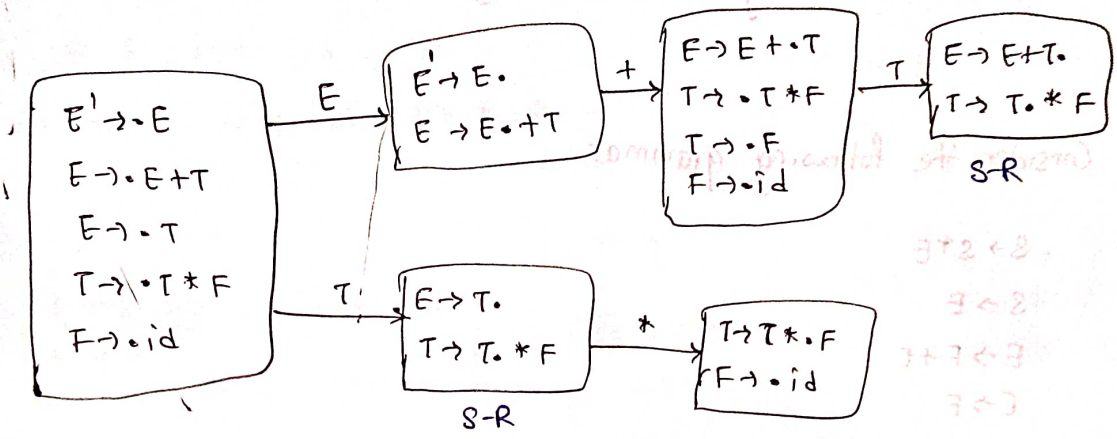
$\therefore$ 2 conflicts.

P/53 same as previous example

It is LR(0)

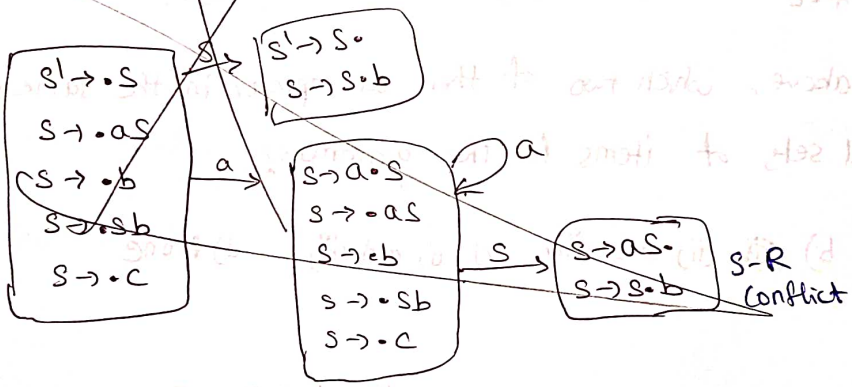
∴ CLR(1)

P/54

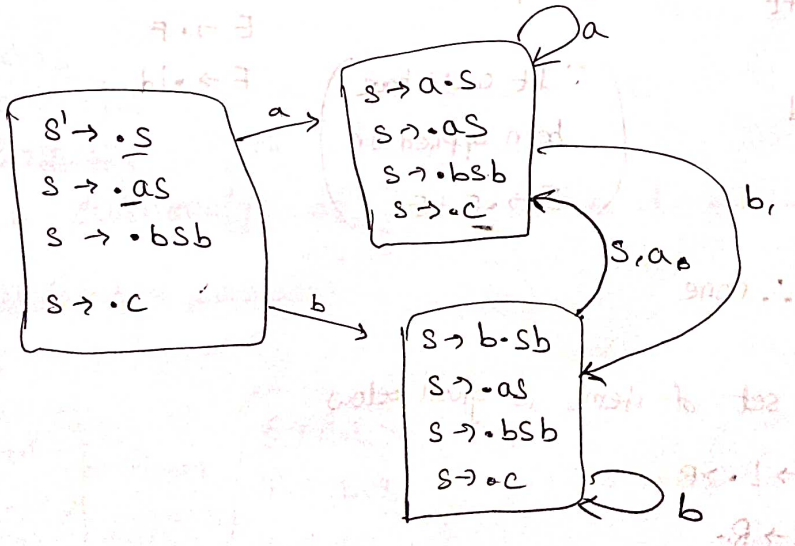


∴ 2 conflicts.

P/55 S→aS | b | sb | c



P/55



~~LR(0)~~

∴ not LR(0) is false

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

ambiguous

∴ doesnot belong to LR(k) family

(Q42 / G-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) $S \to S*.E$

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

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

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) $S \to S*.E$

$E \to .F+E$

$E \to .F$

$F \to .id$

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

$E \to F.$

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

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

$E \to .F+E$

$E \to .F$

$F \to .id$

∴ none

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

$S \to L.>R$

$S \to R.$

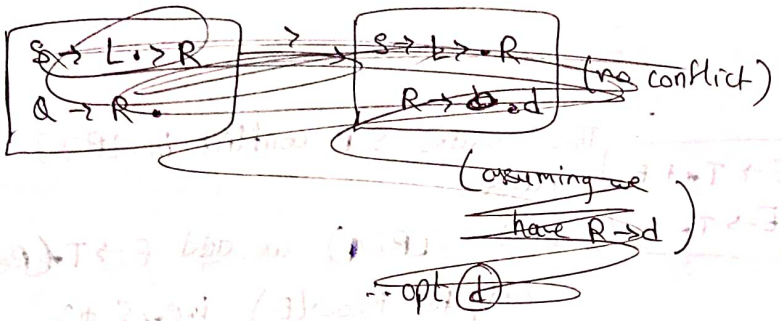
on the input symbol $\leq$ the set has

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

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

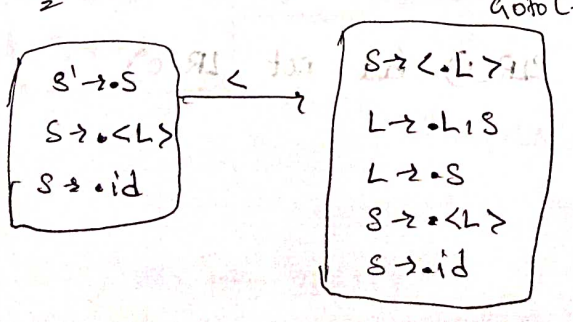
$S' \to S$

$S \to <L> \mid id$

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

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

Sol:



∴ 5 items

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

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

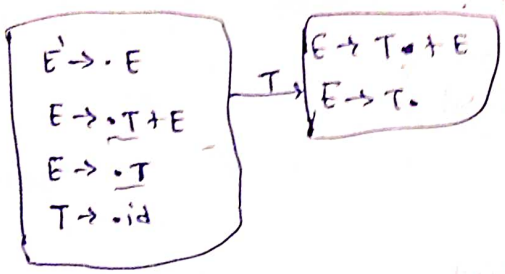
i.e., if $A \to \alpha \cdot$ is in state I then

Action (I, FOLLOW(A)) = Reduce $A \to \alpha$.

Eg: $E \to T + E$

$E \to T$

$T \to id$



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

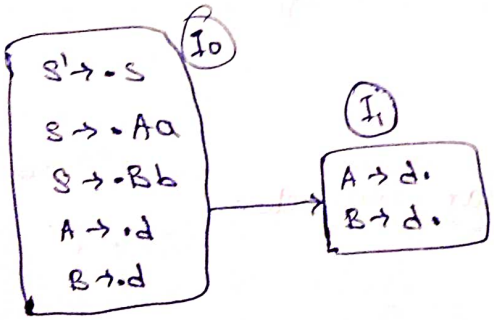
But in SLR(1) we add $E \to T$ (Reduce) only to follow(E) i.e., {$}

∴ 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)

Parser Conflict in SLR(1):

i) Shift - Reduce Conflict:

If any state has items of type

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

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

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

ii) Reduce - Reduce Conflict:

If any state has items of type

$A \to \alpha .$
$B \to \beta .$

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

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

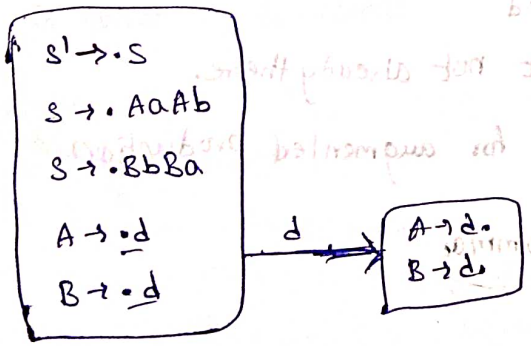
$S \to AaAb$

$S \to BbBa$

$A \to d$

$B \to d$

Sol:



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

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

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

$\therefore$ R-R conflict.

not SLR(1)

5.

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

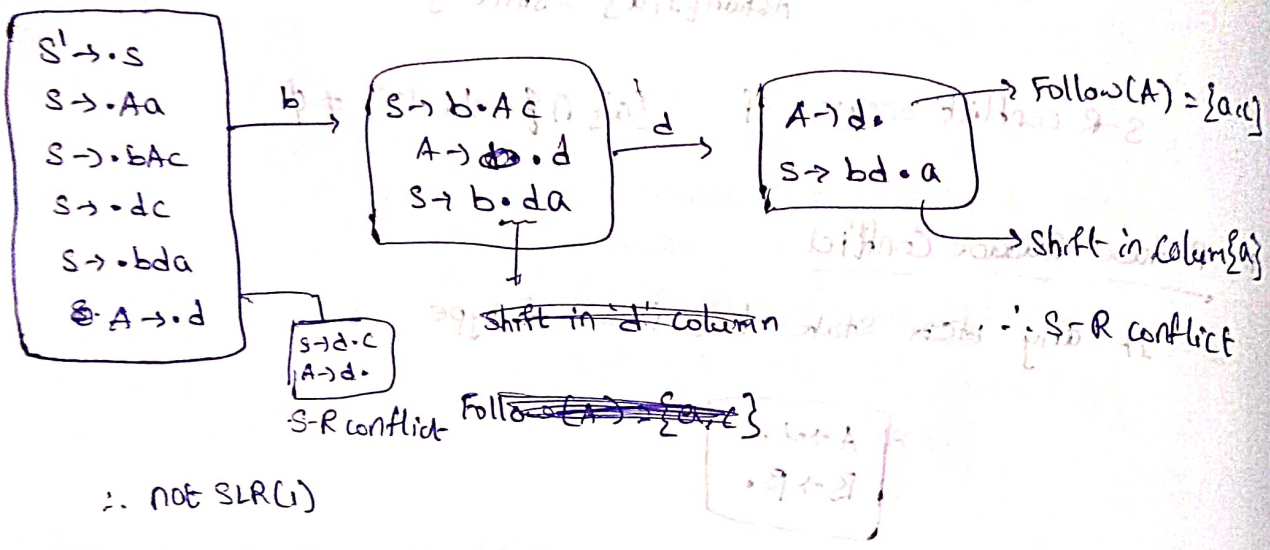
$S \to Aa$

$S \to bAc$

$S \to dc$

$S \to bda$

$A \to d$



∴ not SLR(1)

Note :

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

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.

Eg: Construct LR(1) items for below grammar

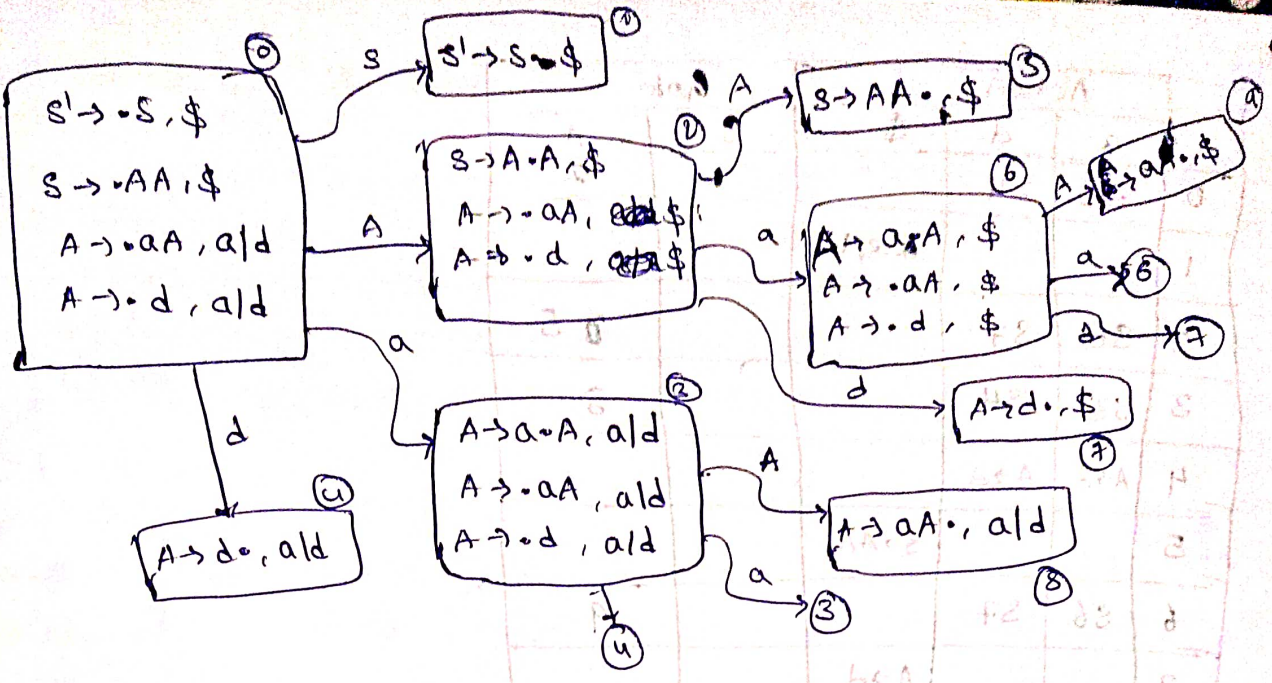
(2)

$S \to AA$

$A \to aA$

$A \to d$

First(A) = {a, d}



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 grammar.

CLR(1) Parse table (Canonical LR):

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

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

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

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.

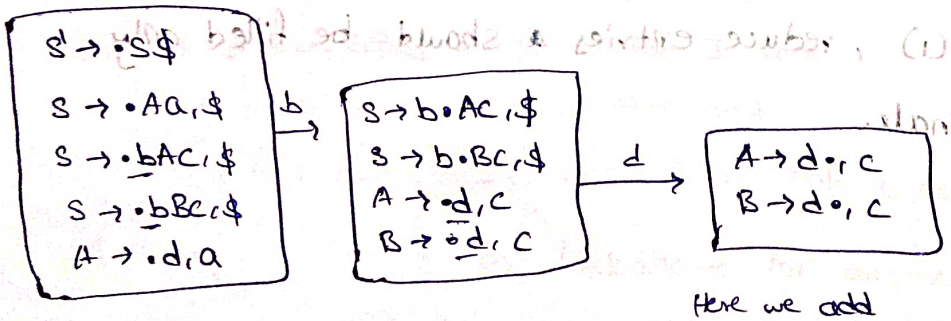
	Action			Goto	
	a	d	$	S	A
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	A→d	A→d			
5			S→AA		
6	S6	S7			9
7			A→d		
8	A→aA	A→aA			
9			A→aA		

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

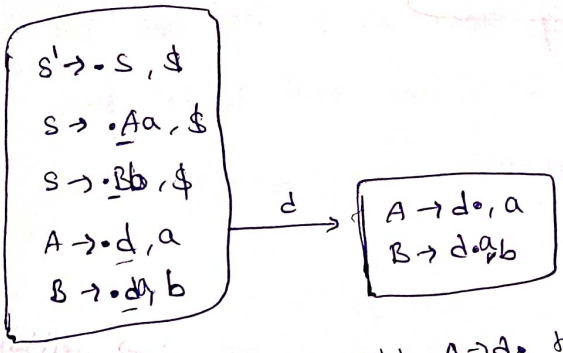


Here we add
A→d to Action[I, c]
B→d to Action[I, c]

∴ Reduce - Reduce Conflict.

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

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



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

$First(aS) \cap First(bS) = \phi$
$First(bS) \cap First(c) = \phi$
$First(aS) \cap First(c) = \phi$

Every LL(1) is CLR(1)

∴ above grammar is CLR(1)

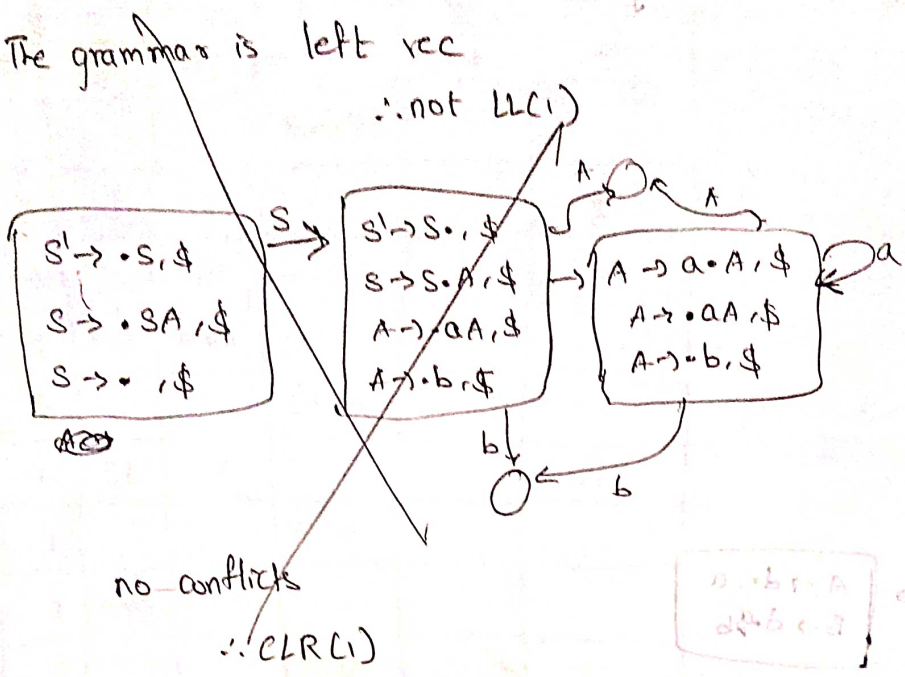
Ques Check whether below grammar is CLR(1) or not

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

Sol:

The grammar is left rec

∴ not LL(1)



no conflicts

∴ CLR(1)

Sol:

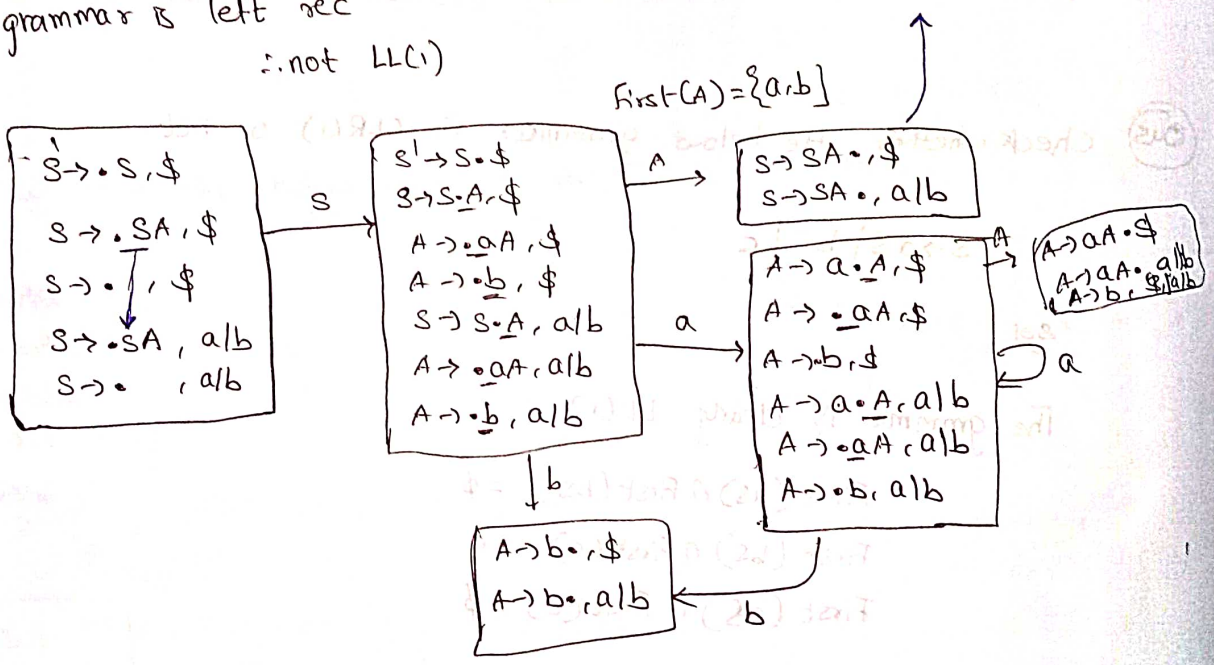
This can also be written like this.

S→SA•, $/a/b

The grammar is left rec

∴ not LL(1)

First(A) = {a, b}



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).

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

→ The drawback of CLR(1) 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.

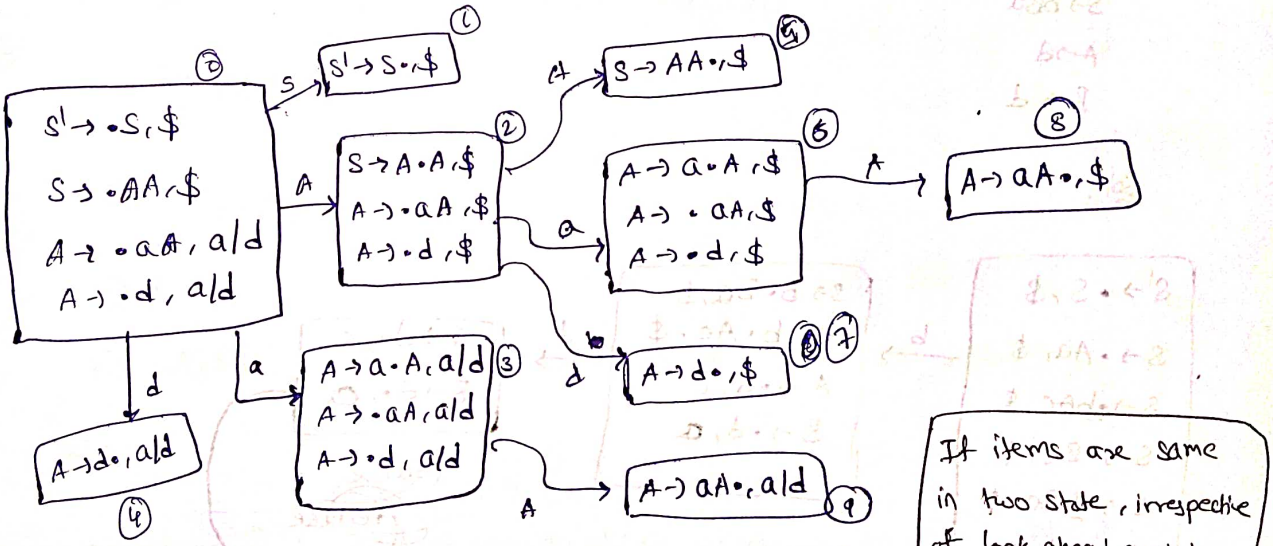
→ we can find LALR states by merging LR states that have the same set of first components and we can take the union of look ahead symbols. ~~to get LALR(1) states~~

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/$

(67)

A → d•, $/a/d

(89)

A → aA•, a/b/$

Now for the above grammar

CLR(1) has 10 states
LALR(1) has 7 states
SLR(1) has 7 states

Note:

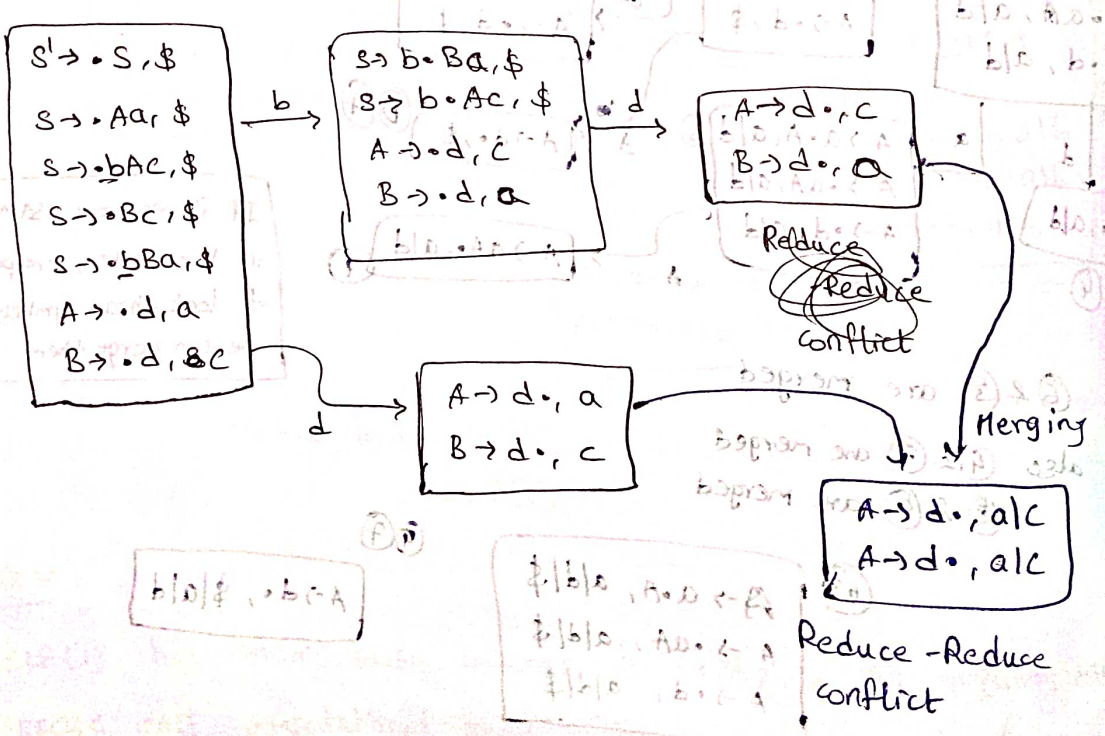
no of states in

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

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

S → Aa
S → bAc
S → Bc
S → bBa
A → d
B → d

Sol:



∴ 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 / G-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 / G-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)

Sol :

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

all first symbols are different

∴ LL(1)

Every LL(1) is LR(1)

∴ opt (c)

Q50 / 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

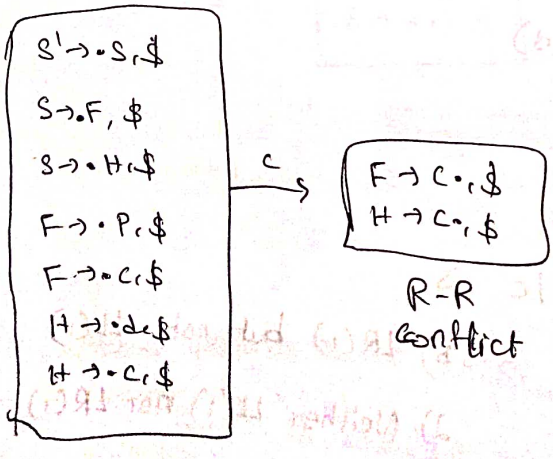
Sol :

$S \to F \mid H$

First(F) ∩ First(H)

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

∴ not LL(1)



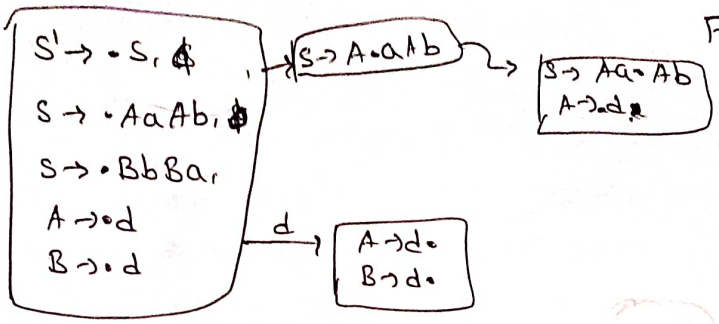
∴ opt (d)

Material Problems

P/57

Follow(A) = {a, b}

Follow(B) = {a, b}



↓

Follow(A) ∩ Follow(B)

= {a, b}

∴ R-R conflict

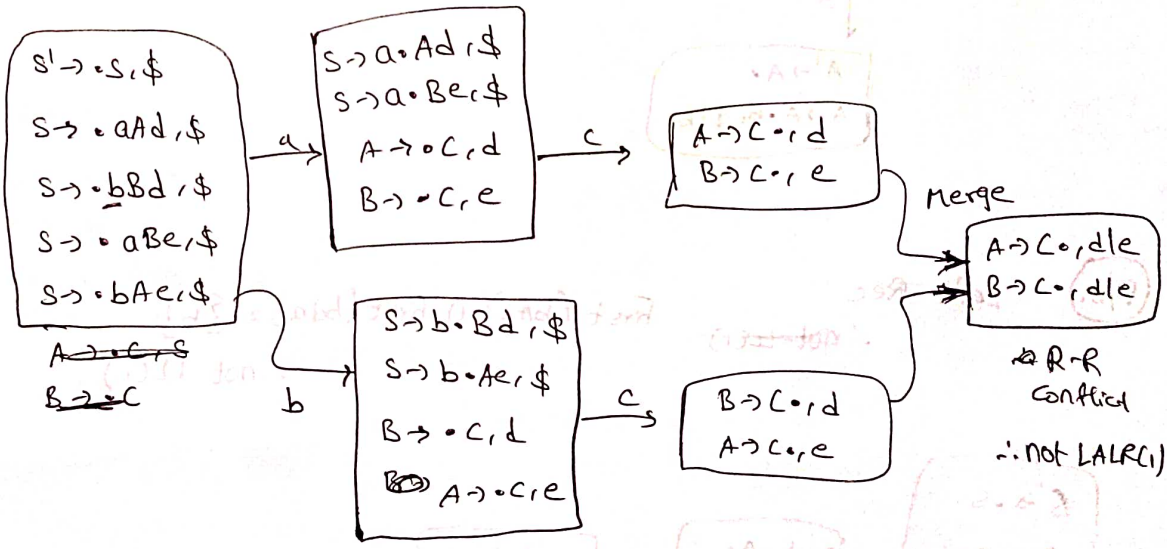
The question is asked ↑ no of conflicting entries

A → d •

B → d •

i.e., 2 entries.

P/58



But it is clearly CLR(1)

∴ LR(1) but not LALR(1)

P/59

$S' \to .S, \$$

$S \to .Aa, \$$

$S \to .bAc, \$$

$S \to .Bc, \$$

$S \to .bBa, \$$

$A \to .d, a$

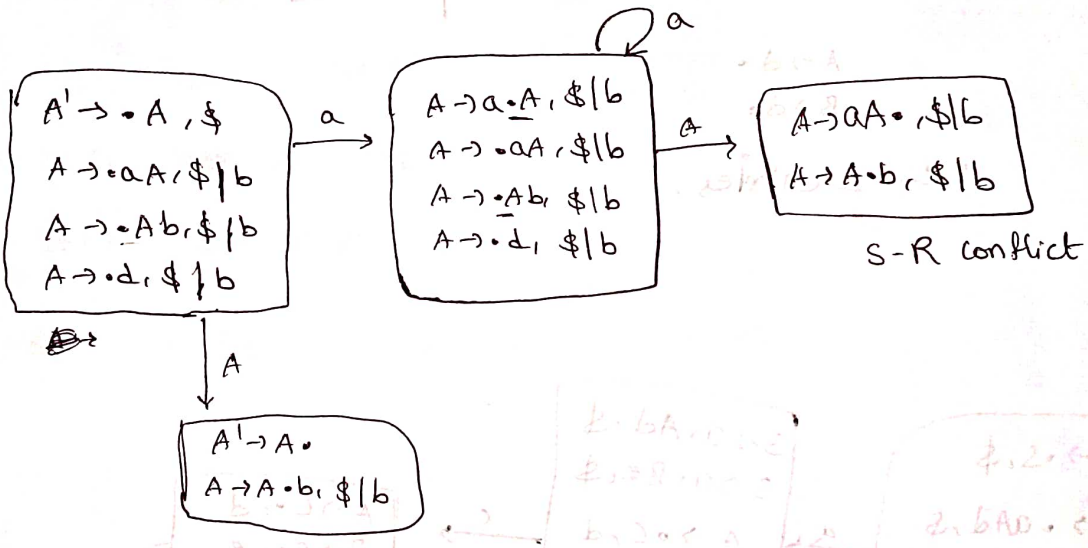
$B \to .d, c$

This is same as Q47

∴ LR(1) but not LALR(1)

28/06/20

P/60



S-R conflict

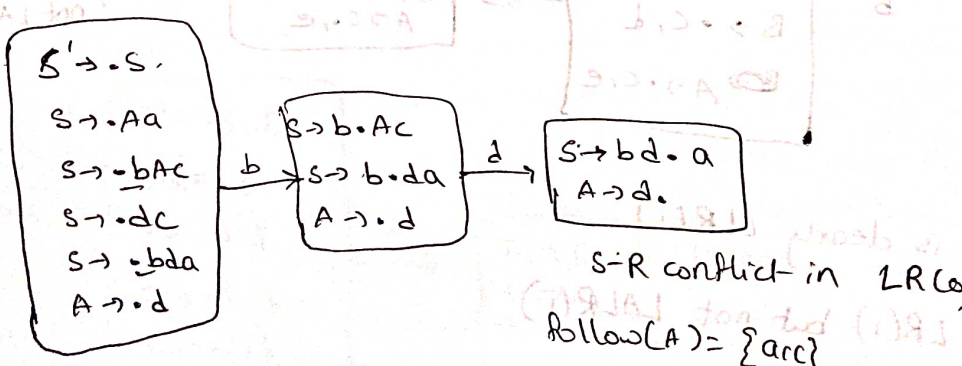
P/61

~~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)

P/62

$S \to \cdot A, \$$

$A \to \cdot AB, \$$

$A \to \cdot, \$$

$A \to \cdot AB, b$

$A \to \cdot, b$

$First(B) = \{b\}$

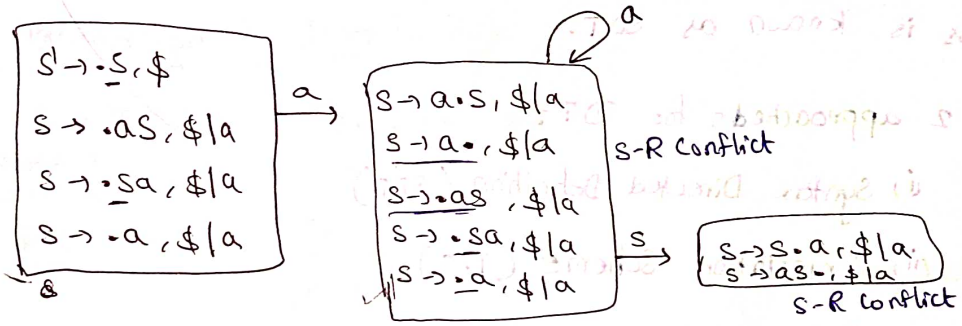
$\therefore S \to \cdot A, \$$

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

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

$\therefore$ opt (c)

P/63



$\therefore$ 2-conflicting states

P/64

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

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

$S \to \cdot R, \$$

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

$R \to \cdot L, \$$

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

$\therefore$ opt (c)

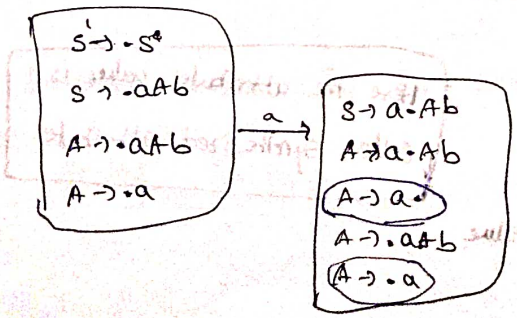
P/65

$S \to aAb$

$A \to aAb \mid a$

$First(aAb) \cap First(a) = \{a\}$

$\therefore$ not LL(1)



SR-Conflict in LR(0) parse

follow (A) = {b}

$\therefore$ we dont have SR conflict in SLR(1)

$\therefore$ LR(1)

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

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

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

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

$\therefore$ 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.

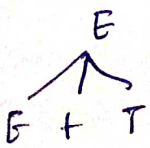
Attributed are of two types:

Synthesized attributes or Inherited attributes.

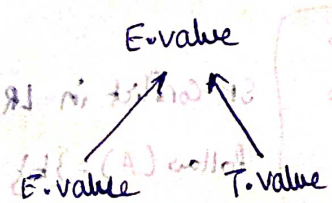
Synthesized attribute:

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

→ 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.

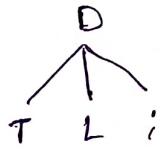
Inherited attributes:

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

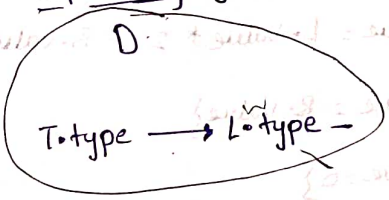
Eg:

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

~~The a~~

~~→ An SDD that uses synthesized att~~

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

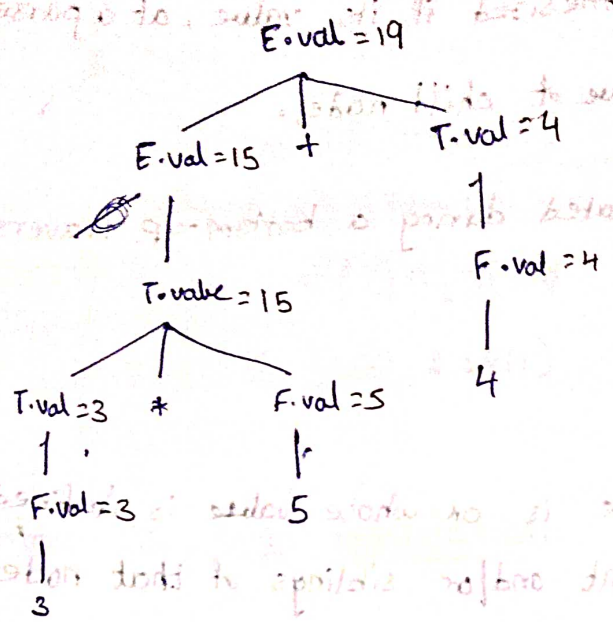
Eg: Consider the following SDD

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 ~~attribute~~ 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



$\therefore$ E.value = 19

(Q51) Consider the following SDD

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

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

$\therefore$ L.value = 11

Q52 / GATE 2004

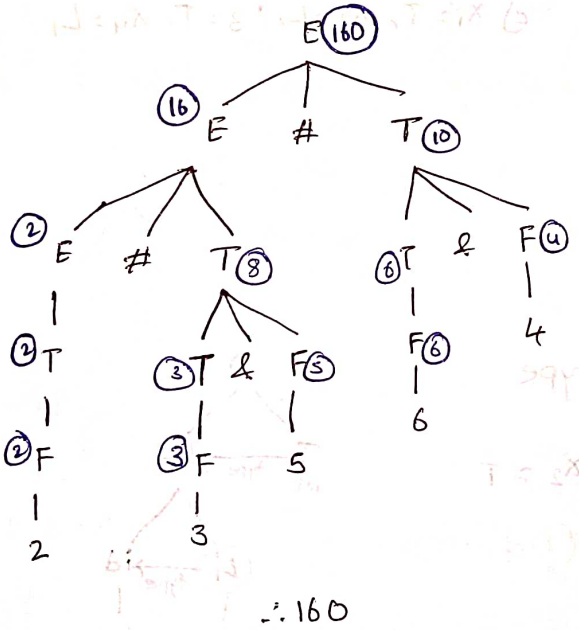
Consider the grammar

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

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 / GATE 4-19

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?

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

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

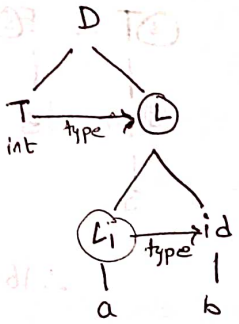
Sol:

$D \to TL$

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

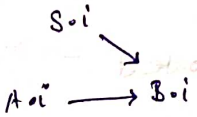
Ex: SDD in (Q51)

→ ~~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)

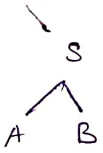
'L' attributed

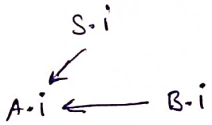
but not 'S' attributed

L-attributed

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

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



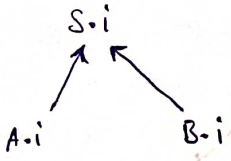


(no left to right evaluation)

∴ 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.

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

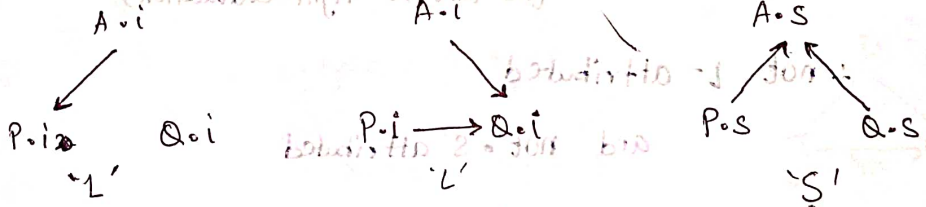
b) Both ① & ② are L-attributed

c) ① is L-attributed

d) ② is L-attributed

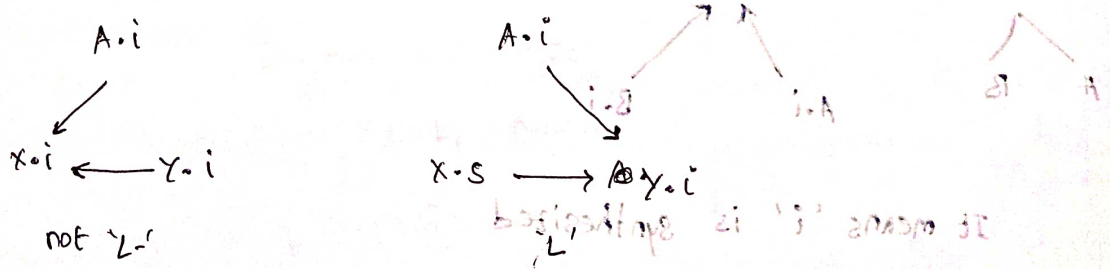
Sol:

Rule 1:



above are L-attribute rules.

Rule 2:

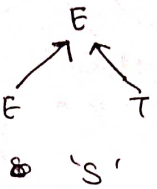


∴ above rule is not L-attribute

∴ opt ©

Material Problems on SDT:

P/2 ~~E → E+T~~



E → T

E → id

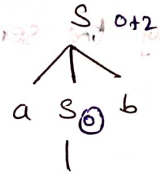
T → S

∴ S-attributed

∴ L-attributed

∴ opt (c)

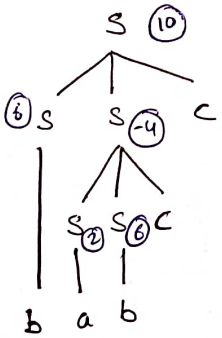
P/3 ~~P/8~~ $S \to ab$



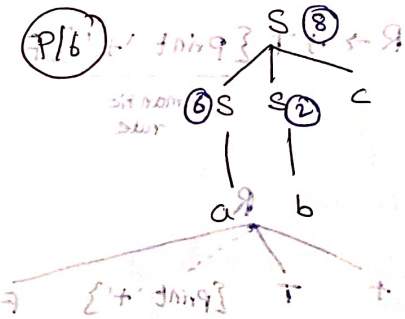
∴ length of string

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

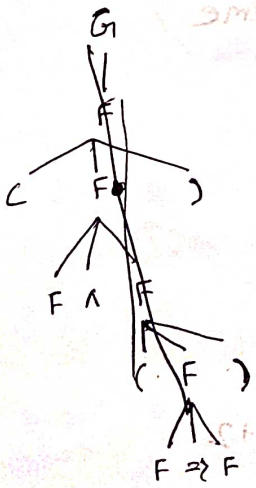
P/5

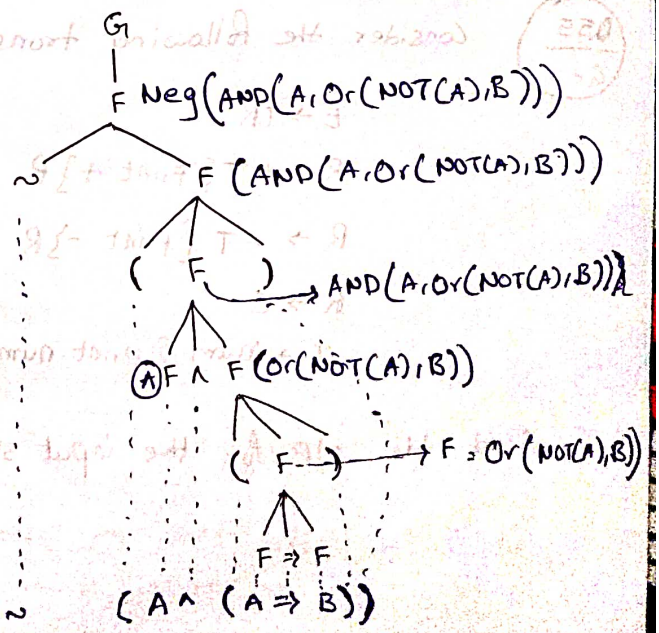


P/6

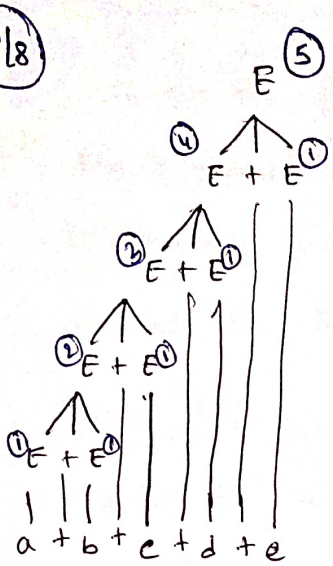


P/7





P18



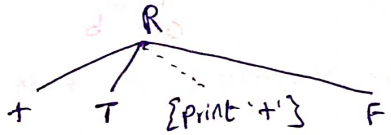
Translation Scheme:

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.

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

semantic rule



Q55 Consider the following translation scheme

$E \rightarrow TR$

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

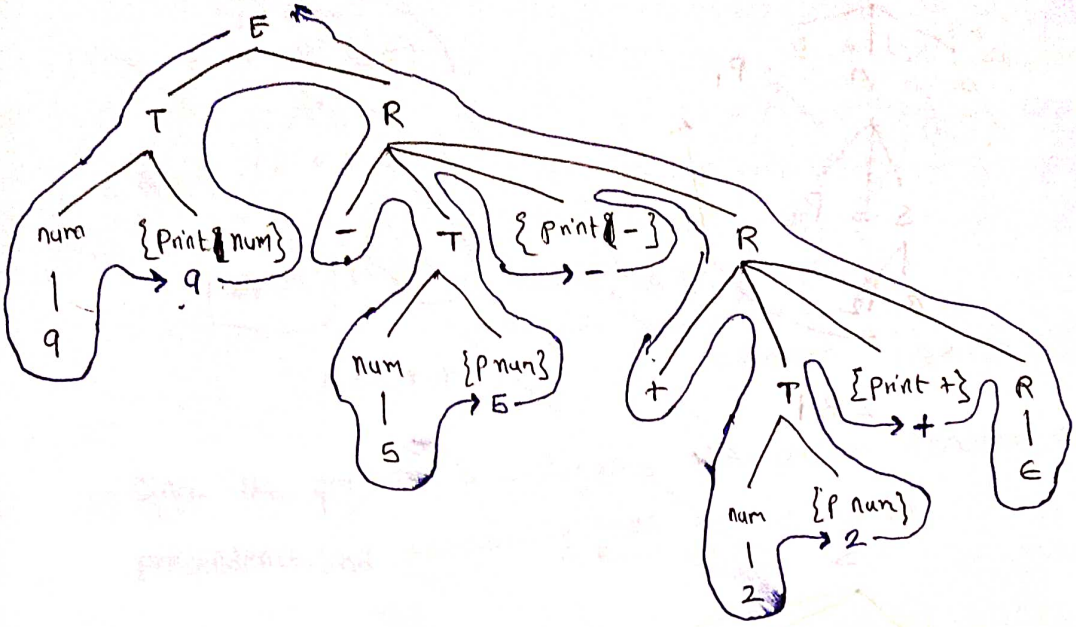
$R \rightarrow -T \{print -\} R$

$R \rightarrow \epsilon$

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

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

Sol:-



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

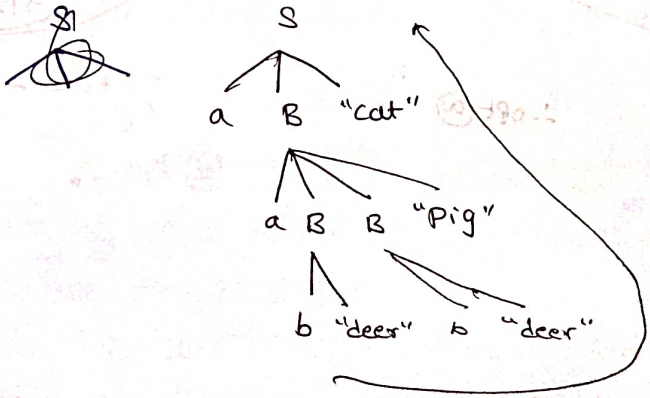
$\therefore$ 95-2+

Material Problems

P/9

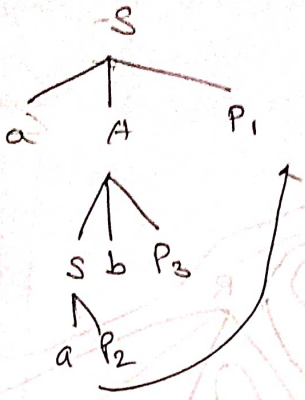
2111

P/10



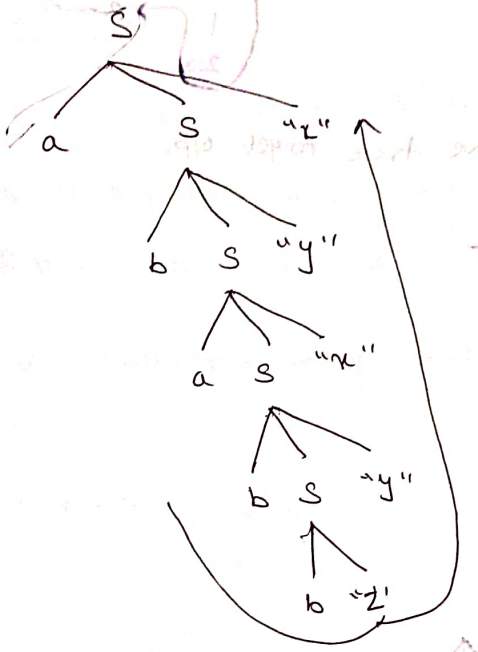
deer, deer, pig, cat

P/11



∴ 231

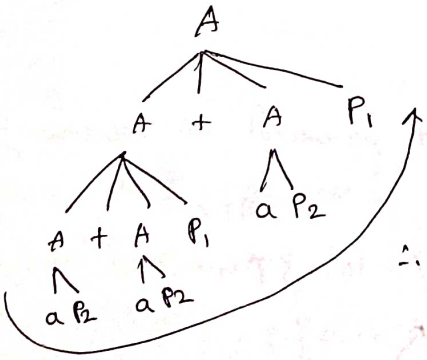
P/12



zyxyx

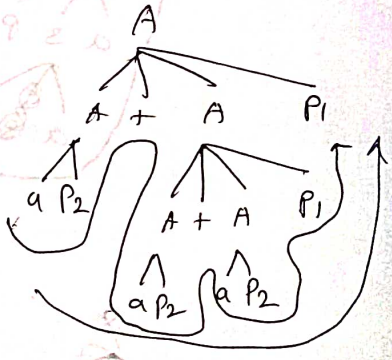
∴ opt (c)

P/13



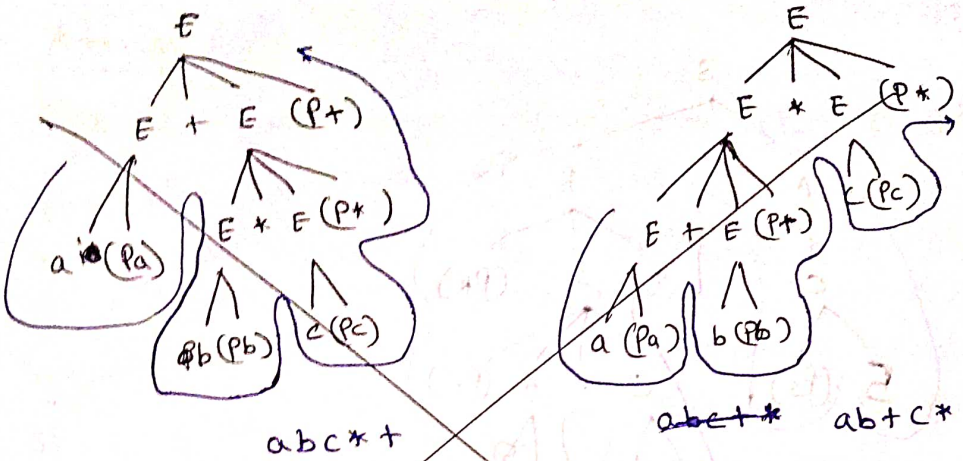
∴ 22121

∴ opt (a)



∴ 22211

(P/M)



Since the grammar is ambiguous the answer depends on precendences and associativity given by the compiler

(P/14) a+b*c

SR parser scans from left to right

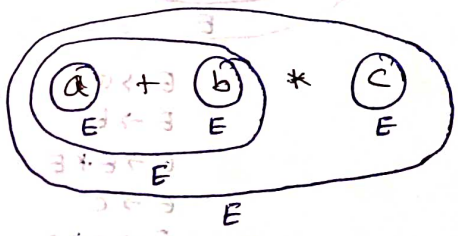
∴ The 1st handle found is a

later the handle is b

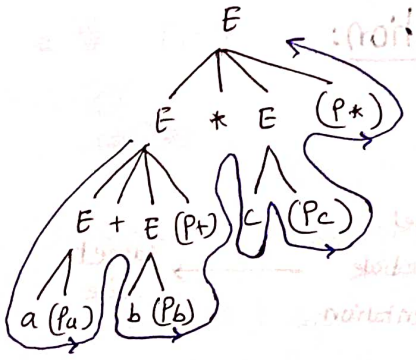
next it is E+E

then c

then E*E

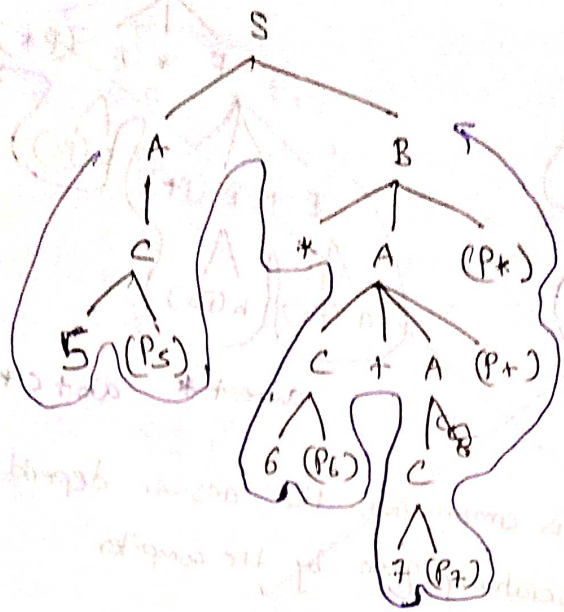


∴ The parse tree is



∴ opt (b)

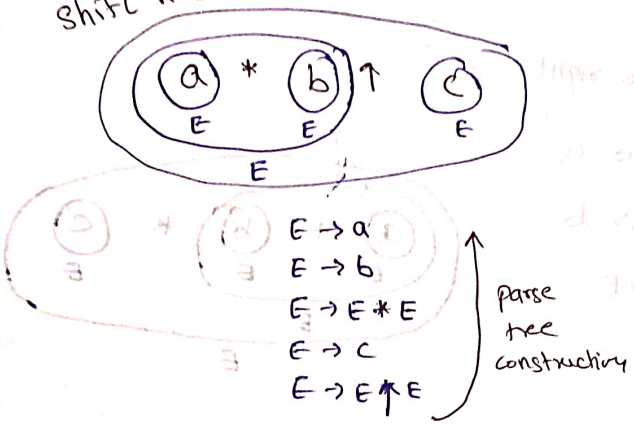
P/15

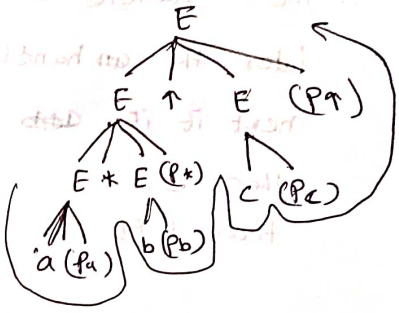


5 6 7 + *

P/16

Shift Reduce Parser





ab * c ↑

Intermediate Code Generation:

Source code → high level intermediate representation (close to soure language)

Eg: Syntax tree, DAG, postfix

→ low level intermediate representation (close to target machine representation) → Target code

Eg: TAC, SSA

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

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

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

∴ not SSA but TAC

Static Single Assignment:

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

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

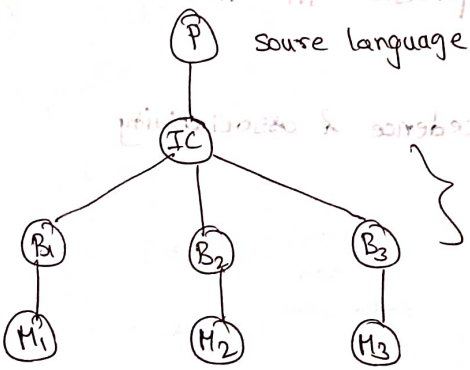
a) make parsing and semantic analysis simpler.

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.

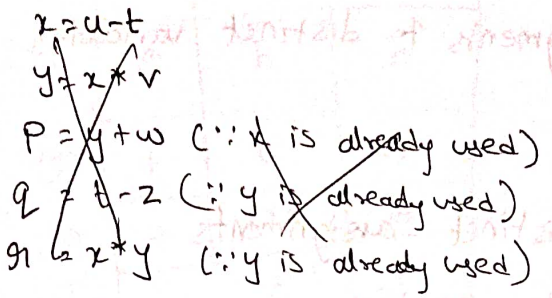
∴ opt (c)

Q57 / G-16 Consider the following code segment

```
x = u - t;
y = x * v;
x = y + w;
y = t - z;
y = x * y;
```

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

Sol :



$T_1 = u - t$

$T_2 = T_1 * y;$

$x = T_2 * w;$ (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

∴ Ans : 3 (The question is asked variable required for conversion process)

Q58 / 4-19 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) $p_1 = a - b$

$q_1 = p * c$

$q_2 = u * v$

$q_2 = p + q$

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

	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)

Eg: Write TAC for below C code

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

TAC:

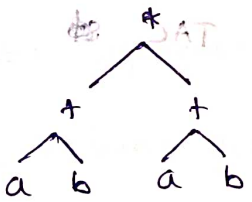
1. i=0
2. if i<10 goto 4
3. goto 7
4. x=y+z
5. i=i+1
6. goto 2
7. ---- (remaining code)

These goto stmts addresses are filled by a process called backpatching

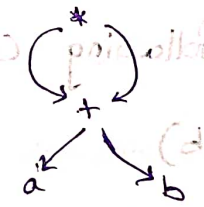
Directed Acyclic Graph (DAG)

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

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

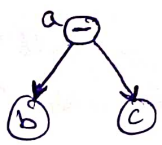
∴ a-c is common sub expression.

Construction of DAG for above stmt:

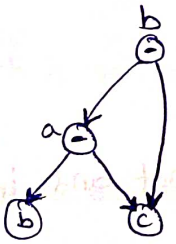
- For given code from top to bottom create a node for every new memory reference.

  If reference already exists used existing one (latest one)

a = b - c

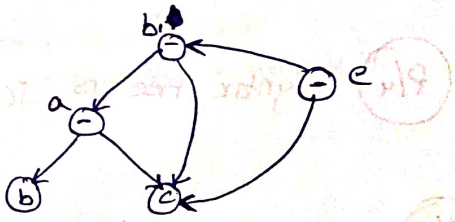


b = a - c

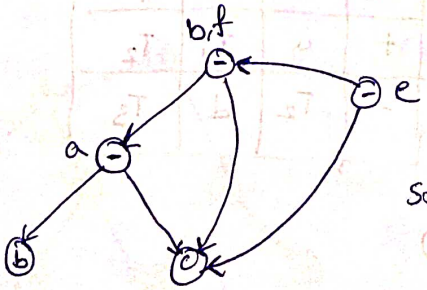


e = b - c

Here we consider latest modified b.



f = a - c



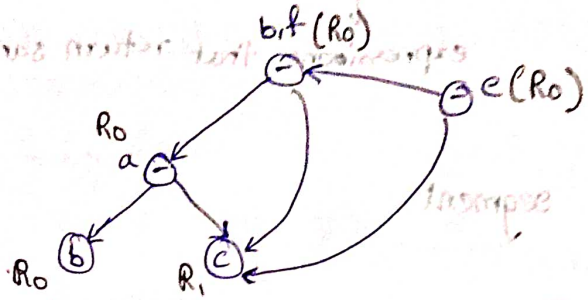
So Here

b = a - c

f = a - c

are identified as common sub expressions.

Finding minimum number of registers required from the 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 R_0 & R_1

For 'a' we store in R_0

After that b,f in R_0

and later e in R_0

Material Problems on IC

(P/1) same as (Q56)

(P/2) 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 copy of TAC

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

$T_1 = b * c$

$T_2 = a + T_1$

$T_3 = T_2 - d$

op	arg1	arg2	Res
*	b	c	T_1
+	a	T_1	T_2
-	T_2	d	T_3

∴ opt (b)

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

Parsalysis 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

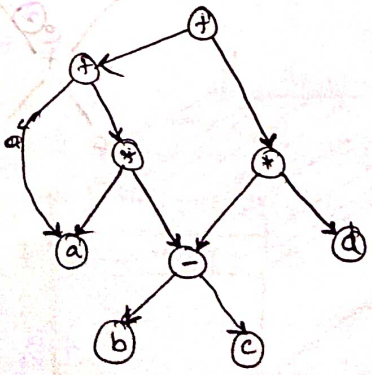
eg: a = a

This can be eliminated (not needed for gate optimization technique)

c) helps with using min of registers

∴ none

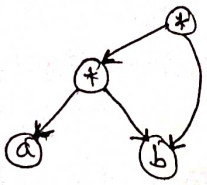
P/11 P/12



(b-c) is evaluated first

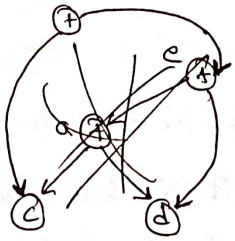
P/11

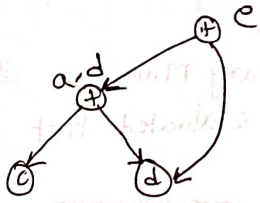
a * b + b



∴ opt (b)

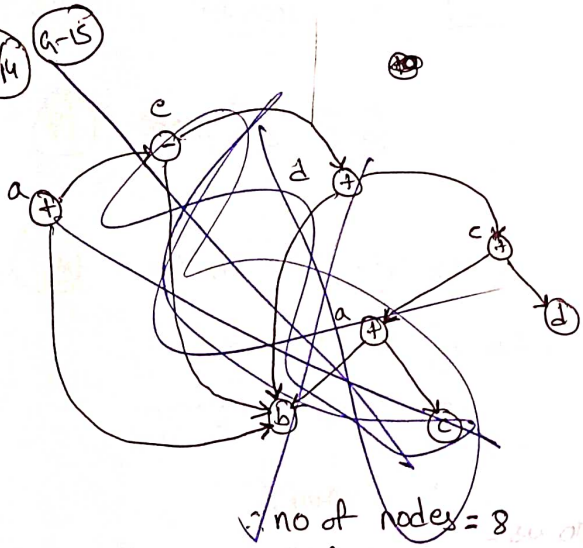
P/13





∴ 4 nodes

P/14 G-15

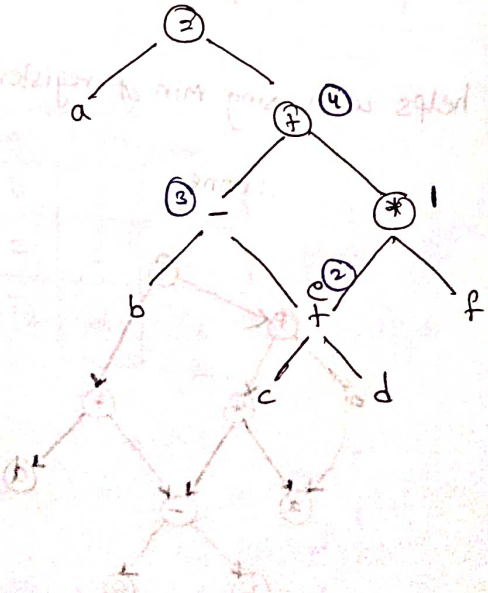


∴ no of nodes = 8

no of edges = 10

P/16

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



P/15T
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
```

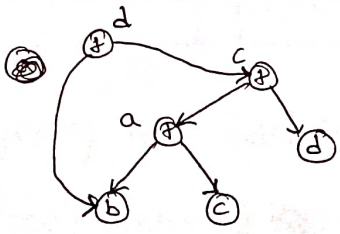
(From substitution
a = e+b
= d-b+b
= d)

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

⇒

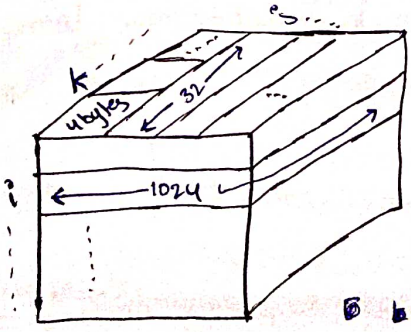
```
a = b+c
c = a+d
d = b+c
e = c      } Dead code because they have no further references
a = d      }
```

∴
```
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
$= \frac{1024}{32} = 32$

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

j value can't be determined

Each k is 4 bytes ⇒ int array
int a[ ][32][8] (from this opt ⓐ)