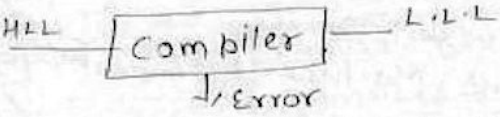
Compiler (4-6) Marks

12/09/19

1. Lexical Analysis → 10%.
2. Parsing → 50%.
3. Syntax Directed Translation → 20%.
4. Intermediate code generation } → 20%.
5. Runtime Environment.

Def^n:→ Compiler is a program that translates a program written in one language into an equivalent program of other language. also detects errors present in a program.

⇒ Errors detected by a compiler of 3 types know as Lexical errors, Syntax Errors, & Semantic Errors.



Lexical Analysis :→ It takes sequence of characters as input and produces token as output. and also detects Lexical errors present in the program.

⇒ To design this phase R.E (Regular expression) and F.A (Finite Automata) Mathematical technique are used.

Syntax Analysis :→ It takes token as input and produces parse tree as output and also detects Syntax errors present in the token.

⇒ To design this phase Contex Free grammer & Push-down Automata mathematical technique are used.

Semantic Analysis :→ It take parse tree as input and produces Annotated parse tree as output also detects Semantic errors.

⇒ Semantic error means meanning less statements present in the program.

Intermediate code generation :→ Translates High level language (HLL) into Three-address code

⇒ Advantage of generating intermediate code is to perform Optimization.

⇒ Intermediate code is Machine independent code.

⇒ To design intermediate code generation Syntax Directed translation Mathematical technique is used.

⇒ To design Semantic Analysis Contex Sensitive grammer are used.

Code Optimization : It translates intermediate code into optimized intermediate code, by removing unnecessary information present in the program.

⇒ This phase reduces time and space required for the target Machine.

⇒ There are 2 types of optimization performed by compilers know as Machine independent optimization and Machine dependent optimization.

⇒ Optimization performed on intermediate code is kla Mlc independent optimization.

⇒ Optimization performed on Assembly language is kla Machine dependent optimization.

Code Generation :→ It translates optimized intermediate code into Assembly language.

⇒ This phase is Machine dependent phase of the compiler.

Front END :→ The phases of compiler which are depending on source language and independent on target Machine kla Front END of compiler.

⇒ Front End phases include lexical analysis, Syntax analysis, Semantic analysis, & intermediate code generation.

Back End :→ The phases of compiler which are dependent on target Machine and independent on source language.

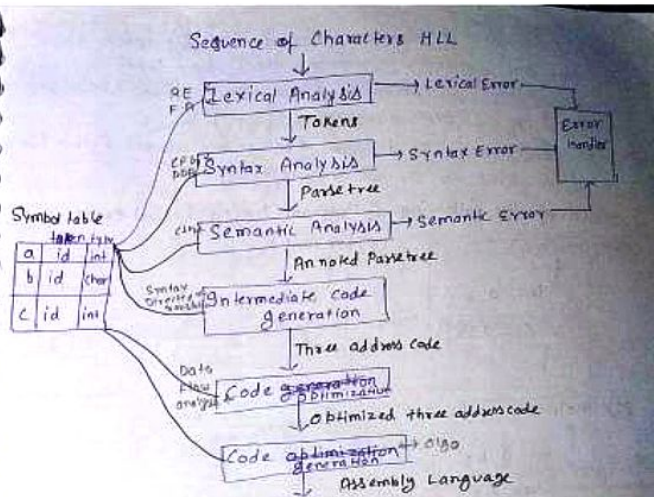
⇒ Back end phase includes code generation phase.

Symbol table :→ It is a data structure that stores information about identifiers and constant present in the program.

⇒ To design symbol table Hash table data structure is used.

⇒ If any phase of the compiler detects error, that information is send to error handler.

⇒ Error handler return error information to the programmer.



Lexical Analysis

• **Token** :→ Sequence of characters in the source program which makes a valid string know as token.

Following are tokens of programming languages

1. Keyword
2. Identifier
3. Operator
4. Constant
5. Special Symbol

• **Lexeme** :→ Sequence of characters in the source program matching with rules of the token kla Lexeme.

⇒ Any sequence of characters in the source program not a Lexeme then Lexical error is generated.

• **Designing of Lexical Analysis**

To design Lexical analyser corresponding to every token regular expression and F.A is constructed.

⇒ If any sequence of character are accepted by F.A then token is produced other wise lexical error is generated.

⇒ If any sequence of character are accepted by white space automata or comment automata then No token is produced. Hence all comment lines and white spaces are removed by Lexical analyser.

⇒ Lexical analyser can't detected Syntax and Semantic errors hence tokens are send to Syntax analysis phase.

Q. Which of the following C Program having Lexical error.

(a)
```
main()
{
  int a, b;
  c = a+b;
}
```

(b)
```
main()
{
  int a,b,a,b;
}
```

(c)
```
main()
{
  int a, b;
  char c;
  a = b+c;
}
```

(d) None

All are compiler time error not Lexical error.

Q. How many token Produce by lexical analyser

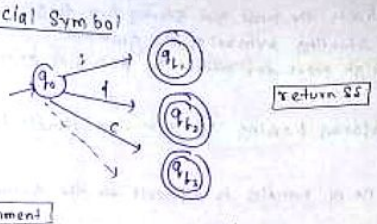
```
main()
{
int a,b;
f1 /*comment*/ out z;
a = b+z; /*gate*/
b= /*exam*/ 20;
}
```

Ans:- 29

```
main()2
{int a,b,c; Ans=31}
a=b+c;
a=b+c;
a=b;
a=b;
```

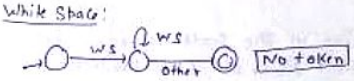
printf("Compiler"); → 5

Special Symbol



Comment

White Space:



17-Aug-19

Syntax :→ Structure of program Statements is know as Syntax.

Parsing :→ Checking whether given string is member of given gramm or not know as Parsing (Membership problem of CFG)

- The program that performs Parsing is know as Parser or Syntax Analyzer.
- Parser taken token and CFG as input and verified token are member of CFG or not.
- If token are member Parser produces parse tree as o/p, otherwise Syntax Error is generated.
- To design Syntax analysis phase CFG and PDA Mathematical Model are used.
- Syntax of programming language is represented by CFG.
- Deterministic PDA is used to design Parser.
- There are 2 type of Parsing algorithm exists to design Parser i/p
  (i) Top-down Parsing algo.
  (ii) Bottom up Parsing algo.
- To design Parser Unambiguous CFG are used.

Bottom-Up Parsing

- Bottom up parser constructs the parse tree staring from the given string and proceeds towards starting symbol of the grammar.
- Bottom up parser uses Right most derivation in Recursive order to construct parse tree.
- Bottom up parser performs parsing by detecting Handle present in the i/p.
- Detection of proper no of handles is difficult for the bottom up parser.
- There are two algorithm in bottom up parsing know as LR-Parsing algorithm and operator precedence parsing algorithm.

Handle:- Handle is a substring in the sentential form that completely matches with R.H.S parts of CFG production.
⇒ Whenever handle is detected it is replaced by corresponding LHS Non-terminal.

Q. How many total no of handles detected by Bottom up parsing parser while performing parsing by using following string and grammar?

(i) xxxyzz

S → xxw
S → y
w → Sz

(ii) n+n+n

E → E+n / Exn / n

(iii) abbcde

S → aABe
A → Abc / b
B → d

(iv) S → TL;
T → int
L → id / L, id

int id, id;

(v) S → As / AB
A → a
B → bC / dB
C → c

aaadbC

(i) S → xxw → xxSz → xxxxyzz → xxxxSzz → xxxxyzz
∴ 5 handles

(ii) E → E+n → E+n+n → n+n+n
∴ 3 handles

(iii) S → aABe → aAde → aAbcde → abbcde
∴ 4 handles

(iv) S → TL; → T id, id; → int id, id;
4 handles

(v) S → As → AAB → AAdB → AAdbC → AAdbc → Aadbc → aadbc
9 handles.

Q. S → aA / a
A → Sb
input : aab

S A
S a b
S a b
a b

(∵ Not reaches string Symbol not handles)

S → aA → aSb → aab
3 handles
(∴ These are the handles)

L-R Parsing Algorithm :-

- L-R Parser is Shift/Reduce parser because parsing is done with the help of following actions -

(i) **Shift Action:-** Shift action means push symbol from i/p buffer to Stack.

(ii) **Reduce Action:-** Reduce action means handle in the stack is replaced by corresponding L.H.S non terminals.

(iii) **Accept Action:-** Accept action means valid syntax hence parse tree is generated.

(iv) **Error action:-** Error action means invalid syntax hence, syntax error is generated.

(i) L-R parsing algorithm using stack for detection of handle.
(ii) All the token sequence are stored in input buffer along with $ as end marker.
(iii) L-R parser using parsing table which is divided into two parts known as Action table and Goto table.
(iv) Action table gives information regarding action to be performed by the parser.
(v) Goto table gives information regarding DFA next state.
(vi) Output of L-R parser is parse tree for valid syntax or syntax error for invalid syntax.
(vii) Let 's' is DFA state in the top of the stack and 'a' is look ahead symbol in the i/p buffer then parser takes decision from parsing table as follows-

(a) T[s,a] = accept then i/p string is valid and parse tree is produced as o/p.
(b) T[s,a] = Blank, then i/p string is invalid and syntax error is generated.
(c) T[s,a] = Shift j
  (i) Push 'a' into stack
  (ii) Push j into top of the stack
  (iii) and increment look ahead symbol.

(d) T[s,a] = reduce A→B
  (a) pop 2*|B| symbol from stack.
  (b) push A into the stack
  (c) then push goto(i,A) into top of the stack where i is previous DFA state in the stack.

- Time complexity of L-R parsing algorithm is O(n) where n is length of the input.

Q. How many shift action and reduce actions are taken by L-R parser the input "ab" by using following grammar and parsing table?

1. S→AB
2. A→a
3. B→b

Action

	a	b	$
I_0	S_3		
I_1			accept
I_2		S_5	
I_3		r_2	
I_4			r_1
I_5			r_3

Goto

S	A	B
1	2	
		4

	Stack	input	Action
①	$0	ab$	[0,a] = S_3
②	$0a3	b$	[3,b] = r_2 (A→a)
③	$0A2	b$	[2,b] = S_5
④	$0A2b5	$	[5,$] = r_3 (B→b)
⑤	$0A2B4	$	[4,$] = r_1 (S→AB)
⑥	$0S1	$	[1,$] = accept

∴ No of shift operation = 2
No of reduce operation = 3

Construction of LR Parsing table :-

To Construct L-R parsing table 4 methods exist known as:-

1. LR(0)
2. SLR(1)
3. CLR(1)
4. LALR(1)

- LR(0) parser is least powerful and CLR(1) parser is most powerful

LR(0) parsing table construction :-

Step 1:- Construct augmented grammar by adding new production. ($S' \to S$)

Step 2:- Construct LR(0) item DFA by using closure() and goto().

- LR(0) item is 'CFG' production having '.' Somewhere in the RHS
- The LR(0) item $A \to xyz$ gives meaning to the parser XYZ present in the stack whence parser can perform reduce action.
- The LR(0) item $A \to xy.z$ gives meaning to the parser 'xy' present in the stack and 'z' not present in the stack. Hence parser can't perform reduce action.
- Total no of LR(0) items corresponding to given grammar are calculated by closure() and goto().
- Even though we can not construct DFA for CFG but we can construct DFA for LR(0) items.
- For the given grammar, total no. of LR(0) items are always finite hence regular (Hence DFA possible)
- The task of closure function is in any LR(0) item '.' followed by non-terminal exist then closure function adds that non-terminal production and places '.' in the first position.
- The task of goto() is it forwards '.' one position towards right side.

Step 3:- Convert DFA transition diagram into transition table (or) parsing table.

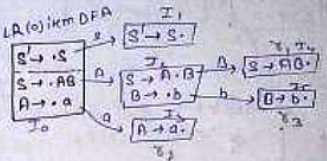
Q Construct LR(0) parsing table for the following grammar.

1. $S \to AB$
2. $A \to a$
3. $B \to b$

LR(0) item

$S \to .AB$
$S \to A.B$
$S \to AB.$
$A \to .a$
$A \to a.$
$B \to .b$
$B \to b.$

LR(0) item DFA



LR(0) Table

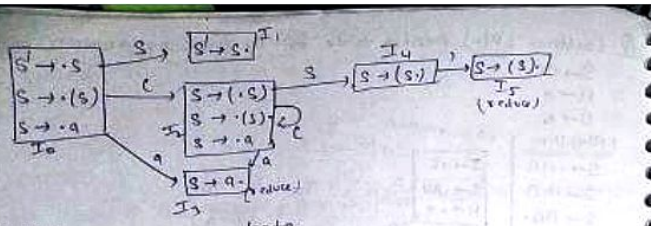
State of DFA	Action (all terminal) a	b	$	GoTo (all Non-terminal) S	A	B
I_0	S_3			1	2	
I_1			Accept			
I_2		S_5				4
I_3	r_2	r_2	r_2			
I_4	r_1	r_1	r_1			
I_5	r_3	r_3	r_3			

Parsing table

Q. Construct LR(0) parsing table for the following grammar.

1. $S \to (S)$
2. $S \to a$

Q Verify given grammar is LR(0) grammar or not!!



Action					Goto
	(	)	a	$	S
I_0	S_2		S_3		1
I_1				accept	
I_2	S_2		S_3		4
I_3	R_2	R_2	R_2	R_2	
I_4		S_5			
I_5	R_1	R_1	R_1	R_1	

Note:- In any LR(0) items DFA, transitions labelled with terminal symbols are shift actions

- In LR(0) item DFA, transition labelled with non terminals are goto table entries.
- In LR(0) items DFA, any state contains complete LR(0) item (∵ at right most end) is reduce action.
- Completed argument production is accept action.

LR(0) grammar detection:-

Any CFG, LR(0) parsing table contains shift/reduce conflict or Reduce/Reduce conflict then that grammar is not LR(0) grammar. Otherwise given grammar is LR(0).

- Every ambiguous grammar LR(0) parsing table should contains shift/reduce conflict or Reduce/Reduce conflict hence every ambiguous grammar is not LR(0) grammar.
- Every LR(0) grammar generates deterministic CFL.

LR(0) grammar

Following are conflict state or inadequate state for LR(0) parser

S/R conflict: $A \to \alpha \cdot a X$ (terminal), $B \to \gamma \cdot$

R/R conflict: $A \to \gamma \cdot$, $B \to \gamma \cdot$

LR(0) items DFA contains S/R conflict state or R/R conflict state then given is not LR(0) grammar.

Q. Which of the following grammar is LR(0) grammar?

(a) $E \to T + E$, $E \to T$, $T \to i$

(b) $S \to Aa$, $S \to Bb$, $A \to d$, $B \to d$

(c) $S \to AA$, $A \to aA$, $A \to a$

(d) $E \to E + n$, $E \to n$

Solution

(a) $E' \to \cdot E$, $E \to \cdot T + E$, $E \to \cdot T$, $T \to \cdot i$

- E → $E' \to E \cdot$
- T → $E \to T \cdot + E$, $E \to T \cdot$ → S/R conflict
- i → $T \to i \cdot$

∴ Not LR(0) grammar

(b) $S' \to \cdot S$, $S \to \cdot A a$, $S \to \cdot B b$, $A \to \cdot d$, $B \to \cdot d$

- S → $S' \to S \cdot$
- A → $S \to A \cdot a$
- B → $S \to B \cdot b$
- d → $A \to d \cdot$, $B \to d \cdot$ → R/R conflict

∴ Not LR(0) grammar.

(c) $S' \to \cdot S$, $S \to \cdot AA$, $A \to \cdot aA$, $A \to \cdot a$

- S → $S' \to S \cdot$
- A → $S \to A \cdot A$
- a → $A \to a \cdot A$, $A \to a \cdot$, $A \to \cdot aA$, $A \to \cdot a$ → S/R conflict.

Conflict occur due to colour due to goto

(Not LR(0) grammar)

(d) $E' \to \cdot E$, $E \to \cdot E + n$, $E \to \cdot n$

- E → $E' \to E \cdot$, $E \to E \cdot + n$ → $E \to E + \cdot n$ → $E \to E + n \cdot$
- n → $E \to n \cdot$

No conflict

∴ LR(0) grammar.

Note:- The LR(0) item $[A \to \cdot\varepsilon]$ is equal to $[A \to \varepsilon\cdot]$ 'ε' transition are not allowed in LR(0) items DFA. Hence LR(0) item corresponding to 'ε' are reduced items in the given state.

Q. Which of the following grammar is LR(0) grammar?

① $S \to aSa$, $S \to bSb$, $S \to \varepsilon$

② $S \to AaAb$, $S \to BbBb$, $A \to \varepsilon$, $B \to \varepsilon$

③ $S \to aAb$, $A \to Aa$, $A \to \varepsilon$

Sol^n

① $S' \to \cdot S$, $S \to \cdot aSa$, $S \to \cdot bSa$, $S \to \cdot\varepsilon$ → $S' \to S\cdot$

S/R conflict. ∴ not LR(0)

(ii) $S' \to \cdot S\cdot$, $S \to \cdot AaAb$, $S \to \cdot BbBb$, $A \to \varepsilon\cdot$, $B \to \varepsilon\cdot$ → R/R conflict

(iii) $S' \to S\cdot$, $S \to A\cdot$

$S' \to \cdot S$, $S \to \cdot AAb$ —S→ $S' \to S\cdot$ —a→ $S \to a\cdot Ab$, $A \to \cdot Aa$, $A \to \varepsilon\cdot$ —A→ $S \to aA\cdot b$, $A \to A\cdot a$

$S \to aA\cdot b$ —b→ $S \to aAb\cdot$

$A \to A\cdot a$ —a→ $A \to Aa\cdot$

No conflict

∴ LR(0) grammar.

DRAWBACK OF LR(0) :-

LR(0) parsing table contains unnecessary reduce action hence Power of LR(0) parser is less than SLR(1) parser.

Q. Construct the following augmented grammar.

$S' \to S$, $S \to \langle L \rangle$, $S \to id$, $L \to L, S$, $L \to S$

How many LR(0) items exist in the state construction by using goto $(I_0, \langle)$? 5

$S' \to \cdot S$, $S \to \cdot\langle L\rangle$, $S \to \cdot id$ —<→ $S \to \langle\cdot L\rangle$, $L \to \cdot L, S$, $L \to \cdot S$, $S \to \cdot\langle L\rangle$, $S \to \cdot id$

SLR(1) [Simple LR(1)]

First set Analysis

- First set of a non-terminal is set of terminals which appears in the first position on the grammar R.H.S part.

① first (abc) = {a}

② $A \to a\alpha \mid b\gamma \mid \varepsilon$, First(A) = {a, b, ε}

③ $A \to B\alpha$, $B \to \varepsilon$, First(A) = First(B)

④ $A \to B\alpha$, $B \to \varepsilon$, First(A) = {First(B) − ε} ∪ {First(α)}

Q. Calculate first set of each non-terminals for the given grammar.

① $S \to AB$, $A \to a \mid bc$, $B \to d \mid \varepsilon$

② $S \to AB$, $A \to a$, $B \to b \mid \varepsilon$

③ $S \to AB$, $A \to \varepsilon$, $B \to d$

④ $S \to AB$, $A \to a \mid \varepsilon$, $B \to d \mid \varepsilon$

⑤ $S \to aAbB \mid bAaB \mid \varepsilon$, $A \to BS$, $B \to BS$

⑥ $S \to AeB \mid cb \mid Ba$, $A \to da \mid Bc$, $B \to g \mid \varepsilon$, $c \to h \mid \varepsilon$

Sol^n

① | | |
|---|---|
| S | {a, b} |
| A | {a, b} |
| B | {d, ε} |

② | | |
|---|---|
| S | {a} |
| A | {a} |
| B | {b, ε} |

③ | | |
|---|---|
| S | {d} |
| A | {ε} |
| B | {d} |

④ | | |
|---|---|
| S | {a, d, ε} |
| A | {a, ε} |
| B | {d, ε} |

⑤ | | |
|---|---|
| S | {a, b, ε} |
| A | {a, b, ε} |
| B | {a, b, ε} |

⑥ | | |
|---|---|
| S | {d, a, g, h, b, a, ε} |
| A | {d, a, g, h, ε} |
| B | {g, ε} |
| C | {h, ε} |

Follow Set Analysis :-

- Follow of non-terminal is set of terminals which appears immediately R.H.S part of that non-terminal in some sentential form.
- Follow set doesn't contain 'ε'.

① Follow(S) = {$}

② A → αBX
Follow(B) = First(X)

③ A → αB
Follow(B) = Follow(A)

④ A → αBX , X → ε
Follow(B) = {First(X) − ε} ∪ follow(A)

Q. Calculate follow set of each non-terminal for the given grammar.

ⓐ S → AB
A → a
B → b

ⓑ S → aSa | bSb | ε

ⓒ S → aSbS | bSaS | ε

ⓓ S → AaAb | BbBa
A → ε
B → ε

ⓔ S → ACB | Cb | Da
A → da | BC
B → g | ε
C → h | ε

Sol^n

ⓐ

	follow
S	{$}
A	{b}
B	{$}

ⓑ

	follow
S	{a, b, $}

ⓒ

	follow
S	{a, b, $}

ⓓ

	follow
S	{$}
A	{a, b}
B	{a, b}

ⓔ

	follow
S	{$}
A	{h, g, $}
B	{a, h, g, $}

ⓕ E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | a

	follow
E	{), $}
E'	{), $}
T	{+,), $}
T'	{+,), $}
F	{*, +,), $}

ⓖ S → aSN | ε
A → c | ε

	follow
S	{$, c}
A	{$, c}

ⓗ S → aABb | bAaB | ε
A → ε
B → ε

	follow
S	{$, b, a}
A	{b, a}
B	{$, b, a}

ⓘ S → iEtSS' | a
S' → eS | ε
E → b

	follow
S	{$, e}
S'	{e, $}
E	{t}

ⓙ S → Aa
A → BD
B → b | ε
D → d | ε

follow(B) = {d, a}

10. Construct SLR(1) parsing table for the following grammar.

1. E → T + E
2. E → T
3. T → i

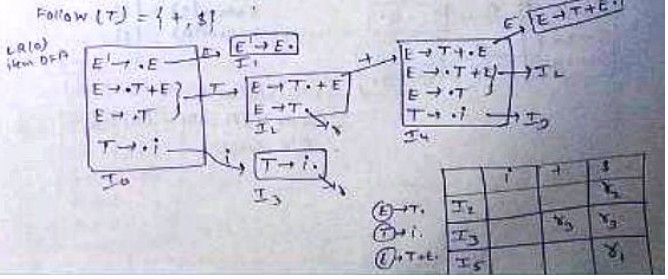
2 Q) Verify given grammar is SLR(1) grammar or not.

- To construct SLR(1) parsing table LR(0) item DFA is used.
- Hence no of Shift action and GoTo table entries are same in both tables.
- No. of reduce actions may be different in both tables
- In SLR(1) parsing table, reduce actions are placed based on follow set of values of LHS non-terminal.

Sol^n

Follow(E) = {$}
Follow(T) = {+, $}

LR(0) item DFA



	i	+	$
I_2			r_2
I_3			r_3
I_5			r_1

(E → T.
(T → i.
(E → T+E.

Q) Given grammar is SLR(1) grammar

- LR(0) items DFA contains Shift/Reduce conflict state or Reduce/Reduce Conflict state then grammar is not SLR(1) grammar.
- Every SLR(1) grammar is unambiguous grammar but every unambiguous grammar need not be SLR(1) grammar.
- Every ambiguous grammar SLR(1) parsing table should contains either Shift/reduce conflict or reduce/reduce conflict.
- Following are the conflict states for SLR(1) Parser.

S/R conflict

(i) $A \to \alpha \cdot a\,X$ (Terminal)
$B \to \gamma\,.$

(ii) if $a \in$ Follow(B)

R/R conflict

(i) $A \to \alpha\,.$
$B \to \gamma\,.$

(ii) Follow(A) $\cap$ follow(B) $\neq \phi$

Q. Which of the following grammar is SLR(1) grammar.

(a) $S \to A$, $S \to a$, $A \to a$

(b) $S \to AA$, $A \to aA$, $A \to a$

(c) $S \to Aa$, $S \to Bb$, $A \to d$, $B \to d$

(a) $S' \to \cdot S$
$S \to \cdot A$
$S \to \cdot a$
$A \to \cdot a$

S: $S' \to S.$
A: $S \to A.$
a: $S \to a.$, $A \to a.$ → R/R conflict

Follow(S) = {$}
Follow(A) = {$}

∵ Follow(S) $\cap$ follow(A) = {$}
∴ R/R conflict.

(b)
$S' \to \cdot S$
$S \to \cdot AA$
$A \to \cdot aA$
$A \to \cdot a$

S: $S' \to S.$
A: $S \to A \cdot A$, $A \to \cdot aA$, $A \to \cdot a$
a: $A \to a \cdot A$, $A \to a.$, $A \to \cdot aA$, $A \to \cdot a$ → S/R conflict

Follow(A) = {a, $}

∴ a ∈ Follow(A).

(c)
$S' \to \cdot S$
$S \to \cdot Aa$
$S \to \cdot Bb$
$A \to \cdot d$
$B \to \cdot d$

S: $S' \to S.$
A: $S \to A \cdot a$
B: $S \to B \cdot b$
d: $A \to d.$, $B \to d.$

Follow(A) = {a}
follow(B) = {b}

∵ follow(A) $\cap$ follow(B) = ϕ
∴ No Conflict
∴ SLR(1) grammar

Q. Which of the following grammar are SLR(1)

(d) $S \to aSa$, $S \to bSb$, $S \to \varepsilon$

(e) $S \to AaAb$, $S \to BbBb$, $A \to \varepsilon$, $B \to \varepsilon$

(f) $S \to A$, $A \to AB$, $A \to \varepsilon$, $B \to aB$, $B \to b$

(d) ~~$S' \to \cdot Sa$~~
$S' \to \cdot S$
$S \to \cdot aSa$
$S \to \cdot bSb$
$S \to \varepsilon\,.$ → S/R conflict

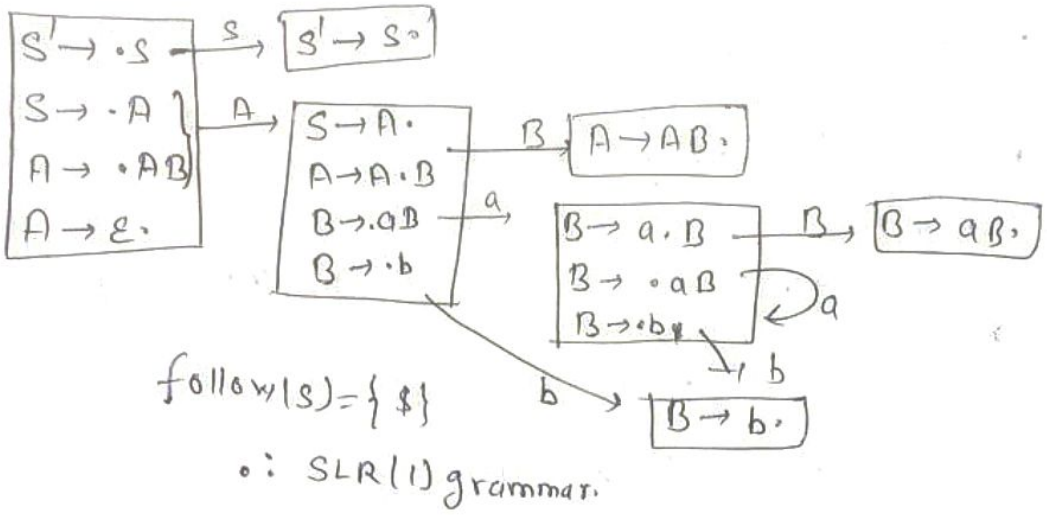
Follow(S) = {$, a, b}
{a, b} ∈ follow(S)
(∴ Not SLR(1) grammar)

(e)
$S' \to \cdot S$
$S \to \cdot AaAa$
$S \to \cdot BbBb$
$A \to \varepsilon\,.$
$B \to \varepsilon\,.$ → R/R conflict

Follow(A) = {a, b}
follow(B) = {a, b}
∴ (Not SLR(1) grammar)

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

(3)



follow(S) = {$}

∴ SLR(1) grammar.

Drawback of SLR(1):-

SLR(1) parsing table contains unnecessary reduce actions compared to CLR(1) parser. Hence, CLR(1) parser is more powerful than SLR(1) parser.

CLR(1) Parser

Canonical LR(1) items

- To construct CLR(1) parser LR(1) DFA ~~item~~ is used.
- LR(1) item is LR(0) item associated with Look ahead Symbol.
- Calculation of Look ahead Symbols for LR(0) item is done by closer f^n.

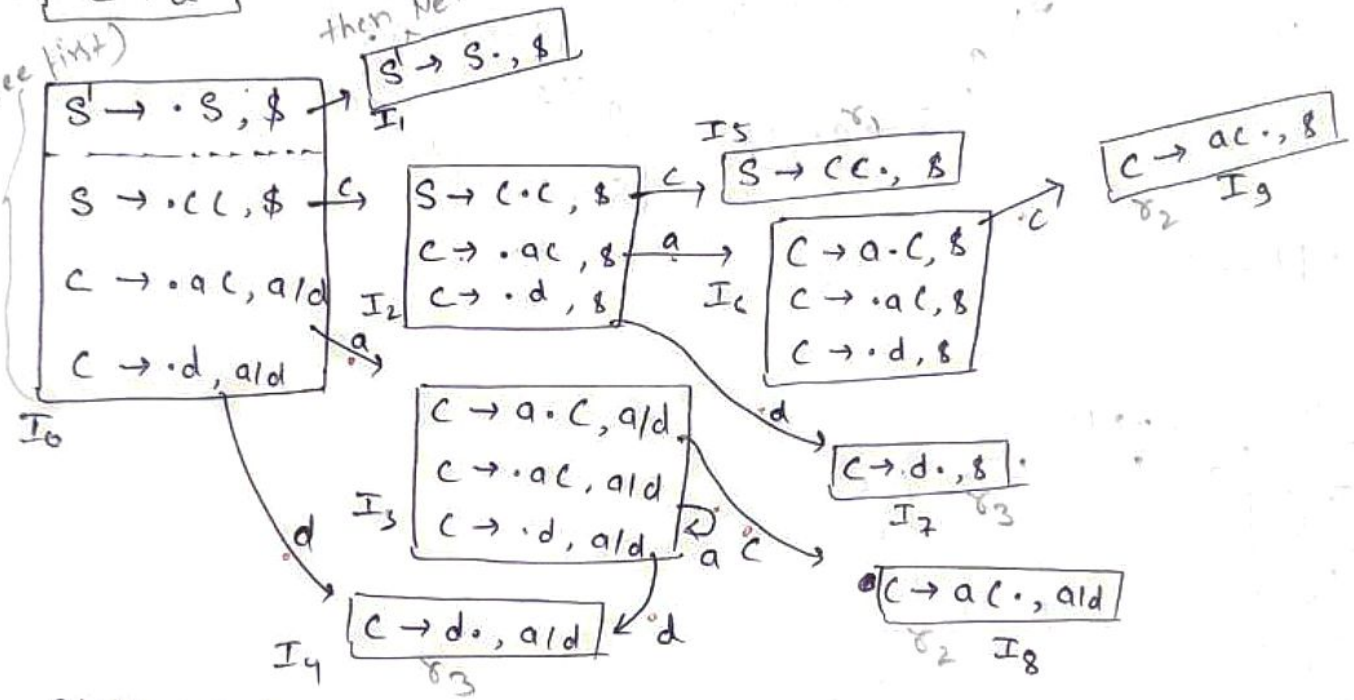
Q Construct CLR(1) Parsing table for the following Grammar.

LR(1) → LR(0) + Lookahead

(In closure only see first)
(Goto Lookahead is Move Same)

S' → S.

1. S → CC
2. C → aC
3. C → d



Shift → 8 (transaction level with terminal)

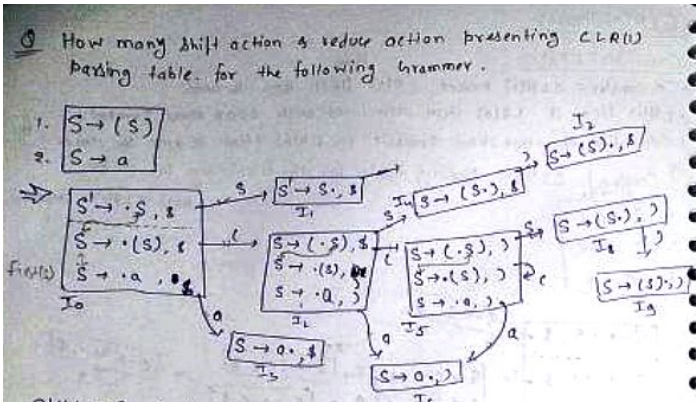
Goto → 5 (" " " Non terminal)

	a	b	$
I_4	r_3	r_3	
I_5			r_1
I_7			r_3
I_8	r_2	r_2	
I_9			r_2

Note → In CLR(1) Parsing table reduce action are placed by using look ahead Symbol of completed LR(1) item.

Q How many shift action & reduce action presenting CLR(1) parsing table. for the following grammer.

1. $S \to (S)$
2. $S \to a$



Shift → 8, goto table → 3

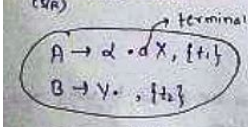
Reduce table → 4

	a	(	)	$
I_3				r_2
I_6			r_2	
I_7				r_1
I_9			r_1	

CLR(1) Grammer Detection

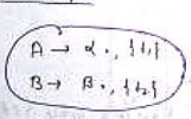
Following are the conflit state for CLR(1) parser.

Shift / Reduce Conflit (SR)



if $a \in \{t_2\}$

R/R confil.



$\{t_1\} \wedge \{t_2\} \neq \phi$

- If LR(1) DFA contains as a Shift reduce conflit state or Reduce-Reduce conflit state then the given gram mar is NOT CLR(1) Grammar.
- Every CLR(1) grammar is Unambiguous Grammar, but every Unambiguous grammar need not be CLR(1)
- Any ambiguous grammar CLR(1) parsing table should contains either S/R conflit or R/R conflit

Q. Which of the following grammar is NOT CLR(1) grammar.

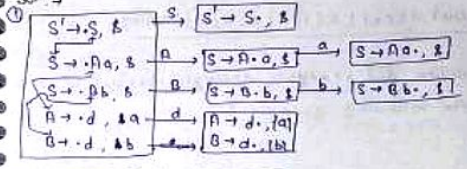
(1) $S \to Aa$, $S \to Bb$, $A \to d$, $B \to d$ — CLR(1)

(2) $S \to a$, $S \to ab$ — CLR(1)

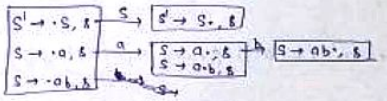
(3) $S \to A$, $A \to AB$, $A \to \varepsilon$, $B \to aB$, $B \to b$ — CLR(1)

(4) $S \to A$, $S \to a$, $A \to a$ — Not CLR(1)

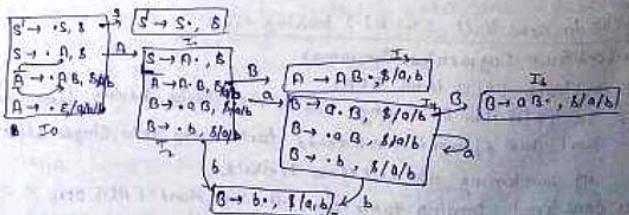
Soln →

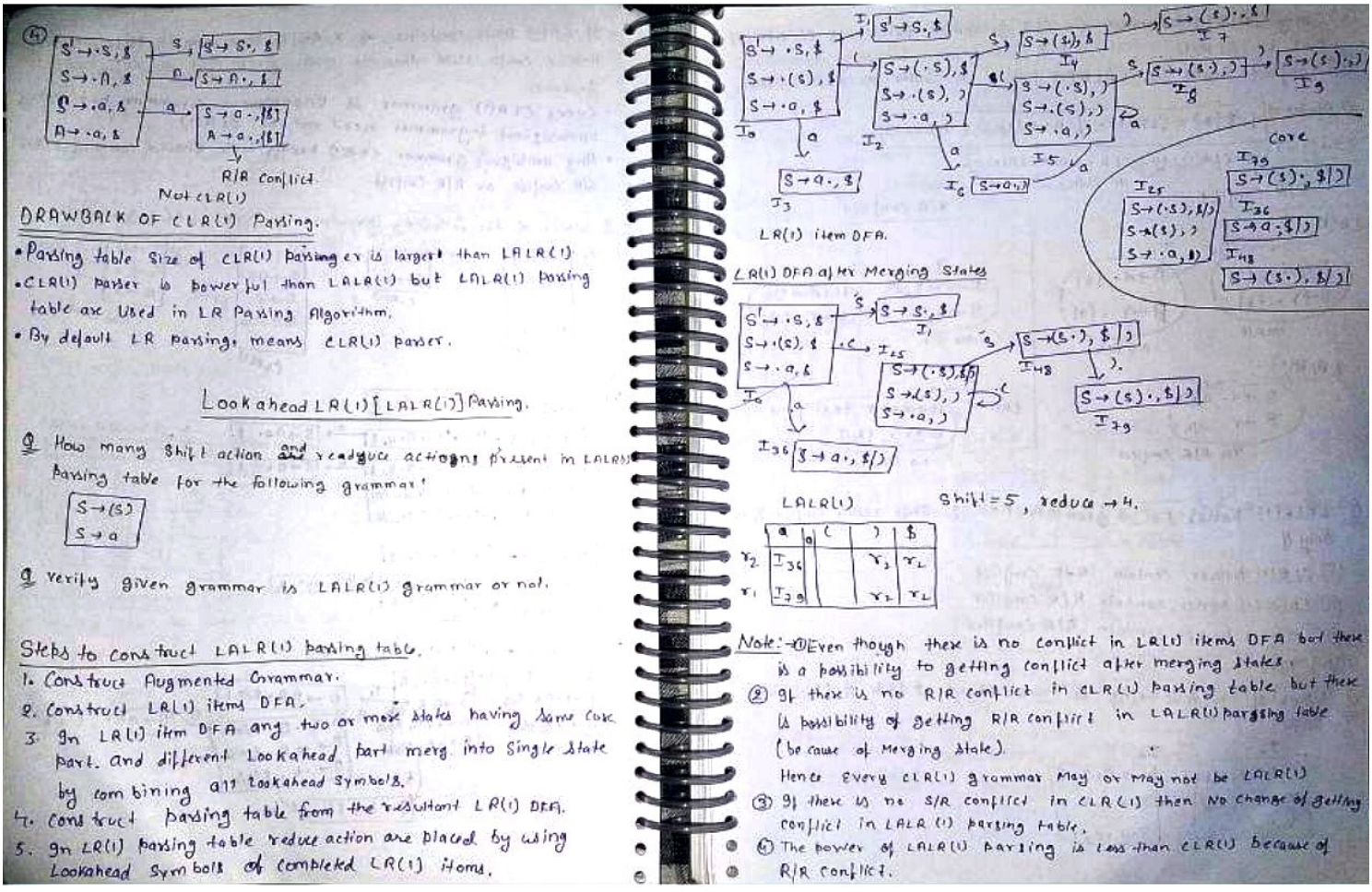


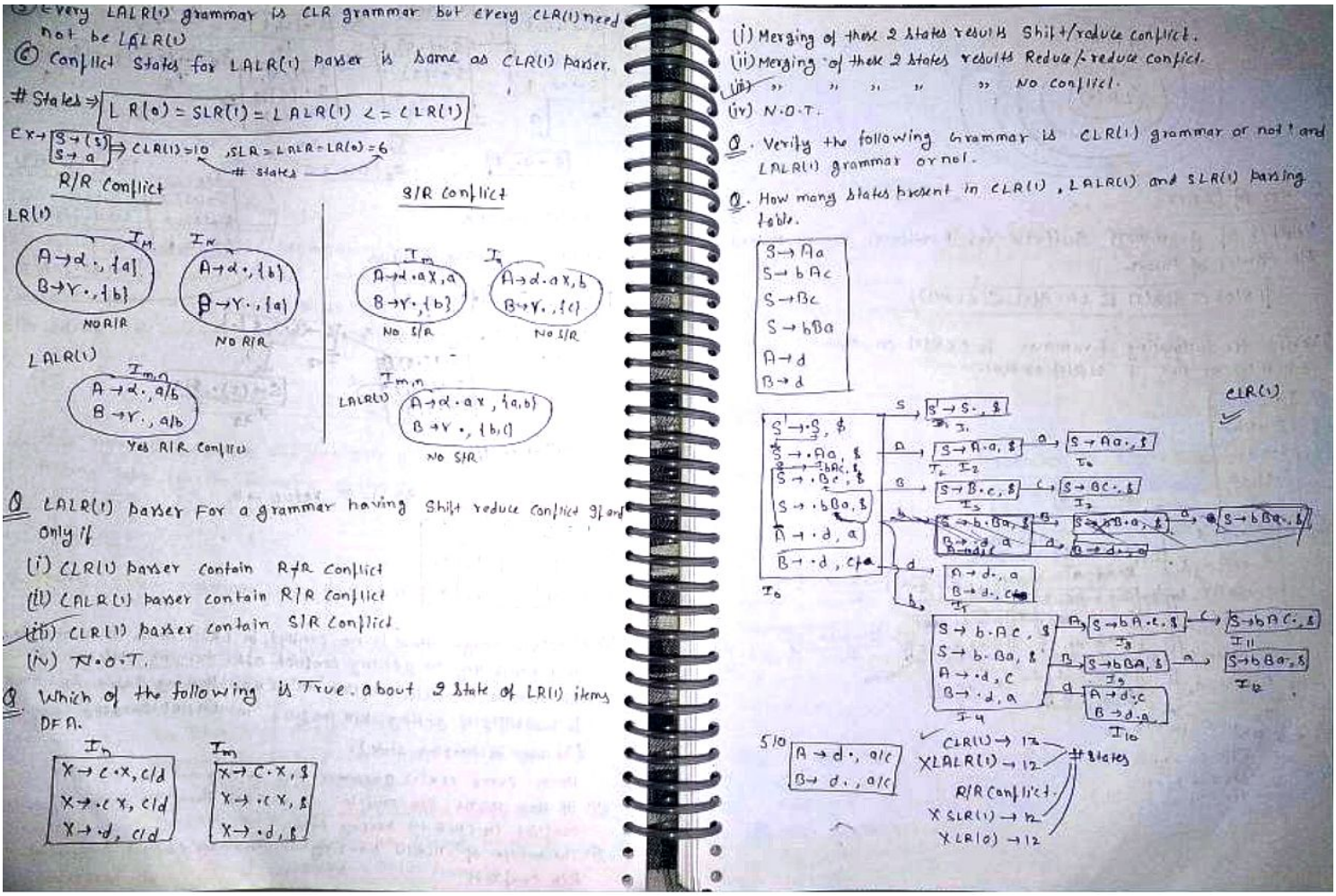
(2)



(3)




S' → .S, $
S → .A, $
S → .a, $
A → .a, $
S → S., $
S → A., $
S → a., $
A → a., $
R/R conflict
Not CLR(1)
DRAWBACK OF CLR(1) Parsing.
• Parsing table size of CLR(1) parsing er is largest than LALR(1)
• CLR(1) parser is powerful than LALR(1) but LALR(1) parsing table are used in LR Parsing Algorithm.
• By default LR parsing means CLR(1) parser.
Lookahead LR(1) [LALR(1)] Parsing.
Q How many shift action and readyuce actions present in LALR(1) parsing table for the following grammar!
S → (S)
S → a
Q Verify given grammar is LALR(1) grammar or not.
Steps to construct LALR(1) parsing table.
1. Construct Augmented Grammar.
2. Construct LR(1) items DFA.
3. In LR(1) item DFA any two or more states having same core part and different Lookahead part merg into single state by combining all lookahead symbols.
4. Construct parsing table from the resultant LR(1) DFA.
5. In LR(1) parsing table reduce action are placed by using Lookahead symbols of completed LR(1) items.
S' → .S, $
S → .(S), $
S → .a, $
S' → S., $
S → (.S), $
S → .(S),)
S → .a,)
S → a., $
S → (S.), $
S → (S), $
S → (S.),)
S → (S),)
S → a.,)
S → (S)., $
S → (S).,)
LR(1) item DFA.
LR(1) DFA after Merging States
S' → .S, $
S → .(S), $
S → .a, $
S → S., $
S → (.S), $/)
S → .(S),)
S → .a,)
S → (S.), $/)
S → a., $/)
S → (S)., $/)
LALR(1)
Shift = 5, reduce → 4.
Note: ① Even though there is no conflict in LR(1) items DFA but there is a possibility to getting conflict after merging states.
② If there is no R/R conflict in CLR(1) parsing table but there is possibility of getting R/R conflict in LALR(1) parsing table (because of Merging state).
Hence every CLR(1) grammar may or may not be LALR(1)
③ If there is no S/R conflict in CLR(1) then no change of getting conflict in LALR(1) parsing table.
④ The power of LALR(1) parsing is less than CLR(1) because of R/R conflict.



Q. LALR(1) parser for a grammar having Shift reduce conflict iff and only if

(i) CLR(1) parser contain R/R conflict

(ii) LALR(1) parser contain R/R conflict

(iii) CLR(1) parser contain S/R conflict

(iv) N.O.T.

Q. Which of the following is True. about 2 state of LR(1) items DFA.

I_n

X→c·X, c/d

X→·cX, c/d

X→·d, c/d

I_m

X→c·X, $

X→·cX, $

X→·d, $

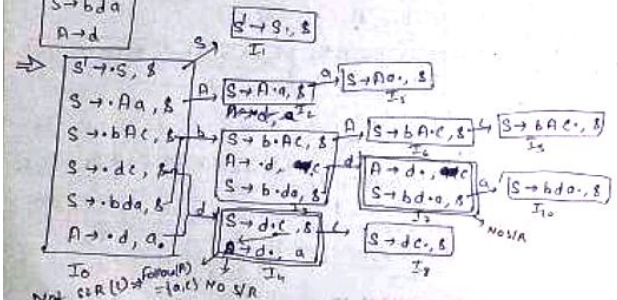
Power of Parser

Number of grammers suitable for particular parsing Method is/a Power of Parser.

$$LR(0) \subset SLR(1) \subset LALR(1) \subset CLR(1)$$

Q Verify the following Grammar is CLR(1) or Not. LALR(1) or not & SLR(1) or not.

$S \to Aa$
$S \to bAc$
$S \to dc$
$S \to bda$
$A \to d$



Not SLR(1) → Follow(A) = {a,c} No S/R

CLR (✓) → //
LALR (✓) → //
SLR (✗) → //

For SLR(1) remove Look Ahead Symbol.

In I_4 State

$S \to d \cdot c$
$A \to d \cdot$

follow(A) = {a, c}

∴ S/R conflict

Not SLR(1) grammar.

☆

	LR(0)	SLR(1)	CLR(1) & LALR(1)
S/R	$A \to \alpha \cdot a\chi$ $B \to \gamma \cdot$	$A \to \alpha \cdot a\chi$ $B \to \gamma \cdot$ if $a \in$ Follow(B)	$A \to \alpha \cdot a\chi$ $B \to \gamma \cdot, \{t_2\}$ if $a \in \{t_2\}$
R/R	$A \to \chi \cdot$ $B \to \gamma \cdot$	$A \to \chi \cdot$ $B \to \gamma \cdot$ Follow(A) ∧ Follow(B) ≠ ϕ	$A \to \chi \cdot, \{t_1\}$ $B \to \gamma \cdot, \{t_2\}$ $\{t_1\} \wedge \{t_2\} \neq \phi$

terminal

Note:→

Ambiguous Grammar in LR Parsing.

- If the Grammar is ambiguous then any LR parsing table should Contain S/R or R/R conflict
- We can resolve conflicts present in the parsing table by using Precedence and associativity of operators.
- Elimination of ambiguity is undecidable problem hence not eliminating ambiguity from the grammar but elimating conflicts present in the parsing table.

Q Resolve conflicts from following Ambiguous grammar.

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

→ ① S/R conflict at State I_5 on Lookahead Symbol "*" is resolved by giving priority for Shift action over reduce action inorder to make "*" is higher precedence than "+"

② S/R conflict at State I_5 on Lookahead Symbol '+' is resolved by giving priority for reduce action in order to make '+' is left associative.

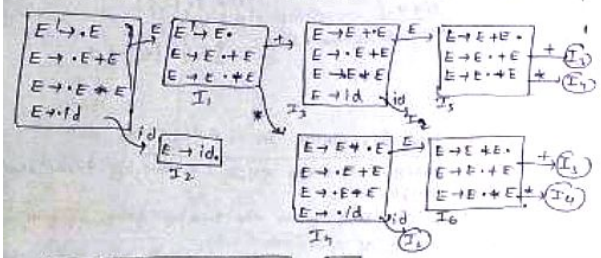
3). S/R conflict at State I_6 on lookahead symbol '+' is resolved by given priority for reduce action in order to make '*' is higher precedence than '+'

4). S/R conflict at State I_6 on lookahead symbol '*' is resolved by given priority for reduce action in order to make '+' is left associative.

⇒ The tool to design Lexical Analysis phase is "LEX"

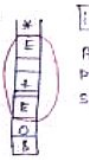
⇒ The tool to design LR Parser is "YACC" Yet Another Compiler Compiler.

⇒ "YACC" tool also using LALR(1) parsing table.



	id	+	*	$	E
I_0	S_2				
I_1		S_3	S_4	Accept	
I_2		r_3	r_3	r_3	
I_3	S_2				5
I_4	S_2				6
I_5		S_3/r_1	S_4/r_1	r_1	
I_6		S_3/r_2	S_4/r_2	r_2	

Ex



id + id * id

As * is having higher precedence then + shift.

Top-down Parsing

- Top-down Parser construct parse tree starting from starting Symbol of the grammar and proceeds towards given string.
- Top-down parser using left-most derivation to construct parse tree.
- If a Non-terminal having multiple productions on the Right hand side part then selecting correct production for tree construction is difficult for top-down parsing.
- There are two types of Top-down parsing Algo exist Ka
  1. with backtracking Algo → Brute force Parsing
  2. Predictive Parsing Algo (or without Backtracking) → LL(1) Parsing, Recursive Descent Parsing

Brute force Parsing :→

- Time complexity of Brute force Parsing is $O(2^n)$
- Brute force parsing requires exponential time complexity, hence not suitable for practical compiler.
- In Brute force parsing if a Non-terminal having multiple production then priority is given for first production
- If required string is not generated parser backtracks and selecting 2nd production then 3rd production & so on.

LL(1) Parsing

Ex :→ S → aAd
A → b|c|a $O(2^n)$

aad

LL(1) Parsing

Step 1. All Tokens stored in input buffer along with $ as end Marker.

2. Let x is Symbol on the top of the stack and a is Lookahead Symbol, then parser takes decision as follows,

Step 1. if x = a = $ then i/p string is valid and parse tree is produced.

2. if x = a ≠ $ then pop x from Stack and increment lookahead Symbol

3. if x is Non terminal in the stack then select production from LL(1) parsing table.

• Table entry $T[x, a] = x \to UVW$ then X is replaced by UVW in reverse order. (Such that U appears on Top of the Stack)

4. If table entry is Blank then Syntax error is generated.

Time complexity of LL(1) parsing algorithm is O(n).

Q. How many steps taken by LL(1) Parser to verify Syntax of I/P a+a*a by using following parsing table. and also identify height of the parse tree constructed by the Parser.

Lookahead Symbol

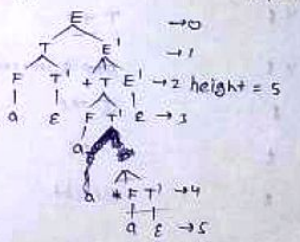
Non terminal	a	+	*	$
E	$E \to TE'$			
E'		$E' \to +TE'$		$E' \to \varepsilon$
T	$T \to FT'$			
T'		$T' \to \varepsilon$	$T' \to *FT'$	$T' \to \varepsilon$
F	$F \to a$			

⇒

	Stack	input	operation
①	$\$E$	a+a*a $	$E \to TE'$
②	$\$E'T$	a+a*a $	$T \to FT'$
③	$\$E'T'F$	a+a*a $	$F \to a$
④	$\$E'T'a$	a+a*a $	Pop & Increment
⑤	$\$E'T'$	+a*a $	$T' \to \varepsilon$
⑥	$\$E'$	+a*a $	$E' \to +TE'$
⑦	$\$E'T+$	+a*a $	Pop & Increment
⑧	$\$E'T$	a*a $	$T \to FT'$
⑨	$\$E'T'F$	a*a $	$F \to a$
⑩	$\$E'T'a$	a*a $	Pop & Increment
⑪	$\$E'T'$	*a $	$T' \to *FT'$
⑫	$\$E'T'F*$	*a $	Pop & Increment
⑬	$\$E'T'F$	a $	$F \to a$
⑭	$\$E'T'a$	$	Pop & Increment
⑮	$\$E'T'$	$	$T \to \varepsilon$
⑯	$\$E'$	$	$E' \to \varepsilon$
⑰	$\$$	$	Accept

* LR → (Right Most derivation) R.M.D in reverse → left to Right Scan

LL(1) → length of LA (Look ahead) → Left Most Derivation (LMD) → left to Right Scanning of I/P



Q. Which of the ~~following~~ syntax error ~~grammer~~ for the given i/p string having grammer parse table.

(a) y
(b) x q $
(c) x y q

	x	q	y	Y	$
S	S→P	S→P	S→P	S→P	S→P
P	P→TQR	P→QR	P→QR	P→TQR	P→QR
Q		Q→q	Q→ε		Q→ε
R			R→Y		R→ε
T	T→x			T→y	

① Stack | I/P | Operation.

Stack	I/P	Operation.
$ S	y $	S→P
$ P	y $	P→TQR
$ R Q T	y $	T→y
$ R Q y	y $	POP & increment
$ R Q	$	Q→ε
$ R	$	R→ε
$	$	Accept.

② ~~x~~

Stack	I/P	Operation.
$ S	x q y $	S→P
$ P	x q y $	P→TQR
$ R Q T	x q y $	T→x
$ R Q x	x q y $	POP & increment
$ R Q	q y $	Q→q
$ R	y $	R→Y
$ Y	y $	POP & increment
$	$	Accept.

③ $ S

LL(1) Parsing table Construction.

Q Construct LL(1) Parsing table for the following grammar.
Q Verify given grammar is LL(1) grammar or not.

$S \to aABC$
$A \to a/bb$
$B \to a/\varepsilon$
$C \to b/\varepsilon$

Hint: पहले First(Terminal) को Value आयेगा तब production देखेंगे, अगर 'ε' आया तो Follow जिस प्रोडक्शन में वहाँ का (Terminal) बैठा देंगे।

Note:→ The size of LL(1) parsing table is "NT * (T+1)" ($)
(NonTerminal) * (Terminal+1)

	a	b	$
S	S→aABC		
A	A→a	A→bb	
B	B→a	B→ε	B→ε
C		C→b	C→ε

S → a ABC
First(S) → a
A → a/bb
First(A) = a, b
B → a/ε
First(B) = {a, ε}
Follow(B) → {b, $} → given ε production
C → b/ε
First of (C) = {b, ε} Follow(C) = {$}

Note: Construct LL(1) parsing table corresponding to given grammar production. Row is selected left hand side Non terminal and Column is selected by using first set values of production right hand side part.

⇒ If 1st set contain ε then follow of Nonterminal is used to select about coulmns.
⇒ Any context Free grammar LL(1) parsing table contains multiple productions in one box, then that grammar is Not LL(1) grammar.
⇒ Every LL(1) grammar is Unambiguous grammar but Every Unambiguous grammar is need not be LL(1)
⇒ Any ~~unambiguous~~ grammar LL(1) parsing table should contains Multiple productions in atleast one box.

U.G
LL(1)

Q Construct Verify following grammar is LL(1) grammar or not.
$S \to aAbB / bAaB/\varepsilon$
$A \to S$
$B \to S$

	a	b	$
A S	S→AbB, S→ε	S→bAaB, S→ε	S→ε
A	A→S	A→S	
B	B→S	B→S	B→S

Follow(S) ⇒ $, a, b
A → S
First(S) → {a, b, ε} → follow(A) = {a, b}
B → S
First(S) → {a, b, ε}
Follow(B) → Full

Not LL(1) grammar bcz Multiple entry.

Q Construct Parsing table & Verify following grammar is LL(1) or not.
$S \to aSA$
$A \to c/\varepsilon$

→

	a	c	$
S	S→aSA		
A		A→c, A→ε	A→ε

S → a SA
First(S) ⇒ {a}
Follow(A) = Follow(S) ⇒ {c, $}
First(A) ⇒ {c, ε}

Not LL(1)

T.M
7 → (c) Within 100 steps then Decidable.
(d) ✓ → D
(e) All of them

(96) $L = \{ \langle M \rangle / M$ is a T.M of length that accepts strings $\leq 10^{14} \}$ → UD (REL) / Partially Decidable

(a) $L_U = \{ \langle w_M, w \rangle / w \in L(M) \} =$ T.M → Yes / No ⇒ REL
(b) $L_d = \{ \ldots \} \ w \notin L(M) \} \to$ Not REL
By defaut T.M May / May not Halt.

* Queue automata ⇒ F.A + 2 Stack.

Note:→ Left Recursive grammar is NOT LL(1) but elimination of left Recursion may result in LL(1) grammar. ($S \to A\alpha/\beta$, $A \to A\alpha/\beta$ → $A \to \beta A'$, $A' \to \alpha A'/\varepsilon$)

⇒ Non deterministic grammar is Not LL(1) but Left factoring of the grammar may results LL(1) grammar.

⇒ In any CFG if a Non-terminal having Multiple production where each production having common prefix then that grammar is Non-deterministic grammar.

Q. Construct left factor grammar for the following CFG.

(i) $S \to ab/ac$

L.F
$S \to aS'$
$S' \to b/c$

(ii) $S \to iEtS/iEtSeS/a$

L.F
$S \to iEtSS'/a$
$S' \to \varepsilon/eS$

Q. Which of the following is sufficient to convert CFG into LL(1) grammar.

(i) Elimination of left recursion alone.
(ii) Left factoring of grammar alone.
(iii) Both (i) & (ii)
(iv) N.O.T

Note:→ Converting CFG into LL(1) grammar is Undecidable problem.

CFG ⇒ CFL ⇒ Not possible of Convertion from CFL to DCFL.
LL(1) ⇒ DCFL

1. Any single production grammar is LL(1)
2. If the grammar production is
   $A \to \alpha_1/\alpha_2/\alpha_3 \dots \alpha_n$
   then
   $first(\alpha_1), first(\alpha_2), \dots first(\alpha_n)$ are pair wise disjoint then grammar is LL(1) Other wise Not LL(1)
3. If the grammar production is $A \to \alpha/\varepsilon$ then
   $First(\alpha) \wedge Follow(A) = \phi$ then the grammar is LL(1)

Q. Which of the following grammar is LL(1) grammar.

(a) $S \to aSa/bSb/\varepsilon$
   $first(a)$, $first(b)$, $follow(S) = \{a, b, \$\}$ → Common element ✗

(b) $S \to aSbS/bSaS/\varepsilon$
   a, b, $follow(S) = \{a, b, \$\}$ ✗

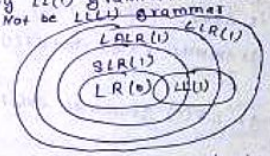
(c) $S \to AB$
   $A \to ab/bc$ ✓

(d) $S \to An$
   $A \to aA/a$ ✗

Q. Which of the following is true?

(i) Every LL(1) grammar is LR(0)
(ii) " " " " SLR(1)
(iii) " " " " LALR(1)
(iv) N.O.T

Note:→ Every LL(1) grammar is CLR(1) grammar but every CLR(1) Need Not be LL(1) grammar



Hence overall bottom up parser is More than power of Top-down Parser

Q. Which of the following is true about given grammar.

$S \to aSa/bSb/c$ → LL(1) ✓ if LL(1) then CLR(1) ✓ LR means CLR

(a) LL(1) & LR(1)
(b) LR(1) but LL(1)
(c) not LL(1) but LR(1)
(d) None.

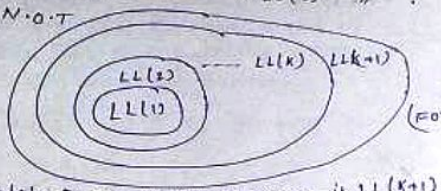
Q Which of the following is true.

(a) Every LL(2) grammar is LL(1) grammar.

(b) " LL(3) " " LL(2) ".

(c) " LL(1) " " LL(2) ".

(d) N.O.T



(For TOP-down)

Note:→ Every LL(k) grammar is LL(k+1) but every LL(k+1) need not be LL(k)

- In Top-down parsing if size of lookahead symbol is ↑es then power of parser is also ↑es.
- In Bottom-up parsing every LR(k) grammar is same as LR(1) grammar.
- In Bottom-up parsing length of the lookahead symbol is ↑es then power of the parser is same.

Note:→ For Ever DCFL LR(1) grammar exist but LL(1) May or May not exist.

→ For every regular lang. LR(1) grammar exist but LL(1) May or may not exist.

Q Which of the following statement is true?

(a) Every Regular grammar is LL(1) (S→aS/a) ×

(b) For every Regular lang LL(1) grammar should exist.

(c) For every DCFL, LL(k) grammars should exist

(d) For every Reg* lang LR(1) grammar should exist.

03/09/19

Comparision b/w bottom-up parser and Top-down parser

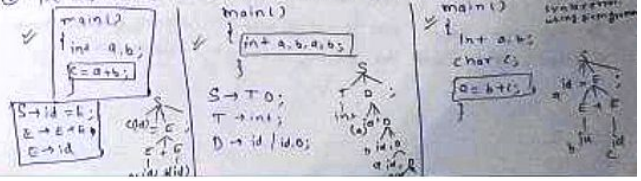
1) Bottom up parser performs parsing by detecting proper number of handles present in the input string.
2) Top-down parser perform parsing by constructing parse tree in which non-terminal is replaced by its Right hand side part.
3) Bottom up parser using Right most derivation in reverse order to construct parse tree.
4) Top-down parser uses Left most derivation to construct parse tree.
5) Bottom-up parser is more power Full than top down parser.
6) The difficulty in bottom-up parser is detection of proper number of handles.
7) The difficulty in Top-down parser is if a non-terminal having multiple productions on the grammar Right hand side part then selecting Correct production for tree construction is difficult.
8) Left recursive grammars are accepted by bottom-up parser but not accepted by top-down parser.
9) Non-deterministic grammar are accepted by bottom-up parser but not by top-down parser.
10) Top-down parser is simple to design compair to bottom-up parser.

Drawback of parser:

Paser detects Syntax errors only but not Semantic errors

Following are some of Semantic error parser can't detects. :-

① using a variable without declaration.
② Multiple declaration of same variable in the same scope.
③ Expression statements which are not type compatible.
④ Array index out of bound.
⑤ Formal and actual parameters dismatch.
⑥ Type dismatches in formal and actual parameters.



Semantic Analysis

```
main()
{
  int i;
  for(i=1;i<10;i++)        Syntax Error.
    printf("Hello");
}
```

- To design Semantic Analysis phase Context Sensitive grammars are used. hence this phase is also known as context sensitive analysis phase.
- CSG's are difficult to implement hence, Simplest form of CSG's are used to design these phase known as Syntax directed Translation.

Fig (2M)

Syntax Directed Translation

1) SDT is a mathematical technique where each C.F.G production is associated with set of semantic rules
2) Semantic rules are performing attribute analysis in ~~Parse~~ Annotated parse tree.

Annotated parse tree :- It is a parse tree where each grammar symbol is associated with attribute information.

→ Attribute is type of a variable or scope of a variable.

→ To design Semantic Analysis phase and Intermediate code generation phase syntax directed translation are constructed for given grammar productions.

⇒ Syntax Directed translation is carried out along with parser

⇒ The parser can be bottom up parser or Top-down parser.

⇒ The parser used in semantic analysis having 2 stacks hence its power is equal to linear bounded automata

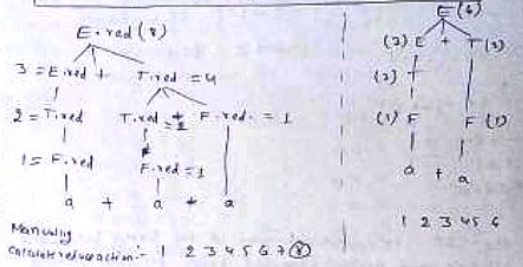
⇒ In the given parser one stack is used for storing grammar symbols and other stack is used for storing attribute information.

Q. Construct Syntax directed translation to calculate total No. of reduce actions taken by bottom up parser. by using following Grammar.

E → E + T
E → T
T → T * F
T → F
F → a

⇒

E → E + T	{E.reduce = E.reduce + T.reduce + 1}
E → T	{E.reduce = T.reduce + 1}
T → T * F	{T.reduce = T.reduce + F.reduce + 1}
T → F	{T.reduce = F.reduce + 1}
F → a	{F.reduce = 1}

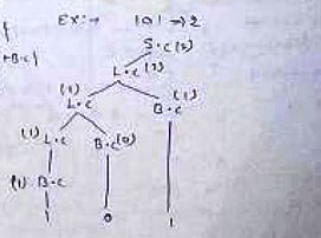


Manually calculate reduce action:- 1 2 3 4 5 6 7 (8)

1 2 3 4 5 6

Q. Construct SDT to calculate total no of one's present in given binary string. where count is attribute. and write Semantic rules for the given grammar production.

S → L {S.c = L.c}
L → L B {L.c = L.c + B.c}
L → B {L.c = B.c}
B → 0 {B.c = 0}
B → 1 {B.c = 1}



Steps to Construct SDT

1). Construct parse tree which includes all grammar productions
2). Construct annotated parse tree by computing attribute information.
3). Generalise rules of attribute computation and attached grammar production as semantic rules.

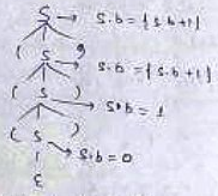
Q. Construct SDT to calculate total no of balanced parenthesis present in the input where count is attribute and write semantic rules for the following grammar productions :-

$S \to (S)$
$S \to \varepsilon$

b = balanced Parenthesis

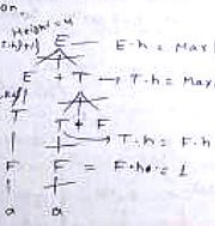
Ex → (((S))) → 3

$S \to (S)$ {S.b = S.b + 1}
$S \to \varepsilon$ {S.b = 0}



Q. Construct the SDT to calculate height of the parse tree. where height is attribute and write semantic rules for the following grammar production.

$E \to E + T$ / E.h = Max(E.h, T.h)+1
$E \to T$ {E.h = T.h}
$T \to T * F$ {T.h = Max(T.h, F.h)+1}
$T \to F$ {T.h = F.h+1}
$F \to a$ {F.h = 0}



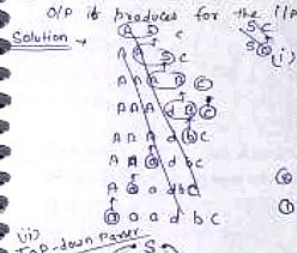
- SDT's is carried out along with bottom up parser for every reduce action the semantic rule corresponding to given production is executed.
- If SDT's is carried out along with Top-down parser then semantic rules are executed in depth first search traversal of parse tree from left to right.

Q. Construct the following SDT's

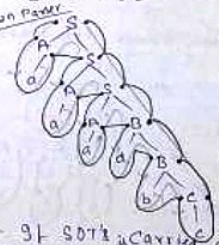
$S \to AS$ {print (1)}
$S \to AB$ {print (2)}
$A \to a$ {print (3)}
$B \to bC$ {print (4)}
$B \to dB$ {print (5)}
$C \to c$ {print (6)}

(i) This SDT is carried out along with Shift-reduce parser then what output it produces for the I/P aaadbc.
(ii) This SDT is carried out along with top-down parser then what O/P it produces for the I/P aaadbc.

Solution →



(ii) Top-down parser



Note:- If SDT's carried out along with Top-down parser or bottom up parser output is same but way of execution of semantic rules is different.

Q. Construct the following SDT.

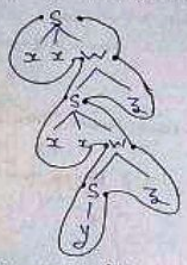
$S \to x\ x\ W$ { print (1) }

$S \to y$ { print (2) }

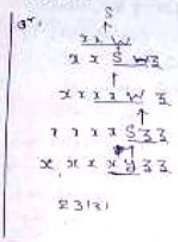
$W \to S\ 3$ { print (3) }

This SDT is carried out along with Top-down parser Then what o/p it produces for the i/p. x x x y 3 3

→



2 3 1 3 1



2 3 1 3 1

Q. Construct the following SDT.

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

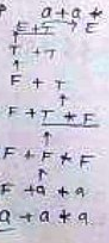
$E \to T$ { }

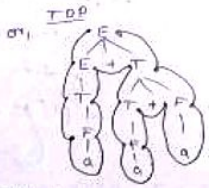
$T \to T * F$ { print (*) }

$T \to F$ { }

$F \to a$ { print (a) }

This SDT is given to shift reduce parser then what o/p it print for the i/p a + a * a





a a a * +

Q. $F \to L$ { F.v = L.v }

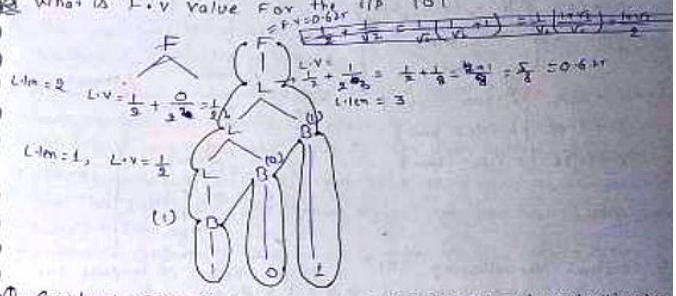
$L \to L_1 B$ { L.len = L_1.len + 1 ; L.v = L_1.v + B.v / $2^{L.len}$ }

$L \to B$ { L.len = 1 ; L.v = B.v / 2 }

$B \to 0$ { B.v = 0 }

$B \to 1$ { B.v = 1 }

What is F.v value for the i/p 101



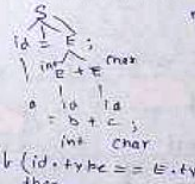
Q. Construct SDT for detection of incompatable type expression by using following grammar production. where type is attribute

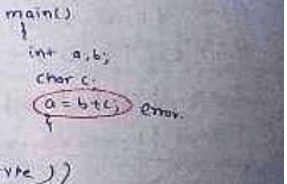
$S \to id = E$;

$E \to E + E$

$E \to id$

Solution





$S \to id = E$; { if (id.type == E.type) then void else error() }

$E \to E_1 + E_2$ { if (E.type = E_2.type) && (E.type = int) then E.type = int else error() }

$E \to id$ { E.type = id.type }

Q. Construct SDT to detect incompatible type expression of a language having integer and boolean data types. Where type is attribute and write semantic rules for the following grammar production.

$E \to E_1 + E_2$ { if (E_1.type == E_2.type) && (E_1.type == int) then E.type = int else error() }

3 + (3 == 6)

3 + false ✗ Semantic error.

$E \to E_1 == E_2$ { if (E_1.type == E_2.type) then E.type = bool else error() }

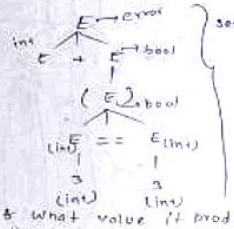
Not detect error by Syntax tree.

$E \to (E_1)$ { if E_1.type = ~~bool~~ then E.type = ~~bool~~ int else error() }

$E \to$ num { E.type = int }

$E \to$ true { E.type = bool }

$E \to$ false { E.type = bool }



Q. Construct the following SDT's & what value it produces for E.val by using Expression 2 # 3 & 5 # 6 & 4

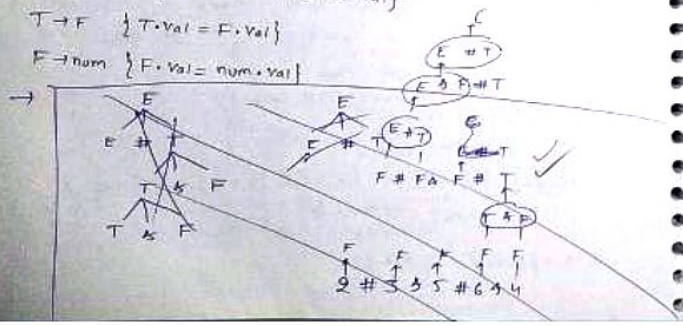
$E \to E \# T$ { E.val = E.val * T.val }

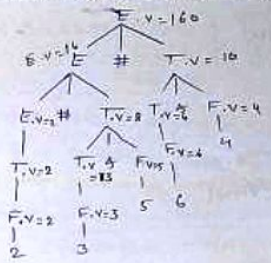
$E \to T$ { E.val = T.val }

$T \to T_1 \& F$ { T.val = T_1.val + F.val }

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

$F \to$ num { F.val = num.val }

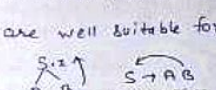


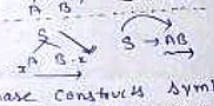


Note:

Q. There are two types of attribute present in SDT know as Synthesized attribute & Inherited attribute.

- If the attribute value at a node in the annoted parse tree. Computed from children i/a Synthesized attribute.
- Synthesized attributes are well suitable for bottom-up parser.
- Inherited attribute: → The attribute value at a node in the annoted parse tree computed either from parent or from left sibling i/a inherited attribute.
- Inherited attributes are well suitable for Top-down parser.

Synthesized attribute 

Inherited attribute. 

- Lexical analysis phase constructs symbol table partially (type information of variables not available to the finite automata).
- Complete construction of symbol table is done by semantic analysis phase.

Q. Construct SDT's to insert type information in the symbol table by using following declaration statement grammar. Where S type is synthesized attribute & T.type is inherited attribute, and add type is a function that assign type information to a variable in the symbol table.

$S \to T\ L$;

$T \to$ int

$T \to$ float

$L \to L_1$, id

$L \to$ id

$S \to TL$; { L.itype = T.stype }

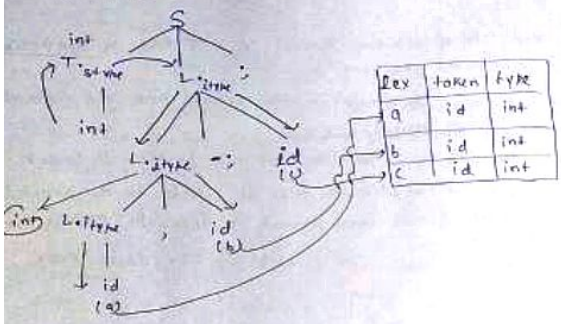
$T \to int$ { T.stype = int }

$T \to float$ { T.stype = float }

$L \to L_1, id$ { L_1.itype = L.itype
addtype ("id.name", L.itype) }

$L \to id$ { addtype ("id.name", L.itype) }

int a, b, c ;



lex	token	type
a	id	int
b	id	int
c	id	int

S-attribute

Def^n :→ The SDT that contains synthesized attributes only Kla S-attribute definition.

L-attribute

Def^n → The SDT that contains both synthesized attributes and inherited attributes Kla L-attribute definition.

Every S-attributed definition is L-attributed definition but every L-attributed need not be S-attributed.

Ex :→ $S \to AB$ {s.x = A.x}, $A \to BC$ {A.x = B.x}, $B \to b$ {print(b)} — both. b/c S attribute Kla L.

Ex :→ $A \to aBCD$ { $A_1 = B.i$, $C.s = C.i$, $B.s = D.s$ } (L-attribute) X (both)

Syntax Directed definition

- It is a formalism where all the semantic rules corresponding to grammar production are placed at right most end of the grammar.

Syntax Directed Translation or Translation Scheme

- It is a formalism where semantic rules may present in b/w grammar symbols.
- In translation scheme all the semantic rules corresponding to Synthesis attributes are placed at Right most end of grammar production. and all the semantic rules corresponding to inherited attributes are placed before a non terminal for which attribute Calculation is performed.
- In translation scheme all semantic rules corresponding to Synthesized attributed and inherited attribute are evaluated by DFS traversal of the parse tree from left to right manner.
- While constructing parse tree from translation scheme an edge should be provided for every semantic rule by assuming the semantic rule as one of the grammar symbol.

Q. Construct Syntax Directed Translation (SDT) from the given Syntax Directed definition (SDD).

SDD

$S \to TL$; { L.itype = T.stype }

$T \to int$ { T.stype = int }

$T \to float$ { T.stype = float }

$L \to L_1, id$ { L_1.itype = L.itype
addtype ("id.name", L.itype) }

$L \to id$ { addtype ("id.name", L.itype) }

SDT or Translation Scheme.

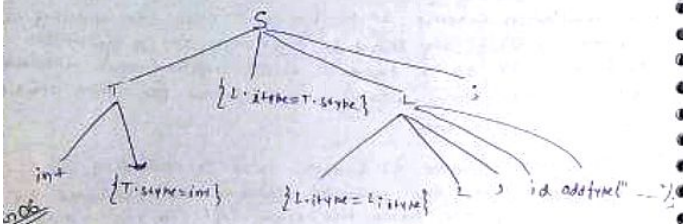
$S \to T \{L.itype = T.stype\}\ L;$

$T \to int \{T.stype = int\}$

$T \to float \{T.stype = float\}$

$L \to \{L_1.itype = L.itype\}\ L, id \{addtype("id.name", L.itype)\}$

$L \to id \{addtype("id.name", L.itype)\}$



Q. What is the output produce by translation Scheme for the i/p

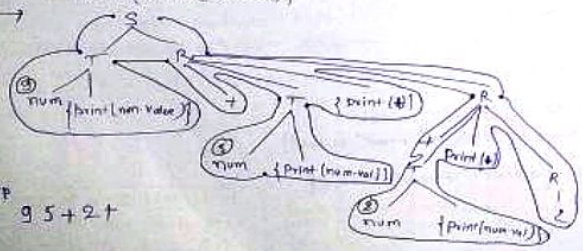
9 + 5 + 2

$S \to T\ R$

$R \to + T \{Print(+)\}\ R$

$R \to \varepsilon$

$T \to num \{Print(num.val)\}$



O/P

9 5 + 2 +

Q. i/p ⇒ 2 * 3 4

For the given i/p construct the parse tree

$S \to E\ R$

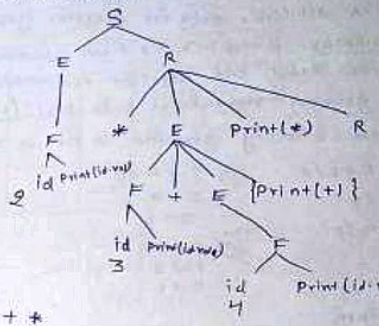
$R \to * E \{print(*)\}\ R$

$R \to \varepsilon$

$E \to F + E \{print(+)\}$

$E \to F$

$F \to id \{print(id.value)\}$



O/P → 2 3 4 + *

Q. $S \to P \{Print(w)\}\ P$

$P \to q \{Print(A)\}\ Q$

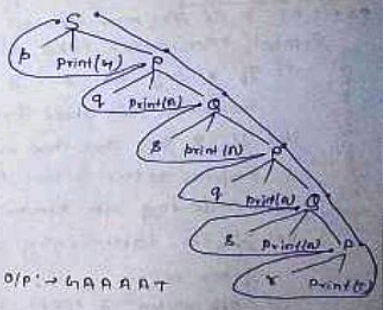
$P \to r \{Print(T)\}$

$P \to \varepsilon \{Print(E)\}$

$Q \to s \{Print(A)\}\ P$

$Q \to \varepsilon \{Print(a)\}$

i/p:- p q s q s r



O/P :- w A A A a A T

Operator precedence parsing

1. Operator precedence parser performs parsing by detecting proper number of handles in every sentential form.
2. Operator precedence parser is Shift reduce parser.
3. Operator precedence parser can take ambigious grammar without having conflicts.
4. Operator precedence parser is less powerful compair to LR parser.
5. This parsing is applicable only for operator grammar.
6. Operator grammar is a Contex Free grammar in which No "ε" production exist and adjacent Non-terminals should not present in grammar right hand side part. (A→ε, A→BC)

7. Which of the following grammar is not a operator grammar.

(a) E→E+E/E*E/a

(b) E→E+T/T
T→T*F/F
F→a

(c) S→AaB
A→a
B→b

(d) S→AB
A→a
B→b

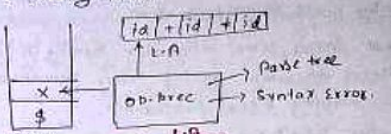
→ A→εX
→ A→BCX

Parsing Algorithm

1. Let x is Top most terminal on the Stack & y is Look ahead Symbol then parser takes decision from parsing table.
   (i) If x <· y or x ≐ y then shift y into the Stack and increment look ahead symbol.
   (ii) If x ·> y then there is handle to the stack hence perform reduce action by replacing handle by its corresponding non-terminal.
   (iii) If the table entry is blank then syntax error is generated.
   (iv) Table entry is accept if the i/p string is valid hence parse tree is generated.
2. Time Complexity of operator precedence of parsing algo is O(n).

(• means precedence)

8. How many shift action and reduce action taken by operator precedence parser to parse the i/p id+id*id by using following grammar and parsing table.



Stack L.A

	id	+	*	$
id		·>	·>	·>
+	<·	≐	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	accept

all possible cases

E→E+E
E→E*E
E→id

Sol^n.

	Stack		Input	Action
①	$	<·	id+id*id$	S
②	$ id	·>	+id*id$	R
③	$ E	<·	+id*id$	S
④	$ E +	<·	id*id$	S
⑤	$ E + id	·>	*id$	R
⑥	$ E + E	<·	*id$	S
⑦	$ E + E *	<·	id$	S
⑧	$ E + E * id	·>	$	R
⑨	$ E + E * E	·>	$	R
⑩	$ E + E	·>	$	R
⑪	$ E		$	Accept.

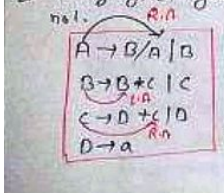
⑤ 5 shift, 5 Reduce Action

Construction of precedence parsing table.

- To construct parsing table precedence and associativity information is analyzed by constructing parse tree.
- In any given parse tree lower level operator having higher precedence and upper level operator having lesser precedence and same level operator having equal precedence.
- Associativity of operator analyzed by using recursion.
- If the production having left recursion then corresponding operator is left associative.
- If the production having right recursion then corresponding operator is right associative.
- Identifier having higher precedence overall other operators
- Doller ($) is having lesser precedence overall other operators
- In any operator grammar both left recursing and right recursion exist on the same production then the grammar is ambiguous grammer.
- If any operator grammar all production contains either left recursion or right recursion but not both. then that grammar is Unambiguous grammar.

Q. Construct operator precedence parsing table from the following grammar.

Q. Verify given grammar is operator precedence grammar or not.



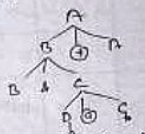
- Operator precedence Grammar

⇒ Any contex free grammar operator precedence parsing table contains multiple entries in one box that grammar is not operator precedence grammar.

⇒ Every O.P.G is operator grammar but every operator grammar need not be operator precedence grammar.

Q. Which of the following is true from the given operator grammar.

① $A \to B \oplus A \mid B$, $B \to B \,\&\, C \mid C$, $C \to D \otimes C \mid D$, $D \to a$



ⓐ & ⊗ is higher precedence than ⓓ {& > ⓓ}

ⓑ both ⓓ and & are left associative.

ⓒ ⊕ is higher precedence than ⓓ

ⓓ N.O.T

Q. Which of the following is false as for the given grammar.

$A \to B \uparrow A \mid A * B \mid B$, $B \to B - B \mid C / B \mid C$, $C \to a$

ⓐ Power(↑) and * are same precedence.

ⓑ '−' and '/' are equal precedence.

ⓒ '−' is higher precedence than '+'

ⓓ '+' is higher precedence than '/'

Q. Which of the following is true about given grammar.

$X \to X \oplus Y / Y$, $Y \to Z * Y / Z$, $Z \to a$

ⓐ ⊕ and * are left associative.

ⓑ ⊕ and * are right associative.

ⓒ ⊕ is left associative and * is right associative

ⓓ N.O.T

Q. Construct operator precedence parsing table for the following ambiguous grammar.

$E \to E+E / E*E / a$

	a	+	*	$
a	⋗	⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	accept

Drawback of Operator precedence parser. (OPP)

- O.P.P are less power full then LR parser because this parsing is applicable only for operator precedence ~~dtat~~ grammar.

10-9-19

Implementing Translation Scheme with Top-down parser

- If the translation scheme carried out along with top-down parser then left recursion should be eliminated from the translation scheme
- If translation scheme carried out along with top-down parser then all semantic rule are executed in Depth first search left to right manner.

Q. Eliminate left recursion from the following translation scheme.

$E \to E+T$ {print(+)}	$E \to T\ E'$
$E \to T$	$E' \to +T$ {print(+)} E' / ε (M)
$T \to T*F$ {print(*)}	$T \to FT'$
$T \to F$	$T' \to *F$ {print(*)} T' / ε (N)
$F \to a$	$F \to a$

$A \to A\alpha / \beta$

$A \to \beta A'$

$A' \to \alpha A' / \varepsilon$

Translation Scheme with bottom-up parser

If translation scheme carried out along with bottom-up parser for every reduce action semantic rules are executed.

All the semantic rules present in the middle of grammar symbol are replaced by new non-terminal.

All the new non-terminals generate "ε" and associated with corresponding semantic rule.

Convert the translation scheme suitable for bottom up parser.

$E \to TE'$	$E \to TE'$
$E' \to +T$ {print(+)} E' / ε (M)	$E' \to +TME' / \varepsilon$
$T \to FT'$	$T \to FT'$
$T' \to *F$ {print(*)} T' / ε (N)	$T' \to *FNT' / \varepsilon$
$F \to a$	$F \to a$
	$M \to \varepsilon$ {print(+)}
	$N \to \varepsilon$ {print(*)}

Drawback of SDT

By using SDT dynamic semantic error can't be identified

By using SDT static semantic error only detected.

- The language in which type checking is done at compile time only ~~k~~ is statically typed language.
- The language in which type checking is done at run time ~~k~~ is Dynamically typed language.
- C language statically typed language.
- Java language is dynamically typed language.
- The language in which no type checking is perform ~~k~~ is untyped languages
- Machine language and Assembly language are untyped languages.

Semantic Error

Static

```
main()
{ int a,b;
  char c;
  a = b + c;
}
```

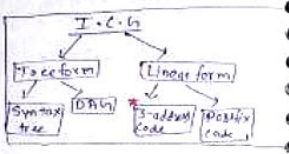
Dynamic

```
main()
{ int a[10];
  printf(a[12]);
}
```

Intermediate Code Generation

- The advantage of generating Intermediate code is to perform machine independent optimization.
- Machine dependent optimization is very difficult to perform by compiler hence intermediate code is generated.
- Intermediate code is generated easily from high level language and translating it into low level language. It also simple.
- Following are different type of intermediate code generated by Compilers.

(i) Syntax tree
(ii) DAG (Directed Acyclic graph)
(iii) 3-address code
(iv) Postfix code



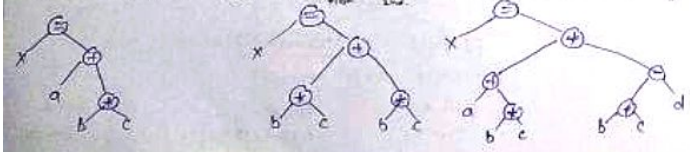
Syntax tree :-

① Syntax tree is optimized form of parse tree.
② Syntax tree represents order of evaluation of expression.
③ Corresponding to expression all operators present at root node and all operands present at leaf node in syntax tree.

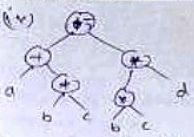
Q. Construct syntax tree for the following expression statements.

Note: In syntax tree common sub expressions are repeatedly evaluated, hence to avoid this drawback directed acyclic graph is used.

① X = a + b * c (ii) X = b * c + b + c (iii) X = (a+b*c)+(b*c-d)



(iv)



{a + (b * c)} → {(b * c) * d}

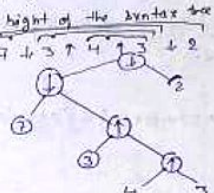
Q. Construct the following expression.

7 ↓ 3 ↑ 4 ↑ 3 ↓ 2

① the precedence of ↑ is higher than ↓.
② ↑ is right associative and ↓ is left associative.

Find height of the syntax tree constructed for given expression.

→ 7 ↓ 3 ↑ 4 ↑ 3 ↓ 2



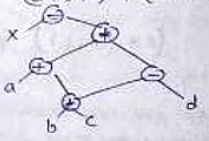
DAG (Directed acyclic graph)

DAG is syntax tree in which common sub expression are eliminated. If DAG is constructed some type of optimization automatically perform i.e. common sub expression elimination optimization.

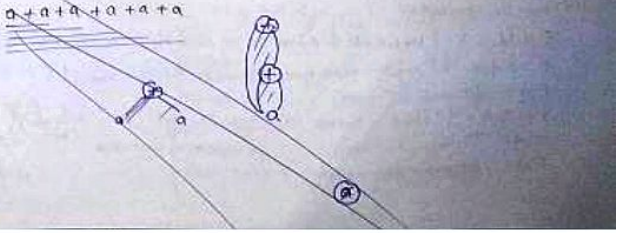
Q. Construct DAG for the following expression.

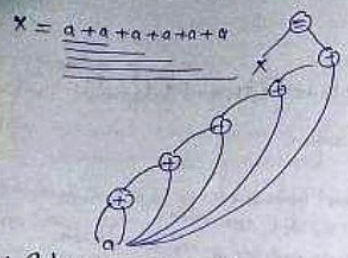
X = (b * c) + (b * c) or b * c + b * c

X = (a + b * c) + (b * c - d)

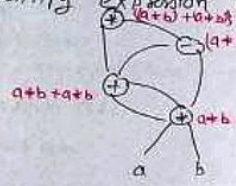


X = a + a + a + a + a + a

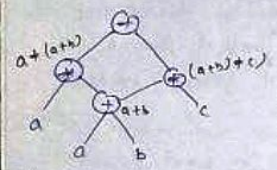




Q. Identify expression corresponding to following DAG.



$\{(a+b)+(a+b)\}*[\{(a+b)+(a+b)\}-(a+b)]$



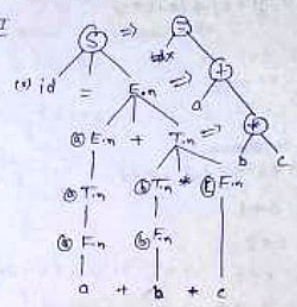
$\{a*(a+b)\} + \{(a+b)+c\}$

Q. Construct SDT to generate syntax tree for the given expression where node is attribute and write semantic rules for the following expression grammar.

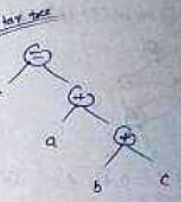
$S \to id = E$ { Makenode (id.node, "=", E.node) }
$E \to E+T$ { E.node = makenode (E.node, "+", T.node) }
$E \to T$ { E.node = T.node }
$T \to T_1 * F$ { T.node = ~~T.node~~ makenode (T.node, "*", F.node) }
$T \to F$ { T.node = F.node }
$F \to a$ { F.node = id.name }

Makenode is a function that creates a root node and assign left subtree and right subtree.
The SDT for syntax tree construction contains only synthesized attribute where node is attribute that gives syntax tree information at every grammar symbol.

S.D.T



Syntax tree



Three address Code

~~Three address~~
Tree form intermediate code is suitable only for expression statement but not suitable for functions declaration statement and flow control statements. hence 3-address code representation of intermediate code is constructed.

- Three address code is suitable for all high level language statement.
- Three address code every high level language statement is represented by using atmost 3 address.
- While generating 3-address code compiler generates temporary variable.
- In 3-address code common sub expressions are repeatedly evaluated.

Following are different form of three-address code statement

Expression
(1) $x = y \text{ op } z$
(2) $x = \text{op } z$
(3) $x = y$
Array
(4) $x[i] = y$
(5) $x = y[i]$
(6) goto L
(7) if a rb goto L1 / goto L2 — flow control & for
(8) function
Param p_1
Param p_2
:
Param p_n
Call name of function, no of arguments

Q. Construct Three-address code for the following expression and find How many temporary variable generated by the compiler.

① $X = a + b * c$ ② $X = a * b + c * d - e * f$

③ $X = (a+b) + (c+d)/(a+b)$ ④ $X = (y+z) * (a+b+c)$

⑤

Soln ① $X = a + b * c$

$t_1 = b * c$

$t_2 = a + t_1$

$X = t_2$

② $X = a * b + c * d - e * f$

$t_1 = a * b$

$t_2 = c * d$

$t_3 = e * f$

$t_4 = t_1 + t_2$

$t_5 = t_4 - t_3$

$X = t_5$

③ $X = (a+b) + (c+d)/(a+b)$

$t_1 = a + b$

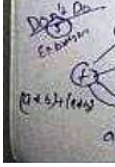
$t_2 = c + d$

$t_3 = a + b$

$t_4 = t_2 / t_3$

$t_5 = t_1 + t_4$

$X = t_5$



④ $X = (y+z) * (a+b+c)$

$t_1 = y + z$

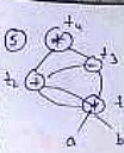
$t_2 = b * c$

$t_3 = a + t_2$

$t_4 = t_1 * t_3$

$X = t_4$

⑤



$t_1 = a * b$

$t_2 = t_1 + t_1$

$t_3 = t_1 - t_3$

$t_4 = t_2 * t_3$

Q. Construct 3-address code for the following if-else statement.

① if (a > b)
{
X = Y + Z;
}
else
{
P = Q + R
}

if a > b goto L1
goto L2
L1: t3 = Y + Z
X = t1
go to LAST
L2: t4 = Q + R
P = t4
LAST

② for (i = 1; i ≤ 10; i++)
{ X = Y + Z * n;
}

i = 1;
L0: if i ≤ 10 goto L1
goto L2
L1: t1 = Z * n
t2 = Y + t1
X = t2
t3 = i + 1
i = t3
goto L0
L2: ---

Array
A[i] = base + (i - low) * w
A[i, j] = ((i * n2 + j) * w) + base - (low1 * n2 + low2) * w

Q. Construct 3-address code for the following expression.

$x = A[i,j]$

Size of the array $A[10][20]$ and size of each word is 4 bytes

$low_1 = low_2 = 1$

$\Rightarrow A[i,j] = (i*20+j)*4 + (base - (1*20+1)*40)$

$t_1 = i*20$

$t_2 = t_1 + j$

$t_3 = t_2 * 4$

$t_4 = base - 84$

$t_5 = t_3 + t_4$

$t_6 = A[t_5]$

$x = t_6$

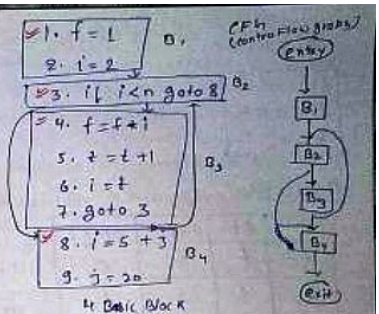
Construction of Basic Block

- Basic Block is sequence of 3-address code instruction which is having single entry and single exist property.
- Basic block is constructed by using leader statemmts
- Following are the leader statement in 3-address code.
  - (i) First statement is always a leader.
  - (ii) Target of conditional or unconditional goto is a leader
  - (iii) The statement immediately followed by conditional or unconditional goto is a leader.
- One leader statement to next leader statement one basic block is created.
- By using basic block control flow graph is constructed

Q. How many basic block possible for following 3-address code instruction.

1. Print
2. $j = 1$
3. $t_1 = i*4$
4. $t_2 = a[t_1]$
5. $t_3 = 4*i$
6. $t_4 = b[t_3]$
7. $t_5 = t_4 * t_1$
8. $t_6 = prod + t_5$
9. $prod = t_6$
10. If $I < 20$ goto (3)
11. $i = 10$

$\therefore$ 3 basic blocks.



Implementation of 3-address code.

- The generated 3-address code stored in array contains four fileds kla Quadruple Notation.
- In Quadruple Notation 2 fields used for storing operands and other 2 fiels operator and results.
- In Quadruple notation compiler unable to differentiate program variable and temp. variable hence temp. variable also placed in Symbol table.
- Removing of result field from Quadruple notation kla triples Notation.
- In triples Notation expression value directly stored in Memory location and accessing from memory location is done whenever required
- Triples implemented by using array data structure hence having fixed size to avoid this drawback dynamic data structure implementation of triples Notation. Know as indirect triples is used in practical compilers.

$X = a + b * c$

$t_1 = b * c$
$t_2 = a + t_1$
$X = t_2$

Quadruple

result	op	oper1	oper2
t_1	*	b	c
t_2	+	a	t_1
X	=	t_2	

triple

	Op	Op1	Op2
(100)	*	b	c
(101)	+	a	100
	=	X	101

Ch-4

(9) $t_1 = -c$
$t_2 = b * t_1$
$t_3 = a + t_2$
$a = t_3$

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

(10)

- **Static Single assignment:** → It is a 3-address code where each expression value assign to new variable.
  → In static single assignment form reusing of variable is not allowed.

(10) $p = a - db$
$q = p * c$
$p = u * v$
$q = p + q$
option (b)

$(a-b) * c$ $(u+v)+g$

(11) ~~1 to 2, 3 to~~
1 | 2 | 3 to 9 | 10 to 11

No Question

Recursive Descent Parsing

Recursive Descent parsing not possible for left recursive grammar.
For left recursive grammar parser will fall into infinite loop.
To construct recursive descent parse a procedure is constructed for every non terminal.
The power of recursive descent parser is less compare to LL(1) parser.

Q: Construct recursive descent parser for given grammar.

$E \rightarrow numT$
$T \rightarrow * numT / \epsilon$

```
main
{ E();
}
```

```
E()
{ if (lookahead == num)
  { match();
    T();
  }
  else
    error();
  if (lookahead == $)
    print("Valid Syntax");
  else
    error();
}
```

```
T()
{ if (lookahead == *)
  { match();
    if (lookahead == num)
    { match();
      T();
    }
    else
      error();
  }
  else
    return null;
}
```

```
match()
{ lookahead = next token;
}
```

```
error()
{ print("Syntax Error");
}
```

2 * 3 * 4

2	*	3	*	4	$

L.A

Top down
- LL(1) is most important. (Table construction, grammer detection)
- Bottom up LR parser.

In LR Parser

- **Viable prefix**

Set of prefixes in the right sentential form which present left hand side part of handle ka viable prefix.

Viable prefix is stack contained in Shift/Reduce parser

Q. Which of the following is valid viable prefix for the given grammar.

$S \to XX$

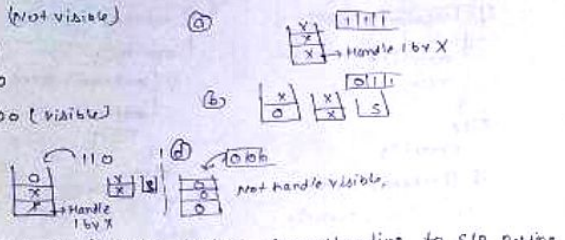
$X \to aX / b$

(a) 111 (not visible)

(b) 011

(c) 110

(d) 000 (visible)



Q. Which of the following is true corresponding to S/R parsing.

(a) Viable prefixes appears only at top of the stack.

(b) " " " " " bottom " " .

(c) Stack contains set of viable prefixes

(d) " never " " " " " .

Ch-12

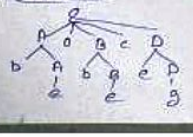
41 → (a) If(z); id(z); ✓ Valid function syntax

(b) For(a,b,c); id(a,b,c); ✓ Valid

(c) While(a,b); id(a,b); ✗ Invalid

(42) N.O.T

(43) Incorrect



$a \to b$, $c \to g$, $a \to e$, $\frac{cab}{X}$ ✓

(7) $S \to Aca / Bcb$

$A \to c$

$B \to c$

LL(1) → length of Lookahead = 1 = {cca, ccb} ✗

LL(2) → length of lookahead = 2 = {cca, ccb} ✗

LL(3) → " " " = 3 = {cca, ccb} ✓

(8) Option (d)

(9) $E \to E+F / F+E / F$

$F \to F-F / id$

Option (b)

(10) → a

(11) → a

(12) → a

(13) → d

(14) → c

- **Preprocessor**

It is a program that translates high level language into mac high level language by executing all '#' statements present in th program.

Assembler :-

- It is a translator that translates assembly language into relocata Machine code
- Relocatable MIC code is not executable MIC code because linking is not perform.

Linker is a programer that makes Multiple Modules into sing Modules.

- Linker translates relocatable MIC code into executable code.

Loader is a program that loads MIC code into Main Memory.

MIC 3 → a, 13 → d

14 → c

Run time Environment

- There are 2 types of Memory allocation Strategies present in the Compiler. Kla Static allocation & Dynamic allocation.
- In static allocation method recursive programs can't be executed and dynamic Data Structure can't be created.

Dynamic allocation

- In this method run time Memory is divided in code area, data area, stack area, heap area.
- For the generated MIC code Memory is allocated in code area,
- For global variables Memory is allocated in Data area,
- For functions Memory is allocated in stack area.
- For every fn call, ~~every~~ activation record is created and pushed into stack Memory.
- If fn return value activation record in popped out from stack hence allocation and de allocation of Memory is completely automatic in stack area. (No additional program required.
- If the Memory is allocated in heap area then allocation and deallocation is done by additional programs
- Hence Memory Managment is completely Mannual.
- In dynamic allocation Method recursive programs can be executed and the dynamic data structure can be created.
- C, C++, Java language dynamic allocation method,