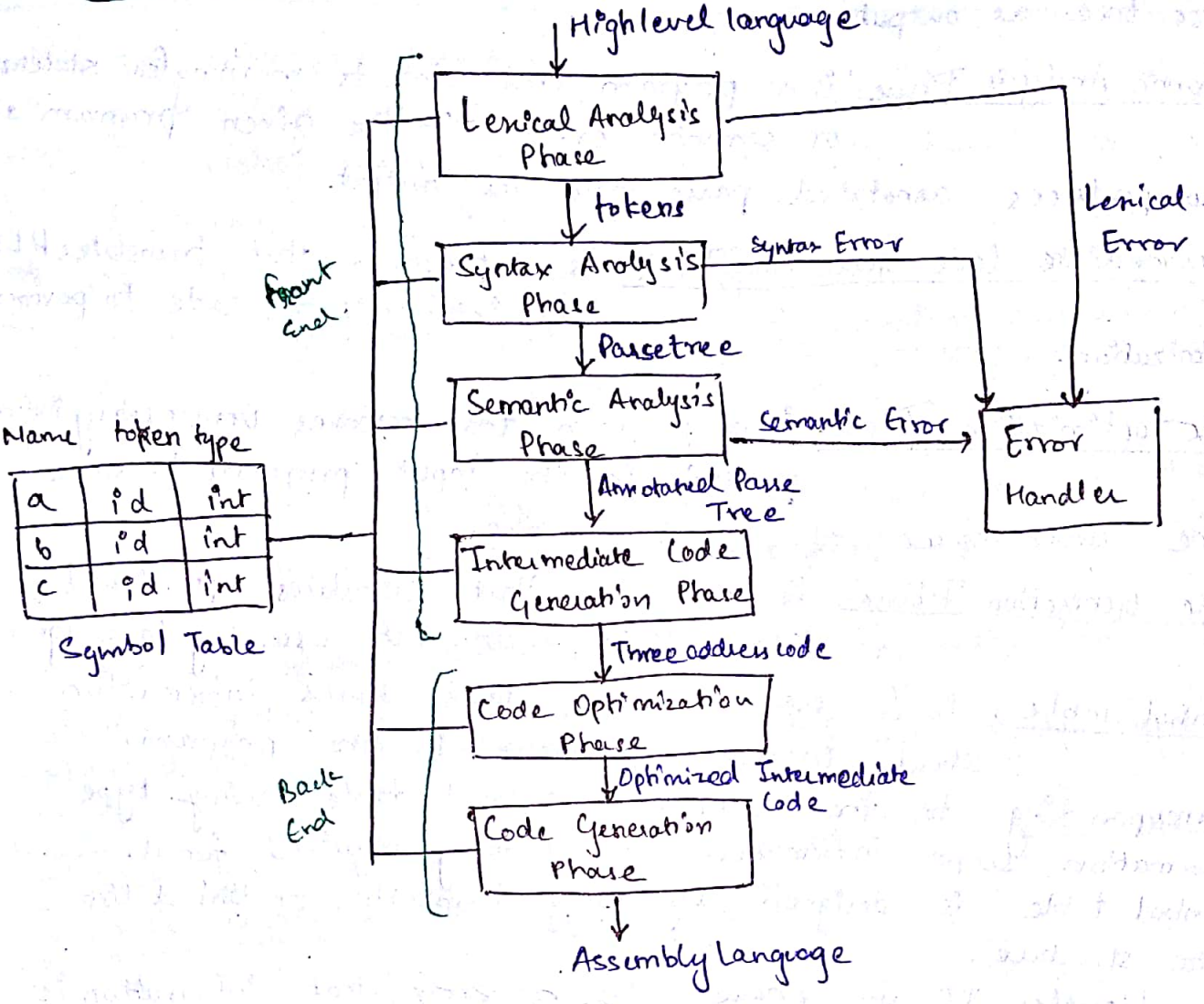
02/08/2017

COMPILER DESIGN:



→ Lexical Analysis 10%

→ Passing 60%

→ Syntax Directed Translation } 30%

→ Intermediate Code Generation

Compiler:

→ Compiler is a program that translates a program written in one language into an equivalent program of any language and also detects errors present in the program.

→ Errors detected by compiler are classified into:
① Lexical Errors ② Syntax Errors ③ Semantic Errors.

Lexical Analysis Phase: It is a program that detects lexical errors present in the HLL program and also produces tokens as output.

Syntax Analysis Phase: is a program that detects syntax errors present in the HLL program and also produces parse tree as output.

Semantic Analysis Phase: is a program that detects meaningless statements or semantic errors in the given program and also produces annotated parse tree as output.

Intermediate Code Generation Phase: is a program that translates HLL into intermediate code to perform optimization.

Code optimization Phase: is a program that removes unnecessary information present in the input program to save time and space of target machine.

Code Generation Phase: is a program that translates optimized intermediate code into assembly language.

Symbol Table: is a data structure that stores information about identifiers present in the program. Corresponding to the identifier symbol table having type information, scope information and memory required for it.

→ Symbol table is designed by using hashtable or linked list data structure.

Error Handler: If any phase detects error, that information is sent to error handler and compilation process is continued.

→ Error handler having information about lexical errors, syntax errors and semantic errors.

NOTE: Lexical ~~ang~~ analysis is designed by using reg. expression & F.A. technique.
- Syntax analysis is designed by using PDA and CFG formal methods.
- Semantic analysis & Intermediate code designed by using CSG or S.D.T.

Front End: The phases of compiler which are depending on source language and independent of the target machine.

→ Front End phases include lexical analysis, syntax analysis, semantic analysis, intermediate code generation and machine independent optimization.

Back End: The phases of compiler which are completely dependent on target machine and independent on the source language.

→ Back end phases include code generation phase and machine dependent optimization.

LEXICAL ANALYSIS PHASE:

→ Lexical analysis reads HLL program character by character and breaks them into tokens.

→ Token describes the category of the input.

→ The following are tokens in programming language:

① Keyword ② Identifiers ③ Constants
④ Operator ⑤ Punctuations

* **Lexeme:** Sequence of characters in the source program that completely matches with rules of a token known as lexeme.

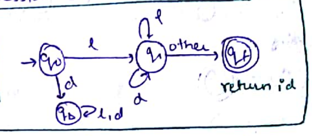
→ If any sequence of characters is not a valid lexeme, then lexical error is produced.

→ Lexical analysis is designed by using F.A. and regular expressions.

NOTE:

→ Detection of tokens, identifier and keyword is by using following regular expression and F.A.:

I.D. $l(l+d)^*$



→ If any sequence of character matching with identifier finite automata, then search operation is performed with keyword table.

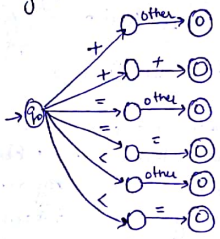
→ If search is success keyword token will be ~~written~~ returned otherwise identifier token will be returned.

→ Lexical analyser cannot detect syntax and semantic errors.

→ For the following "C" program statements, no lexical errors will be detected:

① 9it|'a|,b|,c|;
② int 1| |x|,|y|;|
③ fi|(|b|<|d|)|
④ a|=|b|,|*|c|;|
⑤ fo(i=1;i<10;i++)

→ The following is a F.A. corresponding to token operator.



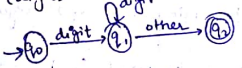
→ If any sequence of characters by which F.A. reaches final state, then token operator is produced.
→ If operator automata reaches final state with other symbol, then, other symbol is excluded from token sequence.

→ The following is the F.A. and reg.ex. for token constant:

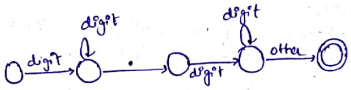
digit = 0 + 1 + ... + 9

Constant : RE = (digit)+

Int constant :

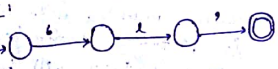


Real constant : (digit)+ . (digit)+

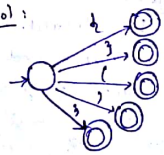


String constant :

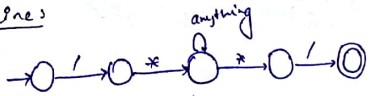
Character constant :



Punctuation Symbol :

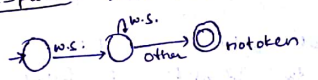


Comment Line :



no token

White space : ws = \b + \t + \n



→ Lexical analyser removes all white spaces present in the program.
→ If any sequence of characters matches with w.s. automata, no token will returned, so that, w.s. is removed.
→ Lexical analyser eliminates all comment lines present in the program.
→ If any sequence of characters matches with comment line automata, no token will be returned.

Q. Identify the no. of tokens produced by following C programs.

① main()
{ int x, y;
fl/*gate*/oat z;
x=/*exam*/10;
y=20;
}

(21)

② int max(int a, int b) (25)
{ if(a>b)
{ return a; /* Max of two nos. */
}
else
return b;

③ main()
{ printf("welcome");
}}}

(12)

Q. Which of the following program results lexical error:

① main()
{ int a,b,'
c=a+b,'
}

No error

② main()
{
int a,b,a/b,a;
}

No error

③ main()
{ int a,b;
char c="Hello";
a=b+c;
}

No error

④ main()
{
if(a==b+c);
}

No error.

→ Symbol table is constructed in the lexical analyser, whenever identifier token is produced.
→ In the symbol table, lexical value and token information is inserted and type of the identifier is filled during semantic analysis phase.
→ Hence, lexical analyser constructs symbol table partially but not completely.

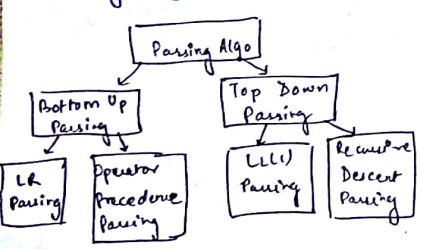
int a,b,c;

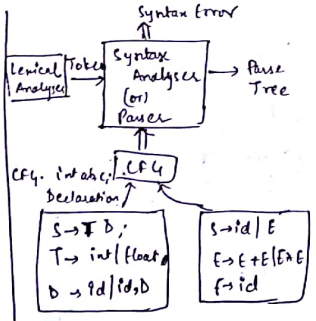
lex	token	type
a	id	—
b	id	—
c	id	—

SYNTAX ANALYSIS OR PARSING:

*Syntax: The way in which tokens arranged together is known as syntax.
*Parsing: Checking whether string is generated from given grammar or not is known as parsing.

Syntax Analyser or Parser: is a program that takes tokens and CFG as input and verifies tokens are member of CFG or not.
→ If tokens are member, parse tree is produced as output, otherwise syntax error is generated.
→ To design parser, two algorithms exist, known as bottom up parsing algorithm and top down parsing algorithm.





→ Syntax of programming languages are represented by CFG.
→ Passing algos. are ~~determ~~ designed by using D-PDA.
→ Only unambiguous grammars are used to design parser.

Bottom Up Parsing:
→ Bottom Up Parser constructs the parse tree, starts from the given string and proceeds towards starting symbol.
→ Bottom Up Parser uses right most derivation in reverse order.
→ Handles detection is major problem for bottom up parser.

*Handle: is a part or substring in the sentential form that is completely matching with the R.H.S. part of grammar production and its replacement by non-terminal leads to starting symbol. [Every matching thing is not a handle.]

Q. How many no. of handles are detected by bottom up parser for the following input string and grammar:

① E → E+T | T
T → T*F | F
F → a

a+a*a

a + T
a + T*F
a + F*F
a + a * F
a + a * a

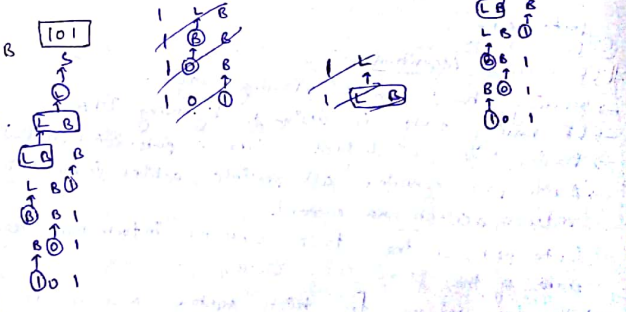
E
E + T
T + T
F + T
a + T

8 handles.

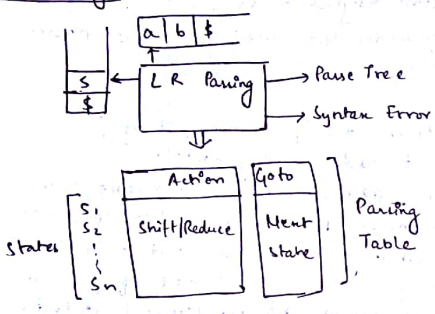
② S → L
L → LB | B
B → 0 | 1

101

7 handles.



L-R Parsing: (Bottom Up):



→ LR parser is shift reduce parser because parsing is done with the help of the following 4 actions:

① Shift Action: Moving input symbols from input buffer to stack is known as shift action.

② Reduce Action: Whenever a handle is detected in the stack, it is replaced by corresponding L.H.S. non-terminal known as reduce action.

③ Accept Action: If the lookahead symbol points to "$" (dollar) & stack contains the starting symbol, accept action is performed.

④ Error Action:

LR Parsing Algorithm:

→ Working based on LR Parsing table.
→ LR Parsing table is designed by using DFA.
→ Parsing table is divided into 2 parts: ① Action Part ② Goto Part
→ Action part contains all possible actions informations (shift, reduce, accept and error).
→ Goto part of the table contains information regarding next state of the DFA.
→ All the tokens for which syntax analysis is performed, are taken in the input buffer along with "$" as end markers.
→ A stack is used in LR parser to identify handles.
→ Let "S" is state of the DFA table present in the top of the stack and "a" is the lookahead symbol.
→ Parser takes decision regarding which action to perform from the parsing table as follows;

① If table action [S,a] (Son a) is "Accept", then parsing is successful and input is valid. [No syntax error] [S,a] = Accept

② If table action [S,a] = Blank, then, there is a syntax error in the input and return it to error handler.

③ If action [S,a] = shift j, then "a" is pushed into the stack and then "j" is pushed into top of the stack and lookahead symbol is incremented.

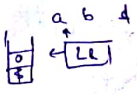
④ If table action [S,a] = reduce A → B, then:
- (a) Pop 2x|B| pop 2 times the length of B symbols from stack.
- (b) push A into stack
- (c) push GoTo(k,A) into top of stack, where "k" is previous DFA state in the stack.

→ The time complexity of LR parsing algorithm is O(n), where "n" is the length of input.

Q. How many shift actions and reduce actions are taken by LR parser to parse the input "ab" by using the following grammar and parsing table.

	Stack	i/p	Action
①	$ | 0	ab$	[0,a] = shift 3
②	$ | 0 | a | 3	b$	[3,b] = r2
③	$ | 0 | A | 2	b$	[2,b] = s5
④	$ | 0 | A | 2 | b | 5	$	[5,$] = r3
⑤	$ | 0 | A | 2 | B | 4	$	[4,$] = r1
⑥	$ | 0 | S | 1	$	(1,$) = accept

	Action			GoTo		
	a	b	$	S	A	B
I0	s3			1	2	
I1			accept			
I2		s5				4
I3		r2				
I4			r1			
I5			r3			



Problem 17:

LR Parsing:

Q. How many shift actions and reduce actions taken by LR Parser to parse the input "id + id x id" by using following grammar and LR parsing table.

① E → E+E
② E → E*E
③ E → id

	id	+	*	$	E
I_0	S_2				1
I_1		S_3	S_4	accept	
I_2		r_2	r_3	r_3	
I_3	S_2				5
I_4	S_2				6
I_5		r_1	S_4	r_1	
I_6		r_2	r_2	r_2	

id+id*id$
↑↑↑↑↑↑

① $|0|id|2
② $|0|id|2
③ pop 2 & length
$|0|E|2
④ $|0|E|1
⑤ $|0|E|1|+|3|id|2
⑥ $|0|E|1|+|3|E|5
⑦ $|0|E|1|+|3|E|5|*|4
⑧ $|0|E|1|+|3|E|5|*|4|id|2
⑨ $|0|E|1|+|3|E|5|*|4|E|6
⑩ $|0|E|1|+|3|E|5
⑪ $|0|E|1

Accept shift = 5 Reduce = 5

LR - Parsing Table Construction:

→ To construct LR parsing table, 4 methods exist, known as:
① LR(0) ② SLR(1) ③ CLR(1) ④ LALR(1)

① LR(0) Parsing Table Construction:

→ To construct LR(0) parsing table for the given grammar:
ⓐ Construct augmented grammar
ⓑ Calculated LR(0) items set, by using closure function and goto function.
ⓒ Construct LR(0) items DFA
ⓓ Construct LR(0) parsing table from DFA.

* Augmented Grammar: means adding a new production S' → S for the original grammar known as augmented grammar.

→ Advantage of construting augmented grammar is in the parsing table.
→ Whenever parser tries to perform reduce action by using augmented production then accept action is called.

* LR(0) Item: Any context free grammar production having (.) dot on the RHS part at some position, is called a LR(0) item.

→ The LR(0) item A → XY.Z means "XY" is there in the stack and parser has to see "Z".
→ In any LR(0) item, the L.H.S part of the (.) dot is present in the stack and RHS part is ready to come into the stack.
→ In any LR(0) item, if (.) dot is at rightmost end, then, reduce action is performed.
→ Corresponding to given grammar, total no of LR(0) items are calculated by using closure() and goto() functions.
→ The task of closure() is, in any LR(0) item, if (.) is followed by non-terminal exists on the RHS part, then closure() adds that non-terminal production and places dot (.) in the first place on the RHS part.

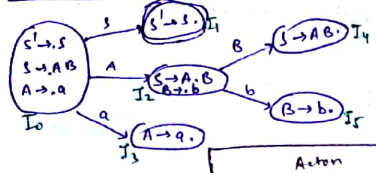
E.g: S → A.B, B → .a || S → .AB, A → .a

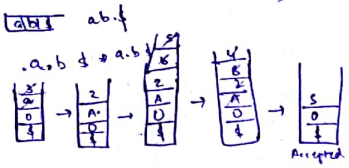
→ The task of goto() if is, it move (.) dot one position ahead towards right side.
→ ~~construct LR(0) parsing table for the following grammar.~~

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

① S → A B
② A → a
③ B → b

LR(0) items DFA





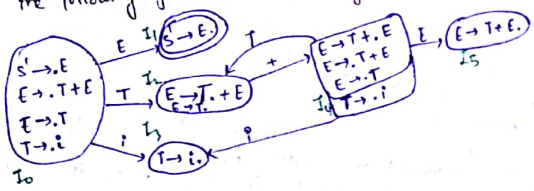
	Action			Goto		
	a	b	$	S	A	B
I_0	S_3			1	2	
I_1			Accept			
I_2		S_5				4
I_3	r_2	r_2	r_2			
I_4	r_1	r_1	r_1			
I_5	r_3	r_3	r_3			

NOTE

→ Any grammar, LR(0) parsing table contains multiple actions in the same box, That grammar is not LR(0) grammar, otherwise, it is LR(0) grammar.

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

① E → T+E
② E → T
③ T → i



	Action			Goto	
	+	i	$	E	T
I_0		S_3		1	2
I_1			Accept		
I_2	S_4/r_2	r_2	r_2		
I_3	r_3	r_3	r_3		
I_4		S_3		5	2
I_5	r_1	r_1	r_1		

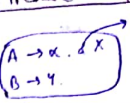
Not LR(0) grammar.

NOTE

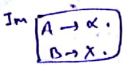
→ If LR(0) items DFA having either shift/reduce conflict state or reduce-reduce conflict state, then, the given grammar is not LR(0) grammar.

→ The following are the conflict states in LR(0) items DFA:

Shift/Reduce Conflict State

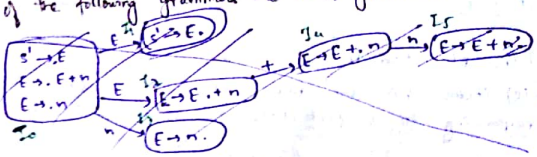


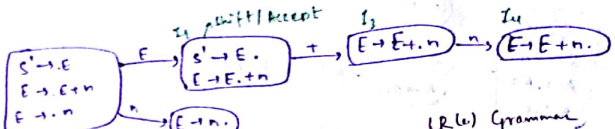
Reduce/Reduce conflict state.



Q. Verify which of the following grammars are LR(0) grammars:

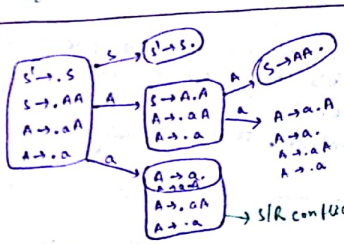
① E → E + n
E → n



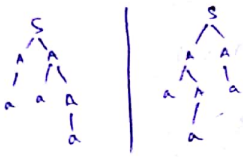


LR(0) Grammar

② S → AA
A → aA
A → a



Not LR(0) grammar.



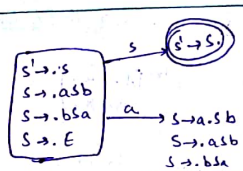
Ambiguous

NOTE All ambiguous grammars are not LR(0) grammars,

→ Any ambiguous grammars, LR(0) parsing table contains either S/R conflict or R/R conflict.

→ All LR(0) grammars are unambiguous grammars, but all unambiguous grammars need not be LR(0).

Q. S → aSb
S → bSa
S → ϵ.

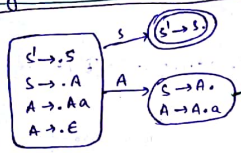


A→.ϵ = A→ϵ.

NOTE The LR(0) item A→.ϵ = A→ϵ.,

→ LR(0) items DFA does not contains ϵ-transitions. Hence, the LR(0) items corresponding to ϵ-productions are reduced in the initial state.

Q. S → A
A → Aa
A → ϵ



Not LR(0) grammar

Drawback of LR(0):

→ In LR(0) parsing table, reduce actions are placed without considering lookahead symbol. This may lead to unnecessary reduce actions in the parsing table.

→ To avoid this drawback, SLR(1) parsing tables are used in LR parser.

first():

→ first() of a non-terminal is set of terminals existing first place of RHS part of grammar production.

→ first() of string abc → first(abc) = {a}

→ If the production is of the form:

① A → aα | bγ then first(A) = {a, b}

② If any production is of the form: A → Bα, where B does not produces ϵ, directly or indirectly, then first(A) = first(B).

③ If any production is of the form: A → Bα, B produces ϵ directly or indirectly, then:
B → ϵ

first(A) = {first(B) − ϵ} ∪ first(α)

Q. Calculate first() of each non-terminal for the following grammar productions.

① E → TE'
E' → +TE' | ϵ
T → FT'
T' → *FT' | ϵ
F → (E) | a

E	{(, a}
E'	{+, ϵ}
T	{(, a}
T'	{*, ϵ}
F	{(, a}

② S → AB
A → a | b
B → d

S	{a, b}
A	{a, b}
B	{d}

③ S → AB
A → a | ϵ
B → d

S	{a, d}
A	{a, ϵ}
B	{d}

④ S → AB
A → a | ϵ
B → b | ϵ

S	{a, b, ϵ}
A	{a, ϵ}
B	{b, ϵ}

⑤ S → ACB | Cb | Ba
A → da | BC
B → g | ϵ
C → h | ϵ

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

⑥ S → aAbA | bBbA | ϵ
A → S
B → S

S	{a, b, ϵ}
A	{a, b, ϵ}
B	{a, b, ϵ}

Follow() set:

→ Follow() of a nonterminal is set of terminals which appears immediately aft RHS part of [after] that non-terminal in some sentential form.

(1) Follow of starting symbol of the grammar is $.

Follow(S) : $

(2) Any production is of the form: $A \to \alpha B X$; $X \neq \epsilon$; Follow(B) = First(X)

(3) $F \to \alpha B$; Follow(B) = Follow(A)

(4) $A \to \alpha B X$, $X \to \epsilon$; Follow(B) = {First(X) − ε} ∪ Follow(A)

NOTE: follow set doesn't contain ε.

Q. Calculate follow() of each nonterminal for the following grammars:

(1) S → AaAb | BbBa
A → ε
B → ε

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

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

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

(3) S → iEtSS' | a
S' → eS | ε
E → b

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

(4) S → aAbA | bBbA | ε
A → S
B → S

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

(5) S → L=R
S → R
L → *R
L → id
R → L

S	{$}
L	{=, $}
R	{$, =}

Q. Find first & follow?
S → aAbB | bAaB | ε
A → S
B → S

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

Simple LR(1) [SLR(1)]

SLR(1)
LR(0)

* Comparison between LR(0) and SLR(1) parsing tables:

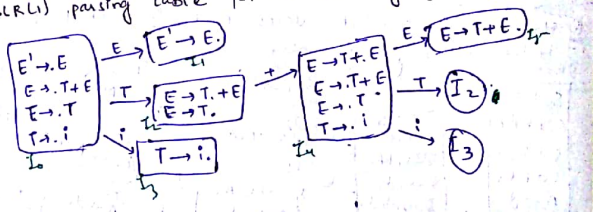
(1) No. of states are same in both tables.
(2) No. of shift actions are same in both tables.
(3) No. of goto table entries are same in both tables.
(4) No. of reduce actions may be different in both tables.
(5) No. of error actions may be different in both tables.
(6) To construct LR(0) and SLR(1) parsing tables, LR(0) items automata (DFA) is used.

NOTE: In SLR(1) parsing table, reduce actions are placed is based on L.H.S. nonterminal's follow() set values.

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

① E → T+E
② E → T
③ T → i

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



	+	i	$
I_0		S_3	
I_1			accept
I_2	S_4		r_2
I_3	r_3		r_3
I_4		S_3	
I_5			r_1

	E	T
I_0	1	2
I_1		
I_2		
I_3		
I_4	5	2
I_5		

NOTE
Any grammar, SLR(1) parsing table contains either SR conflict or RR conflict, then that grammar is not SLR(1) grammar. Otherwise, SLR(1) grammar.

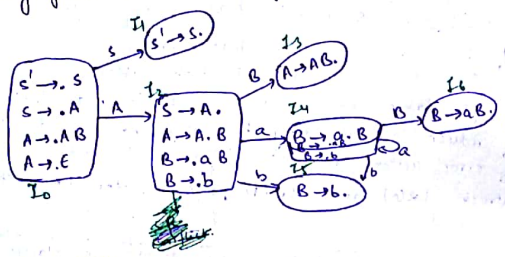
→ The following are conflict states corresponding to SLR(1) parsers.

S/R Conflict: $A \to \alpha . a\beta$, $B \to \gamma .$ (terminal present in follow(B))
if $a \in$ follow(B)

R/R Conflict: $A \to \alpha .$, $B \to \gamma .$
Follow(A) ∩ Follow(B) ≠ ϕ

Q. Verify following grammars are SLR(1) grammars or not.

$S \to A$
$A \to AB$
$A \to \epsilon$
$B \to aB$
$B \to b$



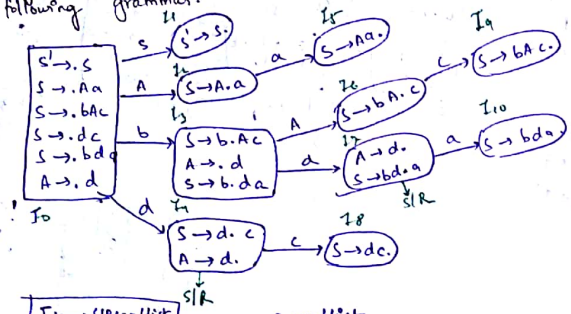
Follow:

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

No conflict.

Q. How many conflict states exist in SLR(1) parser corresponding to the following grammar.

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



Follow:

S	{$}
A	{a, c}

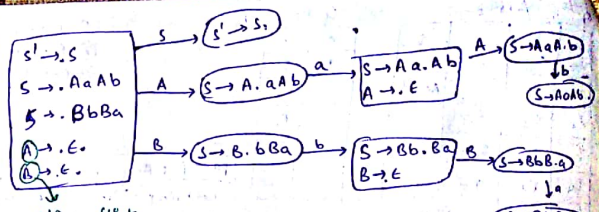
$I_4 \to$ S/R conflict
$I_7 \to$ S/R conflict

2 conflicts

NOTE → All LR(0) grammars are SLR(1) grammars, but all SLR(1) grammars need not be LR(0) grammars.

→ All ambiguous grammars are not SLR(1) grammar.
→ Any ambiguous grammar, SLR(1) parsing table contains either S/R conflict or R/R conflict.
→ All SLR(1) grammars are unambiguous grammars.

Q.
$S \to Aa\,Ab$
$S \to Bb\,Ba$
$A \to \epsilon$
$B \to \epsilon$



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

Not CLR(1) Grammar

Drawback of SLR(1):

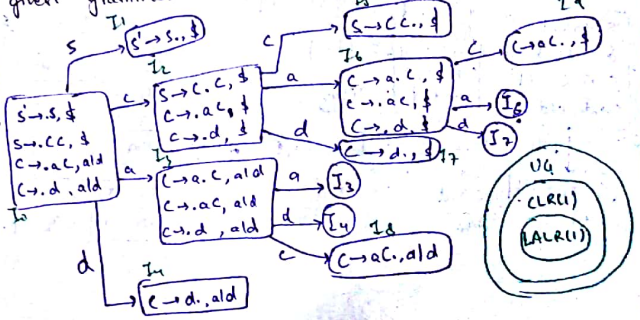
→ In SLR(1) parsing, reduce actions are placed based on follow() set values.
→ follow() set may result more number of elements than actual lookahead, hence, parsing table contains unnecessary reduce actions.
→ To place reduce actions properly, CLR(1) parsing tables are used.

CLR(1) (canonical LR(1)) and Lookahead LR(1) [LALR(1)]:

→ To construct CLR(1) and LALR(1) tables, LR(1) items DFA is used.
⇒ LR(1) item is nothing but any LR(0) item associated with lookahead.

Q. Construct CLR(1) parsing table for the following grammar.
Q. Verify given grammar is CLR(1) grammar or not.
Q. Construct LALR(1) parsing table for the following grammar.
Q. Verify given grammar is LALR(1) grammar or not.

$S \to CC$
$C \to aC$
$C \to d$



	Action			Goto	
	a	d	$	S	C
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

① S→CC
② C→aC
③ C→d

NOTE: Any grammar CLR(1) table contains either s/R conflict or R/R conflict, then that grammar is not CLR(1) grammar.
Otherwise CLR(1) grammar.

Drawback of CLR(1):

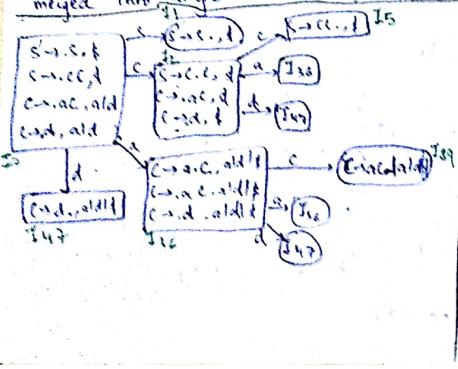
→ The table size of CLR(1) is more compare to LALR(1). To avoid this drawback LALR(1) parsing tables are used in LR parser.

→ General programming language compilers use LALR(1) parsing tables only.

⇒ To construct LALR(1) table, the states of LR(1) items DFA may be merged.

⇒ In LR(1) items DFA any states having same core part and different lookahead part are merged into single state by combining all lookahead symbols.

⇒ For the given grammar, the following states of LR(1) items DFA are merged into single state: I_{36}, I_{47}, I_{89}.

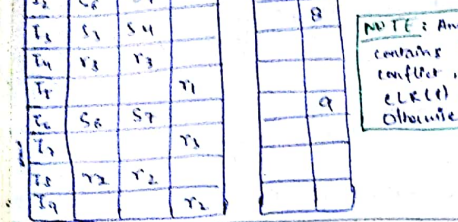


	a	d	$	S	C
I_0	S_{36}			1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}			r_3		
I_5					
I_{89}	r_2	r_2	r_2		

NOTE: Any grammar, LALR(1) table contains s/R or R/R conflict, it is not LALR(1), otherwise it is LALR(1).

Q. Verify following grammar is CLR(1) or not?
Q. Verify following grammar is LALR(1) or not?

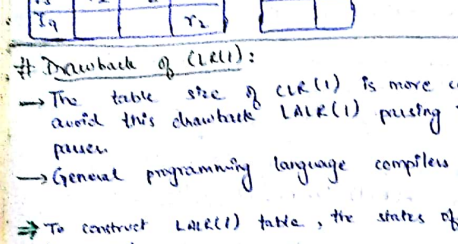
S→(S)
S→a



CLR(1) Grammar ✓

	Action					Goto	
	a	(	)	$	S	
I_0	S_3	S_2			1	
I_1				Accept		
I_2	S_6	S_5			4	
I_3				r_2		
I_4			S_7			
I_5	S_6	S_5		r_2	8	
I_6			r_1			
I_7			S_9			
I_8			r_1			
I_9						

LALR(1):



	Action				Goto	
	a	(	)	$	S	
I_0	S_{36}	S_{25}			1	
I_1				Accept		
I_{25}	S_{36}	S_{25}			48	
I_{36}			r_2	r_2		
I_{48}			S_{59}			
I_{59}			r_1	r_1		

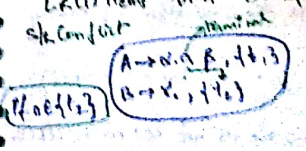
LALR(1) Grammar ✓

No. of states in Parsing Table

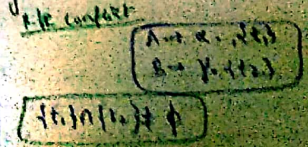
LR(0): CLR(1): LALR(1) ≤ CLR(1)

→ No. of conflict states in LALR following are the conflict states in LR(1) items DFA corresponding to CLR(1) and LALR(1) parsing tables.

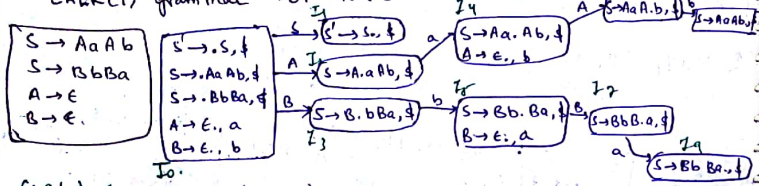
s/R Conflict



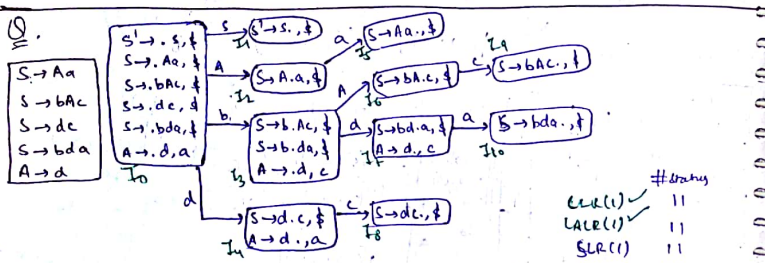
R/R Conflict



Q. Verify the following grammar is CLR(1) grammar or not, LALR(1) grammar or not.



Q.



① Find no. of states in LR(0), SLR(1), CLR(1), & LALR(1) parsing tables.
② Verify that grammar is CLR(1) and LALR(1).

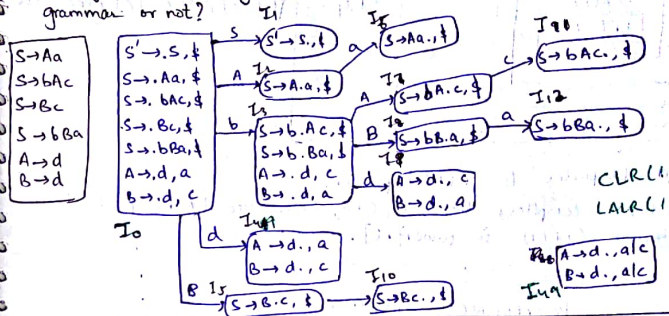
NOTE

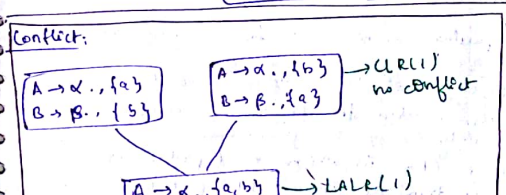
① If there is no R/R conflict in CLR(1) table, then there may be chance of having reduce R/R conflict in LALR(1) table. [Because of merging states]
② If there is no S/R conflict in CLR(1) table, then, no chance of having S/R conflict in LALR(1) table.
③ If there is a S/R conflict in LALR(1) table, then, S/R conflict should exist in CLR(1) table.
④ If there is R/R conflict in CLR(1) table, then, R/R conflict should exist in LALR(1) table.
⑤ If there is no conflict in LALR(1) table, then, no chance of having conflict in CLR(1) table.

Hence, all LALR(1) grammars are CLR(1) grammars, but all CLR(1) grammars need not be LALR(1) grammars.
All, CLR(1) grammars are unambiguous grammars but all ~~all~~ i.e. need not be CLR(1).

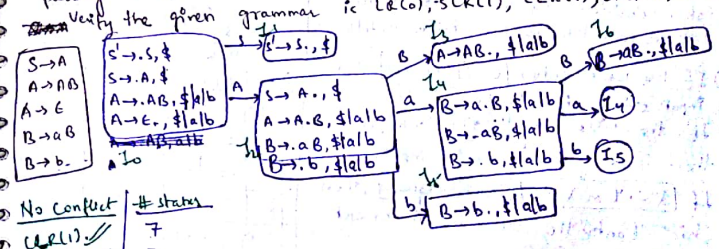
→ Any ambiguous grammar is not CLR(1) and not LALR(1) because, ambiguous grammar parsing table having conflicts.

Q. Verify the following grammar is CLR(1) grammar or not, LALR(1) grammar or not?

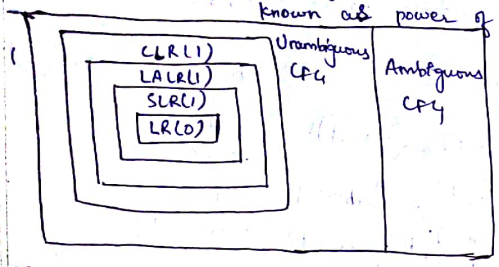




H.W. Find the number of states of LR(0), SLR(1), CLR(1), LALR(1) parsing tables.
Verify the given grammar is LR(0), SLR(1), CLR(1), LALR(1) or not?



*Power of a parser: No. of grammars suitable for particular parser is known as power of that parser.



[LR(0) < SLR(1) < CLALR(1) < CLR(1)]

⇒ By default LR parser is CLR(1) parser.
⇒ The practical compilers like C, C++, Java use LALR(1) parsing table [even though CLR(1) is powerful].

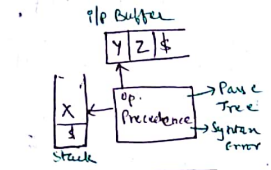
OPERATOR PRECEDENCE PARSING:

→ Operator precedence parsing can be used for ambiguous grammars also.
→ Operator precedence parsing is simple to design compared to LR parser.
→ Operator precedence parser is also shift-reduce parser.
→ O.P. parsing suitable only for operator grammars.
→ Operator grammar is a CFG in which no ε-productions exist and no adjacent non-terminals on the RHS part of the grammars.

Parsing Algorithm:

→ Let X is top most terminal on the stack and Y is lookahead symbols, then parser takes decision regarding shift/reduce action from the parsing table as follows:

① If the table entry [X,Y] = Accept then input string is valid.
② If the table entry [X,Y] = Blank, then there is a syntax error in input.
③ If [X < Y or X ≐ Y] (lesser or equal), then shift Y onto the stack and increment lookahead symbol.
④ If [X > Y] (greater pre.), then there is a handle in the stack and perform reduce action.



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

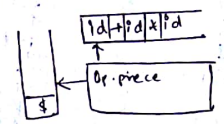
Op. precedence Table.

To perform reduce action, pop symbols from stack, such that the terminal on top of stack has lesser precedence over

→ To perform reduce action, pop the symbols from stack, such that the terminal on top of stack has less precedence over most recently popped out terminal.
→ The time complexity of operator precedence parsing algo. is O(n), where 'n' is the length of input.
→ The size of operator precedence parsing table is (T+1)² (T = terminal)

Q. How many shift and reduce actions are taken by O.P. parser to parse the input "id+id*id" by using the following grammar and parsing table.

E → E+E
E → E*E
E → id



	Stack	Input	action
①	$	<· id+id*id$	shift
②	$id	·> id+id*id$	reduce
③	$E	<· +id*id$	shift
④	$E+	<· id*id$	shift
⑤	$E+id	·> *id$	reduce
⑥	$E+E	<· *id$	shift
⑦	$E+E*	<· id$	shift
⑧	$E+E*id	·> $	reduce
⑨	$E+E*E	·> $	reduce
⑩	$E+E	·> $	reduce
⑪	$E	— $	accept

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

	Stack		Input	action	
①	$	<·	id * id + id $	shift	S-S
②	$ id	·>	* id + id $	reduce	S-R
③	$ E	<·	* id + id $	S	
④	$ E *	<·	id + id $	S	
⑤	$ E * id	->	+ id $	R	
⑥ (id)	$ E * E	·>	+ id $	R	
⑦	$ E	<·	+ id $	S	
⑧	$ E +	<·	id $	S	
⑨	$ E + id	·>	$	R	
⑩	$ E + E	->	$	R	
⑪	$ E		$	accept	

Construction of Operator Precedence parsing table:

→ Operator precedence parsing table construction is different for ambiguous grammars and unambiguous grammars.

→ If the grammar is unambiguous, then, to get precedences of operators parse tree method is used.

→ If the grammar is ambiguous, general programming language or general mathematical precedences are inserted in the table.

NOTE

→ If any operator grammar contains bothe left recursion and right recursion, then, it is ambiguous.

→ In any operator grammar, either left recursion or right recursion, but not both and different precedence operators present at different non-terminals, then it is unambiguous.

*Operator Precedence Grammar:

→ A CFG is said to be operator precedence grammar if it is operator grammar and atmost one precedence relation exists between every pair of terminals.

→ In operator precedence parsing table, "$" is lesser precedence over all other operators.

→ Identifier is having higher precedence over all other operators.

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

A → B/A/B
B → B*C/C
C → D+C/D
D → a

	a	+	*	/	$
a		·>	·>	·>	·>
+	<·	<·	·>	·>	·>
*	<·	<·	·>	·>	·>
/	<·	<·	<·	<·	·>
$	<·	<·	<·	<·	accept

A → A*A/A+A/a Construct op. prec. parsing table for the following ambiguous grammar.

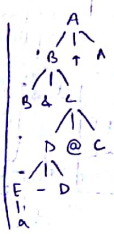
+ → left associative
* → " "

Use general precedence and associative rules

	a	+	*	$
a		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	accept

Q. Which of the following is true about precedence and associativity of operators from the following grammar.

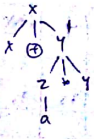
A → B↑A/B
B → B&C/C
C → D@C/D
D → E-D/E
E → a



(a) "@" is lesser precedence than "&" and "-"
(b) Both "&" and "@" are left associative
(c) "-" operator is higher precedence than "↑", "@" and "-" operator is right associative.
(d) All operators present in grammar are right associative

Q. Consider the following grammar: which is true?

X → X⊕Y/Y
Y → Z*Y/Z
Z → a



(a) Both "⊕" and "*" are left associative
(b) Both "⊕" and "*" are right associative.
(c) (d) ⊕ is left associative & * is right associative.
(d) None of these.

Q. Consider the following grammar, which of the following is true:

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

(a) "*" is higher precedence than "+"
(b) "−" is higher precedence than "*"
(c) "+" and "−" have same precedence
(d) "+" is higher precedence than "*"

Drawbacks of Operator Precedence Parsing:

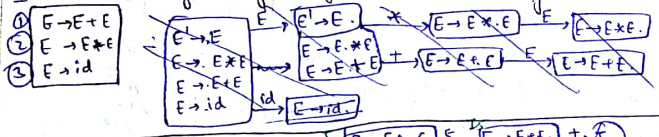
→ The power of operator precedence parser is very less compared to LR parser [suitable only for operator precedence grammars].
→ Hence, LR parser is powerful compared to operator precedence parser in bottom up parsing.

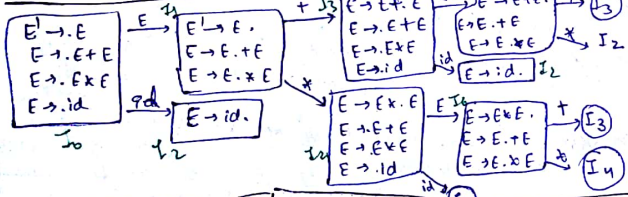
05/08/2017:

Handling Ambiguous Grammar in LR Parsing:

Q. Handle following ambiguous grammar in LR Parsing:

1. $E \to E + E$
2. $E \to E * E$
3. $E \to id$





	id	+	*	$
I_0	S_2			
I_1		S_3	S_4	
I_2		r_3	r_3	r_3
I_3	S_2			
I_4	S_2			
I_5		~~S_3~~/r_1	S_4/r_1	r_1
I_6		~~S_3~~/r_2	~~S_4~~/r_2	r_2

E
1
5
6

$E \{+, *, \$\}$

Case 1: The S/R conflict at I_6 state on "*" is resolved by giving priority for shift action over reduce action in order to make "*" is higher precedence than "+".

Case 2: The S/R conflict at I_5 state on "+" is resolved by giving priority for reduce action over shift action in order to make "+" as left associative.

Case 3: The S/R conflict at I_6 state on "+" is resolved by giving priority for reduce action over shift action in order to make "*" as higher precedence than "+".

Case 4: The S/R conflict at I_6 state on "*" is resolved by giving priority for reduce action over shift action in order to make "*" as left associative.

→ All expression grammar's conflicts in the parsing table are resolved, based on precedence and associativity of operators.
→ To design lexical analyser, there exists a tool, known as LEX tool.
→ To design parser (LR), there exists a tool known as YACC [Yet another compiler compiler].

S/R Conflict:

→ For general ambiguous grammars S/R conflict is resolved by giving priority for shift action over reduce action.
→ If parsing is success for shift action, then shift action is suitable in the parsing table. Otherwise, reduce action is suitable in the parsing table.

R/R Conflict:

→ For general ambiguous grammars, R/R conflict in the parsing table is resolved by giving priority for first reduce action over second reduce action.
→ If the parsing is success for first action, then first reduce action is suitable in the parsing table otherwise, second reduce action is suitable.
→ The LR parser generator tool (YACC) also resolves the conflicts in the same manners.

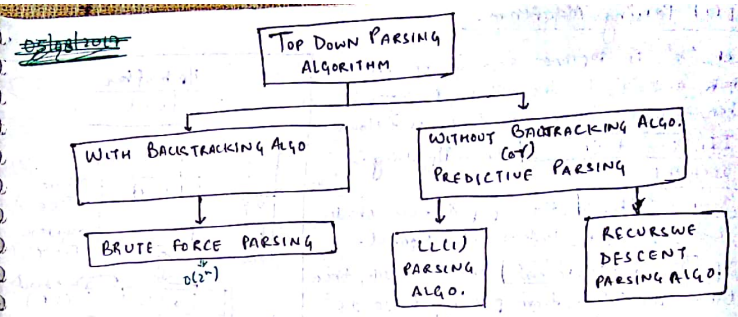
Q. Consider the following expression grammar and it is fed to a YACC tool. Then which of the following is true about the action of YACC tool for the given grammar [precedence and associativities of operators are not known to YACC tool].

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

(a) It detects recursion and eliminates recursion.
(b) It detects R/R conflict and resolves the conflict by giving priority for first reduce action over second reduce action.
(c) It detects S/R conflicts and resolves the conflict by giving priority for shift action over reduce action.
(d) It detects S/R conflict and resolves the conflict by giving priority for reduce action over shift action.

② If the conflicts are resolved in the above manner what precedence and associativity rules, parser will realise? What is the value of the expression "3*2+1"?

(a) Both operators are equal precedence, left associative and expression value 7.
(b) Both operators equal precedence, right associative, expression value 9.
(c) "*" is higher precedence than "+", both operators left associative, expression value 7.
(d) "+" is higher precedence than "*", both are left associative, expression value 9.



→ Top Down parser constructs the parse tree, starts from the starting symbol of the grammar and proceeds until given string.
→ Top Down parser uses leftmost derivation while constructing the parse tree.
→ To construct top-down parser, two type of algorithms exist, known as with backtracking algorithms and w/o backtracking algorithms.
→ The major problem in top-down parsing is, if a non-terminal has multiple productions on the RHS part, then selecting right production for parse tree construction is difficult.

* Brute Force Parsing: In brute force parsing, if a non-terminal having multiple productions on the RHS part of the grammar then priority will be given for first production to construct the parse tree.
→ If the string is not generated then priority will be given to next production and so on.
→ Backtracking parser requires exponential time complexity, hence, not suitable for practical compilers.

Predictive Parsing Algorithms:

→ In predictive parsing, if a non-terminal having multiple productions on the RHS part, then, there is a mechanism to select the right production to construct parse tree.
→ To construct predictive parsers, grammar should not contain left recursion and it should not be non-deterministic grammar.

#LL(1) Parsing Algorithm:

→ Let "X" is topmost symbol on the stack and "a" is the lookahead symbol, then parser takes decisions as follows:

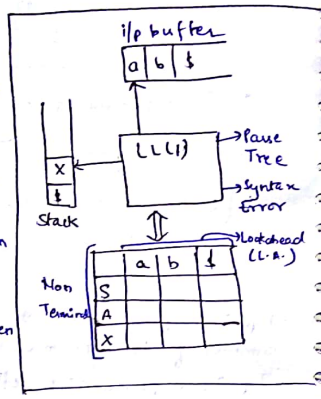
① If [X=a=$] [Accept] then i/p string is accepted.

② [X=a≠$] Pop X, then pop "X" from stack and increment lookahead symbol.

③ If [X is nonterminal] in the stack, then take decision from parsing table as follows:

- If the table entry [X,a]=X→uvw, then replace X by uvw in the stack in reverse order ["u" appears on top of stack].

④ If the table entry [X,a] = blank then there is a syntax error.



→ The time complexity of LL(1) parsing algorithm is O(n), where "n" is the length of input.

→ The size of LL(1) parsing table is (No.of N.T.)*(T+1).

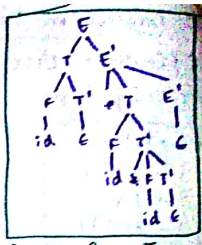
→ Initially starting symbol of the grammar is pushed onto the stack.

Q. How many total no. of steps are taken by LL(1) parser to parse the input "id+id*id" by using following grammar and LL(1) parsing table.

E → TE'
E' → +TE' | ∈
T → FT'
T' → *FT' | ∈
F → id

	id	+	*	$
E	E→TE'			
E'		E'→+TE'		E'→∈
T	T→FT'			
T'		T'→∈	T'→*FT'	T'→∈
F	F→id			

Stack	Input	Action
① $E	id+id*id$	E→TE'
② $E'T	id+id*id$	T→FT'
③ $E'T'F	id+id*id$	F→id
④ $E'T'id	id+id*id$	Pop id
⑤ $E'T'	+id*id$	T'→∈
⑥ $E'	+id*id$	E'→+TE'
⑦ $E'T+	+id*id$	Pop +
⑧ $E'T	id*id$	T→FT'
⑨ $E'T'F	id*id$	F→id
⑩ $E'T'id	id*id$	Pop id
⑪ $E'T'	*id$	T'→*FT'
⑫ $E'TF*	*id$	Pop *
⑬ $E'T'F	id$	F→id
⑭ $E'T'id	id$	Pop id
⑮ $E'	$	T'→∈
⑯ $E	$	E'→∈
⑰ $	$	Accept



Output Parse Tree

"id*id+id" input

Stack	Input	Action
① $E	id*id+id$	E→TE'
② $E'T	id*id+id$	T→FT'
③ $E'T'F	id*id+id$	F→id
④ $E'T'id	id*id+id$	Pop id
⑤ $E'T'	*id+id$	T'→*FT'
⑥ $E'T'F*	*id+id$	Pop *
⑦ $E'T'F	id+id$	F→id
⑧ $E'T'id	id+id$	Pop id
⑨ $E'T'	+id$	T'→∈
⑩ $E'	+id$	E'→+TE'
⑪ $E'T+	+id$	Pop +
⑫ $E'T	id$	T→FT'
⑬ $E'T'F	id$	F→id
⑭ $E'T'id	id$	Pop id
⑮ $E'T'	$	T'→∈
⑯ $E'	$	E'→∈
⑰ $	$	Accept

Construction of LL(1) Parsing Table:

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

$E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to id \mid (E)$

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

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

→ Any grammar, LL(1) parsing table contains multiple productions in the same box, that grammar is not LL(1) grammar.
→ All ambiguous grammars, LL(1) parsing table contains multiple productions.
→ Hence, all ambiguous grammars are not LL(1) grammars.
→ All, LL(1) grammars are unambiguous grammars but all unambiguous grammars need not be LL(1) grammars.

Venn: LL(1)

Q. Construct LL(1) parsing table and also verify given grammar is LL(1) or not.

① $S \to aABC$
$A \to a \mid bb$
$B \to a \mid \epsilon$
$C \to b \mid \epsilon$

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

LL(1) ✓

	a	b	$
S	$S \to aABC$		
A	$A \to a$	$A \to bb$	
B	$B \to a$	$B \to \epsilon$	$B \to \epsilon$
C		$C \to b$	$C \to \epsilon$

② $S \to aAbB \mid bAaB \mid \epsilon$
$A \to S$
$B \to S$

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

Not LL(1)

	a	b	$
S	$S \to aAbB$, $S \to \epsilon$	$S \to bAaB$, $S \to \epsilon$	$S \to \epsilon$
A	$A \to S$	$A \to S$	
B	$B \to S$	$B \to S$	$B \to S$

③ $S \to iEtSS'/a$
$S' \to eS \mid \epsilon$
$E \to b$

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

	i	t	e	a	b	$
S	$S \to iEtSS'$			$S \to a$		
S'			$S' \to eS$, $S' \to \epsilon$			$S' \to \epsilon$
E					$E \to b$	

④ $S \to AaAb \mid BbBa$ LL(1) ✓
$A \to \epsilon$
$B \to \epsilon$

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

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

LL(1) Grammar Detection:

① A single production grammar is always LL(1) grammar.
② Any grammar is of the form $A \to \alpha_1 \mid \alpha_2 \mid \ldots \alpha_n$, then if $first(\alpha_1), first(\alpha_2)$ and so on $first(\alpha_n)$ are pairwise disjoins, then the grammer is LL(1), otherwise not LL(1).
③ Any grammar is of the form $A \to \alpha \mid \epsilon$, then $first(\alpha) \cap follow(A)$ equal to ϕ, then it is LL(1).
$first(\alpha) \cap follow(A) = \phi$
④ Left recursive grammars are not LL(1), but elimination of left recursion may result LL(1) grammars.
⑤ Non-Deterministic grammars are not LL(1) but left factoring of ND grammar may result into LL(1) grammar.

Q. Which of the following grammars are LL(1).

① S → aSb | bSa | c ✓

② S → aSbS | bSaS | ∈ ✗

③ S → aSa | bSb | ∈ ✗

④ S → aS' | ∈ ✓

⑤ S → Sa | b ✗

S → bS' ✓
S' → aS' | ∈ ✓
} LL(1) ✓

*Non-Deterministic Grammars: If a grammar having multiple productions on the RHS part where each production having some common prefix, that grammar is non-deterministic grammar.

Eg. ① S → aa | ab
S → aS'
S' → a | b

② S → ;EtS | ;EtSeS | a
S' → ;EtSS' | a
S' → eS | ∈
E → b

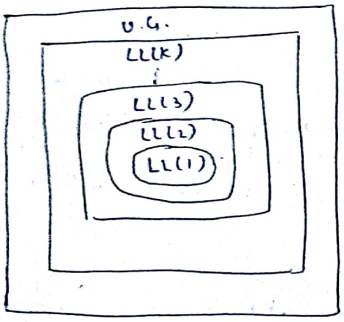
*Left factoring
*Eliminating left factoring does not remove ambiguity

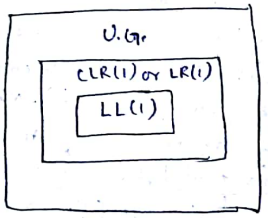
GATE

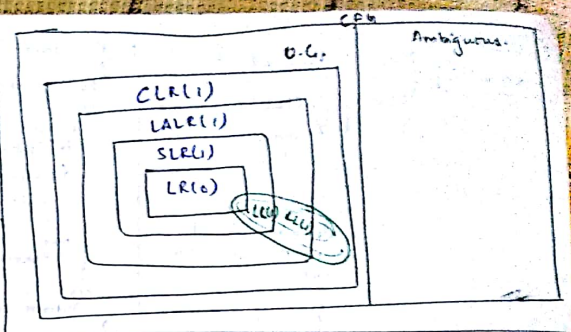
Q. Which of the following is sufficient to convert an arbitrary CFG into LL(1) grammar.

(a) Elimination of left recursion alone.
(b) Factoring the grammar alone.
(c) Both (a) and (b)
(d) None of these.

NOTE: It is undecidable to convert every CFG into LL(1) grammar.







L → Left to Right scanning of i/p.
L → Leftmost derivation
(k) → Lookahead of length k

LR(1) = LR(k)

Q. Consider the following grammars; Given grammar is:
S → aSbl bSalc.

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

RECURSIVE DESCENT PARSING:

$E \rightarrow num\ T$

$T \rightarrow *num\ T \mid \epsilon$

```
Main()
{
 E();
}
```

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

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

```
Match()
{lookahead = nexttoken;
}
error()
{print("syntax error");
}
```

→ The power of recursive descent parser is less compared to LL(1) parsers.

→ Recursive Descent parser not using predictive parsing tables. Hence, for large class of grammars, parser may fall into back tracking.

→ Hence, LL(1) parser is used in top-down parsing to construct without backtracking parsers.

SEMANTIC ANALYSIS PHASE:

→ The following are some of the semantic errors, parser cannot detect:

1. Using a variable without declaration.
2. Declaration of same variable multiple times in the same scope.
3. Expression statements which are not type compatible.
4. Scope violation statements.
5. Mismatch of formal and actual parameters.
6. Type mismatches in formal and actual parameters.

→ To construct semantic analysis phase, CSGs can be used. Hence, semantic analysis phase is also called as context sensitive analysis phase.

→ Content Sensitive Grammars are difficult to implement. Hence, another form of CSGs, known as attribute grammars are used to design semantic analysis phase.

→ Attribute grammar is a CFG where each grammar symbol is associated with a set of attribute values, where attribute can be a type information or scope information of a variable.

→ To compute attribute information of grammar symbols, semantic rules are attached to grammar productions.

Syntax Directed Translation (SDT):

→ SDT is a mechanism where each CFG production is associated with set of semantic rules.

→ Semantic rules are nothing but rules to compute attribute information in the annotated parse tree.

* Annotated Parse Tree: It is a parse tree, where each grammar symbol is decorated with set of attribute information.

→ To construct semantic analysis phase and intermediate code generation phase, SDT is constructed for the requirement.

→ Hence, semantic errors are detected, complete symbol is constructed and intermediate code is generated by constructing SDTs.

#Steps to construct SDT:

1. Construct the parse tree.
2. Construct annotated parse tree.
3. Compute attribute information in the annotated parse tree.
4. Generalised rules of attribute combination in the annotated parse tree, attached to grammar production as semantic rules.

Q Construct the SDT, to detect semantic error in the expression statement of a language having int and boolean datatypes. Where type is attribute that computes type information at every grammar symbol. Then write semantic rules by using following grammar productions:

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

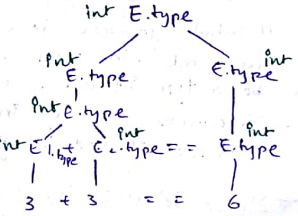
$E \to E_1 == E_2$ { if (E_1.type == E_2.type) then E.type = bool else "semantic error" }

$E \to (E_1)$ → E.type = E_1.type

$E \to$ num → E.type = num.type

$E \to$ true → E.type = bool

$E \to$ false → E.type = bool

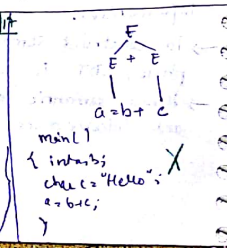


→ SDTs are carried out along with parser.
→ Whenever parser is performing parsing by using grammar productions, the semantic rules associated with grammar productions also executed, hence, verifies semantic features of the language.
→ SDTs can be given to Top-down or bottom-up parser.

#SDT for checking expressions not type compatible: 06/08/24

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

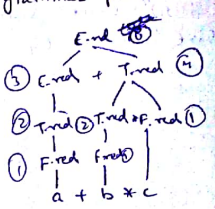
$E \to E_1 + E_2$ { if (E_1.type == E_2.type) and (E_1.type = int) then E.type = int else "Semantic error"; }



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

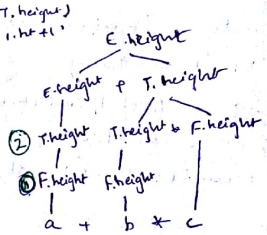
Q Construct the SDT that would calculate total no. of reductions taken by a S/R parser, where "red" is attribute that gives no. of reductions at every place and the root value "E.red" would give total no. of reductions value.
Write semantic rules by using following grammar productions.

$E \to E + T$ {E.red = (E1.red + T.red) + 1}
$E \to T$ {E = T.red + 1}
$T \to T * F$ {T.red = (T1.red + F.red) + 1}
$T \to F$ {T.red = F.red + 1}
$F \to id$ {F.red = 1}



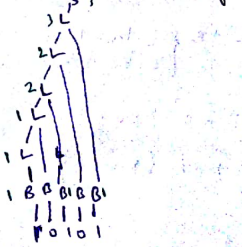
Q Construct the SDT that would compute total height of the parse trees, where height is attribute and the root value E.height would give total height of the parse tree. Write semantic rules for the following grammar productions. if (E_1.height > T.height) then E.ht = E_1.ht + 1 else

$E \to E + T$ {E.height = (E_1.height > T.height) ? E_1.height + 1 : T.height + 1}
$E \to T$ {E.height = T.height + 1}
$T \to T * F$ {T.height = (T1.height > F.height) ? T1.height + 1 : F.height + 1}
$T \to F$ {T.height = F.height + 1}
$F \to id$ {F.height = 1}



Q Construct the SDT that would count total no. of 1's present in the given binary string, where count is attribute and S.count given total no. of 1's value. Write semantic rules for the following grammar productions.

$S \to L$ {S.count = L.count}
$L \to LB$ {L.count = L1.count + B.count}
$L \to B$ {L.count = B.count}
$B \to 1$ {B.count = 1}
$B \to 0$ {B.count = 0}



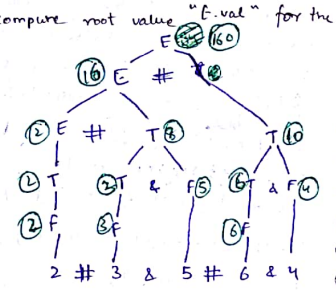
Q. Construct the SDT that would give total no. of balanced parenthesis information, where "count" is attribute and the root value "S.count" would give total no. of balanced parenthesis information.
Write semantic rules for the following grammar productions.

$S \to (S)$ { S.count = S1.count + 1 }
$S \to \epsilon$ { S.count = 0 }

GATE

Q. Consider the following SDT and compute root value "E.val" for the expression 2#3&5#6&4.

$E \to E_1 \# T$ { E.val = E1.val * T.val }
$E \to T$ { E.val = T.val }
$T \to T_1 \& F$ { T.val = T1.val + F.val }
$T \to F$ { T.val = F.val }
$F \to num$ { F.val = num.lexval }



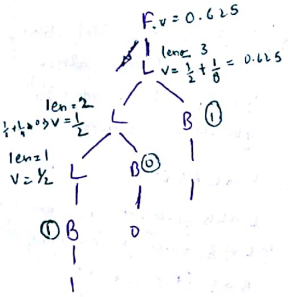
Q. F → L { F.v = L.v }

$L \to L_1 B$ { L.len = L1.len + 1, L.v = L1.v + 2^-L.len * B.v }

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

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

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



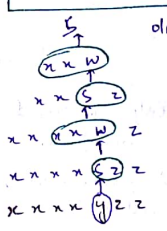
Q. Consider the following SDT:

$S \to xxW$ { print(1) }
$S \to y$ { print(2) }
$W \to Sz$ { print(3) }

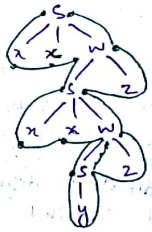
① This SDT is carried out along with S/R parser, then what o/p it will produce for the input "xxxxyzz"

② If this SDT is carried out along with LL(1) parser, then what o/p it will produce for the input "xxxxyzz".

NOTE
→ If the SDT is carried out along with bottom up parser. Corresponding to every reduce action, the semantic rule associated with the production is executed.
→ If the SDT is carried out along with a top down parser, then semantic rules are executed while traversing the parse tree depth first search (left to right manner), in which whenever a non-terminal visited 2 times, the semantic rule corresponding to that non-terminal production is executed.



o/p = 2 3 1 3 1

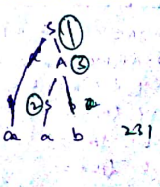


2 3 1 3 1

NOTE
→ If the SDT is carried out along top down parser or bottom up parser, the output is same for the given same input.

Q.
$S \to aA$ {print(1)}
$S \to a$ {print(2)}
$A \to Sb$ {print(3)}
"aab" → o/p ?

2 3 1



2 3 1

Synthesized attribute: If the attribute value at a node is computed from its children, then it is known as synthesized attribute.

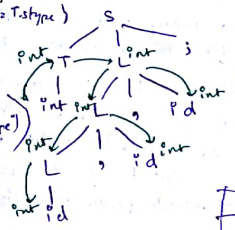
A → x

Inherited attribute: If the attributed value at a node is computed either from parent or from left siblings, then it is known as inherited attribute.

A → x

Q. Construct the SDT to fill type information in the ~~entry~~ symbol table. by using the following grammar productions.

S → TL ; { T.stype = int } { L.itype = T.stype }
T → int { T.stype = int }
T → float { T.stype = float }
L → L , id { L1.itype = L.itype addtype("id.name", L.itype) }
L → id { addtype("id.name", L.itype) }



int a, b, c ;

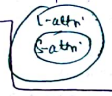
lex	token	type
a	id	int
b	id	int
c	id	int

NOTE

→ Any syntax directed definition which contains only synthesized attributes, then it is called as S-attributed definition.

→ Any syntax directed definition that contains both synthesized attributes and inherited attributes, then it is called as L-attributed definition.

→ All S-attributed definitions are L-attributed definitions, but all L-attributed definitions need not be S-attributed definitions.



S → 2AC {C.x = A.x} I
S → 3ABMC {M.x = A.x + B.x} I
C → 4 {C.x = 4} S
M → ε {M.x = 0}
B → ε {B.x = C
A → ε {A.x =

→ There are two types of formalisms exist to construct semantic analysis phase and intermediate code generation phase. They are:

① Syntax Directed Definition (SDD)
② Syntax Directed Translation Scheme (Translation Scheme)

→ In SDD, all the semantic rules corresponding to synthesized attributes and inherited attributes are placed at right most end of grammar productions.

→ In SDT, semantic rules may not be placed in proper places, hence, attribute values may not be available in the attribute computation time.

→ To avoid this drawback, translation schemes can be constructed.

→ In translation scheme, all the semantic rules corresponding to synthesized attributes and inherited attributes are placed in proper places.

→ Hence, every attribute value is available in the attribute ~~completion time~~ computation time.

→ In translation scheme, all the semantic rules corresponding to synthesized attribute are placed at right most end of grammar productions.

→ In translation scheme, all the semantic rules corresponding to inherited attributes of a non-terminal is placed before that non-terminal and after the depending variable.

→ All SDT, having only synthesized attributes are known as Post fix SDT. because all semantic rules are placed at right most end.

NOTE

→ Translation scheme is a SDT in which semantic rules may be embedded within grammar ~~productions~~ symbols of RHS part of the production.

Q. Convert the following SDD int SDT (Translation scheme).

S → TL ; { L.itype = T.stype }
T → int { T.stype = int }
T → float { T.stype = int }
L → L1 , id [L1.itype = L.itype
addtype("id.name", L.itype)]
L → id [addtype("id.name", L.itype)]

SDT:

S → T {L.itype = T.stype} L ;
T → int {T.stype = int}
T → float {T.stype = float}
L → {L1.itype = Litype} L1 , id {addtype ("id.name", L.itype)}
L → id {addtype ("id.name", L.itype)}

Translation scheme with top-down parsing:

→ If the translation scheme is carried out along with a top-down parser, then semantic rules are executed by traversing the tree DFS (left to right manner).
→ Corresponding to translation scheme, an edge is provided for semantic rule also in the annotated parse tree.
→ If the translation scheme is carried out with top-down parser then left recursion should be eliminated from it.

Q. Eliminate left recursion from the following translation scheme.

E → E + T {print (+)}
E → T
T → T * F {print (*)}
T → F
F → id {print (id.lexval)}

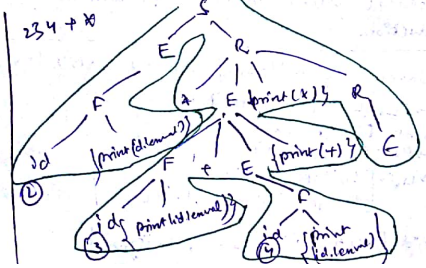
~~E → E + T~~ E → F + T {print (+)} | T,
E → T E'
E' → + T {print (+)} E' | ∈

T → F T'
T' → * F {print (*)} T' | ∈

F → id {print (id.lexval)}

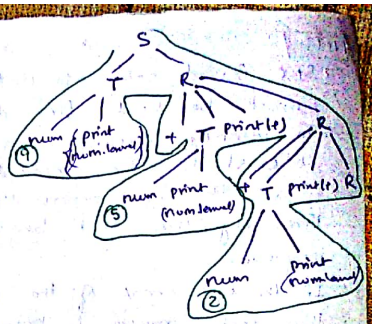
Q. Consider the following translation scheme and what output, it will produce for the input 2*3+4.

1 S → E R
R → * E {print (*)} R
R → ∈
E → F + E {print (+)}
E → F
F → (S)
F → id {print (id.lexval)}



Q. S → T R
R → + T { print (+)} R
R → ∈
T → num {print (num.lexval)}

95 + 2 +



Translation Scheme with bottom up parsing:

→ If the translation scheme is carried out alongwith bottom up parser, handles cannot be detected because of semantic rules presenting within grammar symbols.
→ Hence, translation scheme is converted as follows to carry out bottom up parsing:
→ Translation scheme middle semantic rules are replaced by new-non-terminals which produce "∈" to carry out bottom up parsing.

Q. Convert the following translation scheme suitable for bottom up parser.

E → E + T {print (+)} | T

E → T E'
E' → + T {print (+)} E' | ∈

T → F T'
T' → * F {print (*)} T' | ∈

F → id {print (num.lexval)}

E → T E'
F' → + T M E' | ∈
T → F T'
T' → * F N T' | ∈
F → id {print (id.lexval)}
M → ∈ {print (+)}
N → ∈ {print (*)}

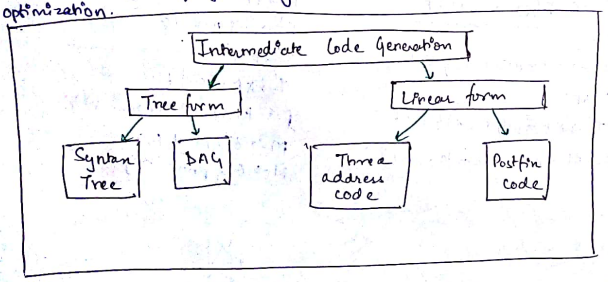
NOTE

→ SDT is used only to detect static semantic errors. Cannot be used for dynamic semantic errors checking.
→ To identify dynamic semantic errors additional program code should be inserted.
→ If all the type errors are detected by compiler at compile time only, then, that language is known as statically typed language.
→ If the type errors of the program, even detected at runtime then that language is known as dynamically typed language.
→ A language in which no type checking is done, then that language is known as untyped language.

E.g. machine language.

Intermediate Code Generation Phase

→ Intermediate code is a type of code which is easily produced from HLL and easily translated into LLL.
→ The following are different forms of intermediate code.
→ The advantage of generating intermediate code is to perform optimization.



	S.T.	DAG	TAC	Pcode
① Declaration statement	X	X	X	X
② i/p, o/p, statement	X	X	✓	X
③ Expression	✓	✓	✓	✓
④ Flow control	X	X	✓	X
⑤ Function/procedures.	X	X	✓	X

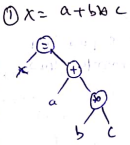
Syntax Tree:

→ Syntax tree is optimized form of the parse tree.
→ Syntax tree is constructed for expression statements.
→ Syntax tree represents order of evaluation of expressions.
→ Syntax tree corresponding to the expression contains all operands at leaf nodes and all operators are at root positions.
→ ~~Syntax tree is optimized form of the parse tree~~

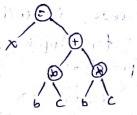
Directed Acyclic Graph (DAG):

→ DAG is a syntax tree in which common subexpressions are eliminated.
→ For expression statement, if DAG is constructed, that automatically performs an optimization known as common subexpression elimination optimization.

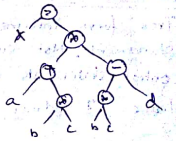
① $x = a + b * c$



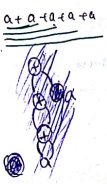
② $x = b * c + b * c$



③ $x = (a + b * c) * (b * c - d)$

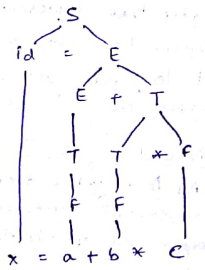


DAG: $x = a + a + a + a + a$



Q. Construct the SDT to construct syntax syntax tree from the expression statements by using following grammar production. where "mknode" is a function that creates root nodes.
Node is attribute that gives syntax tree at every place.
Write semantic rules for the given grammar productions.

$S \rightarrow id = E$ {mknode (id.name, "=", E.node)}
$E \rightarrow E + T$ {E.node = makenode (E1.node, "+", T.node)}
$E \rightarrow T$ {E.node = T.node}
$T \rightarrow T_1 * F$ {T.node = makenode (T1.node, "*", F.node)}
$T \rightarrow F$ {T.node = F.node}
$F \rightarrow id$ {F.node = id.name}



#Drawback of Tree form:

→ Tree form intermediate code is suitable only for expression statements. Not suitable for input output statements, flow control statements and procedures.
→ Hence, 3 address code format is used.
→ In 3 address code, every HLL statement can be represented by using at most 3 addresses.
→ The following are different forms of 3 address code:

① $X = Y$ op Z
② $X =$ op Y
③ $X = Y$
} Expressions

④ $X[i] = Y$
$X = Y[i]$
} Arrays

⑤ goto L } Unconditional Jump

⑥ if a > b goto L_1
goto L_2
} Conditional Jump

⑦ Procedure/Function
P($P_1, P_2, .. P_n$)

Param P_1
Param P_2
:
Param P_n
Call name, #args

Three Address Code

#Three address code for expression statements:

① $X = a + b * c$

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

② $X = (a * b) + (c * d) / (a * b)$

$t_1 = a * b$
$t_2 = c * d$
$t_3 = a * b$
$t_4 = t_2 / t_3$
$t_5 = t_1 + t_4$
$X = t_5$

③ $X = (a + b) + (e + (c - d))$

$t_1 = a + b$
$t_2 = c - d$
$t_3 = e + t_2$
$t_4 = t_1 + t_3$
$X = t_4$

Q. Construct the SDT to generate 3 address code for expression statements. Where "newtemp()" that generates ~~3 address code~~ a new temporary variable on every function call.
gen() is a function, that produces 3 address code, where place is attribute that gives 3 address code at every place.
Write semantic rules by using the following grammar productions.

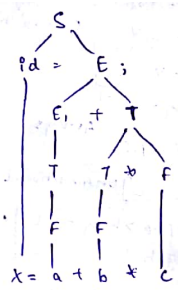
$E \to E + T$ [t = newtemp(); gen(t = E.place + T.place); E.place = t]

$E \to T$ [E.place = T.place]

$T \to T_1 * F$ [t = newtemp(); gen(t = T1.place * F.place); T.place = t]

$T \to F$ [T.place = F.place]

$F \to id$ [F.place = id.name]



3-Address code for arrays:

1-D array

$A[i] = base + (i - low) * w$

2-D array

$A[i,j] = (i * n_2 + j) * w + base = (low_1 * n_2 + low_2) * w$

n-D array

$A[i_1, i_2, \ldots i_n] = (((i_1 n_2 + i_2) n_3 + i_3) \ldots) n_k + i_k)) * w$

$+$

$base - ((((low * n_2) + low_2) n_3 + low_3) \ldots - n_k + low_k)) * w$

Q. Construct the 3address code for following expression:

$X = A[i,j]$, where size of array is $A[10][20]$,
size of each word is 4 bytes
and $low_1 = low_2 = 1$.

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

$t_1 = i * 20$
$t_2 = t_1 + j$
$t_3 = t_2 * 4$
$t_4 = base - 84$
$t_5 = t_3 + t_4$
$t_6 = A[t_5]$
$X = t_6$

Three AC for flow control statements:

If (a>b).
{a=b+c;
}
else
x=4+2;

If a>b goto L_1
goto L_2
L_1: $t_1 = b + c$
$a = t_1$
goto LAST
L_2: $t_1 = 4 + 2$
$x = t_1$
LAST: ---

Q. for (i=1; ic=10; i++)
{x = y + a * b;
}

$t = 1$;
L_1: If $ic = 10$ goto L_2
goto L_2
L_2: $t_1 = a * b$
$t_2 = y + t_1$
$x = t_2$

i=1;
L_0: if ic=10 goto L_4
else goto L_2
L_4: $t_1 = a * b$
$t_2 = y + t_1$
$x = t_2$
$t_3 = i + 1$
$i = t_3$
goto L_0
L_2: ---

Q. How many temporary variables required.

for (i=1; ic=100; i++)
{x = (4+2) * (a+b*c);
} i=1;
L_0: if ic=100 goto L_1
else goto L_2
L_1: $t_1 = 4 + 2$
$t_2 = b * c$
$t_3 = a + t_2$
$t_4 = t_1 * t_3$
$x = t_4$
$t_5 = i + 1$; $i = t_5$
goto L_0
L_2: ---

3AC for functions:

E.g. `Sum(a,b)`

```
Param a
Param b
Call sum, 2
```

`Sum(1, 2)`

```
t1 = 1
t2 = 2
Param t1
Param t2
Call sum,2
```

`Printf("compiler");`

```
t1 = "compiler"
Param t1
Call printf,1
```

```
If (a>b) (or) ((c<d) and (e>f)))
  { x = a+b;
  }
else
    P = Q+R;
```

```
L1:  If (a>b) goto L7
L2:  goto L3
L3:  if (c<d) goto L5
L4:  goto L10
L5:  if (e>f) goto L7
L6:  goto L10
L7:  t1 = a+b
L8:  x = t1
L9:  goto L12
L10: t1 = Q+R
L11:   P = t1
L12:
```

Back Patching, while generating 3address code for flow control statements, initially code is generated and target labels information filled later, known as backpatching.

Basic Block Construction:

→ Basic block is a sequence of 3 AC instructions which is having single entry and single exit property.

→ Basic blocks are constructed by using leader statements.

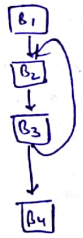
→ The following are leader statements for the given 3 address code:

1. First statement is always leader.
2. The target of conditional or unconditional goto is a leader.
3. The statement immediately followed by conditional or unconditional goto is a leader.
4. One leader statement to until next leader statement, one basic block is constructed.

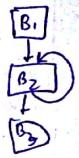
→ Control flow graph is constructed, for all basic blocks which represents the flow of execution among the statements of 3address code.

Q. Construct basic blocks for the following 3 address code statements.

```
1. f = 1
2. i = 2
3. if i<n goto 8
4. f = f * i
5. t = t + 1
6. i = t
7. goto 3
8. i = 5 + j
9. stop
```




```
1.  print
2.  i = 1
3.  t1 = 4*i
4.  t2 = a[t1]
5.  t3 = 4*i,
6.  t4 = b[t3]
7.  t5 = t2*t1
8.  t6 = prod + t5
9.  prod = t6
10. if i<20, goto (3)
11. stop.
```



→ The generated 3 address code is stored in array format with 4 fields known as quadruple.

→ In quadruple representation space is allocated for temporary variables also in the result field.

→ Hence, compiler may insert its value in the symbol table along with variables of the program.

→ To avoid this drawback triples representation is used.

→ In triples notation, expression value is stored in memory locations and accessing of it from memory locations is done whenever required.

Triple :

op	opr1	opr2
*	b	c
+	a	(10)
=	x	(14)

Quadruple

result	op	opr1	opr2
t_1	*	b	c
t_2	+	a	t_1
x	=	t_2	--

Q. Consider the following 3 address code and which of the following is static single assignment form of the given code :

p = a - b
q = p * c
p = u * v
q = p + q

Static single assignment : In this, every expression is assigned to a new variable.

(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) $p_1 = a - b$, $q_1 = p_1 * c$, $p_2 = u * v$, $q_2 = p_4 + q_2$

(d) None

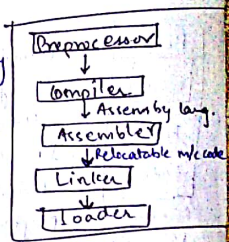
* Interpreter : Interpreter is a language processor that takes HLL program and produces output of it by performing line by line execution.

* Preprocessor : is a translator that executes some of the HLL statements present in the program.

→ The task done by preprocessor is file inclusion and macro evaluation.

* Assembler : is a program that translates assembly language into relocatable m/c code

* Relocatable machine code is not executable code. After performing linking, then, the code is placed in absolute addresses in memory.



* Linker : is a program, that makes different modules of the program into single module.

* Loader : is a program, that loads the program into main memory from secondary memory.

* Pass : One complete scan of the source program is known as one pass.

A phase can be part of the pass

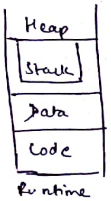
Chapter 2 · (15) Viable Prefix :

	Stack	Input	action
①	t	ab$	S
a	a	b$	R
A	A	b$	S
Ab	Ab	$	R
AB	AB	$	R
a	S	$	Accr.

→ Set of prefixes of right sentential form that appear on LHS part of the handle, like viable prefix.

→ Viable prefixes are nothing but stack content of S/R parser.

#Runtime Environment: → Static allocation Fortran
→ Dynamic allocation



Runtime Memory

→ Compiler uses 2 techniques while allocating, m/m for generated code, k/a static and dynamic allocation.

→ In static allocation strategy, compiler takes decision regarding runtime memory by looking at program text only.

→ By using static allocation strategy, recursive programs cannot be executed and so, dynamic data structures cannot be created.

→ Dynamic storage allocation: In this strategy, operating system allocates some memory to execute the program.

- Compiler divides runtime memory into code area, data area, ~~stack~~ stack area, heap area.
- Compiler allocates space for fixed code at code area.
- The space for global variables is allocated at compile time in data area.
- Whenever a function is called, activation record is pushed into the stack.

→ The space for local variables of a function are kept in activation record.

→ If the func^n returns its value, then activation record is popped out from the stack.

→ If space is allocated in stack m/m, then, memory management is completely automated.

→ If we want to maintain any value throughout the program, that is not kept in stack space. Hence, space from ~~the~~ for that value is given in heap area.

→ Memory management is completely manual in heap area.

→ Additional programs like garbage collection is required to free heap memory.

#Practice:

IoC:
- DFA to language (20 questions)
- Minimization of DFA (15)
- Language to DFA (40)
- F.A. to reg. ex. ~~(50)~~ (15)
- Reg. ex to F.A. (25)
- Lang. to reg. ex / Reg. ex to lang. (30)
- Reg. grammar to reg. ex. & reg. ex to reg grammar (20)
- Reg. lang. detection (40)
- Ambiguity of grammar (20)
- Language to PDA & PDA to lang. (25)
- CFL Detection with reg. & DCFL (30)
- T.M. to language (20)
- Recursive, R.E., not REL detection from T.M. (20)
- Undecidability Table

#compiler: (10 to 20 questions)
- No. of tokens
- Lexical error detection
- Detection of the grammar (LR(0), SLR, CLR, LALR, LL(1), operator precedence)
- SDT construction including translation
- SDT output ~~to~~ scheme
- Construction of DAG with no. of edges and nodes.
- 3 address code for expression statements with temporary variables
- Construction of basic blocks.