DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING AND INFORMATION TECHNOLOGY
YEAR/SEM: III/VI
23CS6401/23IT6401 – COMPILER DESIGN
UNIT II
SYNTAX ANALYSIS

QUESTION BANK

Introduction-Context free grammars –Writing a grammar- Top down parser-Bottom up parser- LR parsers-Constructing an SLR parsing table-LALR parser-CLR parser- Error Handling and Recovery in Syntax Analyzer - YACC

PART - A (2 MARKS)

INTRODUCTION

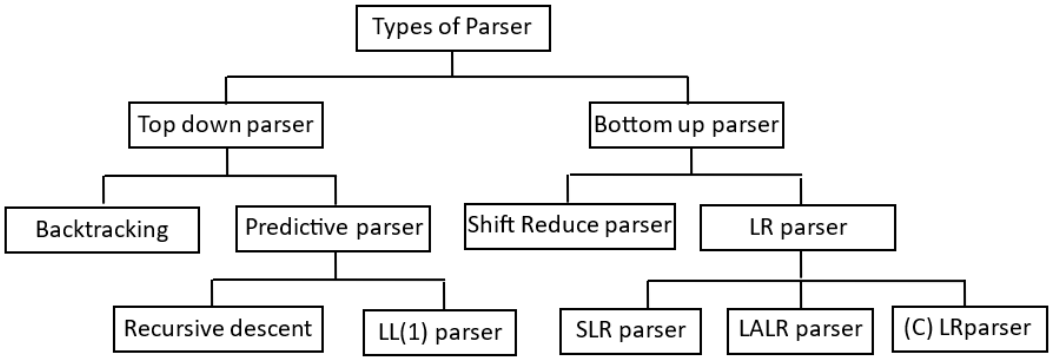
1. **Define syntax analysis in a compiler.** **K1**

The second phase of the compiler is syntax analysis or parsing. The parser (syntax analyzer) receives the source code in the form of tokens from the lexical analyzer and performs syntax analysis, which creates a tree-like intermediate representation that depicts the grammatical structure of the token stream.

2. **What is meant by a parser? List the types of parsers.** **K1**

A parser is a program that breaks data into smaller elements for easy translation into another language.

A parser takes the input in the form of a sequence of tokens or program instructions from the lexical analyzer and constructs a data structure in the form of a parse tree or an abstract syntax tree.



3. **Mention the basic issues in parsing.** **K2**

Two important issues in parsing are

● Specification of syntax and ● Representation of input after parsing.

Specification of syntax means how to write any programming statement in an exact, unambiguous and complete manner.

Representation of input after parsing is important because all the phases of compiler take information from the parse tree being generated. Information suggested by any programming statement should not be differed after building the syntax tree.

4. Distinguish between a parse tree and a syntax tree. **K2**

Parse Trees	Syntax Trees
♦ A parse tree is a graphical representation of the replacement process in a derivation.	♦ A syntax tree is nothing but the compact form of a parse tree.
♦ In parse trees, each interior node symbolizes a grammar rule, and each leaf node symbolizes a terminal.	♦ In syntax trees, each interior node symbolizes an operator & each leaf node symbolizes an operand.
♦ Parse trees provide every characteristic information from the real syntax.	♦ Syntax trees do not provide every characteristic information from the real syntax and that is the reason why they are sometimes prefixed with the term abstract. For example- no rule nodes, no parenthesis etc.
♦ For the same language construct, parse trees are comparatively less dense than syntax trees.	♦ For the same language construct, syntax trees are comparatively more denser than parse trees.

5. What is the need for a parser in compilation? **K2**

- ♦ A parser is needed to detect and report syntax errors clearly and recover from common errors so that compilation can continue.
- ♦ Converts code into a tree structure that represents the hierarchical syntactic structure of the program.
- ♦ An error is detected as soon as the prefix of the input cannot be completed to form a string in the language.

6. Why are lexical analysis and syntax analysis separated? **K2**

The lexical analyzer scans the input program and collects the token from it. On the other hand parser builds a parser tree using these tokens. These are two important activities and these activities are independently carried out by these two phases.

Separating out these two phases has two advantages:

- Firstly, it speed-up the process of compilation.
- Secondly, the errors in the source input can be identified precisely.

CONTEXT FREE GRAMMAR - WRITING A GRAMMAR

7. Define context-free grammar. **K1**

A context-free grammar *G* is defined by the 4-tuples: G= (V, T, P S) where

- ♦ V is a finite set of non-terminals (variable).
- ♦ T is a finite set of terminals.
- ♦ P is a finite set of production rules of the form A→α. Where A is a nonterminal and α is a string of terminals and/or non-terminals. P is a relation from V to (V∪T)*.
- ♦ S is the start symbol (variable S∈ V)

8. What is a sentential form and the yield of a parse tree? **K1**

Sentinels:

Given a grammar G with start symbol S, if S → α, where α may contain nonterminals or terminals, then α is called the sentinel form of G.

Yield or Frontier of the tree:

Each interior node of a parse tree is a non-terminal. The children of a node can be a terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called the yield or frontier of the tree.

9. What is meant by an ambiguous grammar? Why must an ambiguous grammar be eliminated? **K2**

- A grammar G is said to be ambiguous if it produces different parse trees for the same sentence or yield w which has been derived from the same start symbol.
- An ambiguous grammar must be eliminated because it can produce more than one parse tree for the same input string, leading to multiple interpretations of the program. This creates confusion in syntax analysis, makes semantic analysis and code generation unreliable, and can result in incorrect or inconsistent compiled output.

10. What is left recursion in a grammar? State the rule to eliminate left recursion. **K1**

- A context free grammar is said to be left recursive if it has a non-terminal A with two productions in the following form:

  $A \rightarrow A\,\alpha \mid \beta$

  where α and β are sequences of terminals and non-terminals that do not start with A.
- Left recursion in top-down parsing can enter into infinite loop. It creates serious problems, so we have avoided Left recursion.

Rule

$A \rightarrow \beta\, A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

11. Define left factoring and viable prefixes. **K1**

Left factoring

- When a production has more than one alternative with common prefixes then it is necessary to make right choice on production.
- To perform Left factoring for the production, $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

Viable prefixes.

The set of prefixes of right sentential forms that can appear on the stack of a shift- parser are called viable prefixes.

12. What is the dangling else problem? **K1**

The dangling else problem is a programming ambiguity that arises in nested conditional statements, where it is unclear to which "if" statement an "else" clause belongs.

It occurs when an "else" statement is used in conjunction with multiple nested "if" statements. This can lead to confusion about which "if" the "else" is associated with, especially when the code lacks proper indentation or braces. For example:

```
if (condition1) {
if (condition2) {
```

```
// Do something
} // else here is ambiguous
}
else {
// Do something else
}
```

In this case, it is unclear whether the "else" belongs to the first "if" (condition1) or the second "if" (condition2). This ambiguity can lead to unintended behaviour in the program, as different interpretations can yield different results.

TOP DOWN PARSER - BOTTOM UP PARSER

13. Differentiate between a top-down parser and a bottom-up parser. **K2**

Top-down parser.	Bottom up parser
Parse tree can be built from root to leaves	Parse tree can be built from leaves to root
This is simple to implement	This is complex to implement
This is less efficient parsing techniques. Various problems that occurring during top down technique are ambiguity let recursion	When the bottom up parser handles ambiguous grammar conflicts occurs parse table.
It is applicable to small class of languages.	It is applicable to broad class of languages.
Various parsing techniques are • Reduce descent parser • Predictive parser	Various parsing techniques are • Operator precedence parsing • LR parser • Shift reduce

14. What are kernel and non kernel items? **K2**

i) The set of items which include the initial item, S S, and all items whose dots are not at the left end are known as kernel items.

ii) The set of items, which have their dots at the left end, are known as non kernel items.

15. What are the difficulties with top-down parsing? **K2**

- Left recursion
- Backtracking
- The order in which alternates are tried can affect the language accepted. When failure is reported there will be very little idea where the error actually occurred.

16. Why is backtracking avoided in LL(1) parsing? **K2**

Backtracking is avoided in LL(1) parsing because the parser can uniquely select the correct production using one lookahead symbol, making parsing deterministic and efficient.

17. What is meant by a recursive-descent parser? What are the actions in shift-reduce parsing? **K1**

A parser which uses a set of recursive procedures to recognize its input with no backtracking is called a recursive-descent parser.

Actions available in shift-reduce parser

- Shift
- Reduce
- Accept
- Error

18. Define handle and handle pruning. **K1**

Handle

A handle of a string is a substring that matches the right side of a production and reduction to the non-terminal on the left side of the production represents one step the reverse of a rightmost derivation.

Handle pruning

- A rightmost derivation in reverse can be obtained by "handle-pruning".
- The process of discovering a handle & reducing it to the appropriate left-hand side is called handle pruning. i.e) start with a string of terminals w that is to parse. If w is a sentence of the grammar at hand, then $w = \gamma_n$, where γ_n is the n^{th} right sentential form of some as yet unknown rightmost derivation.

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \gamma_{n-1} \Rightarrow \gamma_n = w.$$

LR PARSERS- CONSTRUCTING AN SLR PARSING TABLE - LALR PARSER - CLR PARSER

19. Why are LR parsers more powerful than LL parsers? **K2**

- LR parsers can handle a larger class of context-free grammars.
- They can parse left-recursive grammars directly.
- LL parsers require left recursion elimination and left factoring, but LR parsers do not.
- LR parsers detect syntax errors earlier during parsing.
- They use more context (stack and states), making them suitable for complex language constructs.

20. Define an LR parser. What are the drawbacks of an LR parser? **K2**

LR parser can be used to parse a large class of context-free grammars. The technique is called LR(K) parsing.

- L - Input sequence is processed from left to right.
- R - Rightmost derivation is performed.
- K - At most K symbols of the sequence are used to make decision.

Drawbacks of an LR parser

- LR parser needs an automated parser generator as parsing tables are too complicated to be generated by hand.
- It cannot handle ambiguous grammar without special tricks.

21. Why SLR and LALR are more economical to construct than canonical LR? **K2**

For a comparison of parser size, the SLR and LALR tables for a grammar always have the same number of states, and this number is typically several hundred states for a language like Pascal.

The canonical LR table would typically have several thousand states for the same size language. Thus, it is much easier and more economical to construct SLR and LALR tables than the canonical LR tables.

22. How do operator precedence and associativity help in resolving ambiguity in expressions? **K2**

Operator precedence defines the order in which operators are evaluated, and associativity specifies how operators of the same precedence are grouped. Together, they ensure a unique parse of expressions, thereby resolving ambiguity.

23. What are the LEADING and TRAILING operations in an operator precedence grammar? **K1**

LEADING

- If production is of form $A \rightarrow \alpha a$ or $A \rightarrow Ba\alpha$ where B is Non-terminal, and α can be any string, then the first terminal symbol on R.H.S is Leading(A) = {a}
- If production is of form or $A \rightarrow B\alpha$, if a is in LEADING (B), then a will also be in LEADING (A).

TRAILING

- If production is of form $A \rightarrow \alpha a$ or $A \rightarrow \alpha aB$ where B is Non-terminal, and α can be any TRAILING (A) = {a}
- If production is of form $A \rightarrow \alpha$B. If a is in TRAILING (B), then a will be in TRAILING(A**).**

24. What is shift-reduce parsing? Identify the important rules used in shift-reduce parsing. **K1**

- Shift–reduce parsing is a bottom-up parsing technique in which the parser shifts input symbols onto a stack and reduces them to non-terminals using grammar productions until the start symbol is obtained.

The important rules used in shift–reduce parsing are:

- Shift: Move the next input symbol onto the stack.
- Reduce: Replace a handle on the stack with the corresponding non-terminal.

25. Define LALR grammar. **K1**

- LALR (Lookahead LR) parser is an extension of canonical LR parser.
- LALR parsing tables are constructed by merging the itemsets of canonical LR parser possess similar LR(0) items. Hence, the parsing table is relatively smaller than the CLR.
- If there are no action conflicts then the given grammar is said to be an LALR(1) grammar. The collection of sets of items constructed is called LALR(1) collections.

26. What is meant by operator grammar? Give an example. **K1**

An operator grammar is a context-free grammar that satisfies the following conditions:

1. No two non-terminals appear adjacent to each other on the right-hand side of any production.
2. No ε-productions are allowed.
3. The grammar mainly consists of operators and operands, and is used to define operator precedence and associativity.

Example:

E → E + T | T
T → T * F | F
F → id

ERROR HANDLING AND RECOVERY IN SYNTAX ANALYZER

27. What are the different levels of a syntax error handler? **K1**

- Lexical level such as misspelling an identifier, keyword or operator.
- Syntactic level, such as an arithmetic expression with unbalanced parentheses.
- Semantic level, such as operator applied to an incompatible operand.
- Logical level, such as an infinitely recursive call.

28. List the goals of an error handler used in syntax analysis. **K2**

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

YACC

29. What are the parts of a YACC program? **K1**

There are 3 parts available.

- Declarations
- Translation rules
- Supporting C-routines.

30. What is YACC, and what is its primary purpose in compiler design?

YACC (Yet Another Compiler Compiler) is a tool used to generate a parser from a given grammar. Its primary purpose is to analyze the syntax of a program and build a parse tree or abstract syntax tree for further processing.

PART - B (16 Marks)

1.(i)	**Consider the context-free grammar.** **S → SS + | SS* | a** **and the string aa + a*** **a) Give a leftmost derivation for the string.** **b) Give a rightmost derivation for the string.** **c) Give a parse tree for the string.** **d) Is the grammar ambiguous or unambiguous? Justify your answer. (6)** **Solution:** **a) Leftmost derivation** At each step, the leftmost nonterminal is expanded. S ⇒ SS * ⇒ SS + S * ⇒ a S + S * ⇒ a a + S * ⇒ a a + a * String derived: aa+a*	**10**	**K3**

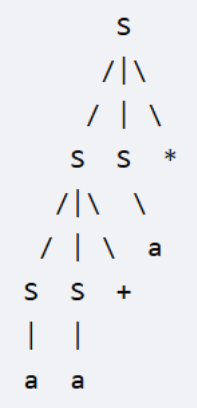
b) Rightmost derivation

At each step, the rightmost nonterminal is expanded.

S ⇒ SS *
⇒ S a *
⇒ SS + a *
⇒ S a + a *
⇒ a a + a *

String derived: aa+a*

c) Parse tree for the string



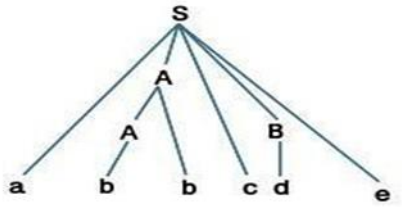
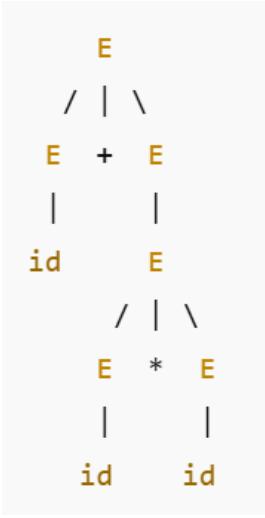
d) Ambiguity of the grammar

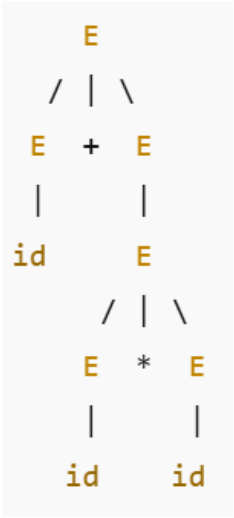
The grammar is unambiguous.

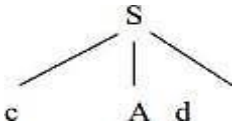
- Every valid postfix expression has exactly one parse tree under this grammar.
- Operators + and * always combine exactly two preceding subexpressions.
- For the string aa+a*, there is only one possible hierarchical grouping:
  - (a a +) a *
- No alternative leftmost/rightmost derivations or parse trees exist for the same string.
- Hence, the grammar is unambiguous.

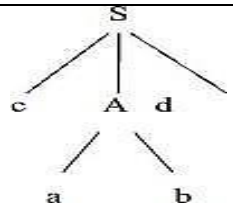
1.(ii)	**Eliminate the left recursion for the grammar:** **E→ E + T | T;** **T → T * F | F;** **F → (E) | id** **Solution:** First eliminate the left recursion for E as E → TE' E' → +TE' | ε Then eliminate for T as T → FT' T'→ *FT' | ε Thus the obtained grammar after eliminating left recursion is E → TE' E' → +TE' | ε T → FT' T' → *FT' | ε F → (E) | id **Algorithm to eliminate left recursion:** 1. Arrange the non-terminals in some order A1, A2 . . . An. 2. for i := 1 to n do begin for j := 1 to i-1 do begin replace each production of the form Ai → Aj γ by the productions Ai → δ1 γ | δ2γ | . . . | δk γ where Aj →δ1 | δ2 | . . . | δk are all the current Aj-productions; end eliminate the immediate left recursion among the Ai-productions end **After eliminating the left-recursion the grammar becomes,** E → TE′ E′ → +TE′ | ε T → FT′ T′ → *FT′ | ε F → (E) | id	**6**	**K3**

	Stack implementation: PROCEDURE / INPUT STRING E( ) — id+id*id T( ) — id+id*id F( ) — id+id*id ADVANCE( ) — id+id*id TPRIME( ) — id+id*id EPRIME( ) — id+id*id ADVANCE( ) — id+id*id T( ) — id+id*id F( ) — id+id*id ADVANCE( ) — id+id*id TPRIME( ) — id+id*id ADVANCE( ) — id+id*id F( ) — id+id*id ADVANCE( ) — id+id*id TPRIME( ) — id+id*id		
2.(i)	**Consider the following CFG:** **S → aABe** **A → Abc | b** **B → d** **Terminals: a,b,c,d** **Non-terminals: S,A,B** **Starting Symbol: S** **(a) Parse the sentence "abbcde" using right-most derivation.** **(b) Parse the sentence "abbcde" using left-most derivation.** **(c) Draw the parse tree.** **Solution:** **S → aABe** **A → Abc | b** **B → d** **(a) Parse the sentence "abbcde" using right-most derivation** S ⇒ aABe ⇒ aAde ⇒ aAbcde ⇒ abbcde **(b) Parse the sentence "abbcde" using left-most derivation** S ⇒ aABe ⇒ aAbcBe ⇒ abbcBe ⇒ abbcde	**8**	**K4**

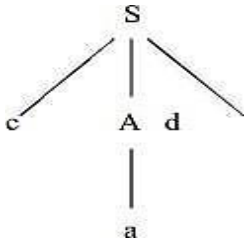
	(c) Draw the parse tree  ***Figure:*** *Parse tree for the expression abbcde*		
2.(ii)	**Analyze the ambiguity of the grammar E → E + E | E * E | (E) | −E | id with respect to the sentence "id + id * id".** **Solution:** If a grammar produces a string with more than one leftmost (or rightmost) derivation, it is ambiguous. **Leftmost Derivation 1** E ⇒ E + E ⇒ id + E ⇒ id + E ∗ E ⇒ id + id ∗ E ⇒ id + id ∗ id **Leftmost Derivation Tree for "id + id * id"**  **Leftmost Derivation 2** E ⇒ E ∗ E ⇒ E + E ∗ E	**8**	**K4**

	⇒ id + E ∗ E ⇒ id + id ∗ E ⇒ id + id ∗ id **Leftmost Derivation Tree for "id + id * id"**  Since the same string id + id * id has two different leftmost derivations, the grammar is ambiguous.		
3.(i)	**Eliminate left recursion and left factoring for the following grammar.** **E → E + T | E - T | T** **T → a | b | (E)** **Solution:** We eliminate left recursion first, then check whether left factoring is required. **Given Grammar** E → E+T | E-T | T T → a | b | (E) **Step 1: Eliminate Left Recursion** The nonterminal E has immediate left recursion of the form E → Eα | β **Productions of E:** • E → E + T • E → E − T • E → T Here, • α_1 = + T • α_2 = − T • β = T **Apply left-recursion elimination rule:** If E → Eα_1 | Eα_2 | β then transform to:	**10**	**K3**

	E → β E' E' → α_1 E' | α_2 E' | ε **Result after eliminating left recursion** E → TE' E' → +TE' | −TE' | ε T → a | b | (E) **Step 2: Check for Left Factoring** **For E:** E → T E' ✓ No common prefix **For E':** E' → + T E' | − T E' | ε ✓ No common prefix **For T:** T → a | b | (E) ✓ No common prefix ♦ Left factoring is not applied because no nonterminal has two or more productions with a common left prefix. ♦ Each alternative begins with a distinct terminal (or ε), allowing unambiguous choice in predictive parsing. **Resultant Grammar after Left Recursion Eliminated & Left Factored** E → TE' E' → +TE' | -TE' | ε T → a | b | (E)		
3.(ii)	**Construct a parse tree for the input string *w*=cad using a top-down parser;** **S→ cAd** **A→ ab | a** **Solution:** Consider the grammar G : S → cAd A → ab |a and the input string w = cad. The parse tree can be constructed using the following top-down approach: **Step 1:** Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.  **Step 2:** The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.	**6**	**K3**



Step 3:
The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'. But the third leaf of tree is b which does not match with the input symbol d.
Hence discard the chosen production and reset the pointer to second position. This is called backtracking.
Step 4: Now try the second alternative for A.



4.	**Construct the FIRST and FOLLOW sets and the predictive parse table for the grammar:** **S → AC$** **C → c | Ɛ** **A → aBCd | BQ | Ɛ** **B → bB | d** **Q → q**	**16**	**K3**

Solution:
1. Finding the first () sets:
First (Q) = {q} First (B) = {b, d}
First (C) = {c, ε}
First (A) = First (aBCd) ∪ First (BQ) ∪ First (ε)
= {a} ∪ First (B) ∪ First (d) ∪{ε}
= {a} ∪ First (bB) ∪ First (d) ∪ {ε}
= {a} ∪ {b} ∪ {d} ∪ {ε}
= {a, b, d, ε}
First(S) = First (AC$) = (First (A) – {ε}) ∪ (First (C) – {ε}) ∪ First (ε)
= ({a, b, d, ε} – {ε}) ∪({c, ε} – {ε}) ∪ {ε}
= {a, b, d, c, ε}

2. Finding Follow () sets:
Follow (S) = {#}

Follow (A) = (First (C) – {ε}) ∪ First ($) = ({c, ε} – {ε}) ∪ {$}
Follow (A) = {c, $}
Follow (B) = (First (C) – {ε}) ∪ First (d) ∪ First (Q)
= {c} ∪ {d} ∪ {q}
= {c, d, q}
Follow (C) = (First ($) ∪ First (d) = {d, $}
Follow (Q) = (First (A) = {c, $}

3. The parsing table for this grammar is:

	a	b	c	d	q	$
S	$S \rightarrow AC\$$	$S \rightarrow AC\$$	$S \rightarrow AC\$$	$S \rightarrow AC\$$		$S \rightarrow AC\$$
A	$A \rightarrow aBCd$	$A \rightarrow BQ$	$A \rightarrow \epsilon$	$A \rightarrow BQ$		$A \rightarrow \epsilon$
C			$C \rightarrow c$			$C \rightarrow \epsilon$
B		$B \rightarrow bB$		$B \rightarrow d$		
Q					$Q \rightarrow q$	

4. Moves made by predictive parser on the input abdcdc$ is

Stack symbol	Input	Remarks
#S	abdcdc$#	S → AC$
#$CA	abdcdc$#	A → aBCd
#$CdCBa	abdcdc$#	Pop a
#$CdCB	bdcdc$#	B → bB
#$CdCBb	bdcdc$#	Pop b
#$CdCB	dcdc$#	B → d
#$CdCd	dcdc$#	Pop d
#$CdC	cdc$#	C → c
#$Cdc	cdc$#	Pop C
#$Cd	dc$#	Pop d
#$C	c$#	C → c
#$c	c$#	Pop c
#$	$#	Pop $
#	#	Accepted

5.(i)	**Construct LR (0) items for the following grammar, G :** **S → S+R | R** **R → R*T | T** **T → (S) |** i	**8**	**K3**

Solution:

Given Grammar (G)

S → S + R | R

R → R * T | T

T → (S) | i

Step 1: Augmented Grammar

Introduce a new start symbol (S'):

S' → S

S → S + R | R

R → R * T | T

T → (S) | i

Step 2: LR(0) Items and Canonical Collection

I_0 = CLOSURE({ S' → ·S })

S' → ·S

S → ·S + R

S → ·R

R → ·R * T

R → ·T

T → ·(S)

T → ·i

GOTO(I_0, S) = I_1

S' → S·

S → S· + R

GOTO(I_0, R) = I_2

S → R·

R → R· * T

GOTO(I_0, T) = I_3

R → T·

GOTO(I_0, '(') = I_4

T → (·S)

S → ·S + R

S → ·R

R → ·R * T

R → ·T

T → ·(S)

T → ·i

GOTO(I_0, i) = I_5

	T → i· **GOTO(I_1, '+') = I_6** S → S + ·R R → ·R * T R → ·T T → ·(S) T → ·i **GOTO(I_2, '*') = I_7** R → R * ·T T → ·(S) T → ·i **GOTO(I_4, S) = I_8** T → (S·) S → S· + R **GOTO(I_6, R) = I_9** S → S + R· R → R· * T **GOTO(I_7, T) = I_{10}** R → R * T· **GOTO(I_8, ')') = I_{11}** T → (S)· **Step 3: Canonical Collection of LR(0) Items** { I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}, I_{11} }		
5.(ii)	**Find the FIRST and FOLLOW sets for the grammar:** **E → TE′** **E → +TE′ | ε** **T → FT′** **T′ → *FT′ | ε** **F → (E) | id** **Solution:** **Computation of FIRST** **E →TE′** Applying Rule (4b) of FIRST Since FIRST (T) does not contain ε, or T does not derive ε. ∴ FIRST (E) = FIRST(TE′) = FIRST(T) ∴ FIRST (E) = {FIRST(T)} (1)	**8**	**K3**

E → +TE′|ε

Applying Rule (3) of FIRST

Comparing E′ → +TE′with X → aα

∴ FIRST(E′) = {+}

Apply Rule (2) on E′ → ε

FIRST (E′) = {ε}

∴ FIRST(E′) = {+, ε} (2)

T→FT′

Apply rule (4b) of FIRST

Since, FIRST(F) does not derive ε

∴ FIRST(T) = FIRST(FT′) = FIRST(F)

∴ FIRST(T) = {FIRST(F)} (3)

T′→*FT′|ε

Comparing with rule (2) & (3) of FIRST, we get

∴ FIRST(T′) = {ε,∗} (4)

F→(E)|id

Comparing with rule (3) of FIRST

∴ FIRST(F) = {(, id} (5)

Combining statement (1), (2), (3),(4),(5)

FIRST (E) = {FIRST(T)}

FIRST(E′) = {+, ε}

FIRST(T) = {FIRST(F)}

FIRST(T′) = {ε,*}

FIRST(F) = {(, id}

∴ FIRST (E) = FIRST(T) = FIRST(F) = {(, id}

FIRST(E′) = {+, ε}

FIRST(T′) = {ε,*}

Computation of FOLLOW

E → TE′

E → +TE′|ε

T → FT′

T′ →∗ FT′|ε

F → (E)|id

Applying Rule (1) FOLLOW
∴ FOLLOW (E) = {$} (1)

E → TE′
Applying Rule (2) FOLLOW

E →	ε	T	E′
A →	α	B	β

∴ A = E, α = ε, B = T, β = E′
Since FIRST (β) = FIRST (E′) contains ε.
∴ Rule (2b) of FOLLOW
FOLLOW (T) = FIRST (E′) − {ε} ∪ FOLLOW (E)
FOLLOW (T) = {+, ε} − {ε} ∪ FOLLOW (E)
∴ FOLLOW (T) = {+} ∪ FOLLOW (E) (2)

Applying Rule (3) of FOLLOW

E →	T	E′
A →	α	β

A = E, α = T, B = E′
∴ FOLLOW (E′) = {FOLLOW (E)} (3)

E′ → +TE′
Applying Rule (2)

E →	T	E′
A →	B	β

A = E, α = +, B = T, β = E′
Since FIRST(β) = FIRST(E′)contains ε.
∴ Rule (2b) of FOLLOW
∴ FOLLOW (T) = FIRST(E′) − {ε} ∪ FOLLOW(E′)
∴ FOLLOW (T) = {+, ε} − {ε} ∪ FOLLOW(E′)
∴ FOLLOW (T) = {+} ∪ FOLLOW(E′) (4)

Applying Rule (3)

E →	+T	E′
A →	α	B

∴ FOLLOW(E′) = {FOLLOW(E′)} (5)

T′ →FT′

Applying Rule (2)

T →	ε	F	T′
A →	α	B	β

Since FIRST(β)derives ε.

∴ Rule (2b) of FOLLOW

∴ FOLLOW (F) = FIRST(T′) − {ε} ∪ FOLLOW(T)

∴ FOLLOW (F) = {∗, ε} − {ε} ∪ FOLLOW(T)

∴ FOLLOW (F) = {∗} ∪ FOLLOW(T) (6)

Applying Rule (3)

T →	F	T′
A →	α	B

∴ FOLLOW(T′) = {FOLLOW(T)} (7)

T →*FT′|ε

Applying Rule (2) FOLLOW

T′ →	*	F	T′
A →	α	B	β

FIRST(β) = FIRST(T′) contains ε or T′ derives ε.

∴ Rule (2b)

∴ FOLLOW (F) = FIRST(T′) − {ε} ∪ FOLLOW(T′)

∴ FOLLOW (F) = {*, ε} − {ε} ∪ FOLLOW(T′)

∴ FOLLOW (F) = {*} ∪ FOLLOW(T′) (8)

Applying Rule (3) of FOLLOW

T′ →	*F	T′
A →	α	B

∴ FOLLOW(T′) = {FOLLOW(T′)} (9)

F→(E)

Applying Rule (2)

F →	(	E	)
A →	α	B	β

FIRST(β) or FIRST( )) = {)} do not contain ε.

∴ Rule (2a)

∴ FOLLOW (E) = FIRST( ))

∴ FOLLOW (E) = {)} (10)

Rule (3) does not apply to this production.

As A → α B has B nonterminal at the right corner of R. H. S of production. But F → (E)has) the terminal on its right corner.

Rule (2) and Rule (3) does not apply to the remaining productions, as they don't match with rules.

Combining Production (1) to (10)

FOLLOW (E) = {$} (1)

FOLLOW (T) = {+} ∪ FOLLOW (E) (2)

FOLLOW (E′) = {FOLLOW (E)} (3)

FOLLOW (T) = {+} ∪ FOLLOW(E′) (4)

FOLLOW(E′) = {FOLLOW(E′)} (5)

FOLLOW (F) = {*} ∪ FOLLOW(T) (6)

FOLLOW(T′) = {FOLLOW(T)} (7)

FOLLOW (F) = {*} ∪ FOLLOW(T′) (8)

FOLLOW(T′) = {FOLLOW(T′)} (9)

FOLLOW (E) = {)} (10)

From (1), (3) and (10)

FOLLOW (E) = FOLLOW(E′) = {$,)} (11)

∴ From rule 4, 7, and 11

∴ FOLLOW (T) = {+} ∪ FOLLOW(E′)

∴ FOLLOW (T) = {+} ∪ {$,)}

∴ FOLLOW (T) = {+,), $}

FOLLOW (T′) = { FOLLOW (T)} = {+,), $} (12)

∴ From statement 6, 8 and 12

FOLLOW (F) = {*} ∪ FOLLOW(T)

	FOLLOW (F) = {*} ∪ {+,), $, } FOLLOW (F) = (*, +,), $} (13) ∴ Statements 11, 12 and 13 give the required answer FOLLOW (E) = FOLLOW(E′) = {), $} FOLLOW (T) = FOLLOW(T′) = {+,), $} FOLLOW (F) = {+,*,), $}		
6.(i)	**Show that the following grammar is LL(1):** **S → AaAb | BbBa** **A → Ɛ** **B → Ɛ** **Solution:** **Step 1:** **Compute FIRST():** FIRST(S) = { FIRST(A) – ∈ } ∪ FIRST(a) ∪ { FIRST(B) – ∈ } ∪ FIRST(b) = {a,b} FIRST(A) = { ∈ } FIRST(B) = { ∈ } **Step 2:** **Compute FOLLOW():** FOLLOW(S) = { $ } FOLLOW(A) = FIRST(a) ∪ FIRST(b) = { a , b } FOLLOW(B) = FIRST(b) ∪ FIRST(a) = { a , b } **Step 3:** **LL(1) Conditions Check** **For nonterminal S** • FIRST(AaAb) = { a } • FIRST(BbBa) = { b } • {a} ∩ {b} = ∅ → No conflict **For nonterminal A** • Only one production: A → ε • Check: FIRST(A) ∩ FOLLOW(A) = {ε} ∩ {a, b} = ∅ → No conflict **For nonterminal B** • Only one production: B → ε • Check: FIRST(B) ∩ FOLLOW(B) = {ε} ∩ {a, b} = ∅ → No conflict	**8**	**K4**

Step 4: Predictive Parsing Table

Non terminal	a	b
S	S → AaAb	S → BbBa
A	A → ε	A → ε
B	B → ε	B → ε

- Each cell has **at most one production**
- No multiple entries in any cell

The grammar satisfies all LL(1) conditions:

- Disjoint FIRST sets for alternative productions
- ε-productions handled safely using FOLLOW sets
- No parsing table conflicts

The given grammar is LL(1).

6.(ii) **Analyze the given grammar and determine the FIRST and FOLLOW sets for all non-terminals, and justify the impact of nullable non-terminals on predictive parsing decisions.** 8 K4

A → BC | EFGH | H
B → b
C → c | ε
E → e | ε
F → CE
G → g
H → h | ε

Solution:

1. Finding first () set:
   1. first (H) = first (h) ∪ first (ε) = {h, ε}
   2. first (G) = first (g) = {g}
   3. first (C) = first (c) ∪ first (ε) = c, ε}
   4. first (E) = first (e) ∪ first (ε) = {e, ε}
   5. first (F) = first (CE) = (first (c) - {ε}) ∪ first (E)
      = (c, ε} {ε}) ∪ {e, ε} = {c, e, ε}
   6. first (B) = first (b)={b}
   7. first (A) = first (BC) ∪ first (EFGH) ∪ first (H)
      = first (B) ∪ (first (E) – { ε}) ∪ first (FGH) ∪ {h, ε}
      = {b, h, ε} ∪ {e} ∪ (first (F) – {ε}) ∪ first (GH)
      = {b, e, h, ε} ∪ {C, e} ∪ first (G)
      = {b, c, e, h, ε} ∪ {g} = {b, c, e, g, h, ε}

2. Finding follow() sets:

1. follow(A) = {$}
2. follow(B) = first(C) – {ε} ∪ follow(A) = {C, $}
3. follow(G) = first(H) – {ε} ∪ follow(A)
   = {h, ε} – {ε} ∪ {$} = {h, $}
4. follow(H) = follow(A) = {$}
5. follow(F) = first(GH) – {ε} = {g}
6. follow(E) = first(FGH) m- {ε} ∪ follow(F)
   = ((first(F) – {ε}) ∪ first(GH)) – {ε} ∪ follow(F)
   = {c, e} ∪ {g} ∪ {g} = {c, e, g}
7. follow(C) = follow(A) ∪ first (E) – {ε} ∪ follow (F)
   = {$} ∪ {e, ε} ∪ {g} = {e, g, $}

7.(i) **Construct a predictive parsing table for the grammar:** **8** **K4**

S → a | ↑ | (T)
T→ T, S | S
Show how the string "(a, a)" is parsed using the predictive parser.

Solution:
Eliminate left recursion:

S → a | ↑ | (T)
T → ST′
T′ → ,ST′ | ε

Compute FIRST and FOLLOW

FIRST(S) = { a, ↑, (}
FIRST(T) = {a, ↑, (}
FIRST(T′) = {, , ε}
FOLLOW(S) = { , ,), $}
FOLLOW(T) = FOLLOW(T′) = {)}

Table 2.4: Predictive Parsing table for the above grammar

Non terminals	Input symbol					
	a	↑	(	)	,	$
S	S→a	S→↑	S→(T)			
T	T→ST′	T→ST′	T→ST′			
T′				T′→∈	T′→,ST′	

Table 2.5:Parse the string (a,a) using Predictive Parsing table

State	Input	Actions
$S	(a, a)$	S→(T)
$)T(	(a, a)$	
$)T	a, a)$	T→ST′
$)T′S	a, a)$	S→a
$)T′a	a, a)$	
$)T′	,a)$	T′→ST′
$)T′S,	a)$	
$)T′S	a)$	S→a
$)T′a	a)$	
$)T′	)$	T′→ε
$)	)$	
$	$	ACCEPT

7.(ii) **Analyze the following grammars, find the problems that prevent recursive-descent parsing, modify them if needed, and then write the recursive-descent parser procedures.** **8** **K4**

a) S→+SS | −SS | a

b) S→S(S)S | ε

Solution:

a) S → +SS | −SS | a

In this grammar, there is No left recursion, No common prefix and suitable for recursive-descent parsing

Recursive-Descent Parser

```
S():
   if lookahead == '+':
      match('+')
      S( )
      S( )
   else if lookahead == '-':
      match('-')
      S( )
      S( )
   else if lookahead == 'a':
      match('a')
   else:
      error
```

(b) S → S(S)S | ε

In this grammar, it has left recursion (S starts with S), and it causes infinite recursion in RD parser

Step 1: Remove Left Recursion

After removing left recursion, the grammar becomes,

	S → S'
	S' → (S)S S' | ε
	Recursive-Descent Parser
	S():
	S_prime()
	S_prime():
	if lookahead == '(':
	match('(')
	S()
	match(')')
	S()
	S_prime()
	else:
	return // ε		

8.	**Analyze the non-recursive implementation of a predictive parser for the given grammar by examining the role of the parsing stack, input buffer, and parsing table.** **E → E + E | T** **T → T * F | F** **F → (E) | id** **Solution:** Given grammar E → E + E E → T T → T * F T → F F → (E) F → id **Remove the left recursion:** Suppose if the given grammar is left Recursive then convert the given grammar (and ∈) into non-left recursive grammar (as it goes to infinite loop). E → TE' E' → +TE' | ε T → FT' T' → *FT' | ε F → (E) | id **First( ):** FIRST(E) = { (, id } FIRST(E′) = { + , ∈ } FIRST(T) = { (, id } FIRST(T′) = { * , ∈ } FIRST(F) = { (, id } **Follow( ):**	**16**	**K4**

FOLLOW(E) = { $,) }
FOLLOW(E′) = { $,) }
FOLLOW(T) = { + , $,) }
FOLLOW(T′) = { + , $,) }
FOLLOW(F) = { + , * , $) }

Table: Predictive Parsing Table for the above grammar

NON – TERMINAL	id	+	*	(	)	$
E	E→ TE'			E→ TE'		
E′		E→ TE'			E′ → ∈	E′ → ε
T	T → FT′			T → FT′		
T′		T′ → ε	T → *FT′		T′ → ∈	T′ → ε
F	F → id			F → ε		

Table: Stack implementation using above predictive parsing table

Stack	Input	Output
$E	id+id*id $	
$E′T	id+id*id $	E → TE′
$E′T′F	id+id*id $	T → FT′
$E′T′id	id+id*id $	F → id
$E′T′	+id*id $	
$E′	+id*id $	T′ → ε
$E′T+	+id*id $	E′ → +TE′
$E′T	id*id $	
$E′T′F	id*id $	T → FT′
$E′T′id	id*id $	F → id
$E′T′	*id $	
$E′T′F*	*id $	T′→ *FT′
$E′T′F	id $	
$E′T′id	id $	F → id
$E′T′	$	
$E′	$	T′ →ε
$	$	E′ → ε

9.(i)	**Write the LR parsing algorithm and check whether the grammar is SLR(1). Justify your answer.** **S → L=R | R** **L → *R | id** **R → L** **Solution:** Canonical collection of sets of LR (0) items for the above grammar is	**8**	**K4**

I0:
S→ •S
S→ •L=R
S→•R
L→•*R
L→•id
R→•L

I1:
S•→S.

I2:
S→L•=R
R→L•

I3:
S→R•

I4:
L→*•R
R→•L
L→•*R
L→•id

I5:
L→id•

I6:
S→L=•R
R→•L
L→•*R
L→•id

I7:
L→*R•

I8:
R→L•

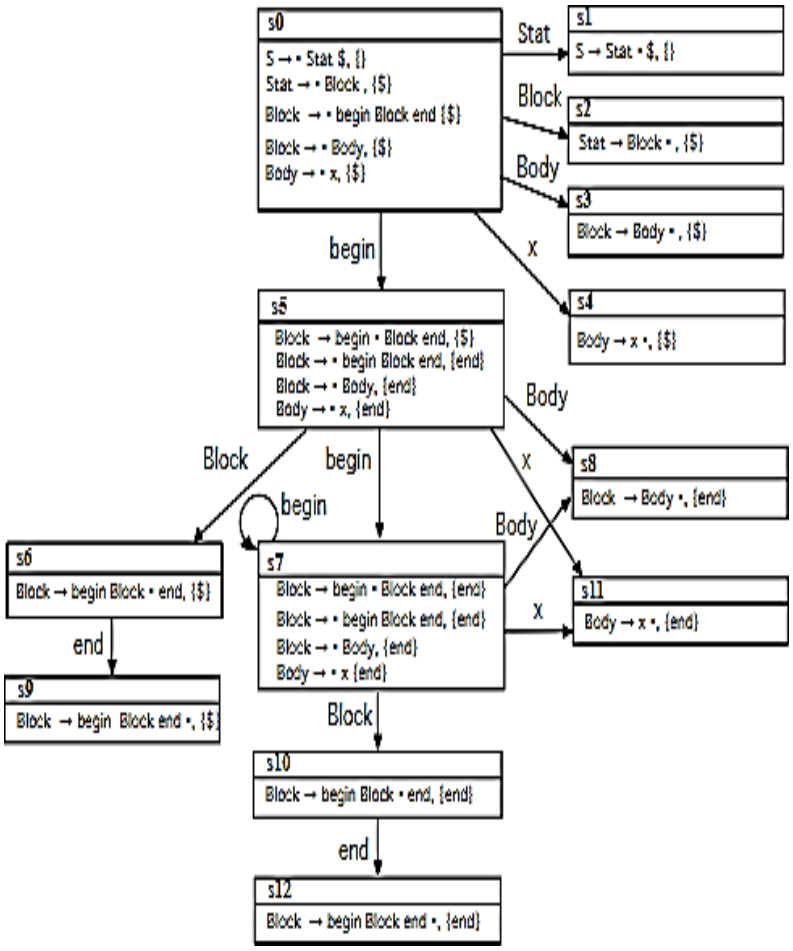
I9:
S→L=R•

Before we can build the parsing table, we need to compute the FOLLOW sets:
FOLLOW(S') = {$}
FOLLOW(S) = {$}
FOLLOW(L) = {$,=}
FOLLOW(R) = {$,=}

Table: Canonical collection of sets of LR (0) items

	ACTION				GOTO		
	id	=	*	$	S	L	R
I0	s3			s5	1	2	4
I1				Acc			
I2		s6/r[R→L]					

		I3		r(L→id)		r(L→id)			
		I4				r(S→R)			
		I5	s3		s5			7	8
		I6	s3		s5			7	9
		I7		r(R→L)		r(R→L)			
		I8		r(L→*R)		r(L→*R)			
		I9				r(S→L=R)			
	Consider the set of Items I2. The first item in this set makes action [2, =] be shift 6. Since Follow(R) has =, the second item sets action [2, =] to reduce R->L. Thus the entry action [2, =] is multiply defined. So it is not SLR (1).								
9.(ii)	**Consider the Context-Free Grammar (CFG) depicted below, where "begin", "end" and "x" are all terminal symbols of the grammar and Stat is considered the starting symbol for this grammar. Productions are numbered in parentheses, and you can abbreviate "begin" to "b" and "end" to "e" respectively.** **(1) Stat → Block** **(2) Block → begin Block end** **(3) Block → Body** **(4) Body → x** **(a) Compute the set of LR(1) items for this grammar and draw the corresponding DFA. Do not forget to augment the grammar with the initial production S → Stat $ as the production (0).** **(b) Construct the corresponding LR parsing table.** **Solution :** **(a) The set of LR(1) items and the corresponding DFA are shown below:**							**8**	**K4**



b) The LR(1) parsing table Is depicted below.

State	Action				goto		
	begin	end	x	$	Stat	Block	Body
0	s5		s4		g1	g2	g4
1				accept			
2				r(1)			
3				r(3)			
4				r(4)			
5	s7		s11			g6	g8
6		s9					
7		s12				g10	g8
8		r(3)					
9				r(2)			
10		s12					
11		r(4)					
12		r(2)					

10.	**Given the grammar** **S → AS | b** **A → SA | a** **Construct an SLR parsing table for the string "baab."** **Solution:** **Given grammar:** S → AS S → b A → SA A →a **Augmented grammar:** S′ → S S → AS S → b A → SA A→a **I0:** S′ → •S S → •AS S →•b A→•SA A → •a **I1:** goto(I0, S) S′→S• A→S•A A→•SA A→•a S→•AS S→•b **I2:** goto(I0, A) S→A•S S→•AS S→•b A→•SA A→•a **I3:** goto(I0, b) S→b• **I4:** goto(I0, a) A→a• **I5:**	**16**	**K3**

goto(I1, A)
A→SA•
S→A•S
S→•AS
S→•b
A→•SA
A→•a

I6:

goto(I1, S)
A→S•A
A→•SA
A→•a
S→•AS
S→•b
goto(I1, a) = I4
goto(I1, b) = I3

I7:

goto(I2, S)
S→AS•
A→S•A
A→•SA
A→•a
S→•AS
S→•b
goto(I2, A) = I2
goto(I2, b) = I3
goto(I2, a) = I4
goto(I5, A) = I2
goto(I5, S) = I7
goto(I5, a) = I4
goto(I5,b) = I3
goto(I6, A) = I5
goto(I6, S) = I6
goto(I6, a) = I4
goto(I6,b) = I3
goto(I7, A) = I5
goto(I7, S) = I6
goto(I7, a) = I4
goto(I7,b) = I3
First(S)={b, a}
First(A)={a, b}
Follow(S)={$,a,b}
Follow(A)={a,b}

Table: SLR parsing table

States	Action			Goto	
	a	b	$	S	A
0	S4	S3		1	2
1	S4	S3	acc	6	5
2	S4	S3		7	2
3	r2	r2	r2		
4	r4	r4			
5	r3 s4	r3 s3		7	2
6	S4	S3		6	5
7	S4 r1	S3 r1	r1	6	5

Parsing the string baab:

0	baab$	shift 3
0b3	aab$	reduce by S->b
0S1	aab$	shift 4
0S1a4	ab$	reduce by A->a
0S1A5	ab$	reduce by A->SA
0A2	ab$	shift 4
0A2a4	b$	reduce by A->a
0A2A2	b$	shift 3
0A2A2b3	$	reduce by S->b
0A2A2S7	$	reduce by S->AS
0A2S7	$	reduce by S->AS
0S1	$	accept

11. **Construct an SLR parsing table for the grammar:** — **16** — **K3**

E → E + T | T
T → TF | F
F → F* | a | b

Solution:

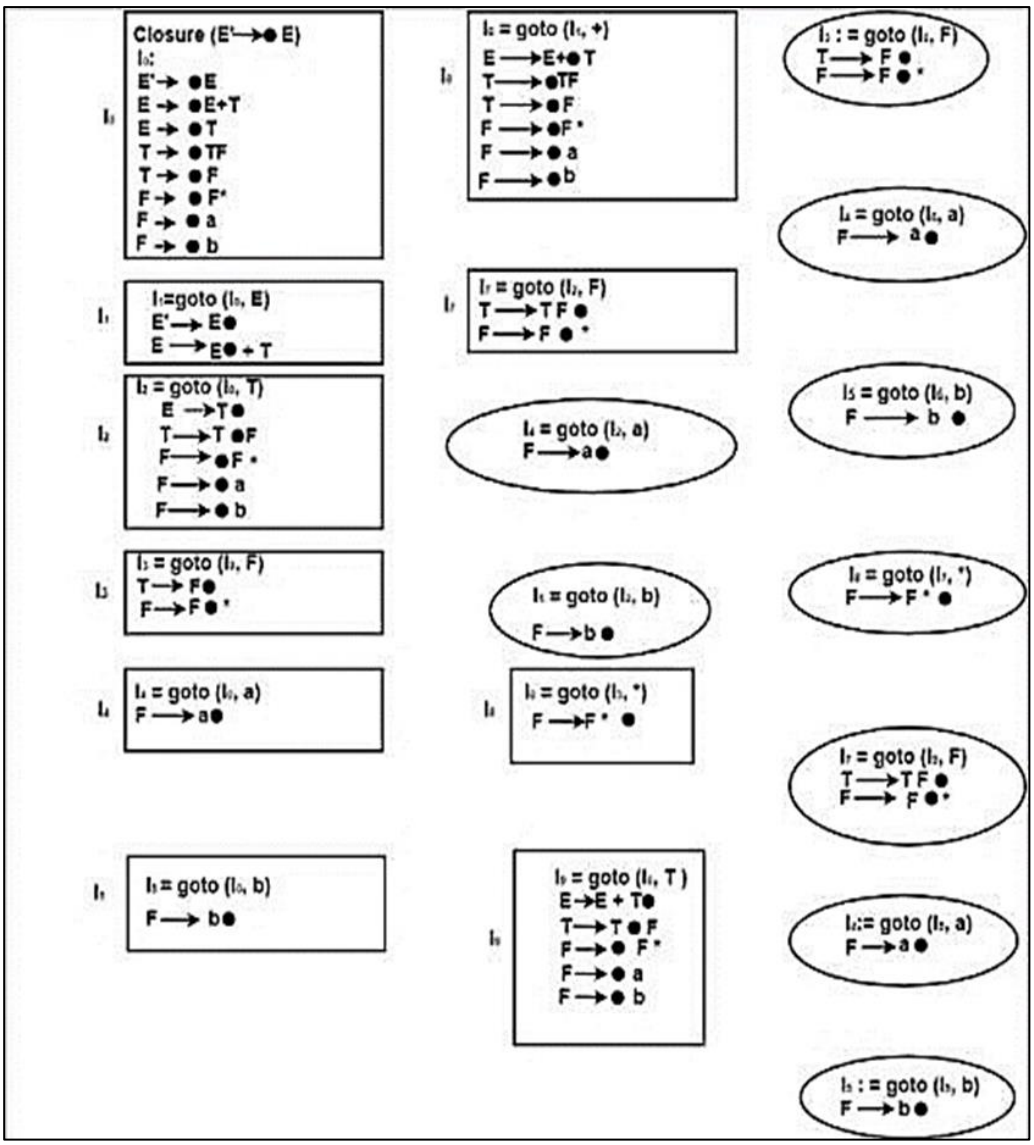
Step 1:

Construct the augmented grammar and number the productions.

(0) E′ → E
(1) E → E + T
(2) E → T
(3) T → TF
(4) T → F
(5) F → F ∗
(6) F → a
(7) F → b

Step 2:
Find closure & goto Functions to construct LR (0) items.
Box represents the New states, and the circle represents the Repeating State.

Computation of FOLLOW
We can find out
FOLLOW(E) = {+, $}
FOLLOW(T) = {+, a, b, $}
FOLLOW(F) = {+,*, a, b, $}



LR Parsing Table

State	Action					goto		
	+	*	a	b	$	E	T	F
0			s4	s5		1	2	3
1	s6				accept			
2	r2		s4	s5	r2			7
3	r4	s8	r4	r4	r4			
4	r6	r6	r6	r6	r6			
5	r6	r6	r6	r6	r6			
6			s4	s5			9	3
7	r3	s8	r3	r3	r3			
8	r5	r5	r5	r5	r5			
9	r1		s4	s5	r1			7

Parsing for Input String a * b + a

Stack	Input String	Action
0	a * b + a $	Shift
0 a 4	* b + a $	Reduce by F → a.
0 F 3	* b + a $	Shift
0 F 3 * 8	b + a $	Reduce by F → F *
0 F 3	b + a $	Reduce by T → F
0 T 2	b + a $	Shift
0 T 2 b 5	+a $	Reduce by F → b
0 T 2 F 7	+a $	Reduce by T → TF
0 T 2	+a $	Reduce by E → T
0 E 1	+a $	Shift
0 E 1 + 6	a$	Shift
0 E 1 + 6 a 4	$	Reduce by F → a
0 E 1 + 6 F 3	$	Reduce by T → F
0 E 1 + 6 T 9	$	Reduce by E → E + T
0 E 1	$	Accept

12.	**Consider the grammar:** **S → L=R** **S → R** **L → *R** **L → id**	**16**	**K4**

	R → L **Analyze the LALR parsing method for the above grammar. List the canonical collection of items and construct the parse table.** **Solution:** **Given grammar:** S → L=R S → R L → *R L → id R → L **Augmented grammar:** S• → S S → L=R S → R L → *R L → id R → L **Canonical collection of LR(1) items** **I0:** S• → •S, $ S → •L=R, $ S → •R, $ L → •*R, = L → •id, = R → •L, $ **I1:** goto(I0, S) S → S•, $ **I2:** goto(I0, L) S → L•=R, $ R → L•, $ **I3**: goto(I0, R) S → R•, $ **I4**: goto(I0, *) L → *•R, = R → •L, = L → •*R, = L → •id, = **I5:** goto(I0, id) L → id•, = **I6**:		

goto(I2, =)
S → L=•R, $
R → •L, $
L → •*R, $
L → •id, $

I7:

goto(I4, R)
L → *R•, =

I8:

goto(I4, L)
R → L•, =
goto(I4, *)=I4
goto(I4, id)=I5

I9:

goto(I6, R)
S → L=R•, $

I10:

goto(I6, L)
R → L•, $

I11:

goto(I6, *)
L → *•R, $
R → •L, $
L → •*R, $
L → •id, $

I12:

goto(I6, id)
L → id•, $

I13:

goto(I11, R)
L → *R•, $
goto(I11, L)=I10
goto(I11, *)=I11
goto (I11, id)=I12

Table: LR (1) table construction

States	action				goto		
	=	*	id	$	S	L	R
0		s4	s5		1	2	3
1				Acc			
2	s6			r5			
3				r2			
4		s4	s5			8	7
5	r4						
6		s11	s12			10	9
7	r3						
8	r5						
9				r1			
10				r5			
11		s11	s12			10	13
12				r4			
13				r3			

This grammar is LR(1), since it does not produce any multi-defined entry in its parsing table.

LALR table construction:

I4 and I11 are similar. Combine them as I411 or I4:

L → *•R, =/$
R → •L, =/$
L → •*R, =/$
L → •id, =/$

I5 and I12 are similar. Combine them as I512 or I5:

L → id•, =/$

I7 and I13 are similar. Combine them as I713 or I7:

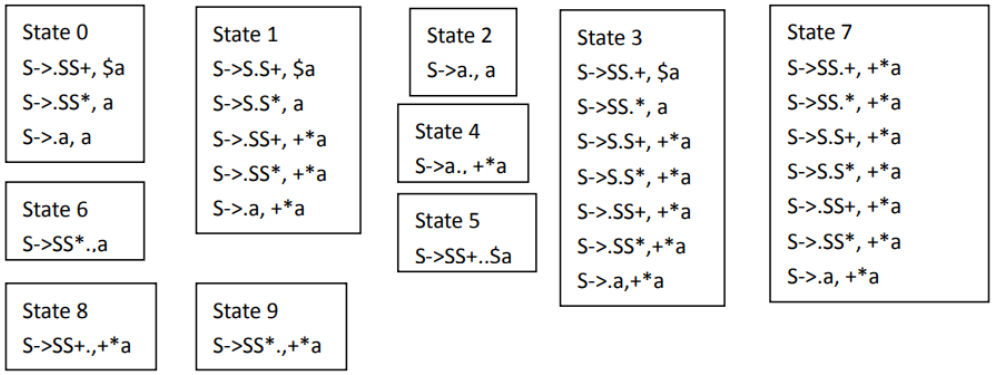
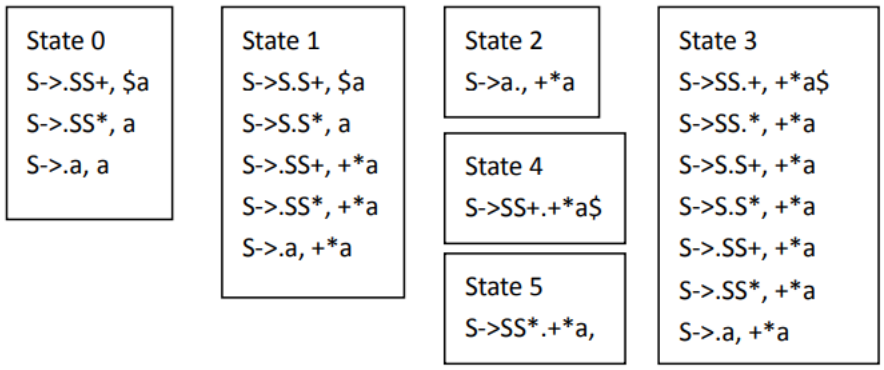
L → *R•, =/$

I8 and I10 are similar. Combine them as I810 or I8:

R → L•, =/$

Table: LALR table construction

States	**action**				**goto**		
	=	*	**id**	**$**	**S**	**L**	**R**
0		s4	s5		1	2	3
1				Acc			
2	s6			r5			
3				r2			
4		s4	s5			8	7
5	r4			r4			
6		s4	s5			8	9
7	r3			r3			
8	r5			r5			
9				r1			

13.	**Construct the** **a) canonical LR, and** **b) LALR sets of items for the grammar S → SS+ | SS* | a** **a) Canonical LR**  **b) LALR sets of items** 	**16**	**K4**
14.	**Construct a CLR parsing table for the grammar:** **S → AA** **A → Aa | b** **Solution:** **Step 1:** Grammar Augmentation S′ → •S … Rule 0 S → •AA … Rule 1 A→ •aA ... Rule 2 A → •b … Rule 3 **Step 2:** Closure operation = I0 S′ → •S S → AA	**16**	**K3**

A → •aA
A→ •b
Goto (I0, S) = I1
S′→ S• //**
Goto(I0, A) = I2
S → A•A
A → •aA
A → •b
Goto(I0, a) = I3
A →a•A
A → •aA
Goto(I0,b) = I4
A →b• //**
Goto(I2, A) = I5
S→AA•
Goto (I2, a) = I3
Goto (I2, b) = I4
Goto (I3, A) = I6
A → aA•
Goto (I3, a) = I3
Goto (I3, b) = I

Rules for construction of parsing table from Canonical collections

Action part: For Terminal Symbols

- If A → α•aβ is state Ix in Items and goto(Ix,a)=Iy then set action [Ix,a]=Sy (represented as shift to state Iy]
- If A→ α•is in Ix, then set action[Ix,f] to reduce A -> α for all symbols "f" where "f" is in Follow(A) (Use rule number)
- If S′ → S• is in Ix then set action[Ix,$] = accept

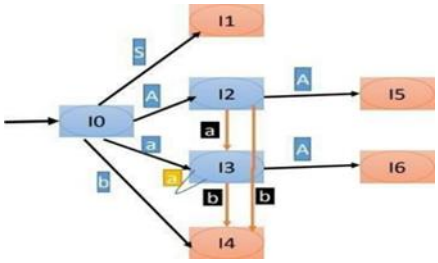
Go To Part: For Non Terminal Symbols

- If goto(Ix, A) = Iy, then goto(Ix,A) in table = Y
- It is numeric value of state Y.
- All other entries are considered as error.
- Initial state is S′→•S

PARSING TABLE

	a	b	$	S	A
	ACTION			GOTO	
I0	S3	S4		1	2
I1			Accept		
I2	S3	S4			5
I3	S3	S4			6
I4	r3	r3	r3		
I5			r1		
I6	r2	r2	r2		

DFA



15.(ii) Differentiate SLR, LALR, and CLR parsers. — 6 — K2

SLR parser	LALR parser	Canonical LR
SLR parser is the smallest in size	SLR and LALR have the same size	The canonical LR parser is the largest in size
It is the easiest method based on the FOLLOW function	This method is applicable to a wider class than SLR	This method is more powerful than SLR and LALR.
This method exposes fewer syntactic features than the LR parser	Most of the syntactic features of a language are expressed in LALR	This method exposes fewer syntactic features than the LR parser.
Error detection is not immediate in SLR	Error detection is not immediate in SLR	Immediate error detection is done by the LR parser
It requires less time and space complexity	The time and space complexity is higher in LALR, but efficient methods exist for constructing an LALR parser directly.	The time and space complexity is more for canonical LR parser.