V th semester

Automata Theory & Compiler Design

8/10/24

Q) 1. Explain frontend / backend of compiler
2. Explain all steps of a compiler
3. Translate a statement to assembly code

UNIT - I (and II, III)

* **Language Processors**
  1. Compiler – generates obj code, whole code debugged at and interpreter
  2. Interpreter – line by line / no obj code generated
  3. Assembler
  4. Preprocessors
  5. Linker / Loader

Python, Java → have both compiler and interpreter
C, C++, C# → only compiler

* **Compiler**

usually, both compiler and interpreter together are prefered

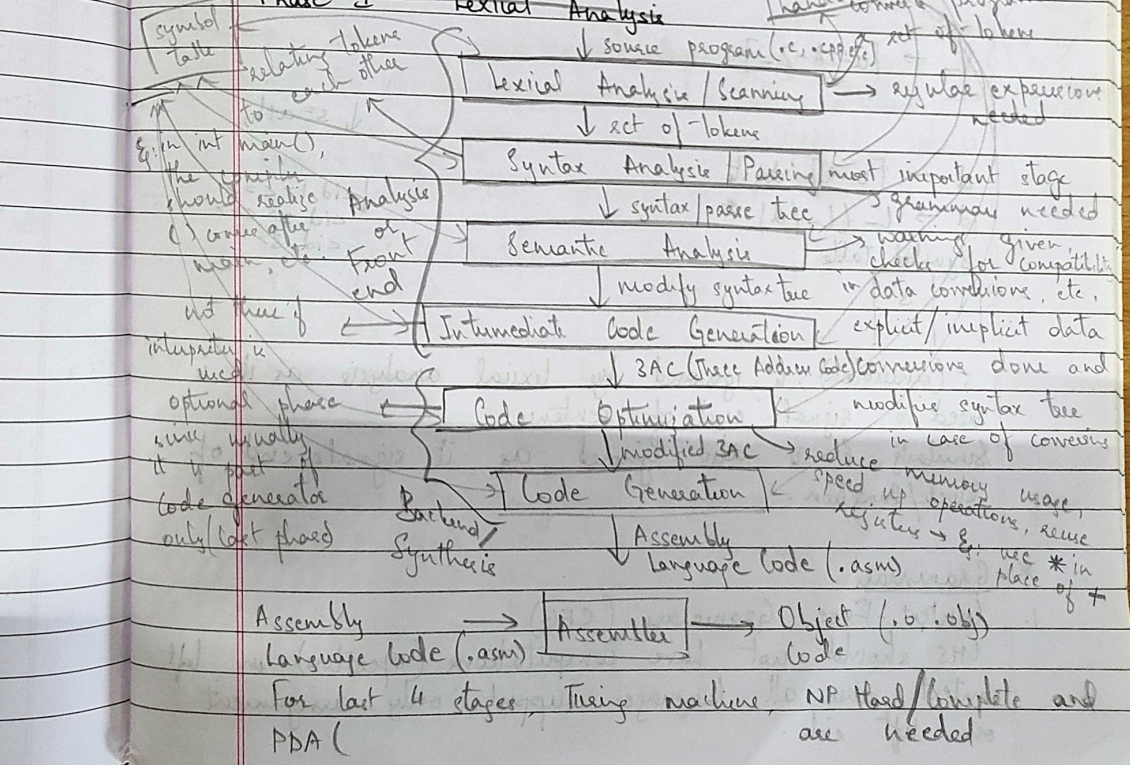
Works in five phases

token – smallest, indivisible part of code / programming construct

C tokens – keywords, variables / identifiers, constants, operators

Phases are:

→ Phase I – Lexical Analysis



Assembly language code (.asm) → Assembler → Object (.o, .obj) code

For last 4 stages, Turing machine, NP Hard / Complete and PDA (are needed

stored as <token-type, attribute>
Eg: <id, 1>

→ Symbol table – stores details of all tokens used. Each token with its details/attributes is stored here.

→ Error handler – stores specific types of errors and displays them with line number and description at time of compilation.

Both symbol table and error handler are attached to all phases of compilation.

Eg: s = a+b;
lexical analysis – checks if s, a, b, +, =, etc are all valid tokens, using regular expressions
a-zA-Z hyphen(-) is a lexical character to show continuous values
L_ → [a-zA-Z_] → regular expression
↳ indicate that L_ is a variable if it follows the regular expression
d → [0-9] digits
id → L_ (L_/d) *

scan input left to right, one character at a time

↓ s=a+b;
Lexical Analysis
↓ <id,1> <=> <id,2> <+> <id,3>

Symbol Table

1	s

; (semicolon) is ignored by lexical analysis as it is used to signal end of sentence.
Similarly, { is also ignored as it signals end of block/function.

* Grammars
1. Context-Free Grammars (CFG)
LHS should not have computations (operators) on left context since all languages support only assignment on left side.

Syntax analysis output
↳ Syntax tree — id1 = + id2 id3

Semantic Analysis Output — id1 = + (int to float a) (int to b)

Intermediate Code Generation
↳ Three Address Code (3AC)
→ If we have s = a+b*c+d, it is split into multiple divisions — each with one computation operator and it in registers. t1, t2, t3 → Temporary registers
s = a+b*c+d
t2 = a+t1 t3 = t2+d then finally s = t3
→ This format is called 3AC — max 3 addresses allowed per computational statement (?)

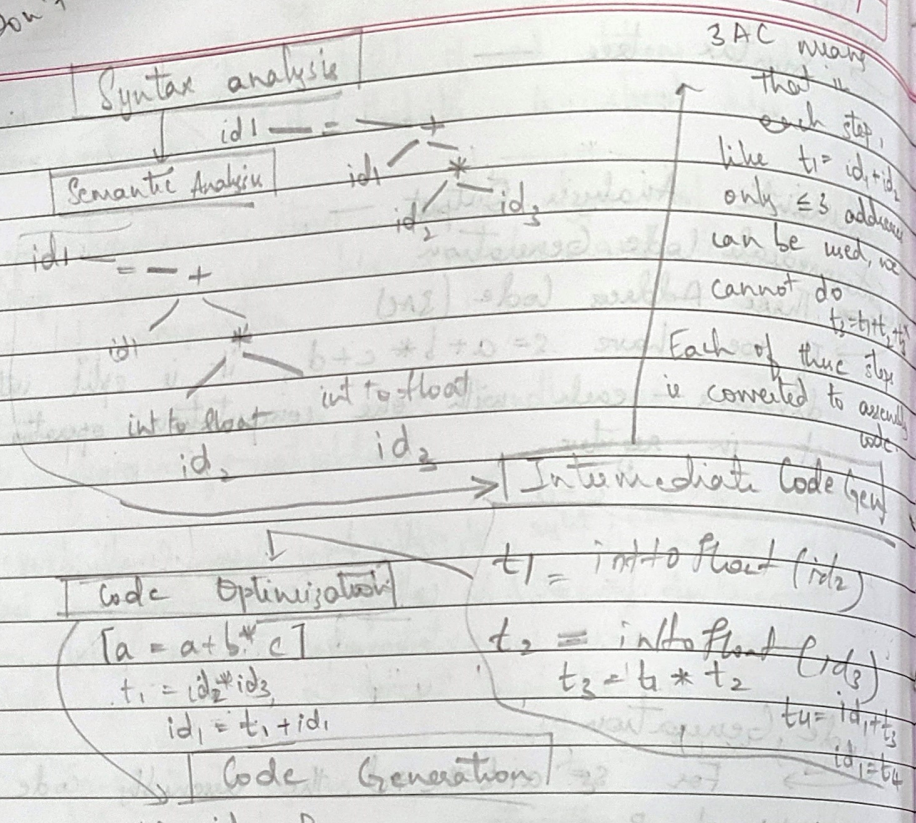
Code Generation
↳ For s = a+b*c+d, the assembly code is
LD b R1 → register
LD c R2
MUL R1,R1,R2 → store answer in R1

For id1 = id2 + id3
LD id2, R1
LD id3, R2
ADD ~~id id~~ R1, R1, R2
~~ST~~ ~~ST id1 R1~~ ST R1 id1 ← store

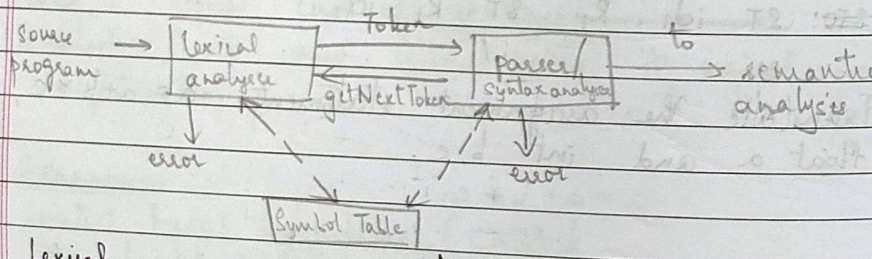
Q) Translate the assignment statement: a = a+b*c, float a and int b, c.

a = a+b*c
↓
Lexical Analysis
<id,1> <=> <id,1> <+> <id,2> <*> <id,3>

Don't use variable names anywhere after lexical analysis



Role of Lexical Analyzer / Scanner



Lexical errors are part of syntax errors
One other important function of lexical

Synchronizing tokens — ; } — end markers

tokenization

from (grouping characters into tokens) is to keep tra
number of newline characters seen — using synchro
tokens.

Pattern — regular expression (RE) generated / description
Lexeme — characters in high level language (HLL) mat
Token — name of token description of patte
RE → if → 'i' 'f'

(if) → 'if'
token

For regular difinition —
eve

→ Variable / Identifier Rule (Regular Expression / Pattern)
letter ← l_ → [a-zA-Z_]
digit ← d → [0-9]
id → l_(l/d)* → 0 or more occurrence of preceding pattern
digit grouping

Q) Write down token, lexeme and pattern for ~~each~~ fragment:

```
if(i ==j)
z=0;
else
z=1;
```

Ans:

Token	Lexeme	
if	char i, f	if →
(	char (	(→
<id,1>	char i	id → l
==	char ==	== →
<id,2>	char j	id → l
)	char)	) →
<id,3>	char z	id →
=	char =	= →
0	0	
else	char e,l,s,e	else →
1	1	

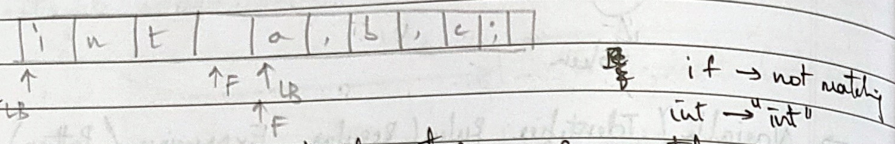
; , \n, \t are ignored
nothing to be written twice
Once registered in symbol table, should not be added again

can write const 0 but should be defined

arthop → arithmetic operator
logop → logical operator
} in symbol table → token names → so ignored

* **Input Buffering**
Lexical analyser takes help of two pointers:
1. Lexeme Begin (LB)
2. Forward

Both pointing to first memory location
Only Forward pointer moves to match ~~lexeme~~ token with regular expression



Tokenization

From LB to F, all characters till eof matches, so sent as one token → continues to parser, which identifies it as a token.
Then, LB and F move to next location
F retract if match is not found. If ! whitespace, F retracts one position if next token does not match.
F pointer is used for checking if any of the token match the regular expressions.

→ eof — end of file / ~~end of buffer~~
used to specify

→ Sentinel used for end of buffer specification because eof used to specify only end of input and now, memory space is also saved.

* **Lex Regular Expressions**

Expression	Matches	Example
\c	character c, literally	*
"s"	string s, literally	"**"
.	any character but newline	a.*b
^	beginning of a line	^a
$	end of a line	a$
[^s]	any one character not in string s	[^a]

r*	zero or more string	
r?	zero or one	
r{m,n}	between m and n occurrence of r	a{1,2}
a1	concatenation	
r1|r2	r1 or an r2	a|b
(r)	same as r (grouping)	(a|b)
r1/r2	r1 identified only when followed by r2 / r1 when followed by r2	(i/am)
R+	one or more strings	

no operator

0 or more occurences of a

Eg: [^(aeiou)] — accepts all consonants, not vowels
epsilon (null string)
$a^* = \{\epsilon, a, aa, aaa, aaaa, ..\}$
$a^+ = \{a, aa, aaa, aaaa, ...\}$
atleast one occurrence of necessary
$(+|-)? = \{+5, 5, -5\}$

16/10/24 * **Specification of Tokens**
1. Regular expressions — notation for specifying lexeme pa
2. Alphabet → finite set of symbols — denoted by Σ
3. {0,1} → binary alphabet — denoted by Σ = {0,1}
4. Language — countable set of strings over some fixed alphabet Eg: If Σ = {0,1}, language = {0, 1, 00, 01, 10, 11, 100

→ **Operations on Languages**
| — union (|) → least precedence
|| — concatenation → 2nd highest precedence
||| — unary operation (closure) < kleene (*) → highest precedence → closure; positive (+) → closure
symbols to denote
0* → can be {ε, 0, 00..
Eg: If 0+ → one or more o of 0

* **Terminology of Strings (s)**
1. prefix (s) — obtained by removing zero or more char from end of s — can be s, ε (null) Eg: hello → prefix: s, hell, h
2. suffix (s) — obtained by removing zero or more characters (chars) from beginning of s
3. substring (s) — obtained by deleting any prefix

( ) → grouping of regular expression
ε → null string

any suffix from s.

4. proper prefix (s) / proper suffix (s) — prefix/suffix that are not s or ε (epsilon) → null string
(for hello, proper prefix/proper suffix cannot be hello, ε (null))

(for hello, subsequence can be hl, lo, ...)

5. subsequence (s) – any string formed by deleting zero or more not necessarily consecutive positions of s.

a|b – can take a or b, not both

RE

Eg: 1) (a|b) (a|b) = { aa, ab, ba, bb } ← possible set of strings that can be generated by given RE

2) a(a|b)* a = {aa, aaa, aba, abba, abaa }
↳ equivalent to (a* b*)*
↳ any string starting and ending with 'a'

Q) Give description of the language accepted by RE in 2)
For 2), it will accept all strings starting and ending with a and give the set.

* **Regular definitions**
Give description of all characters in a regular expression
Eg: l → [a-zA-Z_]
d → [0-9]
id → l (l | d)*
} this is regular definition for identifier
↳ this is regular expression for identifier

Q) Write a regular expression to accept 2022 batch CS (C4) students' USN.
~~1MS22CY*~~ (give always not name)
USN → 1MS2 "(2|3)" C4 "(0|4) (d)(d)
(leads to → or, → or)
d → [0-9]
Can also write as: USN → 1MS2 "(2|3)" C4 "(a)
d → [0-9]
a → (0-4) (d)(d)

token name denoted by <token name, optional attribute>
for < ⇒ <relop, <>

* For string of length n,
1. number of prefix / suffix = n+1
2. number of proper prefix / proper suffix = n-1
3. number of substrings =
4. number of subsequences =

19/10/24 * **Recognition of Tokens**
(Transition diagram) — HLL → to → tokens (high level language)
→ called Deterministic
1. start state / initial state → denoted by circle as start → ⓪ number
2. final state / accepting state → denoted by double circle ◎ → Token

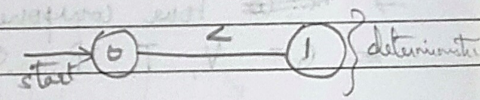
Regular expression for relational operator
relop → (< | > | <= | >= | == | <>) definition of token
(relop – name of token; <> → not equal to)

a+b according to lex → one or more occurrences of a followed by b

All the states denoted by node as ○

→ Transition diagram is deterministic because there is only one deterministic path for any character (<, >, <=) from one state.

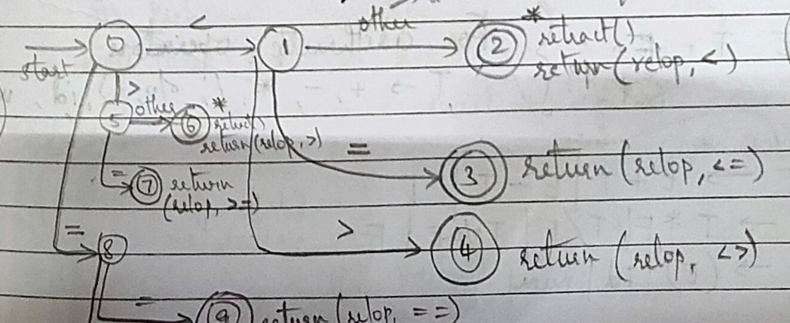
Eg: Transition diagram —



Longest matching prefix / pattern is token (is either identifier/digit)

draw arrows always

for a<=b



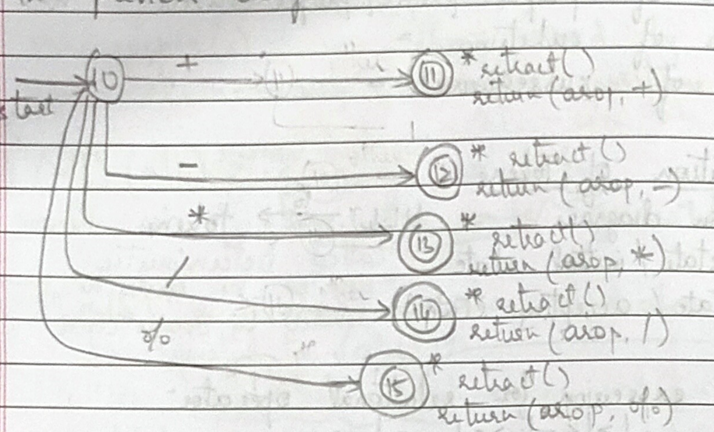
example Transition diagram

If asked to draw for both relop and arop, draw relop first and where it stops, continue with next for arop.

Page No.
Date. / /

Eg: Draw transition diagram for arop → + | - | * | / | %

→ continue from 10 as start state since we end at 9 in previous diagram



* Context Free Grammar (CFG)

Lexical analyzer does not check for errors. It passes tokens generated to syntax analyzer, which checks if pattern is followed. Like for relational operators pattern is token relop token
id/num id/num
id → identifier/variable

→ ~~....~~ Four components of CFG:

G = V, T, P, S → start symbol

V → set of non-terminal; T → set of Terminals/tokens; P → set of production/grammar for particular programming construct

E (expression) T (term) F (factor) } For arithmetic operators, T → +, -, *, /, %, (), id, num [highest priority]

start symbol, non-terminal → derives

E → E + T | E - T | T
T → T * F | T / F | T % F | F
F → (E) | id | num (highest precedence operator)

} CFG for arithmetic expressions

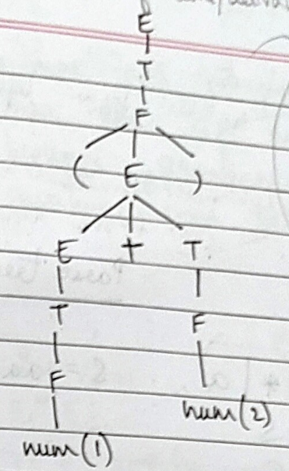
For CFG, there is always LHS derives RHS → definitions/derivations

LHS → non-terminal

All upper-case characters are non-terminals

Page No.
Date. / /

For (1+2),



which tree? → (eg: parse tree)

21/10/24 Q) Write ~~CFG~~ CFG to accept strings of 0's and 1's where equal number of 0's are followed by equal number of 1's.

Eg: {01, 0011, 000111...}. Also identify the component of CFG G = (V, T, P, S).

S → 0S1 | 01

verify by checking if given example strings can be generated using the CFG

Components ~~are~~ of G = (V, T, P, S) are:

V = S

T = 0, 1

P = (CFG) → S → 0S1 | 01 (write same grammar)

S = S

this → S → 0S1 | 01

Q) for string 0011$ → end of string

i) Leftmost derivation
ii) Rightmost derivation
iii) Parse / Derivation Tree

} Do all the

Parsing → Top-Down Approach → Left Most Derivation
Parsing → Bottom-Up Approach → Rightmost Derivation
→ LALR - gcc, (Python) use this → use automatically written parser
→ Java → use handwritten parser

Universal Approach

left most (leftmost definition to be expanded)

For 0011 → definition of S to get 0011

$S \xrightarrow{lm} 0S1$

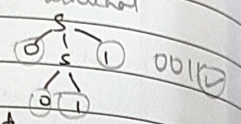
$\xrightarrow{lm} 0011$

→ rightmost (rightmost definition to be expanded first) → start from right non-terminal

$S \xrightarrow{rm} 0S1$

$\xrightarrow{rm} 0011$

Parse tree →



not ambiguous

Q) $(S \to SS+ \mid SS* \mid a)$ $S = aa+a*\$$

i) LMD for input S

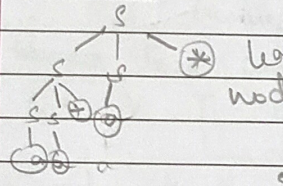
ii) RMD for ~~inp~~ S

iii) Parse tree for S

can't use SS+ first because last character that should be matched is *

LMD	RMD
$S \xrightarrow{lm} SS*$	$S \xrightarrow{rm} SS*$
$\xrightarrow{lm} SS+S*$	$\xrightarrow{rm} Sa*$
$\xrightarrow{lm} aS+S*$	$\xrightarrow{rm} SS+a*$
$\xrightarrow{lm} aa+S*$	$\xrightarrow{rm} Sa+a*$
$\xrightarrow{lm} aa+a*$	$\xrightarrow{rm} aa+a*$

Parse tree:



leaf nodes will give expression

Ambiguous Grammar

this grammar accept all arithmetic operator expressions except for % (modulus)

Q) $(E \to E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid id \mid num)$

input : id + id * id $

i) Find left most derivation (LMD)

LMD

$E \xrightarrow{lm} E*E$

$E \xrightarrow{lm} E+E*E$

$E \xrightarrow{lm} id+E*E$

$E \xrightarrow{lm} id+id*E$

$E \xrightarrow{lm} id+id*id$

If any grammar can get input string accepted by using more than one left most ~~definition~~ derivation (LMD) is called Ambiguous grammar.

For previous question, even below LMD is accepting input string :

LMD

$E \xrightarrow{lm} E+E$

$E \xrightarrow{lm} ~~E+E*E~~ id+E$

$E \xrightarrow{lm} id+E*E$

$E \xrightarrow{lm} id+id*E$

$E \xrightarrow{lm} id+id*id$

~~For the S → SS+ | SS* | a grammar,~~

~~another LMD~~

$S \xrightarrow{lm}$

Q) Given $E \to E+T \mid E-T \mid T$, $T \to T*F \mid T/F \mid T\%F \mid F$

$F \to (E) \mid id \mid num$, is this an ambiguous grammar if input = id + id * id $.

One LMD

$E \xrightarrow{lm} E+T$

$\xrightarrow{lm} T+T$

$\xrightarrow{lm} F+T$

$\xrightarrow{lm} id+T*F$

$\xrightarrow{lm} id+F*F$

$\xrightarrow{lm} id+id*F$

$\xrightarrow{lm} id+id*id$

This grammar is not ambiguous as there is not other LMD

→ Goal is to remove ambiguity from grammars before giving it to the compiler.

→ Dangling else problem — in Turbo C — solved using grammar

Q) (Stmt → if E then stmt | if E then stmt else stmt | other ; statement

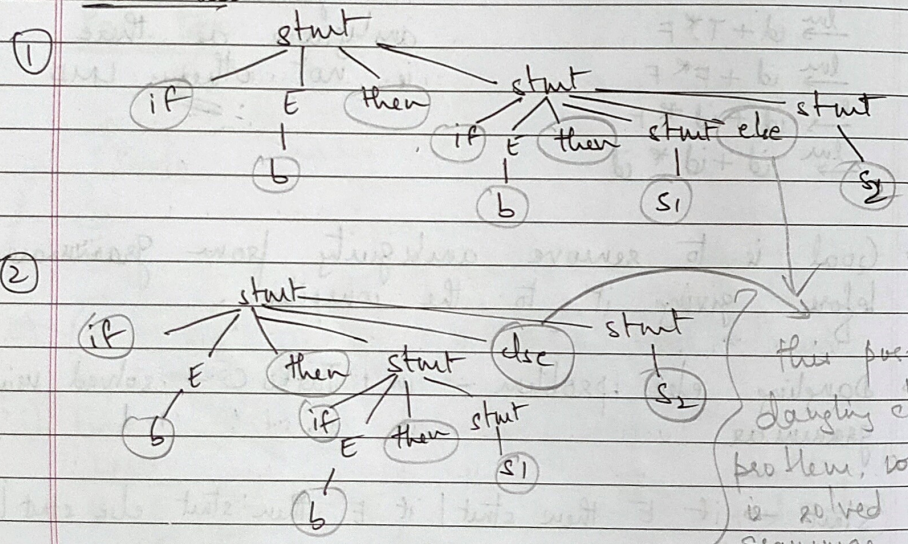
E → b) ; grammar

input: if b then if b then S_1 else S_2.
Check whether the above grammar is ambiguous and draw its parse tree.

LMD
stmt $\xrightarrow{lm}$ if E then stmt else stmt
$\xrightarrow{lm}$ if b then ~~stmt if E then~~ stmt else stmt
$\xrightarrow{lm}$ if b then if b then stmt else stmt
$\xrightarrow{lm}$ if b then if b then ~~other~~ S_1 else ~~if stmt~~
$\xrightarrow{lm}$ if b then if b then ~~other~~ S_1 else ~~other~~ S_2

LMD
stmt $\xrightarrow{lm}$ if E then stmt
$\xrightarrow{lm}$ if b then stmt
$\xrightarrow{lm}$ if b then if b E then stmt else stmt
$\xrightarrow{lm}$ if b then if b then stmt else stmt
$\xrightarrow{lm}$ if b then if b then ~~other~~ S_1 else ~~other~~ S_2
$\xrightarrow{lm}$ if b then if b then S_1 else S_2
∴ This is an ambiguous grammar

Parse tree



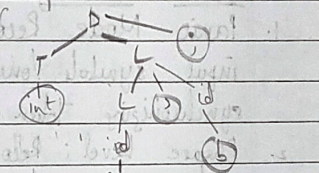
this poses dangling else problem, which is solved using grammar on next page

→ Solution to dangling else problem using grammar:
stmt → matched_stmt | open_stmt
matched_stmt → if E then matched_stmt else matched_stmt | other
open_stmt → if E then matched_stmt else open_stmt | if E then stmt
E → b

12/09/24 Q) 1) Write CFG for accepting a simple type declarations in C.
Eg: int a, b, c;
2) Write regular expression using character class to accept 5 vowels in order (lowercase).
3) Write regular expression to accept SELECT statement in SQL. SQL keyword SELECT can be written as SELECT, Select, select, sElect, sElect, etc.
4) Consider the CFG given: S → aSbS | bSaS | ε. Draw parse tree to accept the string "aabbab"

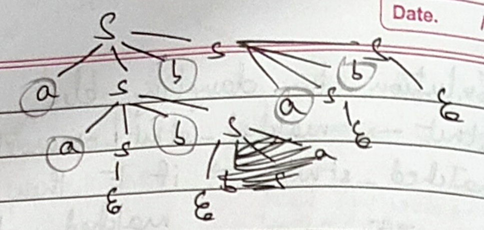
Ans
1) ~~S →~~ D → T L ';'
T → "int" | "char" | "float" | "double" | "boolean"
L → L,'id' | id

Example parse tree



2) VOWELS → (cons)* a* (cons)* e* (cons)* i* (cons)* o* (cons)* u* (cons)*
bracket not required here
this takes empty string too, give + if ε shouldn't be taken
cons → [^(aeiou)] or cons → [b-df-hj-np-tv-z]
or VOWELS → (cons|a)*(cons|e)*(cons|i)* (cons|o)* (cons|u)*
3) SELECT → (S|s)(E|e)(L|l)(E|e)(C|c)(T|t) accept baba type of strings
or SELECT → [Ss][Ee][Ll][Ee][Cc][Tt] → for char class.
S|s|E|e|L|l|E|e|C|c|T|t
no need of '|' for or
[aA] gives either A or a

4) Parse tree:



Tell "string is not matched" if parse tree not possible

* Questions
→ Unit II
Regular Expression
Lexical Analysis process
Theory
→ Unit III
CFG
All types of parsing

23/10/24 * Syntax Error Handling

(parser does)
1. Lexical errors - misspelling of identifiers, keywords, operators, etc.
2. Syntactic errors - misplaced semicolon, extra or missing braces.

(next phase)
3. Semantic errors - type mismatches between operators and operands, return statement with type "void", etc.
4. Logical errors - incorrect reasoning. (not part of error handling here)

* Error Recovery Strategy

1. Panic Mode Recovery - on discovering an error, parser discards input symbols one at a time until one of a designated set of (synchronizing) token is found. (only this is used by compilers today)
2. Phrase Level Recovery - parser performs local correction such as replace, delete extra character, insert missing character.
3. Error production - G grammar production for possible common error routine (in → in research)
4. Global Correction - general deterministic algorithm for error correction (→ in simulation)

(3, 4 not implemented by any compiler)

LL - should be scanned from left-side only, and left-most derivation
LL(0) -
LL(1) - how many lookahead characters it is supposed to take / how many further characters it sees before parsing

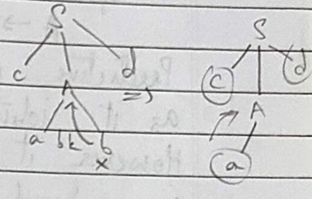
* Parsing
- Top Down
  - Predictive parsing → take only LL(1) CFGs
  - Recursive Descent parsing (gcc) → takes CFG of all forms
- Bottom Up (take only LR CFGs)
  - SLR-
  - CLR-
  - LALR (python)

Look Ahead

* Recursive Descent Parser

G: { S → cAd
A → ab|a }

input: cad



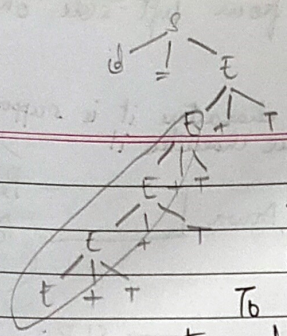
Backtracking used here.
When there is mismatch, it goes back to previously expanded non-terminal (A here) and check if A has any more definitions then input is matched
Only g++ / gcc are using this
This is handwritten, not machine-generated
Deterministic back-tracking used here.
Any CFG in any form will be accepted here since

* Common Problems in Predictive Parser (CFG in LL(1) form, no ambiguity in CFG)
LL(1) is the only input accepted by predictive parser, so LL has to be converted to LL(1).
Main problem of predictive parser is that it follows LMD

Eg: For grammar
E → E+T | E-T | T
T → T*F | T/F | T%F | F
F → (E) | id | num

The input s = a+b+c+d+e÷f+g+h-x It is a very difficult to find LMD for this.

LR - L stands for input to be scanned from right side and R means right most derivation



This gives a left-most drawing tree which is difficult for parser to comprehend.

To solve this, we ~~use~~ ~~to~~ have to eliminate left recursion. Left-most non-terminal is same as LHS

If left-most common factors are same, we have to perform left-factoring, and make it into LL(1) using deterministic algorithm

→ Eg: S → cAd

A → ab | abd

Predictive parsing is most efficient parsing method as it identifies all types of errors.

However it is time-consuming, compiler efficiency is reduced, and it is complex.

12 marks

Q) Write CFG, check if it has left recursion, eliminate it, and do left-factoring if needed.

→ To eliminate left Recursion → terminals, non-Terminals

α, β, γ → can be any string (including null (ε)) in grammar

~~If CFG:~~

if CFG is in form (A → Aα | β) ← single non-terminal ← condition for left recursion

then replace with

A → βA'

A' → αA' | ε

this is general formula to eliminate direct left recursion without changing meaning of original grammar

Eg: G : E → E+T | T (α = +T, β = T)

T → T*F | F

F → (E) | id

This is left-recursive

We are trying to make left-recursive grammar into right-recursive grammar and then use ε to stop / Terminate.

Elimination of left recursion:

E → TE'

E' → +TE' | ε

T → FT'

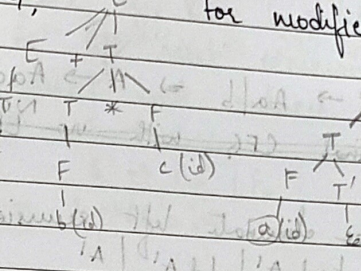
T' → *FT' | ε

F → (E) | id

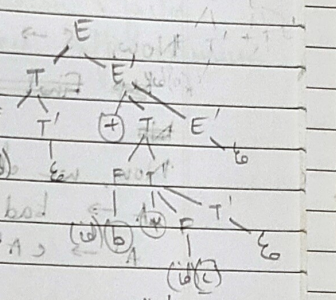
This is the recursion free grammar

For a+b* c, draw parse tree for the original and modified CFGs.

For original CFG:



For modified CFG:



12/10/22 * Algorithm for Elimination of Left Recursion (will be taught later)

i/p: G with no cycles and ε productions (harmful)

o/p: equivalent G with no LR (Left Recursion) (now we can consider ε productions)

Method:

Arrange all non-terminals in same order A1, A2, ..., An

for (each i from 1 to n) { ← total number of NON-Terminals

for (each j from 1 to i-1) {

Replace the G of form Ai → Ajγ into

Ai → δ1γ | δ2γ | ... | δkγ where Aj → δ1 | δ2 | ... | δk

}

Eliminate all immediate left recursion from Ai productions

}

Eg: S → Aa | b

A → Ac | Sad | bd | ε

First, A1 = S and A2 = A) in order that they appear

$A_i \to A_j \gamma$ $\quad n = 2$

$i = 1$

$j = 1 \to 1-1$ (no reference to A_0)

$i = 2$

$j = 1 \to 2-1 = 1$ (exists)

$A_2 \to A_1 \gamma$

$A \to S\gamma \Rightarrow A \to Sad$ (γ can be any string)

The matches $A \to Sad$ in given CFG, so replace γ with ad

Replacing S with its definition

Now, $S \to Aa \mid b \Rightarrow A \to Aaad \mid bad$

~~Key~~ ~~Final CFG with no LR~~ Now: $A \to \underbrace{Ac}_{\alpha_1} \mid \underbrace{Aaad}_{\alpha_2} \mid \underbrace{bad}_{\beta_1} \mid \underbrace{bd}_{\beta_2} \mid \underbrace{\varepsilon}_{\beta_3}$

~~bdo~~

Now, we eliminate left recursion as:

$A \to bad A' \mid bd A' \mid A'$

$A' \to cA' \mid aadA' \mid \varepsilon$

* Left Most Common Factoring

In $S \to Aa \mid b$

$A \to bad A' \mid bd A' \mid A'$

$A' \to CA' \mid aadA' \mid \varepsilon$

For $A \to \alpha\beta_1 \mid \alpha\beta_2$, do left factoring using $A \to \alpha A'$, $A' \to \beta_1 \mid \beta_2$

For this, $A \to b_1 B \mid A'$

$B \to adA' \mid dA'$

Q) Given $S \to 0S1 \mid 01$. Is this G suitable for top-down non-recursive predictive parsing?

It has no left recursion but has left most common factor (0).

Do left factoring as: $S \to 0S'$

$S' \to S1 \mid 1$

CFGs should always look like $A \to \alpha$, where A is free of left and right context, it is alone and α is any string (including ε)

Q) Given $S \to iEtS \mid iEtSeS \mid a$

$E \to id$

It has iEtS as common left most factor.

~~$S \to S' \mid s'es \mid a$~~

~~$S' \to iEt$~~

$S \to iEtSS' \mid a$

$S' \to \varepsilon \mid es$

$E \to id$

No left recursion

Q) $S \to SA \mid A$. Check if this has left recursion or left most common factor.

$A \to Sa \mid a$

It has both direct and indirect left recursion.

For $S \to SA \mid A \Rightarrow S \to AS'$

$S' \to AS' \mid \varepsilon$

For $A \to Sa \mid a \Rightarrow A \to AS'a \mid a$

$A \to aA'$

$A' \to S'aA' \mid \varepsilon$

starting with S, so may have indirect left recursion

No left recursion $\Rightarrow$ CFG with

$S \to AS'$

$S' \to AS' \mid \varepsilon$

$A \to aA'$

$A' \to S'aA' \mid \varepsilon$

It has no left most common factor

$\Rightarrow$ 6 Algorithms for predictive parsing

29/10/24 * Compute First (x) — token passed to lexical analyser

1. If X is a terminal, first(X) = {X} — zero or more derivations give ε (e.g. $X \to 4_1 4_2$, $4_1 \to \varepsilon$, $4_2 \to \varepsilon$)
2. If $X \to 4_1 4_2 \ldots 4_n$, $4_1, 4_2, \ldots, 4_n \overset{*}{\Rightarrow} \varepsilon$, then
   $First(X) = First(4_1) \cup First(4_2) \cup \ldots First(4_n) \cup \varepsilon$
   If $4_1, 4_2, \ldots, 4_{n-1} \overset{*}{\Rightarrow} \varepsilon$, then ~~$First(X) = First(4_1) \cup First(4_n)$~~
   $First(X) = First(4_1) \cup First(4_2) \ldots \cup First(4_{n-1})$
3. $First(\varepsilon) = \{\varepsilon\}$

$X \to a \Rightarrow$ {ε: $X \to 4_1 4_2 4_3$, $4_1 \to \varepsilon$, $4_2 \to \varepsilon$, $4_3 \to a$}

do only one for exam

Eg: Given G: $E \to TE'$
$E' \to +TE' | \varepsilon$
$T \to FT'$
$T' \to *FT' | \varepsilon$
$F \to (E) | id$

Compute first(x) for all non-terminals.

Non-terminal	First → what each non-terminal can start with
E	{(, id}
E'	{+, ε}
T	{(, id}
T'	{*, ε}
F	{(, id}

Q) Given $S \to ABC$
$A \to a | \varepsilon$
$B \to b | \varepsilon$
$C \to e | \varepsilon$

What is first(s)?

First(s) = {a, b, e, ε}

Non-terminal	First
S	{a, b, e, ε}
A	{a, ε}
B	{b, ε}
C	{e, ε}

Q) Given $S \to ABC$
$A \to a | \varepsilon$
$B \to b$
$C \to e$

Compute First(s)

First(s) = {a, b}

Non-terminal	First
A	{a, ε}
B	{b}
C	{e}

In exam, if asked to compute Follow using algorithm, do the full definition of what terminals can be expected/allowed after a non-terminal

* Compute Follow(A)

1. Place $ in Follow(S), where S is the start symbol.
2. If G is in the form $A \to \alpha B \beta$, where First(β) does not contain ε, then everything in First(β) except ε should be in Follow(B).
3. If G is in the form $A \to \alpha B$ or $A \to \alpha B \beta$, and First(β) contains ε, then everything in Follow(A) is in Follow(B).

Eg:

Non-terminal	Follow
E	{$,)
E'	{$,)
T	{+, $,)
T'	{+, $,)
F	{*, +, $,)

Using algorithm:
$E \to TE'$ and $A \to TE'$
$A \to \alpha B$ $A \to \alpha B\beta$
① F(E) ⇒ F(E')
② F(T) ⇒ First(E') − ε
③ F(E') ⇒ F(T)
Now, $E' \to +TE'$ $E' \to +TE'$
$A \to \alpha B\beta$ or $A \to \alpha B$
④ F(E') ⇒ F(T)
⑤ F(T) ⇒ F(T')
⑥ F(F) ⇒ First(T') − ε
⑦ F(T) ⇒ F(F)
⑧ F(T') ⇒ F(F)
⑨ F(E) ⇒)

Now, $T \to FT'$

Use do this

Using Shortcut:

Non-terminal	Follow
E	{$,)
E'	{$,)
T	{+, $,)
T'	{+, $,)
F	{*, +, $,)

$E \to TE'$
Any non-terminal at end of production has follow equal to follow of start of production
Here, follow(E') = follow(E)

20/10/24 Q) a) Is this grammar suitable to be parsed using predictive parsing? Modify, if required.
Given $S \to aAbA | aAbc | ScA$
$A \to aAbab | b$

b) Compute First and Follow of all the non-terminals:
i) $S \to +cs | *ss | a$
ii) $S \to aSbS | bSas | \varepsilon$

2) Check whether the G is suitable for predictive parsing. Modify, if required. Compute first and follow of all the non-terminals:

i) $S \to S+S \mid S*S \mid S+ \mid (S) \mid a$

ii) $S \to A$; $A \to Bc \mid DBC$; $B \to Bb \mid \varepsilon$; $C \to c \mid \varepsilon$; $D \to a \mid d$

iii) $S \to S(S)S \mid \varepsilon$

1) a) ~~There is direct and~~

$S \to ScA$ is of the form $A \to A\alpha \mid \beta \Rightarrow A \to \beta A'$, $A' \to \alpha A' \mid \varepsilon$

$\therefore S \to aAbAA' \mid aAbcA'$

$A' \to cAA' \mid \varepsilon$

$A \to aAbab \mid b \Rightarrow$ This now has no left recursion

Left factoring $\Rightarrow$ $S \to aAbS'$

$S' \to AA' \mid cA'$

$A' \to cAA' \mid \varepsilon$

$A \to aAbab \mid b$

1) b) i)

Non-terminal	First	Follow
S	{+, a, *}	{+, *, a, $}

always add $ to follow(s) when S is start symbol

→ do using algorithm to practice

ii)

Non-terminal	First	Follow
S	{a, b, ε}	{a, b, $}

2) i) $S \to S+S \mid S* \mid S+ \mid (S) \mid a$

This has left recursion

~~$S \to S* \mid S+ \mid (S) \mid a \quad S*S' \mid S+S' \mid (S)S' \mid aS'$~~

~~$S' \to +SS' \mid \varepsilon$~~

$S \to (S)S' \mid aS'$

$S' \to +SS' \mid *S' \mid +S' \mid \varepsilon$

~~This has no left most common factor, so no left factoring needed~~

Non-terminal

There is one leftmost common factor (+) in S'

Now, $S \to (S)S' \mid aS'$

$S' \to +S'' \mid *S' \mid \varepsilon$

$S'' \to SS' \mid S'$

stop here for exam, no need to do left recursive check

Non-terminals	First	Follow
S	{(, a}	{$, +, *,)}
S'	{+, *, ε}	{$, +, *,)}
S''	{(, a, +, *, ε}	{$, +, *,)}

no ε here since it is ε

iii) $S \to S(S)S \mid \varepsilon$

Eliminating left recursion: $S \to S'$, $S' \to (S)SS' \mid \varepsilon$

No leftmost common factor

Non-terminal	First	Follow
S	{(, ε}	{$, (,)}
S'	{(, ε}	{$, (,)}

ii) $S \to A$ $B \to Bb \mid \varepsilon$ $D \to a \mid d$

$A \to Bc \mid DBC$ $C \to c \mid \varepsilon$

For $B \to Bb \mid \varepsilon$, $B \to B'$ $B' \to bB' \mid \varepsilon$

Now, new grammar is: $S \to A$ $A \to Bc \mid DBC$

$B \to B'$ $B' \to bB' \mid \varepsilon$

$C \to c \mid \varepsilon$ $D \to a \mid d$

No leftmost common factor

if S does appear in RHS, then only $

Non-terminal	First	Follow
S	{b, ε, a, d, c}	{$}
A	{b, ε, a, d, c}	{$} — same as S
B	{b, ε}	{c, $}
B'	{b, ε}	{c, $} — what follows B
C	{c, ε}	{$}
D	{a, d}	{b, c, $}

because B can be ε, then it is DC

A → DBC

CFG normal form: $A \to \alpha$ (non-terminal → terminal)
→ only for CFGs

4/11/24 * Predictive Parse Table M[A, a]

① If G is of the form $A \to \alpha$, compute First(α) and for each terminal a in First(α), add $A \to \alpha$ to M[A, a]
Row show non-terminals and columns show input symbols

Eg: $S \to 0S1 \mid 01$
This has 0 as leftmost common factor.
We change it as: $S \to 0S'$, $S' \to S1 \mid 1$
Non-terminals in above grammar are S, S' (↳ start symbol)
Terminals are 0 and 1, and take $ (↳ end of line)

NT → non terminal

Parse table:

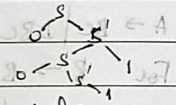
NT \ input symbol	0	1	$ → tokens...
S	$S \to 0S'$		
S'	$S' \to S1$	$S' \to 1$	

$A \to \alpha$

For $S \to 0S'$, First(0S') = 0
M[A, a] = M[S, 0]

For $S' \to S1$, First(S1) = 0
M[S', 0] ⇒ add $S \to S1$ in table for

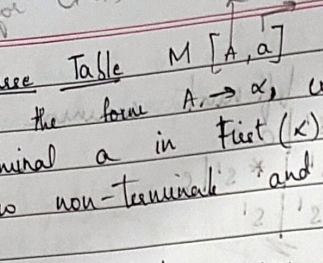
Q) Using $S \to 0S'$, $S' \to S1 \mid 1$, 0011$ ⇒



should first have $ and start symbol(s)

Stack	Input (scanned from left to right)	Action
S$	0011$	
0S'$	↑ pop since matched	Match 0 (0 and 0 matched)
pop 0 since it is matched, S'$	011$	match S' and 0 given CFG
S1$		
0S'1$		Match 0
pop 0, S'1$	11$	
11$	1$	Match 1
pop 1, 1$		Match 1
pop 1, $	$	0011 → Accepted

Q) Given $S \to CC$, $C \to cC \mid d$ and input: cdd$



Stack	Input	Action
S$	cdd$	
CC$		
cCC$		
matching → pop		matched c
CC$	dd$	
dC$		
matching → pop		matched d
C$	d$	
d$		
matching → pop		matched d
$	$	cdd → Accepted

Parse table:

NT \ input symbol	c	d	$
S	$S \to CC$	$S \to CC$	
C	$C \to cC$	$C \to d$	

For $S \to CC$, First(CC) = {c, d}
M[S, c] = M[S, d]

For $C \to cC$, First(cC) = {c}
M[C, c] = $C \to cC$
For $C \to d$, First(d) = {d}
M[C, d] = $C \to d$

4/11/24 * Predictive Parse Table

② If First(α) contains ε, then compute Follow(A). For each terminal b in Follow(A), add $A \to \alpha$ to M[A, b].
If Follow(A) contains $, add $A \to \alpha$ to M[A, $]

Q)

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

Predictive Parse table:

non terminals (NT) \ input symbols	+	*	(	)	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	~~$T' \to \epsilon$~~		$F \to (E)$	~~$T' \to \epsilon$~~	$F \to id$	~~$T' \to \epsilon$~~

$E \to TE'$ First $(TE') = \{(, id\}$

$E' \to +TE'$, First $(+TE') = \{+\}$

also $E' \to \epsilon$, Follow $(E') = \{\$,)\}$

$T \to FT'$, First $(FT') = \{(, id\}$

$T' \to *FT'$, First $(*FT') = \{*\}$

$T' \to \epsilon$, Follow $(T') = \{\$, +,)\}$

$F \to (E)$, First $((E)) = \{(\}$

$F \to id$, First $(id) = \{id\}$

Do parsing for input: id + id * id $

Stack	Input	Action
$E\$$	id + id * id $	
$TE'\$$		
$FT'E'\$$		
$id\ T'E'\$$	matched − pop	matched id
$T'E'\$$	+ id * id $	
$E'\$$		
$+TE'\$$	matched − pop	matched +
$TE'\$$	id * id $	
$FT'E'\$$		
$id\ T'E'\$$	matched − pop	matched id
$T'E'\$$	* id $	
$*FT'E'\$$	matched − pop	matched *
$FT'E'\$$	id $	
$id\ T'E'\$$	matched − pop	matched id
$T'E'\$$	$	
$E'\$$		
$\epsilon\$$	$	

~~Accepted~~

id + id * id

↳ Accepted

Q) Difference between recursive descent and predictive parsing.

Both are top-down parsers. Recursive descent parser (RDP) has recursion so uses backtracking which predictive parser does not.

Q) Do predictive parsing by constructing parse table for:

1) $S \to S(S)S \mid \epsilon$

2) $S \to iEtSS'$, $S' \to eS \mid \epsilon$, $E \to id$

1)

NT \ input symbols	(	)	$
S	$S \to S'$	$S \to S'$	$S \to S'$
S'	$S' \to (S)SS'$	~~$S' \to (S)SS'$~~	~~$S' \to (S)SS'$~~
	$S' \to \epsilon$	$S' \to \epsilon$	$S' \to \epsilon$

If there are multiple entries it means grammar is not LL(1) and parser cannot parse it.

For $S \to S(S)S \mid \epsilon$, we remove left recursion as

$S \to S'$, $S' \to (S)SS' \mid \epsilon$ → also First (S') has ϵ

$S \to S'$, First $(S') = \{(, \epsilon\}$, Follow $(S) = \{\$, (,)\}$

$S' \to (S)SS'$, First $((SSS') = \{(\}$, ~~Follow $(S') = \{(, \$\}$~~

$S' \to \epsilon$ Follow $(S') = \{(, \$,)\}$

it is ambiguous

2) Parse table:

NT \ input symbol	i	t	e	id	$
S	S→iEtSS'				
S'			S'→eS / S'→ε		S'→ε
E				E→id	

S → iEtSS' First(iEtSS') = {i}
S' → eS First(eS) = {e}
S' → ε Follow(S') = {$, e}
E → id First(id) = {id}

This grammar is also not in LL(1) form and is ambiguous as there are multiple entries for M[S', e]

for more marks, construct table and check for multiple entries

Same for 1)

Only for cross-checking (for 1-2 marks only)

6/11/24 * Algorithm for Checking if G is LL(1)

Any CFG of the form A → α | β,

1. For the G: A → α/β, α and β should not derive any string beginning with a same terminal, ~~that is, α or β ⇒* ε~~
2. Any one of the G can derive ε, that is, α or β ⇒* ε
3. If β ⇒* ε, then Follow(A) should not have the terminal 'a', in the First of definition of α and vice-versa.

A → ε (zero derivation)
S → A (one derivation) ⇒
A → ε

In A → α/ε, First(α) should not start with the terminal that is in Follow(A).

→ suitable for non-recursive

BDP (Predictive parser)

LR → A → Aα | β
↳ A → βA'
A' → αA' | ε

A → βA'
A' → αA' | β

Page No.
Date.

9) a) Check if S → (L), L → L,S | id is LL(1) or
Remove left recursion, left common factors, c
create parse table, and look for multiple ent
S → (L) Removing left recursion → L → id L'
L' → ,SL'

Parse table:

NT \ input symbols	(	)	id	,	$
S	S→(L)				
L			L→idL'		
L'		L'→ε		L'→,SL'	

S → (L) First((L)) = {(} Follow(S)
L → idL' First(idL') = {id} Follow(L)
L' → ,SL' First(,SL') = {,} Follow(L'
L' → ε Follow(L') = {)} This is LL(1) e
there are no mu
definitions

b) S → S(S)S | ε

Removing Left Recursion, S → S' S' → (S)SS
Using algorithm First((S)SS') = {(} Follow(S)
Since ~~Follow(S) = {$}~~ Follow(S') = {(,), $} Follow(S
~~Since Follow~~ Using (3) from Algorithm First
to check if G is LL(1), this is not LL(1)

8) Given S → ++s | *ss | a, parse the input +* aa$
the (predictive parsing algorithm) → in picture (phone)

Parse table:

NT \ input	+	*	a	$
S	S→+SS	S→*SS	S→a	

w$ → represent a in aa+$

Stack	Input	Action
~~+aaa~~ $ S	+*aaa$	
+S$	matched - pop	matched +
S$	*aaa$	
*SS$	matched - pop	matched *
SS$	aaa$	
aSS$	matched - pop	matched a
SS$	aa$	
aS$	matched - pop	matched a
S$	~~matched - pop~~	
a$	matched - pop	matched a
$	$	+*aaa$ Accepted

important

* Learn predictive passing algorithm for exam with correct notations (in phone - picture)

Q) With the help of predictive passing algorithm, do passing for an input with a given grammar write and explain algorithm for predictive passing, draw parse table and the stack thing

* Error Recovery in Predictive Parsing

→ Panic Mode Recovery

synchronizing token - synch - delimiters - signify end of statement, end of file, end of function - }, ; in C

Steps:

1. If CFG is of form $A \to \alpha$, compute Follow(A) for all non-terminals
2. Check if M[A, $] or M[A,)] exists. Write sych in Parse table for M[A, $] and M[A,)] if they are empty.

Q) Given G: $E \to TE'$, $E' \to +TE' | \epsilon$, $T \to FT'$, $T' \to *FT' | \epsilon$, $F \to (E) | id$

Non-terminal	Follow
E	$,)
E'	$,)
T	
T'	
F	$,)

Do this after constructing parse table using normal method

If M[A, e] is empty add synch there

Then, draw stack, input and action table as:

1. If M[A, a] = blank, skip input symbol
2. If M[A, a] = synch, pop ~~would not be exact table~~ non terminal on stack top
3. If token on stack top mismatches input symbol, pop stack.

→ Panic Mode Recovery Steps

Parse Table:

NT \ input symbol	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$	synch	synch	$F \to (E)$	synch	synch

After this, do the stack | input | action using panic mode recovery steps (in phone - picture)

Left Recursion
Left Factoring
First
Follow
Parse table
Parse tree
Error recovery

Algorithm in predictive parsing

For exam, remember:

1. M[A, a] for parse table, where
   M → parse table notation
   a → terminal → any lower case letter
   A → non-terminal → any capital letter

all bottom-up parsers do

11/11/24 * Bottom-Up Parsing → reductions

→ LR Parsers

Q) $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

Input string: id * id $

RMD: $E \xrightarrow{rmd} T \xrightarrow{rmd} T*F \xrightarrow{rmd} T*id \xrightarrow{rmd} F*id \xrightarrow{rmd} id*id$

For all bottom-up parsers, ~~input~~ ~~RFG~~ grammar can be of any form. Here, we take input string and reduce it to given S. if possible

id has been reduced to F, so called reduction

id has

handle and handle pruning — will be explained

RSF — Right Sentential Form — in right most derivation, all the intermediate forms of representation from end to source

same for LSF — Left Sentential Form but in left most derivation, from source to end.

* Shift Reduce Parser

Main actions in all bottom-up parsers are:

1. Shift
2. Reduce
3. Whether input to be Accepted
4. Error

→ Process of Shift Reduce Parser

Q)

Stack	Input	Handle	Action
$	id * id $		shift id
$ id	* id $	id	reduce F→id
$ F	* id $	F	reduce T→F
$ T	* id $		shift *
$ T *	id $		shift id
$ T * id	$	id	reduce F→id
$ T * F	$	T*F	reduce T→T*F
$ T	$	T	reduce E→T
$ E	$		Accepted

shifted to top of stack

left to right

pop id and push F

* shifted

can be reduced/taken as T*, here take longest matching string

longest matching string is taken always like T*, and not T

when start symbol of grammar is at top of stack, input is accepted

→ All symbols in RSF which are reduced are called handles. Handles appear at the top of SRP stack. Handles present at right hand side of RMD which have possibility of getting reduced.

→ LR parsing method is used to tell parser when to shift and when to reduce.

→ Handle pruning —

$S \xrightarrow{rmd} 0S1 \xrightarrow{rmd} 0011$

Q) Implement SRP for $S \to 0S1 \mid 01$ given input as 0011 $, and identify the handles.

Stack	Input	Handle	Action
$	0011 $		shift 0
$ 0	011 $		shift 0
$ 00	11 $		shift 1
$ 001	1 $	01	reduce S→01
$ 0S	1 $		shift 1
$ 0S1	$	0S1	reduce S→0S1
$ S	$		Accepted

has

not a handle

* Simple LR Parser / LR(0) Parser — Basic LR parser

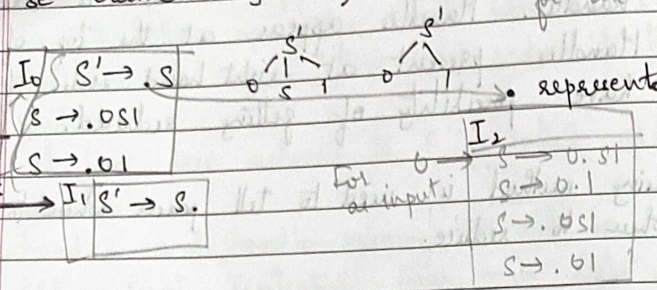
SR(0) left-to-right

Tell parser to stop when start symbol of grammar is at top of stack.

Page No.
Date. / /

Q) $S \to 0S1 \mid 01$
Only when start symbol is there on RHS, we can tell the parser to stop parsing (or its definition)

Step 1: Identify the start symbol. The start symbol must be derived by any other non-terminal → important for all Bottom-Up parser



12/11/24 Bottom Up Parsing

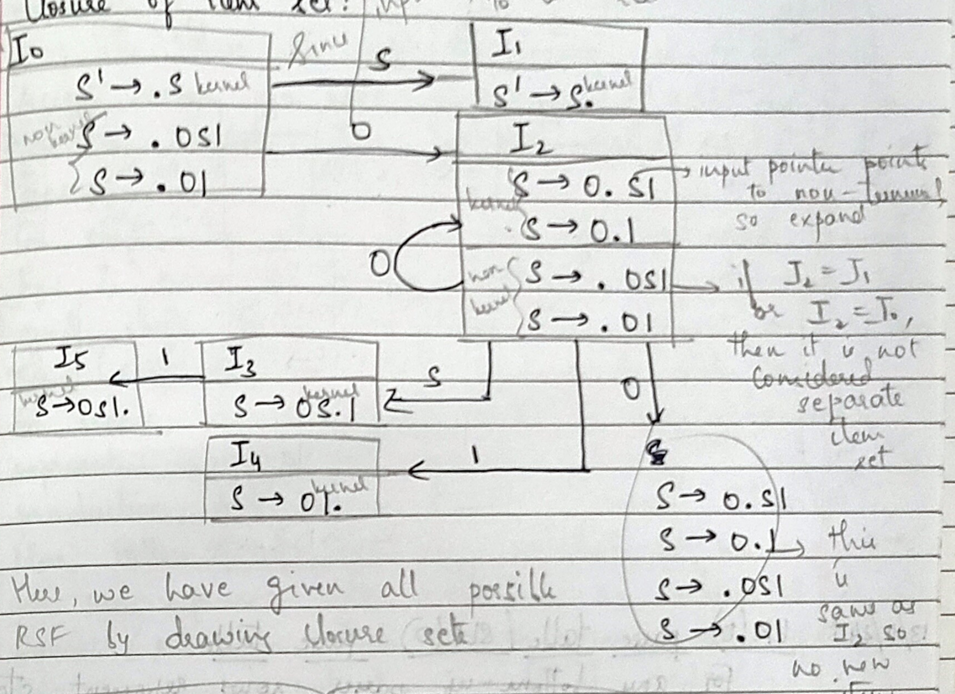
* Closure of Item Sets
If I is a set of items in G, then closure (I) is the set of items constructed from the two rules given:
1. Initially add every item in I to closure (I)
2. If $A \to \alpha . B\beta$ is in closure(I) and $B \to \gamma$ is a production, add $B \to \gamma$ to closure(I) if it is not already there.

→ Kernel items – Initial item $S' \to .S$ and all items whose dots are not at the left end.
→ Non-kernel items – All items whose dots are at left end, except $S' \to .S$
→ Handle – substring that matches the (body of a production) RHS of Grammar, and whose reduction represents one step along the reverse of an RMD. (can be asked) write this

Eg: $S \to 0S1 \mid 01$
Step 1: $S' \to .S$ input pointer points at definition of S(start)

Page No.
Date. / /

Closure of item set: input pointer is pointing to terminal, our item set is done



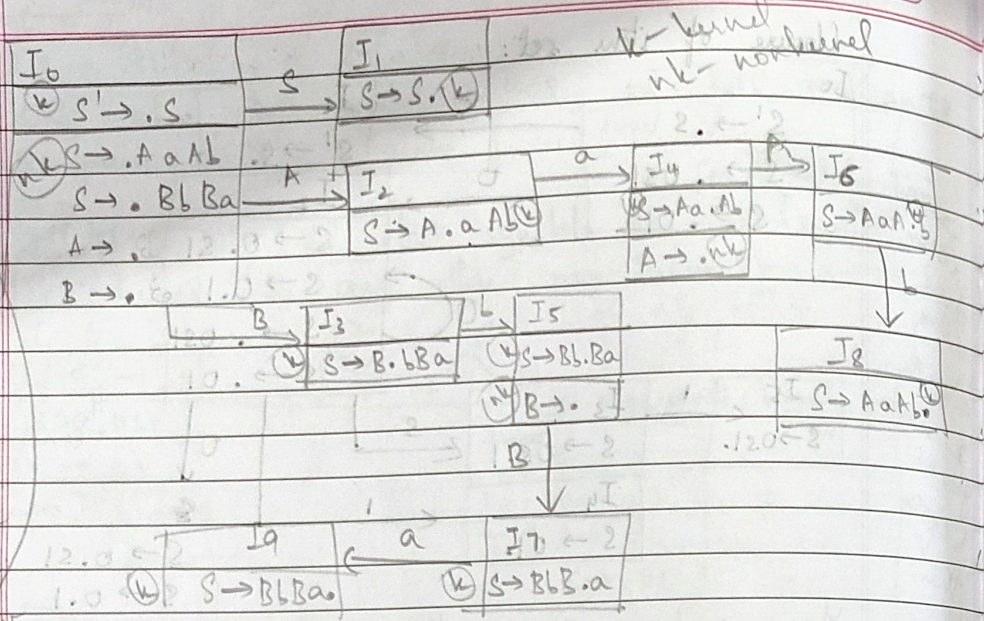
Here, we have given all possible RSF by drawing closure sets

This is a deterministic automata as there are no multiple paths
This is DFA (Deterministic Finite Automata)

Bottom up parser stops parsing when for $S \to \ldots$, we get $S' \to S.$ in any one of the item sets. Also, we should have all handles followed by . in any one of the item sets to show that the input has been passed correctly. Eg: when handle = start symbol, stop.
Wherever we are adding new productions for definition of non-terminals, those newly added are called non-kernels and others are called kernels.

Q) Given $S \to AaAb \mid BbBa$
$A \to \epsilon$
$B \to \epsilon$
Compute the closure of item sets in LR(0) parser

Don't write $A \to .\varepsilon$ or no marks



13/11/24* **LR(0) parse table / SLR(0) parse table**

For any bottom-up parser, rows represent state/item and columns represent Action (terminals) and GOTO (non-terminals)

Action: accept, error, shift, reduce

Q) For $S \to AaAb$ (1), $S \to BbBa$ (2), $A \to \varepsilon$ (3), $B \to \varepsilon$ (4), LR(0) parse table

do using item sets above

State	Action a	Action b	Action $	GOTO S	GOTO A	GOTO B
0	r_3/r_4	r_3/r_4		1	2	3
1			accept			
2	S_4					
3		S_5				
4	r_3	r_3			6	
5	r_4	r_4				7
6		S_8				
7	S_9					
8			r_1			
9			r_2			

reduce reduce conflict

GOTO(I_0, S) = I_1 — from I_0 when input is S go to I_1

↳ only scan over non-terminals

↳ tells where next input pointer should be moving

Accept when we have start symbol., such as S. in previous example. Then, for that state, for state, $, accept

For shift, see where there are transitions over terminals

Eg: I_2 moves to I_4 on input 'a' ~~so add shift to~~ and shifts 'a'. In parse table, represent [2, a] as S_4 (shift to I_4) → shift input a to state I_4

For reduce, whenever . (dot) is at the ~~rightmost~~, and ~~represents~~ rightmost end, compute follow of head of production, such as for $A \to .$, compute Follow(A)

Here, Follow(A) = {a, b}. Now, in item I_0, we see $A \to .$ we can represent reduce $A \to \varepsilon$ as r_3 where 3 is number/order in which $A \to \varepsilon$ appears in original grammar and put r_3 in [0, a] and [0, b].

r_3/r_4 represents conflict — there can be two types of conflicts — shift-reduce or reduce-reduce conflicts.

Follow(S) = {$}

For verification, we must check that there are r_i where $i = 1$–4 here atleast once in the parse table.

↳ number of grammars in the question

Q) For $S \to 0S1 \mid 01$ (1) (2), construct parse table

State	Action 0	Action 1	Action $	GOTO S
0	S_2			1
1			accept	
2	S_2	S_4		3
3		S_5		
4		r_2	r_2	
5		r_1	r_1	

5 says r_1 so reduce first grammar. 0, 0 represent row ↓ column ↓ in parse table. If it says s_2, shift 0 and add 2 to stack top

19/11/24 * **SLR Parsing** → for any LR parser

Q) Given $S \to 0S1$, $S \to 01$, input = 0011\$

initially, should contain start → represent handles

Stack	Symbol	Input	Action
0	0	0011 \$	
02	00	011 \$	
022	001	11 \$	
0224	0S	1 \$	
023	0S1	\$	
0235	S		
01		accept	

done using parse table in previous page

pop as many states as matched

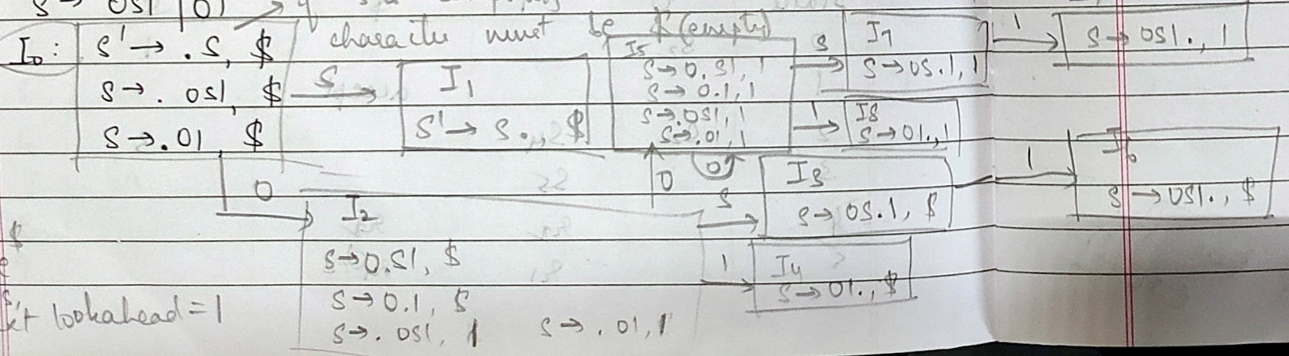
→ **Algorithm**

Take string w\$ take pointer ip pointing to w, check in parse table for M[St, ip], if s_4, shift and push 4 to stack. If r_1, reduce first grammar from symbol to head of production of first grammar.

Number of elements reduced in symbol = number of states popped from stack.

* **Canonical LR Parser (CLR(1))** → any LSF allows for checking
with one look-ahead

Q) $S \to 0S1 \mid 01$ → if S is completely scanned, the next lookahead character must be \$ (empty)



For $S \to 0.S1, \$$ when we expand S, the next lookahead = 1

→ **Algorithm**

If $A \to \alpha.B\beta, r$, then $B \to .a$, First(βr)

A and B are non-terminals, and α, β are terminals

eg.: $S \to 0.S1, \$$

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

$S \to .0S1, 1$

$S \to .01, 1$

Follow(S) = \$

Here, no need to compute follow and all. Just write in I4 for r2 for S →

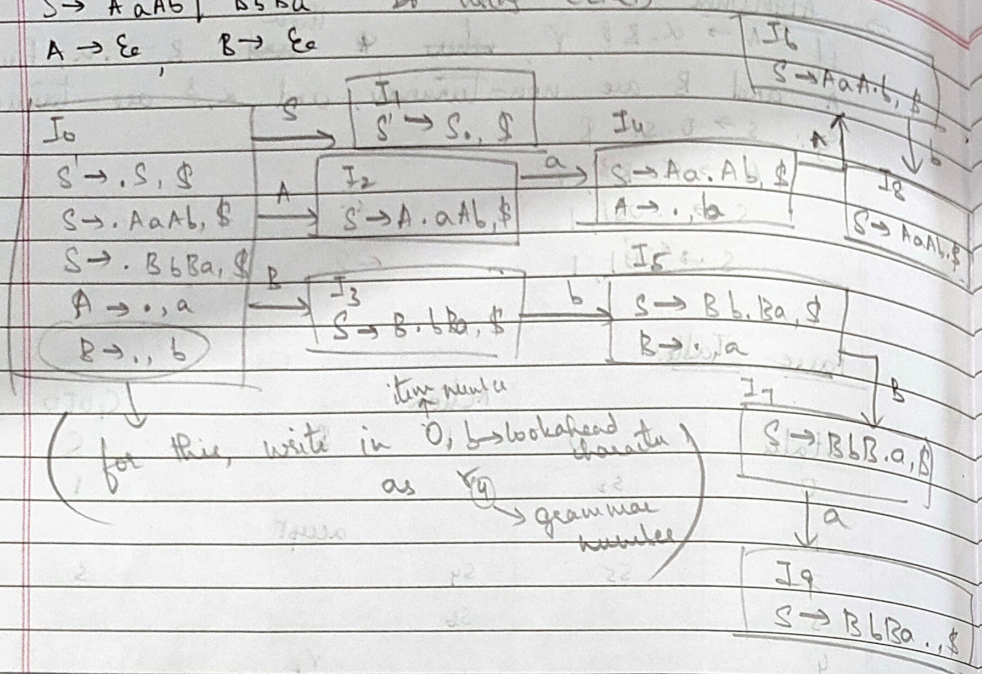
Parse Table:

State	Action 0	Action 1	Action \$	GOTO S
0	s2			1
1			accept	
2	s5	s4		3
3		s6		
4			r2	
5	s5	s8		7
6			r1	
7		s9		
8		r2		
9		r1		

Page No.
Date. / /

(8) $S \to AaAb \mid BbBa$ Do using CLR(1)

$A \to \varepsilon$, $B \to \varepsilon$



(for this, write in I0, b → lookahead of production as r4 → grammar number)

item number

Passe Table:

State	Action a	Action b	Action $	GOTO S	GOTO A	GOTO B
0	r_3	r_4	~~acc~~	1	2	3
1			accept	~~accept~~		
2	s_4					
3		s_5				
4		r_3			6	
5	r_4					7
6		s_8				
7	s_9					
8			r_1			
9			r_2			

→ no conflicts here

will have for CLR-1