Book
Auth → Aho, Ulman

(JIT compiler)

Date Jan 12th, 2023

Compiler Design

Introduction

* Compiler is a program that translates a source program (a program written in a source language) & translate it to the equivalent program (target program, program written in a target language.)

Source Program → [Compiler] → Target Program

- .c/.cpp
- .java

- .obj
- .asm

---

(2)

obj1
obj2
objn

→ DLL → [Linker] → .exe

---

(3) .exe → [Loader] → [memory]

(1) .asm → [Assembler] → .o

Source Program
Input → [Interpreter] → Output

x) The machine language program

Date .....

Assembler → The compiler may produce an assembly language program as its output. Because it is easier to produce as output & its easier to debug. The assembly lang. program is then processed by the program called assembler that produces the relocatable object code as its output.

Linker - Large Programs are often compiled in pieces. So the relocatable code may have to be linked together with other relocatable object files. library files into the code. that actually runs on the machine. The linker is also responsible for this task. It gives the exe.

Loader → The loader finally puts the executable file into memory for execution.

Structure of the Compiler

It is divided into two parts phases

i) Analysis (front-end)

ii) Synthesis (Back-end)

The Analysis breaks up the source program into constituent pieces & imposes the grammatical structure on them

Spiral

Date .....

It then uses the structure to create an intermediate representation (IR) of source program. The Analysis also collects the information about the source program & stores it in a data structure called a symbol table, which is passed along with IR to the synthesis part.

Synthesis part constructs the desired target program from the IR & information in the symbol table.

Phases of a Compiler (fig 1.6)

character stream

↓

Lexical Analyzer

↓ Token Stream

Syntax Analyzer

↓ Syntax tree

Semantic Analyzer

↓ Syntax tree

Intermediate Code Generator

↓ IR

Machine - Independent Code Optimizer

↓ Intermediate Representation

Code Generator

↓ Target Machine Code

Machine Dependent Code Optimizer.

↓ target machine code

Spiral

Symbol Table

Date .....

Lexical Analysis

- It is also known as Lexical Analysis or scanning.
- It is done by the program called lexical analyser.

The first phase of a compiler is called lexical analyser. It reads the stream of characters and groups the characters into meaningful sequences called lexemes.

For each lexeme, the lexical analyser produces as output a token of the form

<token_name, attribute value>

Ex → int a,b,c;

<INTEGER><id,1> <COMMA> <id,2><COMMA>
<id,3> <SEMI-COLON>

Symbol table →

1	a	int	4
2	b	int	4
3	c	int	4

position = initial + rate * 60

Translation of an Assignment Statement

Date .....

fig 1.7

position = initial + rate * 60

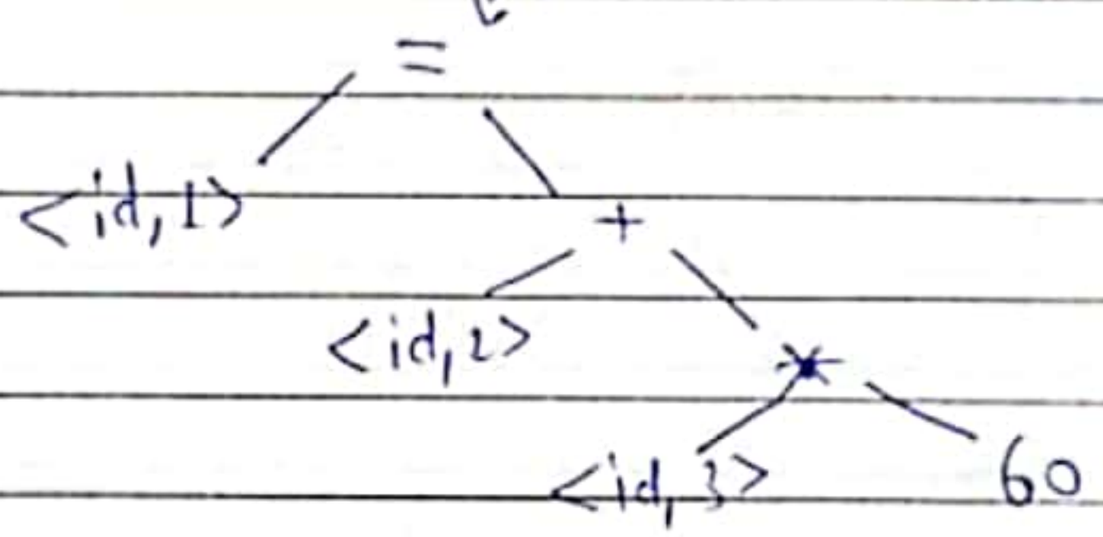
↓

Lexical Analyzer

<id,1> <=> <id,2> <+> <id,3> <*> <60>

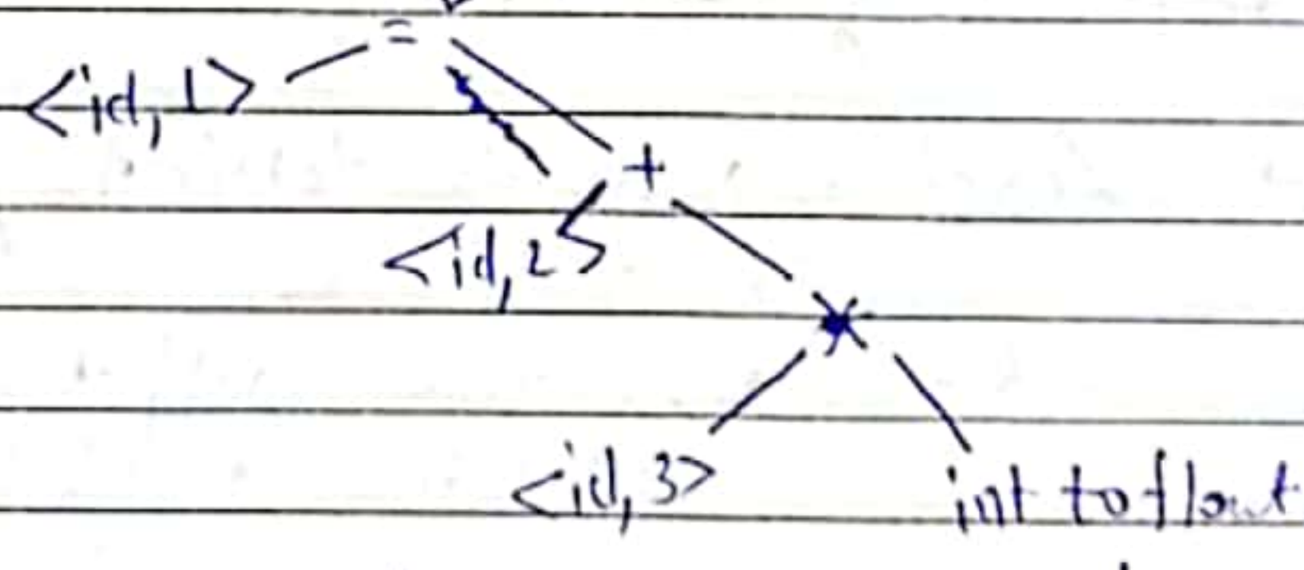
↓

Syntax Analyzer



↓

Semantic Analyzer



↓

Intermediate Code Generator

t_1 = int to float (60)

t_2 = id3 * t_1

t_3 = id2 + t_2

id1 = t_3

↓

Code Optimizer

t_1 = id3 * 60.0

↓

Code Generator

LDF R2, id3

MULF R1, R2, #60.0

LDF R1, id2

ADDF R1, R1, R2

(ch-2 in sk-lpha)

Date Jan. 16th 2023

Chapter-3 Lexical Analysis

Syntax:

```
whitespaces
pattern .. {action}
pattern    {action}
:
pattern    {action}
```

Rule for Syntax

```
%%
definition
%%
Rules
%%
auxiliary func^
```

1) Read the input characters
2) Group them into lexemes.
3) Sequence of tokens for each lexeme.

Note → Lexemes → Words derived from input character stream

Token → Lexemes mapped into token name & an attribute value.

$X = a + b * 2$

Tokens = { <id, 1>, <=>, <id, 2>, <+>, <id, 3>, <*>, <num> }

Spiral

Date .....

Lexeme → sequence of characters recognized by a pattern.

pattern → structure that is matched by multiple strings
↳ [_a-z A-Z][a-z A-Z 0-9_]*

Ex → printf("Total = %d \n", score);

Lexeme →
<id, 1><left parenthesis><string literal, "Total = %d \n">
<comma><id, 2><parenthesis><semicolon>

- We want to define 1 token for each keyword
- Pattern for the keyword will be same as the keyword itself.
- One token identifies all the identifiers.
- Token for each punctuation symbol such (,), <, >, <=, etc.
- The token name influences the parsing actions.

Handling Lexical Errors

There are 4 recovery actions that lexical analyzer takes.

1) Delete 1 character from remaining input.
2) Insert a missing character in the remaining input.
3) Replace a character by another character.
4) Transpose two adjacent characters.

Date .....

Alphabet :- finite set of symbols

String :- Word made up of finite sequence of symbols drawn from the alphabet.

* |s| → length of string

* ε → string of length 0.

Language :- Countable set of strings over some fixed alphabet
(may be uncountable in some cases).

Prefix → by removing 0 or more symbols from the end of string.

Total (n+1) prefixes for string of length n+1.

Suffix → by removing 0 or more symbols from the starting of string.

Substring → by removing any prefix or any suffix from the string

Total $\frac{n(n+1)}{2} + 1$ substrings for string of length n

$\left(\sum_{i=1}^{n} i + 1\right)$

Proper prefix by removing atleast one character but not all from the string.

Subsequence → deleting 0 or more but not necessarily consecutive symbols of s.

Total → 2^n

Spiral

Date .....

Union of L_1 & L_2 →

$$L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$$

Concatenation of L_1, L_2 →

$$L_1 L_2 = \{ st \mid s \in L_1 \text{ and } t \in L_2 \}$$

Kleene's Star →

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Kleene's Closure →

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

Note →

$|LUD| = 62$
$|LD| = 520$

$L = \{a, b, \ldots z, A, B, \ldots Z\}$
$D = \{0, 1, 2, \ldots 9\}$

$L(LUD)^*$ → Set of all strings starting with a letter.

$L(a) = \{a\}$ → Regular set
(Reg exp)

The Unary Order of Precedence → ① The unary operator * has the highest precedence & left association.
② Concatenation has second highest precedence & left association.
③ Union has the lowest precedence & left association.

Spiral

yytext → Accepted strings by the patter
↳ pointer to the char.

Date .....

E.g. - Parenthesizing the reg exp. $a\,b^+\,c$

$(a|((b)^+\,c))$

Regular Definition

letter_ → A|B|....|Z|a|b|....Z|_
digit → 0|1|2....|9
id → letter_ (letter_|digit)*
digits → digit +
optional fraction → .digits| ε
optional Exponent → (E(+|-|ε) digits) | ε
Number → digits optional Fraction optional Exponent

In other way,

letter_ → [A-Z a-Z _]
digit → [0-9]
id → letter_
digits → digit +
number → digits (.digits)? (E[+-]? digits)?

Indian Mobile Number
→ [6-9][0-9]{9}

^ → Beginning of a line; $ → End of the line
^[a-z] → Beginning of a line

[^a-z] → Negation of a-z

r_1/r_2 → r_1 when followed by r_2

^[^aeiou]*$ → Any line that does not contain any lowercase vowel.

Date .....

Q Write regular definition for the following languages:-

i) All strings of lowercase letter that contain vowels in order

ii) All strings of lowercase letters in which the letters are lexicographically (ordered (in ascending order)

iii) String literal

Ans →

str → [a e i o u]* a* [b-d]* e* [f-h]* i* [j-n]* o* [p-t]* u*

alpha

str → a* alpha e* alpha i* alpha o* alpha u* alpha

alpha → [^ a e i o u]

Recommended →

Given

(a) str → con* a (con|a)* e (con|e)* i (con|i)* o (con|o)* u (con|u)*

con → [^ a e i o u] or [b-d f-h j-n p-t v-z]

(b) str → a* b* c* ... z*

(c) x str → [a-z A-Z 0-9 _]*
[^ "]

Spiral

"hi\hello"

Date .....

Ans →

" [^"] "

Ans →

" (("//") | [^"]) * "

* → Single line comment →

(\/\/) [^\n]

or

" // " [^\n]

feb 2nd. 2023

In the process of lexical analysis, if it encounters the white space, it does not return to the parser, rather it restarts the process.

Q → Lex program that copies a file replacing each non-empty sequence of white space by a single blank

%{

#include<stdio.h>

%}

Date .....

```
%%
[" "\t]+    { fprintf(yyout," "); }
.|\n        { fprintf(yyout,"%s",yytext); }
%%
int main(){
    yyin = fopen("input.txt","r");
    yyout = fopen("output.txt","w");
    yylex();
    fclose(yyin);
    fclose(yyout);
    return 0;
}
```

feb 6th, 2023

Assume the file is a sequence of words (group of letters) separated by whitespace.

i) If the first letter is a constant, move it to the right of the word & then add "ay"

ii) If the first letter is vowel, just add "ay" to the end of the word.

```
%}
%%
%}
```

Date .....

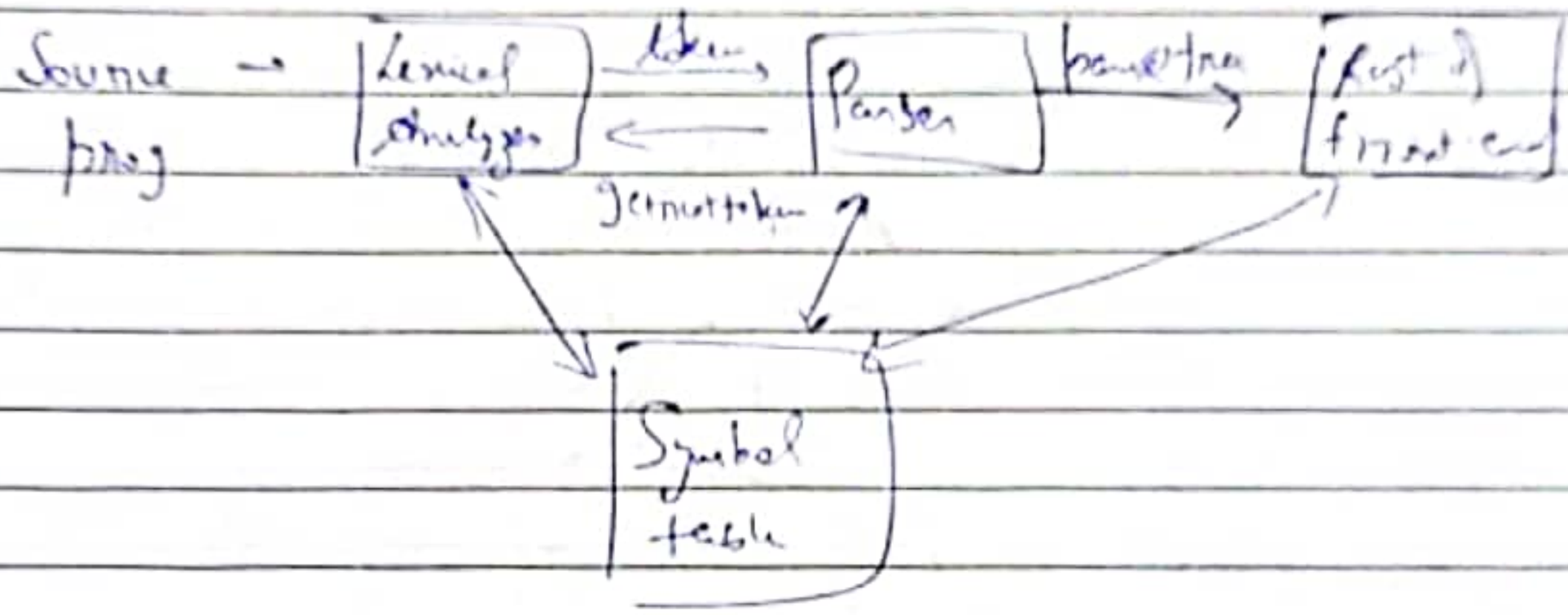
```
delim       [\t ]
ws          {delim}+
letter      [a-zA-Z]
word        {letter}+
%%
{ws}        printf("%s", yytext);}
{word}      {if(starts_vowel(yytext))
              printf("%say", yytext);
            else
              printf("%s%c ay", yytext+1, yytext[0]);
            }
%%
int startS_vowel(char *s){
  switch(s[0]){
    Case 'a':
    Case 'e':
    Case 'o':
    Case 'i':
    Case 'u': return 1;
    default : Return 0;
  }
int main(){
  yylex();
  return 0;
}
```

```
int arr[] = {1,2,3}
for(int i=0; i<3; i++)
  printf("%d", arr[i]);
```

Spiral

Date .....

Syntax Analysis



$E \longrightarrow E + T \mid T$

$T \longrightarrow T * F \mid F$

$F \longrightarrow (E) \mid id$

$id \longrightarrow [letter][letter \mid digit]^*$

$E \longrightarrow E + E \mid E * E \mid - E \mid (E)$

$E \Rightarrow -E$ for $-(id + id)$

$-(E)$

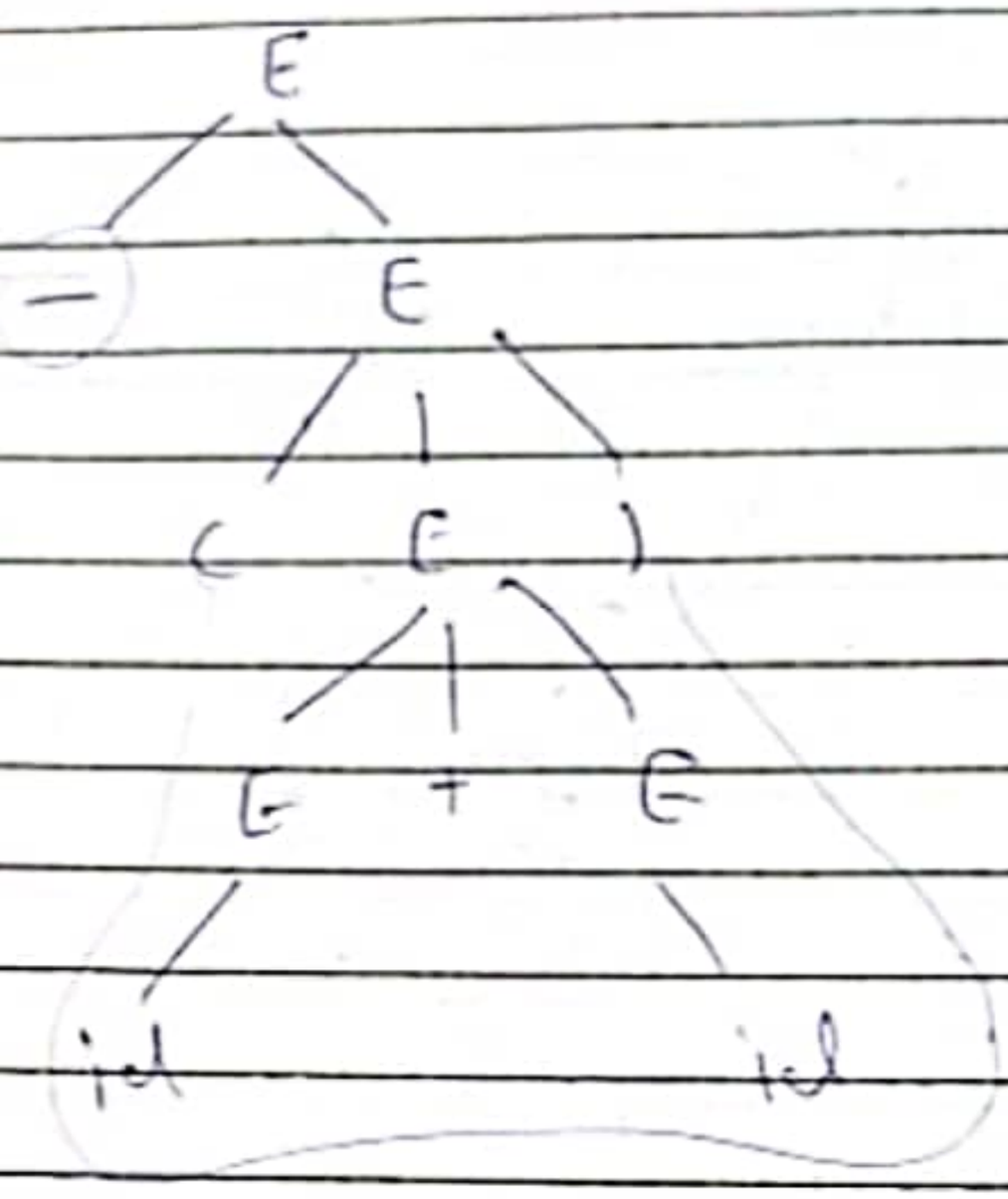
$-(E + E)$

$-(id + E)$

$-(id + id)$

Date .....

Parse Tree



Parser ⟶ bp - down

Bottom - up

T/D - Down Parser

① LL

Left to right

Leftmost Derivation

② LR → Left to right; Rightmost Derivation

LR(0), SLR(1), LALR(1), CLR(1) Spiral

Date .....

Parsing Algorithm

↓

Parsing Table

↓

Parse a Expression

$S \rightarrow SS+ \mid SS* \mid a$

Leftmost derivation of aaa+a*

$S \Rightarrow SS*$
$\Rightarrow SS+S*$
$\Rightarrow aa+a*$

$SS*$
$SS+S*$
$aS+S*$
$aa+S*$
$aa+a*$

lm

$S \underset{lm}{\Rightarrow} SS*$

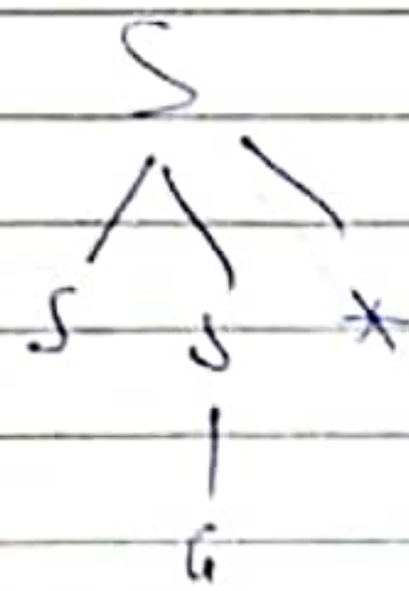
$\underset{lm}{\Rightarrow} SaS*$

$\underset{lm}{\Rightarrow} SS+aS*$

$\underset{lm}{\Rightarrow} Sa+aS*$

$\underset{lm}{\Rightarrow} aa+aS*$

$\underset{lm}{\Rightarrow} aa+a*$



Date .....

Q $S \longrightarrow S(S)S \mid \epsilon$

String (()())

$S \Rightarrow S(S)S$

$\Rightarrow S(S)S(S)S$

$\Rightarrow \epsilon(S)S(S)S$

$\Rightarrow \epsilon(\epsilon)S(S)S$

$\Rightarrow (\epsilon)(S)S$

$\Rightarrow (\epsilon)(\epsilon)$

$\Rightarrow (\,)(\,)$

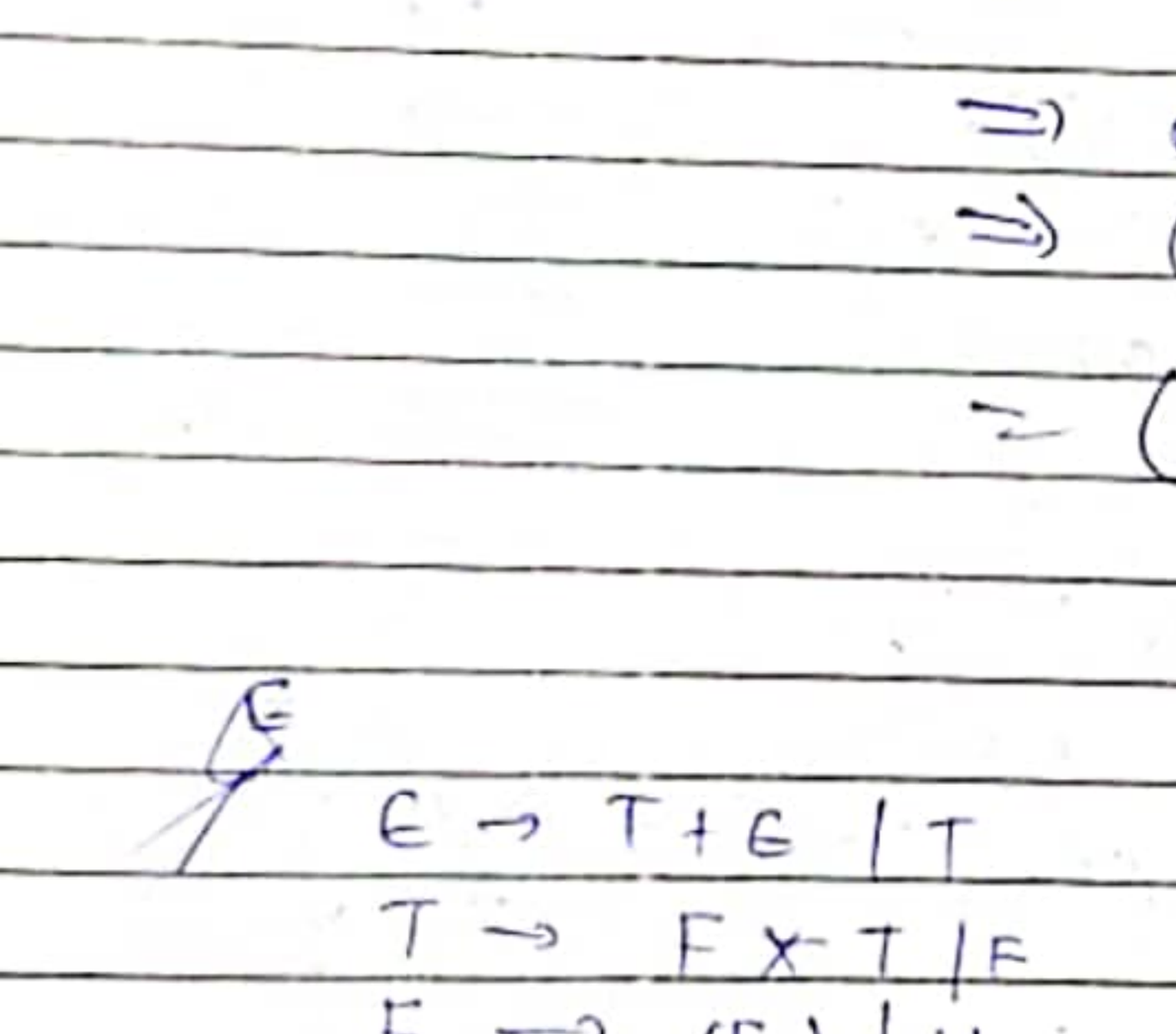
E

$E \rightarrow T+E \mid T$

$T \rightarrow F*T \mid F$

$F \rightarrow (E) \mid id$

id + id * id



Three versions of Expression Grammar

V1:

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

V2:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Spiral

Date .....

V3:

$E \longrightarrow TE'$

$E' \longrightarrow +TE' \mid \epsilon$

$T \longrightarrow FT'$

$T' \longrightarrow *FT' \mid \epsilon$

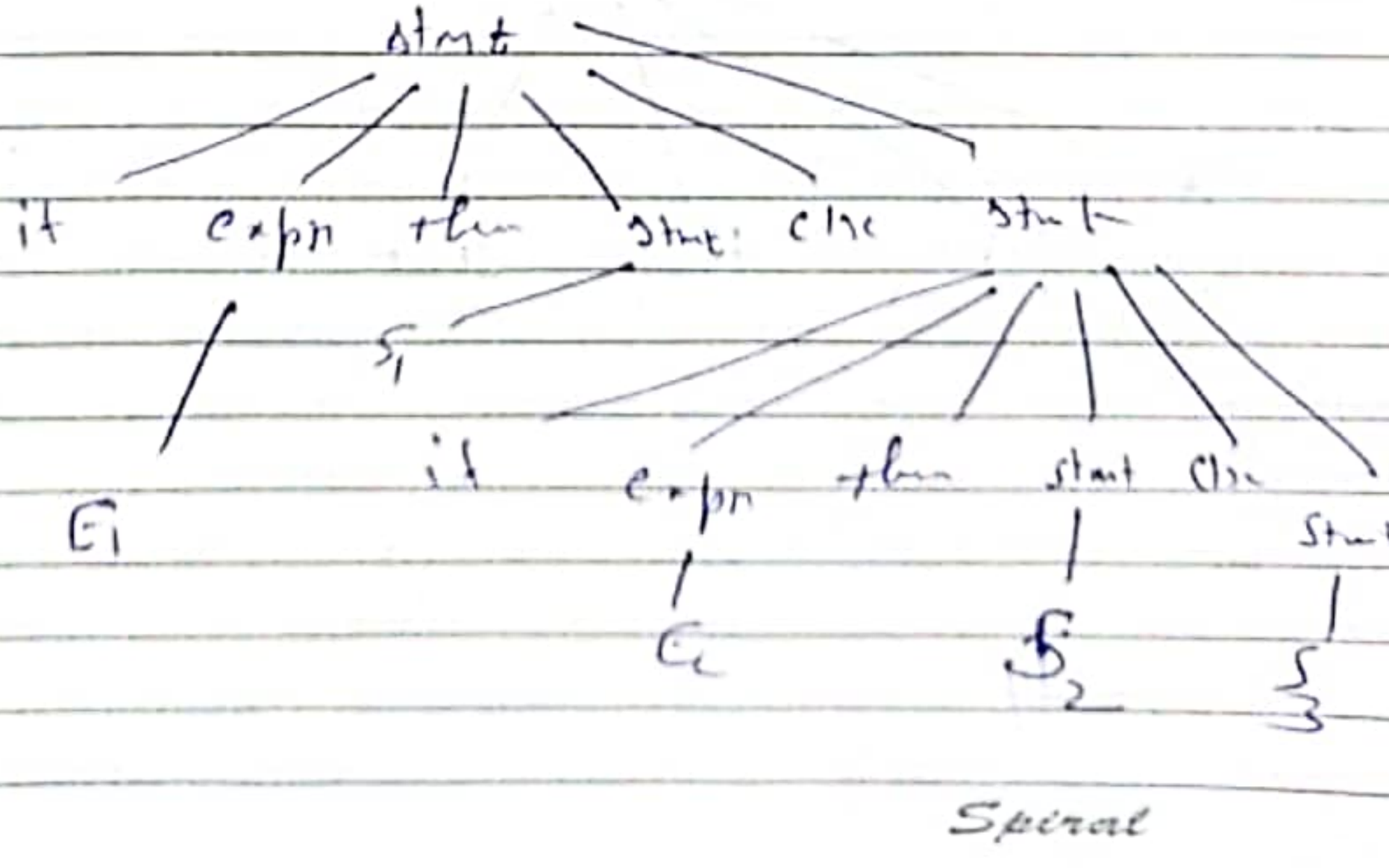
$F \longrightarrow (E) \mid id$

Parsing ek Grammar

~~Stmt~~ → ~~If Expr then S_1 else if~~
~~E_2 then S_2 else S_3~~

Stmt → if expr then Stmt |
if expn then stmt else Stmt |
other

O_3 → if E_1 then S_1 else if E_2 then S_2 else E_3



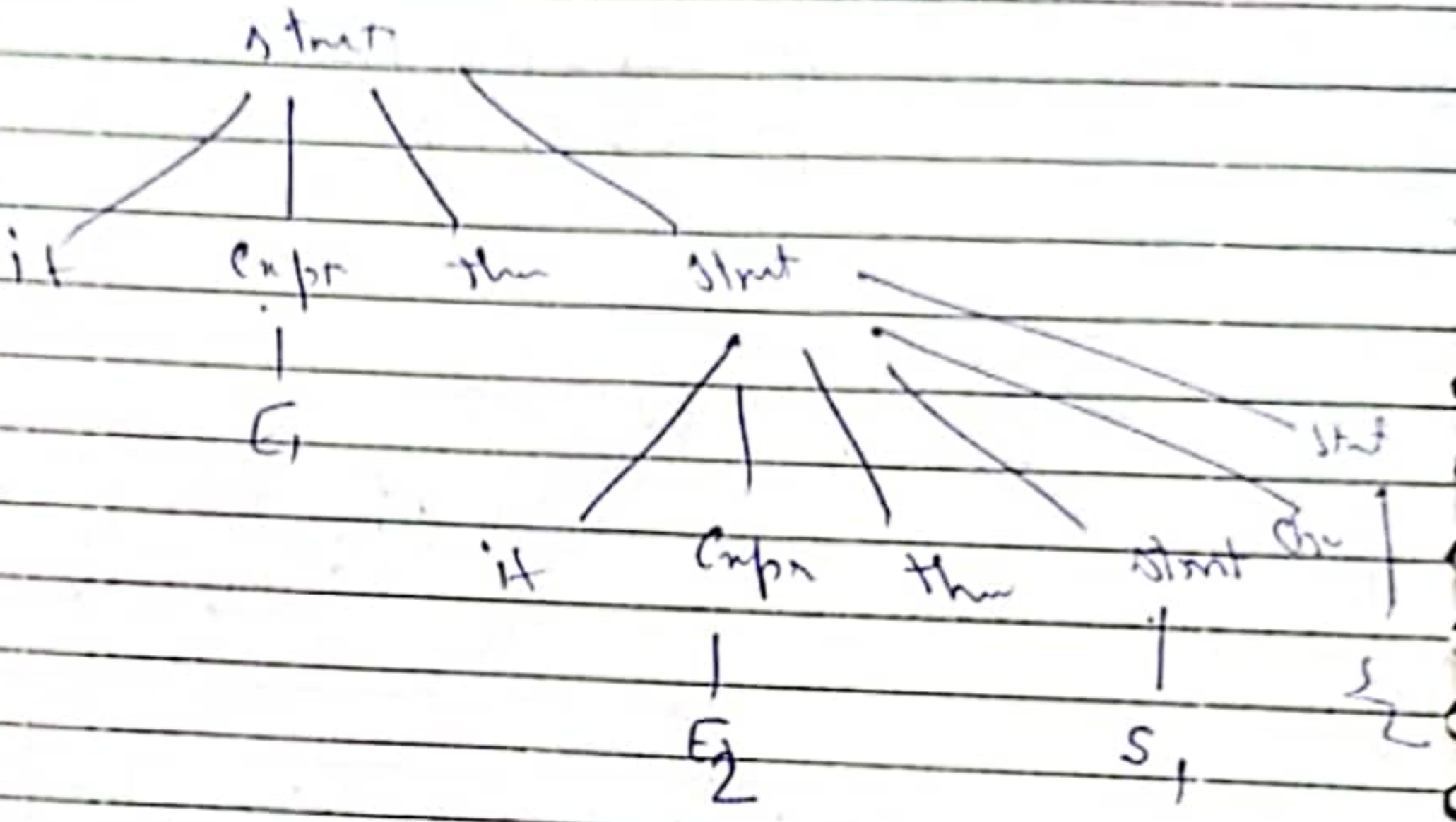
Spiral

Date .....

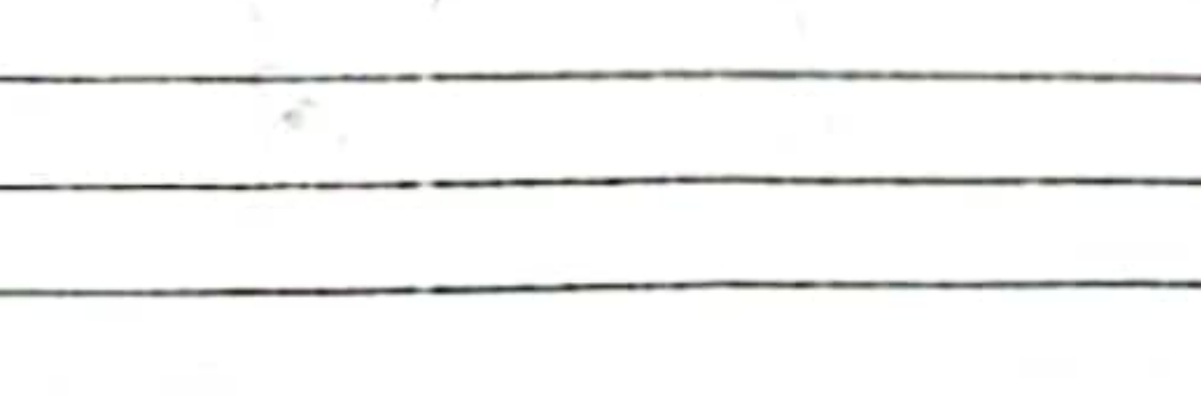
Q -

if E_1 then if G_2 then S_1 else S_2

(i)



(ii)



Spiral

Dangling else

Date .....
(Unambiguous)

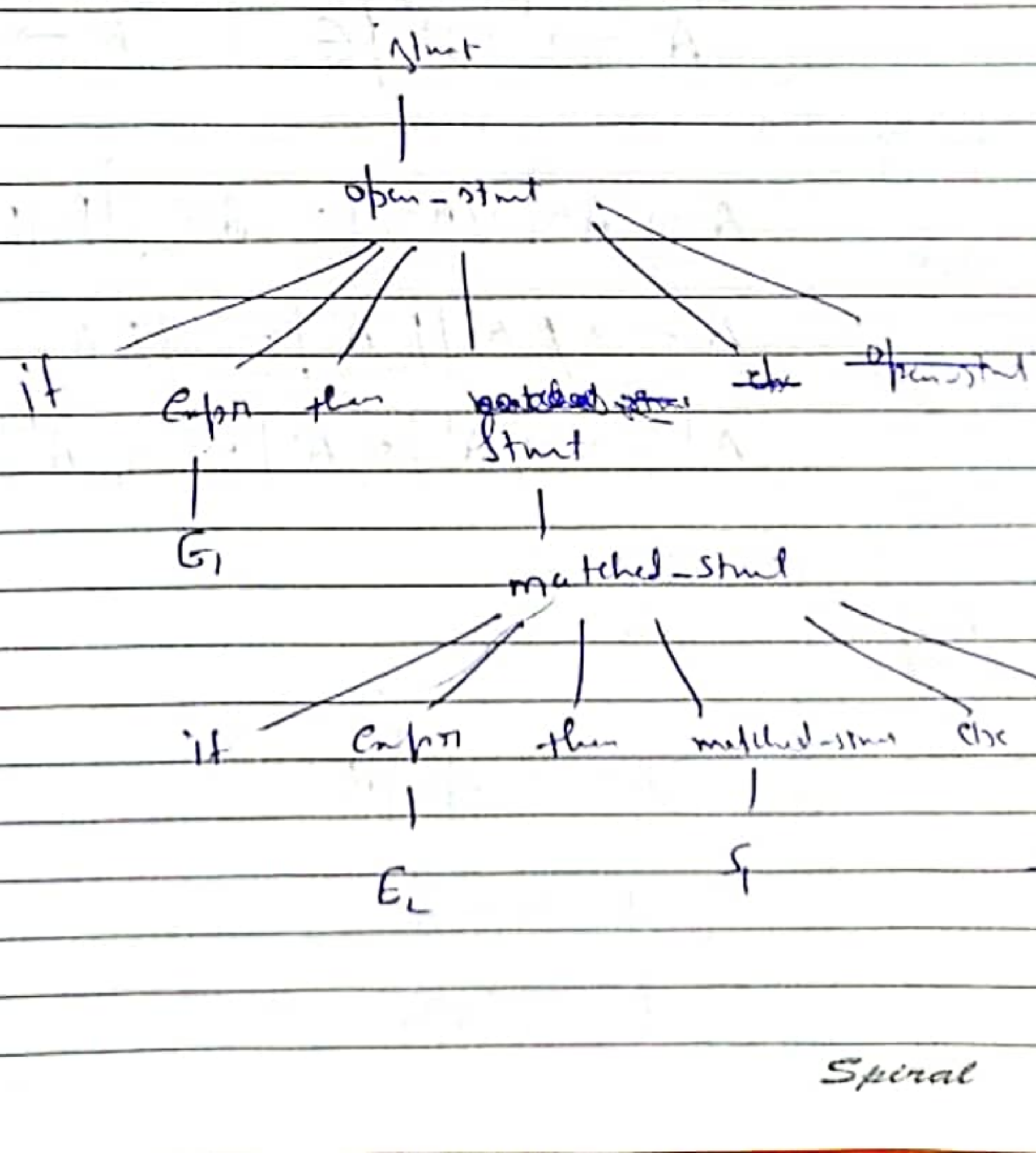
V2:

stmt → matched-stmt | open_stmt

matched-stmt → if expr then matched-stmt
else matched-stmt | other

open_stmt → if expr then stmt |
if expr then matched-stmt
else open_stmt

Q. - if G_1 then if G_2 then S_1 else S_2



Spiral

Date .....

Elimination of Left Recursion

A grammar is said to be left recursive

if $A \rightarrow A\alpha \mid \beta$

then A.I. is

$$A \rightarrow \beta A' \qquad A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon \qquad A' \rightarrow \alpha A' \mid \epsilon$$

In general

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots A\alpha_m \mid \beta_1 \mid \beta_2 \cdots \mid \beta_n$$

Now

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \alpha_m A' \mid \epsilon$$

Left Factoring

i) $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

Sol)

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

Spiral

$A \to \alpha A_1 | \alpha A_2 | \alpha A_3$

Date ....

Using this left factored grammar, do top-down parsing

Note: A left recursive grammar can cause a (recursive-descent) top-down parser, even one with backtracking, to go into an infinite loop.

Q → FIRST of each non-terminal

$S \to ABCDE$

$A \to a | \epsilon$

$B \to b | \epsilon$

$C \to c$

$D \to d | \epsilon$

$E \to e | \epsilon$

FIRST(E) = {e, ε}

FIRST(D) = ~~{d, ε}~~ {d, ε}

FIRST(C) = {c}

FIRST(B) = {b, ε}

FIRST(A) = {a, ε}

FIRST(S) = {a} ∪ FIRST(BCDE)

= {a} ∪ {b} ∪ FIRST(CDE)

= {a} ∪ {b} ∪ {c}

= {a, b, c}

Spiral

Date feb 13th 2023

① Expression grammar

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

	FIRST(·)	Follow(·)
E	FIRST(TE') = {(, id}	{$} ∪ FIRST(') = {$,)}
E'	{+, ϵ}	{$,)}
T	FIRST(FT') = {(, id}	FIRST(E') = {+} ∪ follow(E')
T'	{*, ϵ}	= {+, $,)}
F	{(, id}	{+, $,)}
		{*} ∪ Follow(T) = {*, +, $}

②

$S \rightarrow ABCDE$

$A \rightarrow a \mid \epsilon$

$B \rightarrow b \mid \epsilon$

$C \rightarrow c$

$D \rightarrow d \mid \epsilon$

$E \rightarrow e \mid \epsilon$

	FIRST(·)	Follow(·)
S	{a, b, c}	{$}
A	{a, ϵ}	{b, c} = FIRST(BCDE) ∪ FIRST(CDE)
B	{b, ϵ}	FIRST(CDE) = {c}
C	{c}	{d, e, $} = FIRST(DE) ∪ FIRST(ϵ) ∪ Follow(S)
D	{d, ϵ}	{e, $} = FIRST(E) ∪ Follow(S)
E	{e, ϵ}	{$} = Follow(S)

Spiral

Date .....

Q 31

$S \longrightarrow ACB \mid CbB \mid Ba$

$A \longrightarrow da \mid BC$

$B \longrightarrow g \mid \epsilon$

$C \longrightarrow h \mid \epsilon$

	FIRST(.)	Follow(.)
S	{d, g, h, a, b, ε}	{$}
A	{d, g, h, ε}	FIRST(CB) U FIRST(a) U follow(S) = {h, b, $}
B	{g, ε}	FIRST(a) U follow(S) = {a, b, h, $}
C	{h, ε}	{b, g, $, h}

FIRST(S) = FIRST(ACB) U FIRST(CbB) U FIRST(Ba)

= FIRST(A - {ε}) U FIRST(C - {ε})

U FIRST(B - {ε}) U {ε} U FIRST(C - {ε})

U {b} U FIRST(B - {ε}) U {a}

= {d, g, h, a, b, ε}

LL(1) grammar

→ Predictive parser or Recursive Descent Parser

→ No left recursive or ambiguous grammar can be LL(1).

A grammar can be in LL(1) iff whenever $A \longrightarrow \alpha \mid \beta$ are two distinct productions of the grammar then following condition hold:

Date .....

1) For no terminal 'a' do both α & β derive Strings beginning with a.

2) Atmost one of α & β can derive the empty string.

3) If $\beta \Rightarrow^{*} \epsilon$, then α doesn't derive any string beginning with a terminal in Follow(A). Likewise if $\alpha \Rightarrow^{*} \epsilon$ then β doesn't derive any string beginning with a terminal in Follow(A).

$$A \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_n$$

If any FIRST(α_1), FIRST(α_2) ... FIRST(α_n) contains same element / terminal then the grammar is not in LL(1).

*) $A \rightarrow \alpha | \epsilon$

if FIRST(α) $\cap$ FOLLOW(A) $\neq \phi$ the grammar is not in LL(1)

1) $S \rightarrow (L) | a$

$L \rightarrow SL'$

$L' \rightarrow \epsilon | ,SL$

	FIRST( )	follow( )
S	{ (, a}	{ $, , ,) }
L	{ (, a}	{) }
L'	{ ϵ, , }	{) }

Spiral

Date .....

LL(1) Parsing Table

NT	T	a	(	)	,	$
S		$S \to a$	$S \to (L)$			
L		$L \to SL'$	$L \to SL'$			
L'				$L' \to \epsilon$	$L' \to SL'$	

Since there is no conflict, hence it is in LL(1)

2) $S \longrightarrow aSbS \mid bSaS \mid \epsilon$

$FIRST(S) = \{a, b, \epsilon\}$

$FOLLOW(S) = \{b, a, \$\}$

	a	b	$
S	$S \to aSbS$ $S \to \epsilon$	$S \to bSaS$ $S \to \epsilon$	$S \to \epsilon$

Since there is a conflict, hence it is not in LL(1).

3) $S \longrightarrow AaAb \mid BbBa$

$A \longrightarrow \epsilon$

$B \longrightarrow \epsilon$

$FIRST(A) = \{\epsilon\}$

$FIRST(B) = \{\epsilon\}$

$FIRST(S) = \{a\} \cup \{b\}$

$= \{a, b\}$

Spiral

Date .....

Follow (A) = {a, b}
Follow (B) = {b, a}
Follow (S) = {$}

	a	b	$
S	S → AaAb	S → BbBa	
A	A → ϵ	A → ϵ	
B	B → ϵ	B → ϵ	

4) S → aSA | ϵ
A → c | ϵ

FIRST(S) = {a, ϵ}
FIRST(A) = {c, ϵ}

Follow(S) = {$, ϵ}
Follow(A) = {$, c}

	a	c	$
S	S → aSA	S → ϵ	S → ϵ
A		A → c A → ϵ	A → ϵ

conflict

Since there is a conflict, hence it is not in LL(1).

Spiral

Date .....

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

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

	FIRST(.)	FOLLOW(.)
S	{i, a}	{e, $}
S'	{e, ε}	{e, $}
E	{b}	{t}

NT \ T	Input Symbol i	t	a	b	e	$
S	$S \rightarrow iEtSS'$		$S \rightarrow a$			
S'.					$S' \rightarrow eS$ $S \rightarrow \epsilon$	$S \rightarrow \epsilon$
E				$E \rightarrow b$		

Fig 4.19

Model of a Table driven

Date .....

Expression Grammar :-

$E \rightarrow iE'$

$E' \rightarrow +TE' | \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \epsilon$

$F \rightarrow (E) | id$

Top - Down Parsing

Matched	Stack	Input	Action
	$E\$$	$id+id*id\$$	
	$TE'\$$	$id+id*id\$$	$E \rightarrow TE'$
	$FT'E'\$$	$id+id*id\$$	$T \rightarrow FT'$
	$idT'E'\$$	$id+id*id\$$	$F \rightarrow id$
id	$T'E'\$$	$+id*id\$$	match id
	$E'\$$	$+id*id\$$	$T' \rightarrow \epsilon$
	$+TE'\$$	$+id*id\$$	$E' \rightarrow +TE'$
$id+$	$TE'\$$	$id*id\$$	match +
	$FT'E'\$$	$id*id\$$	$T \rightarrow FT'$
	$idT'E'\$$	$id*id\$$	$F \rightarrow id$
$id+id$	$T'E'\$$	$*id\$$	match id
	$*FT'E'\$$	$*id\$$	$T' \rightarrow FT'$
$id+id*$	$FT'E'\$$	$id\$$	match *
	$idT'E'\$$	$id\$$	$F \rightarrow id$
$id+id*id$	$T'E'\$$	$\$$	match id
	$E'\$$	$\$$	$T' \rightarrow \epsilon$
	$\$$	$\$$	$E' \rightarrow \epsilon$

Spiral

Date .....

Bottom - Up Parsing (LR- parser)

Left to Right — Rightmost Derivation in reverse

LR

LR(0) | SLR(1) | LALR(1) | CLR(1)

↓

used by ~~YACC~~ YACC

* Right Sentential form

R.S-F	Handle	Reducing Production
$id_1 * id_2$	id_1	$F \to id$
$F * id_2$	F	$T \to F$
$T * id_2$	id_2	$F \to id$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$

id * id

E ⇒ T

⇒ T * E

⇒ T * F

⇒ T * id

⇒ F * id

⇒ id * id

* Shift Reduce Parsing

* Stack holds grammar symbols & an input buffer holds rest of string to be parsed.

Spiral

Date .....

Stack	input	action
$	$id_1 * id_2 \$$	shift
id_1 \$	$* id_2 \$$	reduce by $F \to id$
F \$	$* id_2 \$$	reduce by $T \to F$
T \$	$* id_2 \$$	shift
* T \$	$id_2 \$$	shift
id_2 * T \$	\$	reduce by $F \to id$
F * T \$	\$	reduce by $T \to T * F$
T \$	\$	reduce $E \to T$
E \$	\$	Accept

Stack → \$, \$ id_1, \$F, \$T, \$T*, \$T* id_2, \$T*F, \$T, \$E →

4 Possible actions, shift-reduce parser can make
→ Shift
Reduce
Accept
Error / Reject

Conflicts during shift Reduce Parsing

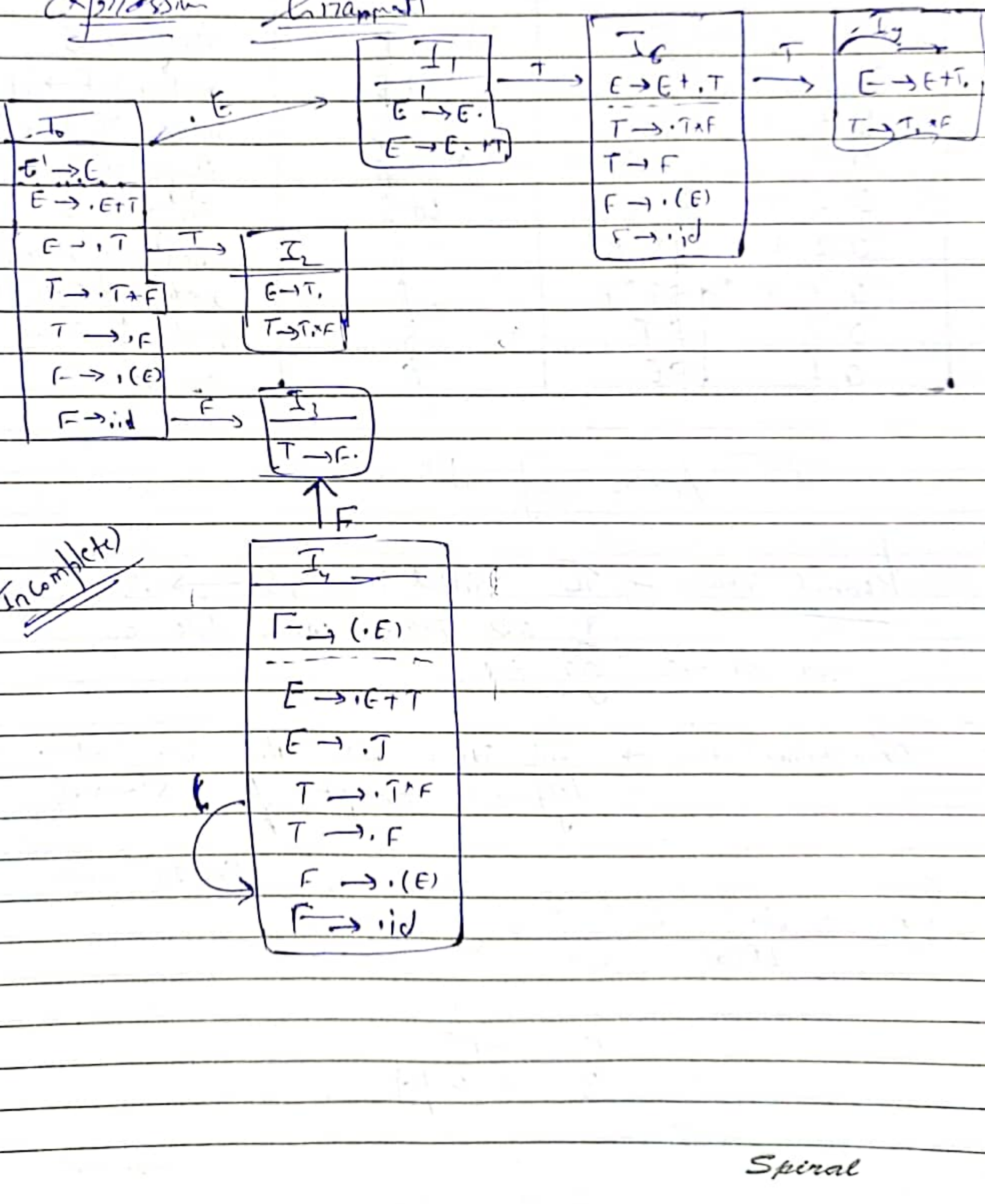
* Shift / Reduce Parser Conflict
* Reduce / Reduce Parser Conflict

Spiral

Date feb 20th 2025

1- Items & LR(0) Automaton (fig)

Expression Grammar



(Incomplete)

Spiral

Date .....

Stack		Input	Action
state	Grammar Symbol		
⓪	$	id * id $	shift to 5
0 5	$id	* id $	reduce by F → id
0 3	$F	* id $	reduce by T → F
0 2	$T	* id $	shift to 7
0 2 7	$T*	id $	shift to 5
0 2 7 5	$T*id	$	reduce by F → id
0 2 7 10	$T*F	$	reduce by T → T*F
0 2	$T	$	reduce by E → T
0 1	$E	$	accept

The parse of id * id

Kernel items → The initial items, $S' \longrightarrow .S$ & all items whose dots are not at the left end.

Non-kernel items → All items with their dots at left-end except for $S' \longrightarrow .S$

feb 23rd 2023

Ques → Check whether the grammar is in LR(0) or not.

$$E \rightarrow BB \quad (1)$$
$$B \rightarrow cB \ (2) \mid d \ (3)$$

Spiral

Date .....

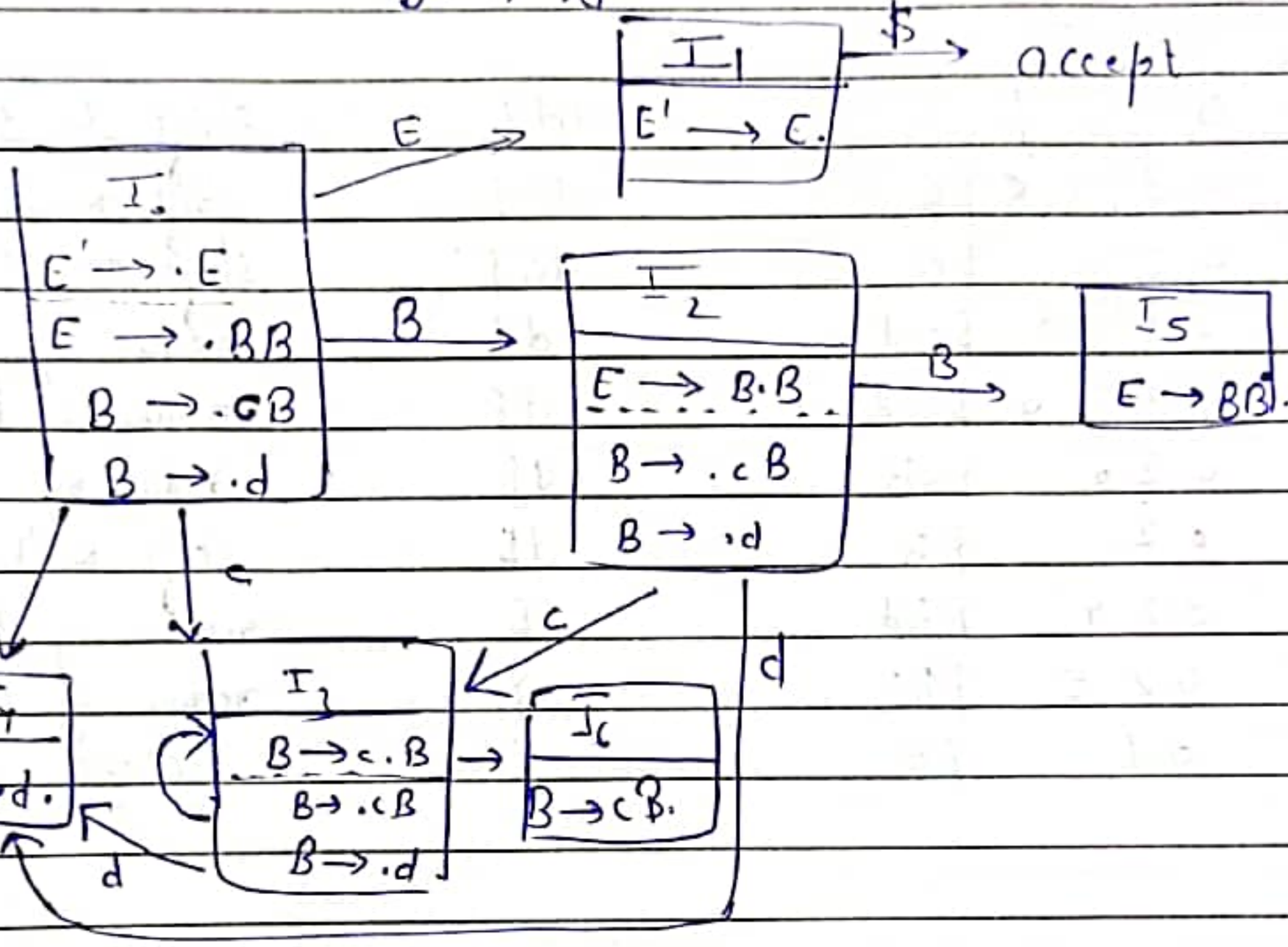
Ex:- Augmented Grammar,

$E' \longrightarrow .E$

$E \longrightarrow .BB$

$B \longrightarrow .cB$

$B \longrightarrow .d$



State	Action			GoTo	
	c	d	$	E	B
0	s3	s4		1	2
1			Accp		
2	s3	s4			5
3	s3	s4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

Parsing Table

Spiral

Date .....

for string "cddd"

stack	Input	Actim
state Symbol		
0 $	cccdd$	shift to 3
0 3 @ $c	cddd$	shift to 3
0 3 3 $cc	ddd$	shift to 4
0 3 3 4 $ccd	d$	reduce by B → d
0 3 3 6 $ccB	d$	reduce by B → cB
0 3 6 $cB	d$	reduce by B → cB
0 2 $B	d$	shift to 4
0 2 4 $Bd	$	reduce by B → d
0 2 5 $BB	$	reduce by E → BB
0 1 $E	$	accept

Q → E → BB
B → cB | d

	FIRST(·)	Follow(·)
E	{c, d}	{$}
B	{c, d}	{c, d, $}

Spiral

Date .....

LR(1) Parsing Table

State	Action			Goto	
	c	d	$	E	B
0	s_3	s_4		1	2
1			Acc		
2	s_3	s_4			5
3	s_3	s_4			6
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

$$LR(1) = \overset{\epsilon}{\$}LR(1)\ \overset{\rho}{\$}$$

Automaton for

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

Augmented Grammar

$S' \rightarrow .S$

$S \rightarrow .AaAb$

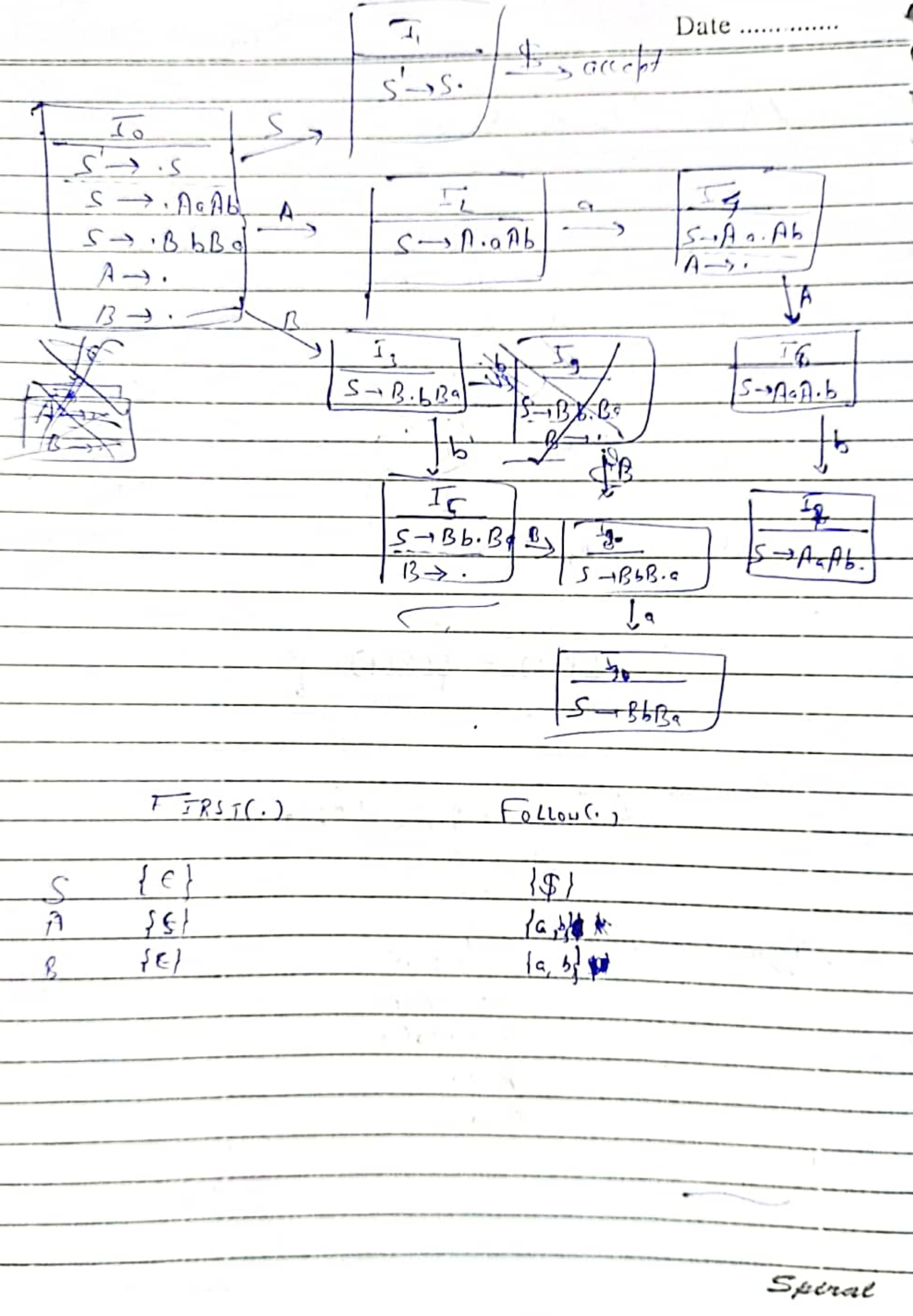
$S \rightarrow .BbBa$

$A \rightarrow .$

$B \rightarrow .$

Spiral

Date .....



	FIRST(.)	FOLLOW(.)
S	{ε}	{$}
A	{a}	{a, b}
B	{ε}	{a, b}

Spiral

Date .....

State	Action			GOTO		
	a	b	$	S	A	B
0				1	2	3
1			acc			
2	s4					
3		s5				
4					6	
5						7
6		s8				
7		s9				
8	~~r2~~		r2			
9			r1			

(i) Viable prefixes → The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes

(ii) → Expression Grammar

P — 25

Stack	Input	Action
$	id + id $	Shift id
$id	+ id $	Reduce by E → id
$F	+ id $	Reduce by T → F
$T	+ id $	shift +
$T+	id $	shift id
$T+id	$	Reduce by F → id
$T+F	$	Reduce by T → T+F
$E	$	Reduce by E → E+T
		accept

Spiral

Date .....

More Powerful LR Parser

1) "Canonical LR" or just "LR" or CLR(1) or LR(1)

2) "Look Ahead LR" or LALR

Both uses LR(1) items = LR(0) item + Lookahead
(,)

LR(0) → Put the reduce in full row
SLR(1) → Put the reduce in follow row
CLR(1) → " " " " lookahead column
& LALR(1)

Q →
$$E \rightarrow BB$$
$$B \rightarrow cB$$
$$B \rightarrow d$$

Check whether the grammar is in LALR or not

$$E' \rightarrow .E, \$$$
$$E \rightarrow .BB, \$$$
$$B \rightarrow cB \mid d, \quad c \mid d$$

Case 1 → if $A \rightarrow \alpha\beta B, c$

Date Feb 27th, 23

Q → check whether the following grammar is in LALR(1) or not.

$E \longrightarrow BB$

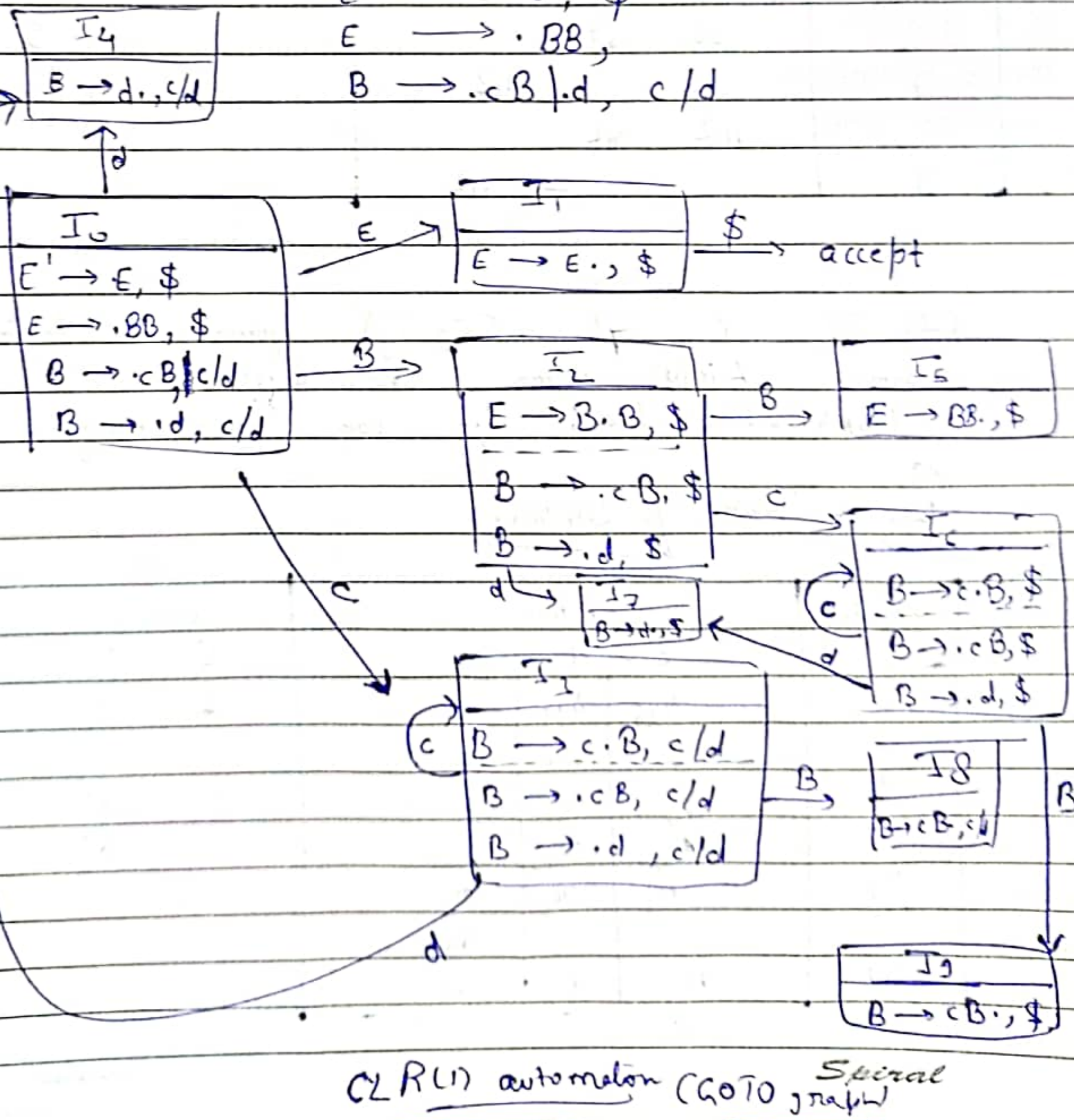
$B \longrightarrow cB \mid d$

Augmented Grammar

$E' \longrightarrow .E, \$$

$E \longrightarrow .BB, \$$

$B \longrightarrow .cB \mid .d, \ c/d$



CLR(1) automaton (GOTO graph) Spiral

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

Date .....

Parsing table ~~CLR(1)~~ CLR(1)

State	Action			GoTo	
	c	d	$	E	B
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5	~~r1~~		r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

(I_3, I_6), (I_4, I_7), (I_8, I_9) pairs are having same LR(0) items but differ in lookahead. Hence they can be merged.

Parsing Table for LALR(1)

State	Action			GoTo	
	c	d	$	E	B
0	s_{36}	s_{47}		1	2
1			acc		
2	s_{36}	s_{47}			5
36	s_{36}	s_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Spiral

Date .....

Ex 4.7.5 →

$S \longrightarrow Aa \mid bAc \mid Bc \mid bBa$

$A \longrightarrow d$

$B \longrightarrow d$

is in LR(1) but not LALR(1).

Ans → LR Automaton →

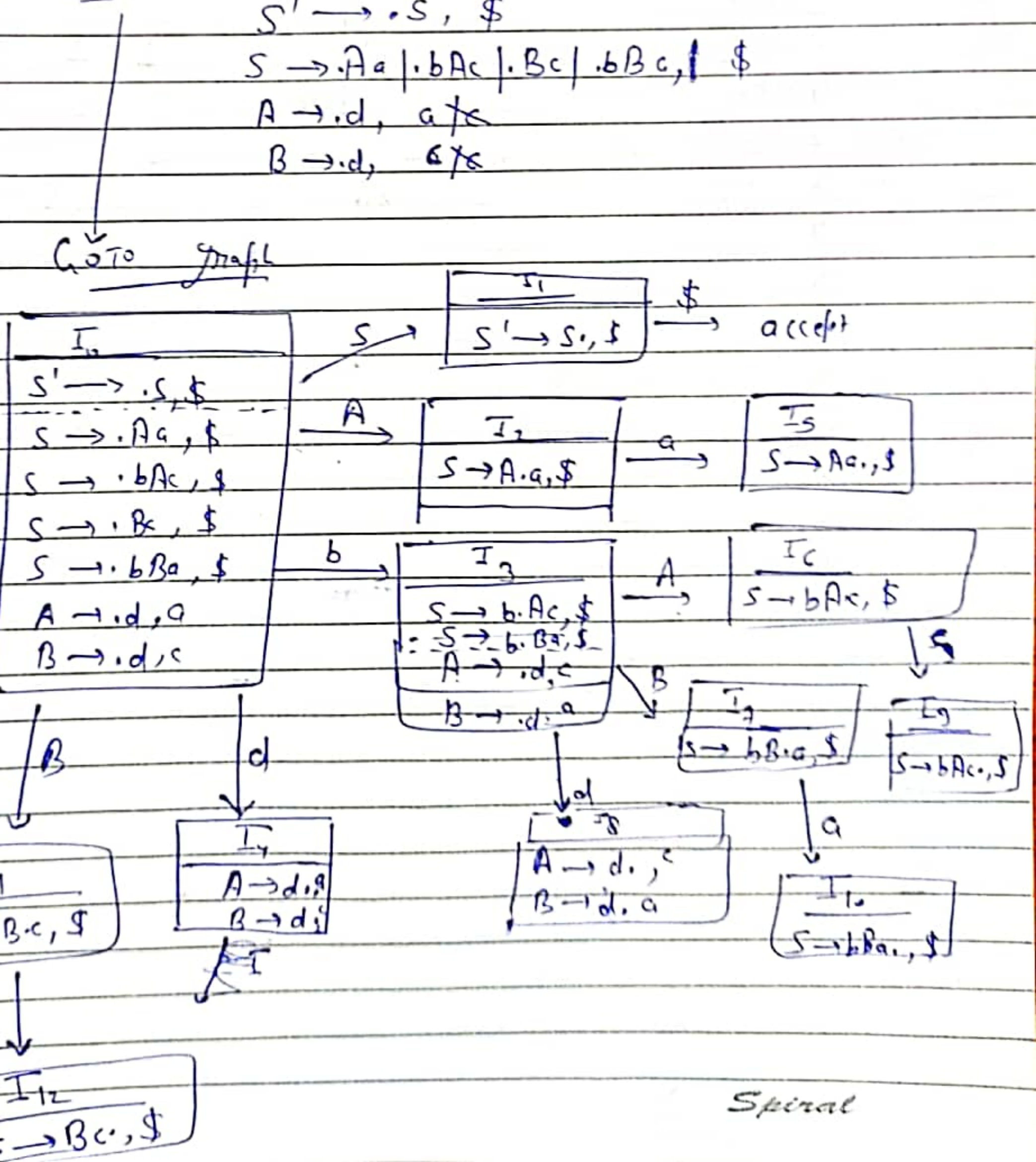
$S' \longrightarrow .S, \$$

$S \longrightarrow .Aa \mid .bAc \mid .Bc \mid .bBa, \$$

$A \longrightarrow .d, a$

$B \longrightarrow .d, c$

GOTO Graph



Spiral

Date .....

Parsing Table

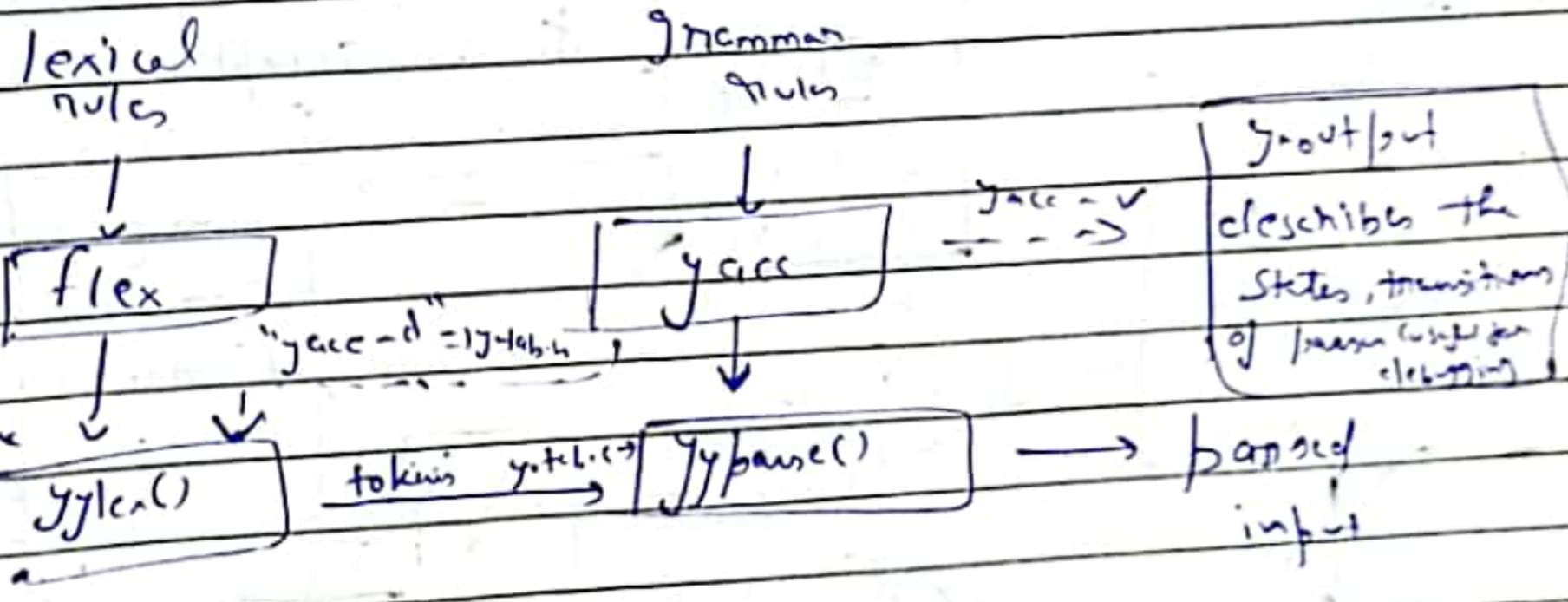
State	Action					GOTO		
	a	b	c	d	$	S	A	B
0		s3		s4		1	2	11
1					acc.			
2	s5							
3				s8			6	7
4								
5								
6								
7								
8								
9								
10								
11								
12								

Verbatim → as it is
(word to word)

Date .....

*) Yacc → Yet Another Compiler Compiler
*) package → bison
*) Parser generator

Input → a set of grammar rules & actions → [yacc (or bison)] → Output: C code implementing a parser
func": yyparse()
file [default]: y.tab.c



Note - In windows bison -yvd options are used
bison -yv Enchant = all testing for clause abs.

yyparse → called once from main (user-supplied)
Repeatedly calls yylex() until done
*) on syntax error, calls yyerror().

Spiral

Mon 2nd, 2023

Date Feb 3.....

Using Ambiguous Grammars

Continuation

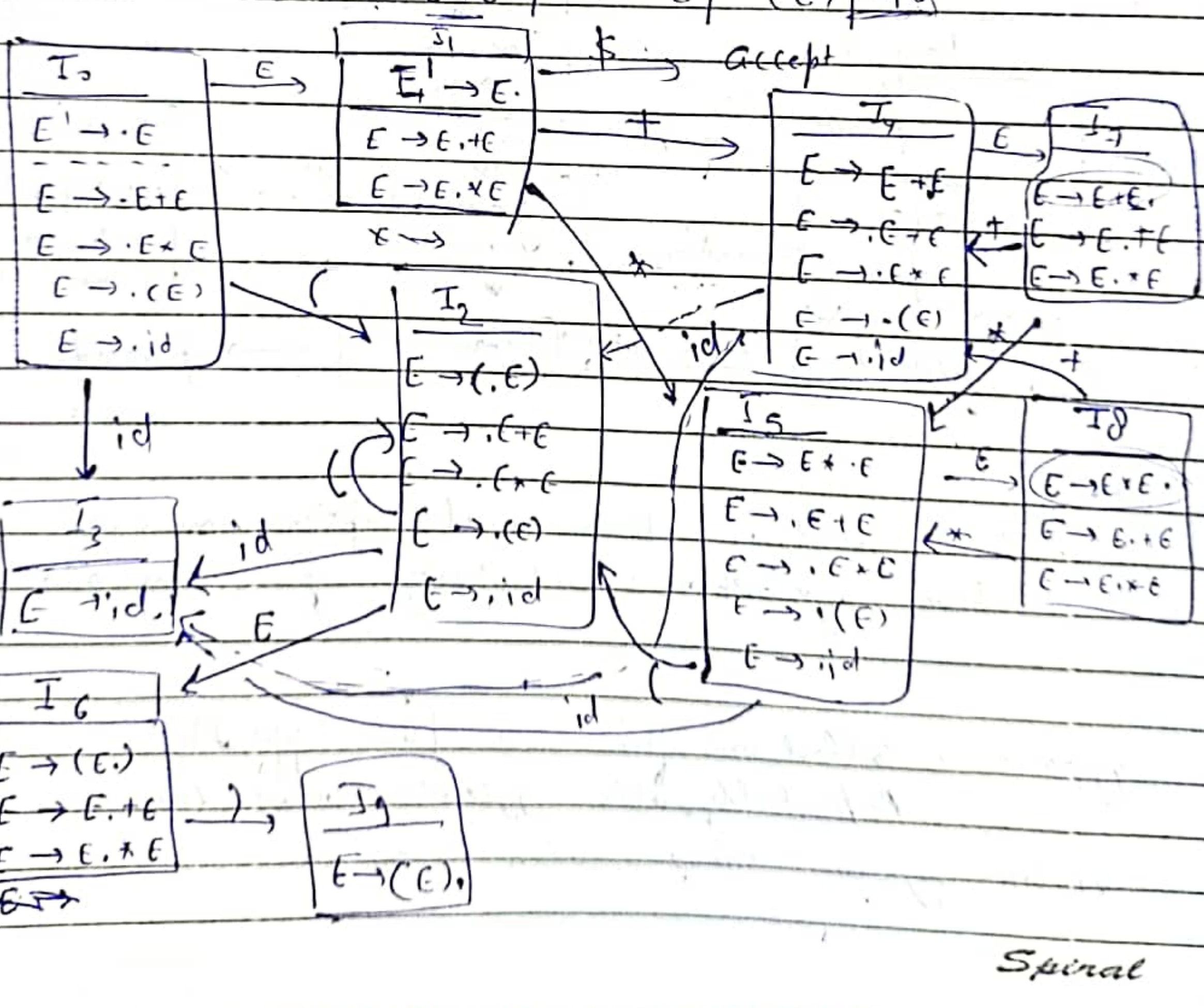
Precedence & Associativity to Resolve Conflicts

LR(0) Automata

Automaton for Expression Grammar V1

$E' \rightarrow \cdot E$

$E \rightarrow \overset{①}{\cdot} E+E \mid \overset{②}{\cdot} E*E \mid \cdot(E) \mid \overset{④}{\cdot} id$



Date .....

Parsing Table for SLR(1)

State	Action						Goto
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc.	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3	~~s6~~		s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1 / s4	r1 / s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

String → ① id + id * id
② id + id + id

Stack State	Stack Symbol	Input	Action
0	$	id+id*id$	shift to 3
03	$id	+id*id$	reduce by E → id
01	$E	+id*id$	shift to 4
014	$E+	id*id$	shift to 3
0143	$E+id	*id$	Reduce by E → id
0147	$E+E	*id$	shift to 5
01475	$E+E*	id$	shift 3
014753	$E+E*id	$	Reduce by E → id
01475 8	$E+E*E	$	Reduce by E → E*E
0147	$E+E	$	Reduce by E → E+E
01	$E	$	Accept

Date .....

hw

4.8.2 The "Dangling - Else" Ambiguity

$S' \rightarrow S$

$S \rightarrow iS$

4.8.3 - Errors Recovery in L-R parsing

Date March 17th, 2023

Chapter – 5

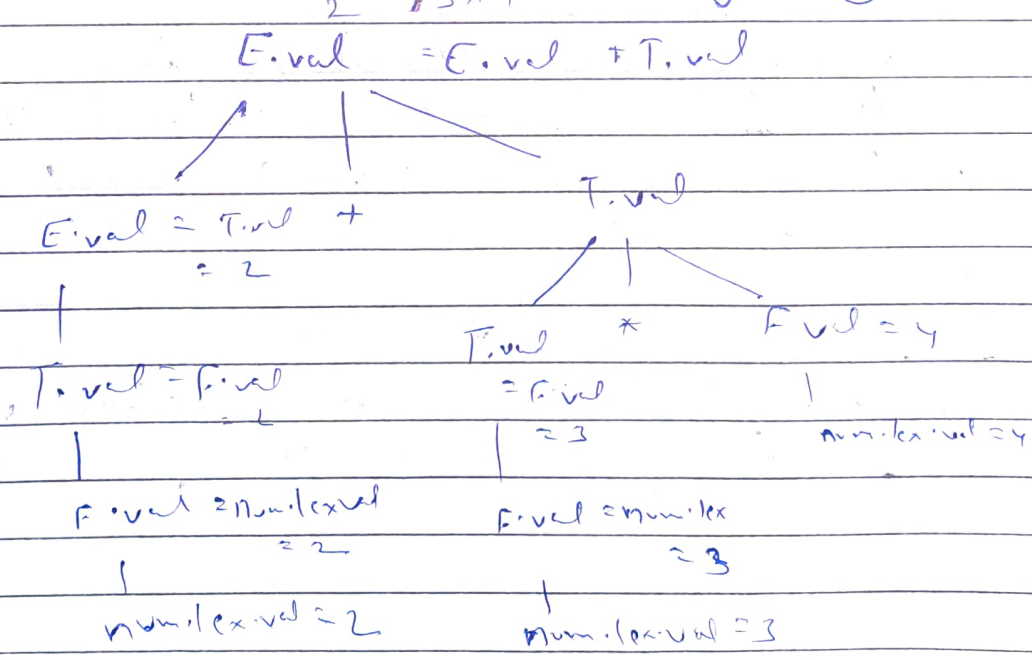
Syntax Directed Translation (SDT)

Grammar + Semantic Rules

$E \rightarrow E\ '+'\ E \quad \{\$\$ = \$1\ '+'\ \$3\}$

# $E \rightarrow E + T$	$\{E.val = E_1.val + T.val\}$	– (1)
$E \rightarrow T$	$\{E.val = T.val\}$	– (2)
$T \rightarrow T * F$	$\{T.val = T_1.val * F.val\}$	– (3)
$T \rightarrow F$	$\{T.val = F.val\}$	– (4)
$F \rightarrow num$	$\{F.val = num.lex.val\}$	– (5)

The attribute which propagates values to its parents which always take, it value from the child an called synthesized attribute



Spiral

Date .....

Syntax Directed Translation

Embeds program fragments called symmetric actions within production bodies. They are generally enclosed {}.

Aim → To make the translating machine efficient.

Attributes → are associated with grammar symbols (Terminal / Non-terminal) & rules are associated with the production.

X is a symbol & a is an attribute of X.
We write X.a
Then X.a denote value of a at a particular parse tree nodes.

Types of Attributes

A synthesized Attribute for a Non-terminal A at a parse tree Node N is defined by a semantic rule associated with the production at N.

A synthesized attributes at node N is defined only in terms of attribute values at the children of N & of N itself.

Date .....

H An inherited attribute of a Non-terminal at a parse tree Node N is defined by the semantic rule N at with the production at the parent of N.

An inherited attribute at Node N is defined only in terms of attributes values at N's parent, N itself, N's sibling.

Q → (newline)

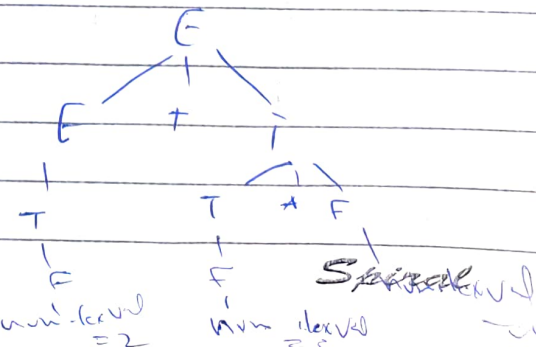
$L \rightarrow E_n$	{L.val = F.val}
$E \rightarrow E_1 + T$	{E.val = E_1.val + T.val}
$E \rightarrow T$	{E.val = T.val}
$T \rightarrow T_1 * F$	{T.val = T_1.val * F.val}
$T \rightarrow F$	{T.val = F.val}
$F \rightarrow (E)$	{F.val = E.val}
$F \rightarrow$ digit	{F.val = digit.lexval}

Ans 3 * 5 + 4n {fig 5.3}

Q →

$E \rightarrow E_1 + T$	{printf('+');} (1)
$\mid T$	{ } (2)
$T \rightarrow T * F$	{printf('*');} (3)
$\mid F$	{ } (4)
$F \rightarrow$ num	{printf(num.lexval);} (5)

2+3*4



In exam | Panic mode Recovery? | for exam | Identify the inh & syn attr in the semantic Rules & write the value at root & also the value at each node

Date March 20th, 2023

*) Dependency on the sibling will be always from left to right

*) DFS tree

Prod Rules

$T \rightarrow FT'$

$T' \rightarrow *FT_1'$

$T' \rightarrow \epsilon$

$F \rightarrow digit$

Semantic Rules

$T'.inh = F.val$

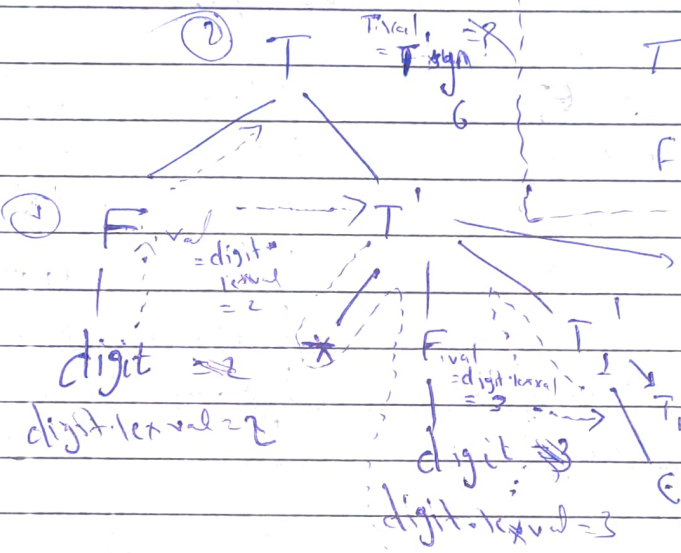
$T.val = T'.syn$

$T_1'.inh = T'.inh \times F.val$

$T'.syn = T_1'.syn$

$T'.syn = T'.inh$

$F.val = digit.lexval$



(1) F.val = synthesized

(2) T.val = synthesized

March 20, 2023

2:30 pm

2:45 pm →

$N \rightarrow L$

$L \rightarrow LB \mid B$

$B \rightarrow 0 \mid 1$

Spiral

Date .....

Rewriting the grammar as

$N \rightarrow L$ { N.count = L.count }

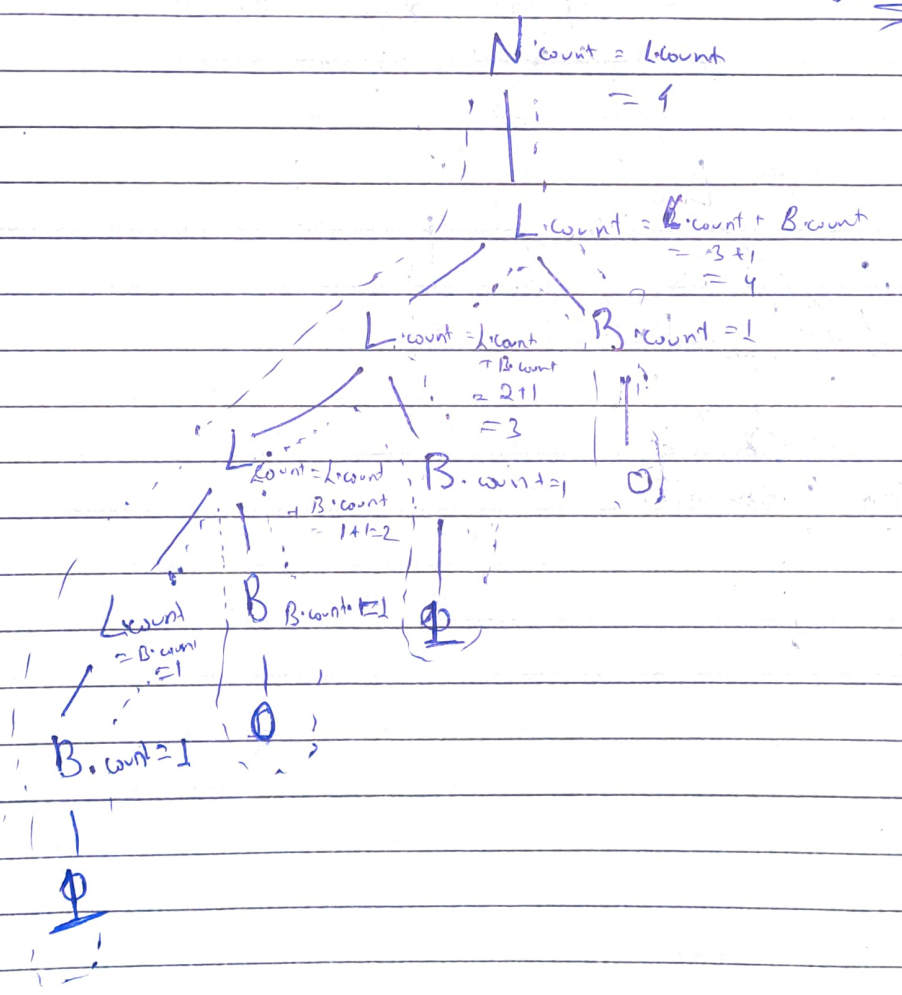
$L \rightarrow L\ B$ { L.count = L_1.count + B.count }

$\mid B$ { L.count = B.count }

$B \rightarrow 0$ { B.count = 1 }

$\mid 1$ { B.count = 1 }

for 1010



Spiral

1010

Binary to Decimal

Date .....

$N \rightarrow L$ { N.dval = L.dval ; N.count = L.count }

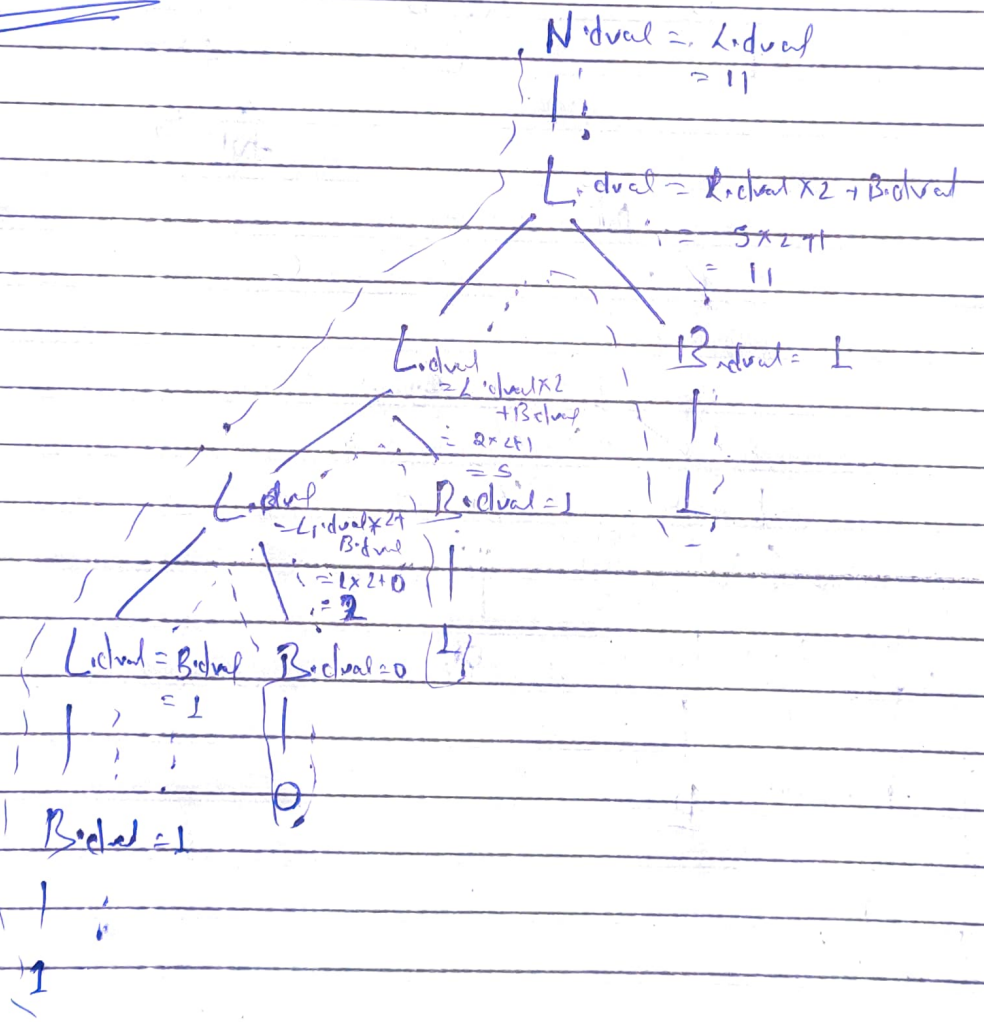
$L \rightarrow L_1 B$ { L.count = L_1.count + B.count ; L.dval = L_1.dval × 2 + B.dval }

| B

$B \rightarrow 0$ { B.count = 1 , B.dval = 0 }

| 1 { B.count = 1 , B.dval = 1 }

for 1011



Spiral

9.5
1001

dval = decimal value
Specify it explicitly

Binary → Decimal (float)

Date .....

$S \to L \cdot L \mid L$
$L \to LB \mid B$
$B \to 0 \mid 1$

S.count = L.count

Rewriting

$S \to L_1 \cdot L_2$ { S.count = L_1.count + L_2.count ; S.dval = L_1.dval + L_2.dval / $2^{L_2.count}$ }

$S \to L$ { S.count = L.count ; S.dval = L.dval }

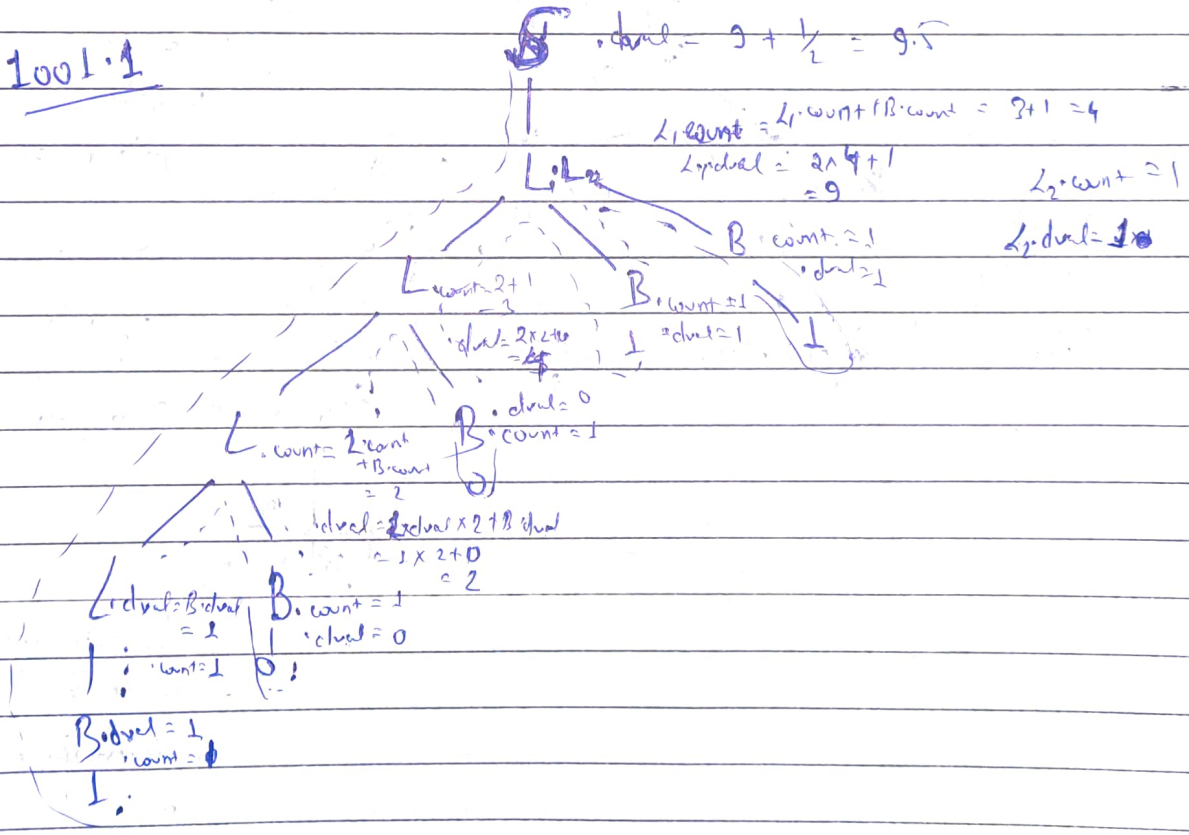
$L \to LB$ { L.dval = 2 × L_1.dval + B.dval ; L.count = L_1.count + B.count }

$\mid B$ { L.dval = B.dval ; L.count = B.count }

$B \to 0$ { B.dval = 0 ; B.count = 1 }

$\mid 1$ { B.dval = 1 ; B.count = 1 }

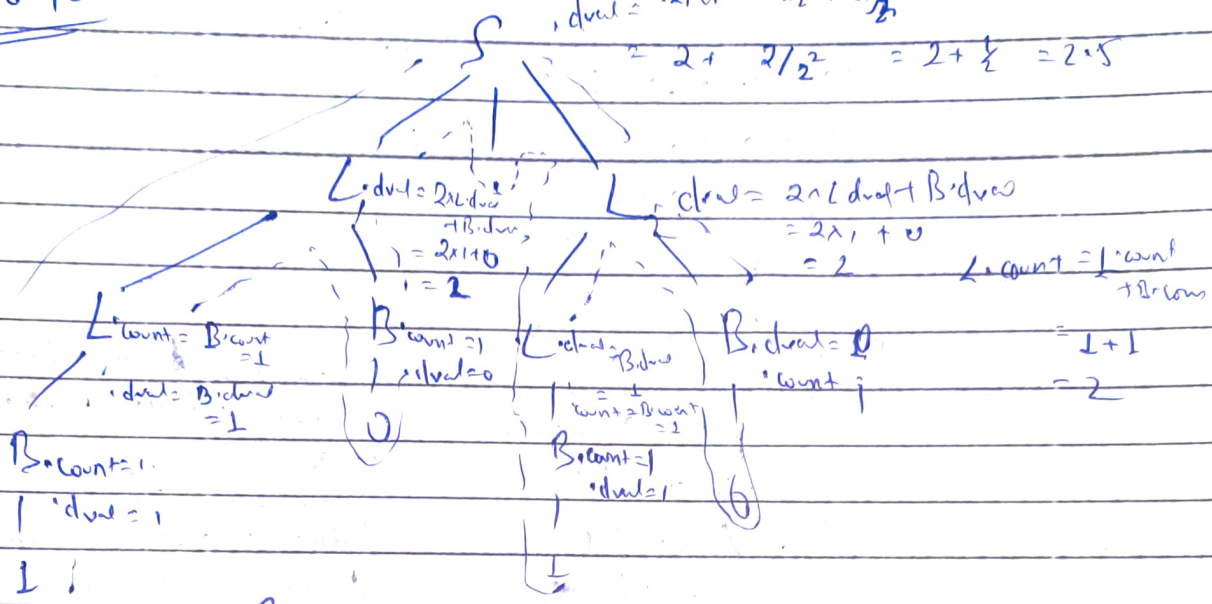
1001.1



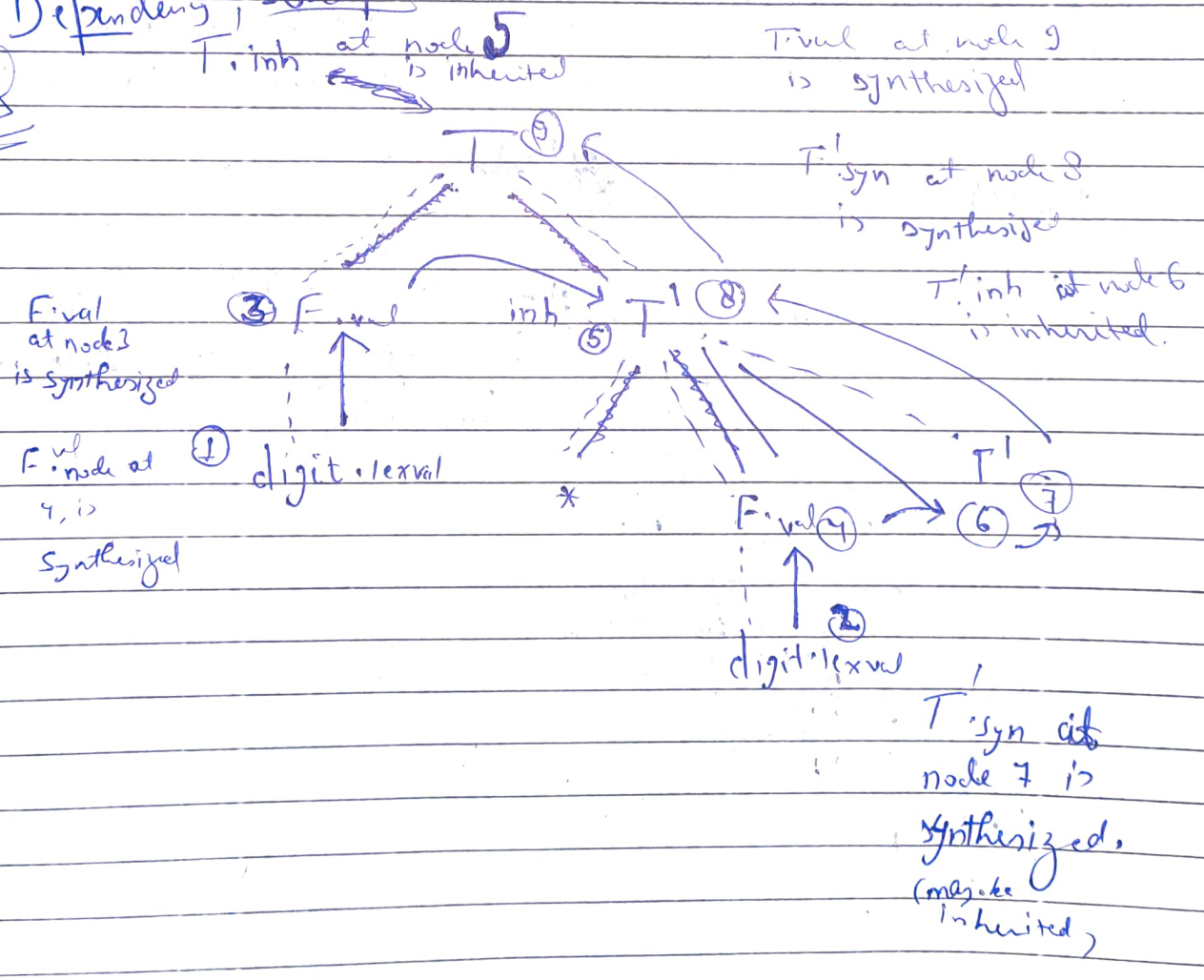
Special

Date .....

10.10



Dependency graph



Topological Sort

1, 3, 5, 2, 4, 6, 7, 8, 9,

Spiral

Date .....

Ex 5.2.4 → L-attributed SDD

$S \rightarrow L.L \mid L$

$L \rightarrow LB \mid B$

$B \rightarrow 0 \mid 1$

Chapter-6

Intermediate Code Generation

3-address Code

for $x + y * z$

$t_1 = y * z$

$t_2 = x + t_1$

t_1, t_2 are compiler-generated temporary names.

for evaluation of $a + a * (b - c) + (b - c) * d$

$t_1 = b - c$

$t_2 = a * t_1$

$t_4 = a + t_2$

$t_3 = t_1 * d$

$t_5 = t_3 + t_4$

a[i] · *(a+i*4)

Date .....

Three different ways to represent 3-address code
Quadruples
Triples
Indirect Triples

Three address instructions can of form :-

→ x = y op z
→ x = op y
→ x = y
→ Goto L
→ If x goto L & if false x goto M
→ if x relop y goto L
relop (<, ==, >=, etc)
→ Procedure call

→ Indexed copy instruction x = y[i] & x[i] = y
→
→ x = &y
x = *y
*x = y

* do i = i+1 ; while (a[i] < v) ;

L : t1 = i+1
i = t1
t2 = i*4
t3 = a[t2]
if t3 < v goto L

{if array element (int) takes 4 units of space}

Spiral

Date .....

Quadruples

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

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = t_2 + t_2$
$a = t_3$

} 3 address code

	op	arg1	arg2	result
0	minus	c	-	t_1
1	*	b	t_1	t_2
2	+	t_2	t_2	t_3
3	=	t_3	-	a

Triples

	op	arg1	arg2
0	minus	c	-
1	*	b	(0)
2	+	(1)	(1)
3	=	a	(2)

Indirect Triples

Instructions

35	(0)
36	(1)
37	(2)
38	(3)

	op	arg1	arg2
0	minus	c	-
1	*	b	(0)
2	+	(1)	(1)
3	=	a	(2)

Spiral

Unification

March 27th, 2023
Date .....

Types & Declarations

① Type checking → uses logical rule to reason about the behaviours of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operation

② Translation Application :- From the type of a name, a compiler can determine the storage that will be needed for that name

Type Expressions → int [2][3]

2 → Arrays of 3

array(2, array(3, integer))

* Type Equivalence

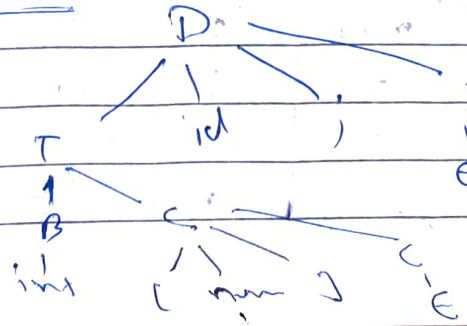
if two type expr. are equal then return a certain type else error.

Name Equivalent or Structurally Equivalent

Declarations

int [num] id ;

D → T id ; D | ∈
T → B C
B → int | float
C → ∈ | [num] C



Spiral

Type & Declaration

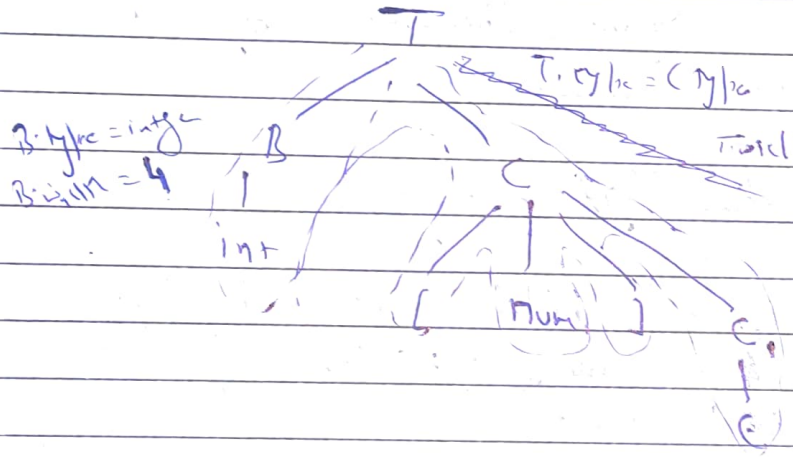
Date .....

Semantic Action

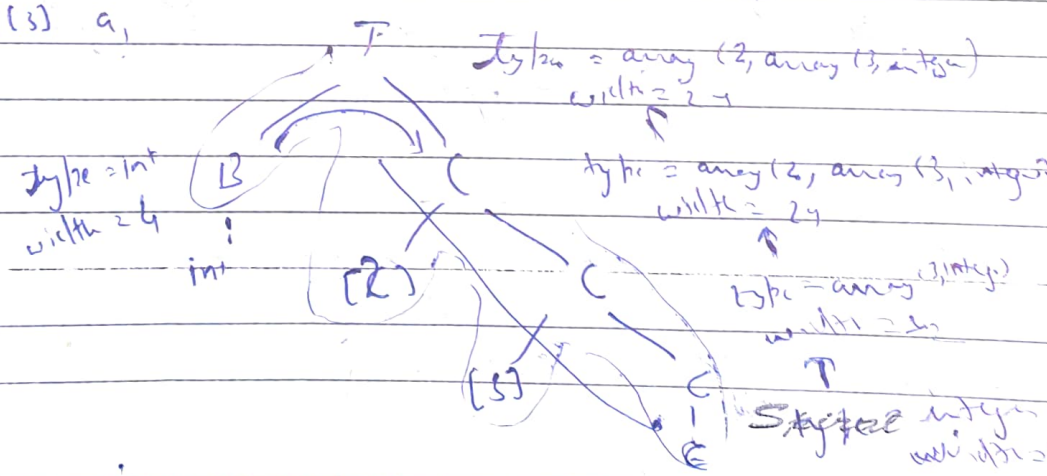
$T \rightarrow B$	$t = B.type$	$w = B.width$
C	$T.type = C.type$	$T.width = C.width$
$B \rightarrow int$	$B.type = integer$,	$B.width = 4$;
$B \rightarrow float$	$B.type = float$,	$B.width = 8$;
$C \rightarrow \epsilon$	$C.type = t$, $C.width = w$	
	$C.width = num.value \times C_1.width$	

int [num] a;

num = 2



int [2][3] a;



Date April 3rd, 2023

Type checking

It is used to catch errors in program. Each operation executed in a program respects the type system of the language. A sound type system eliminates the need for runtime checking for type errors.
PL is strongly typed if the compiler guarantees that the program it accepts will run without any type error.
The type checking can be done at static & dynamic.

Python is dynamically typed language. Python interpreter does type checking only when code is reached.

Type synthesis & Type Inference

Type Synthesis - It builds up the type of an whole expression from the types of it subexpression.

Type Inference → Determines the type of language construct from the way it is used.

Spiral

Date .....

* Compiler infers the missing type information based on contextual information

Type Conversions :- $2 * 3.14$

$t_1 = (float)\ 2$

$t_2 = t_1 * 3.14$

$E \longrightarrow E_1 + E_2 \quad \{ E.type = max(E_1.type, E_2.type);$

$a_1 = widen(E_1.addr, E_1.type, E.type)$

$a_2 = widen(E_2.addr, E_2.type, E.type)$

$gen(E.addr = a_1 + a_2); \}$

Control flow

* Alter the flow of control
* Logical values compute pro.

Ex → 6.2.1 → The if statement

if ($x < 100 \mid\mid x > 200 \;\&\&\; x != y$) $x = 0$;

if $x < 100$ goto L_2

ifFalse $x > 200$ goto L_1

ifFalse $x != y$ goto L_1

L_2: $x = 0$

L_1:

Spiral
Jumping Code

Date .....

Example 6.22 → Improvised version

```
if( x<100 || x>200 && x!=y) x=0;

    if (x<100) goto L1
        goto L2
L2: if x>200 goto L3
        goto L4
L3: if x!=y goto L1
        goto L4
L1: x=0

L4:
```

chapter - 7 Runtime Environment

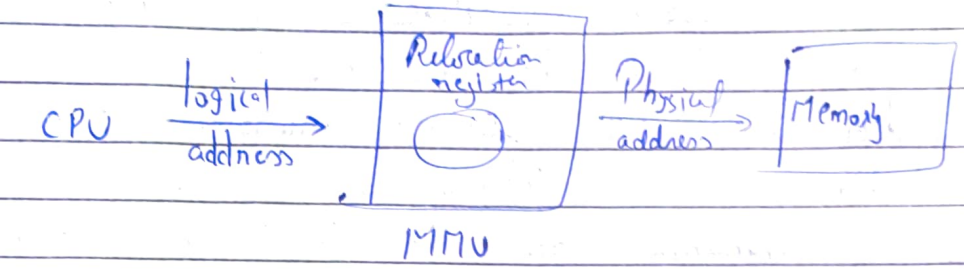
Issues →

i) Layout of address locations
ii) Allocation of address to objects
iii) Mechanism to access variables
iv) Linkages between procedures
v) Passing parameters

Spiral

Date .....

Storage Organization



The global constants & the data generated by the compiler is placed in statically determined called "static" which may be known at compile time. The size of the generated target code is fixed at compile time so that the compiler can place the executable target code in a statically determined area "code" which is usually placed in the low end of the memory.

Heap & Stack

To maximize the utilization of space at runtime, the other two areas called stack & heap are at the opposite ends of the remainder of the address space. These areas are dynamic i.e., their size can change as the program executes. The stack is used to store data structures called activation records generated during procedure calls.

Date .....

Static vs Dynamic Storage Allocation

The layout & allocation of data to memory locations in the runtime environment are the key issues in storage management. The storage allocation is distinguished by two i.e. static & dynamic and the time at which the it is decided is called static & dynamic time.

Dynamic Storage Allocation

Stack & Heap → Stack stores the names local to a procedure whereas heap allocates the memory to the object where they are created like in C, malloc & ~~return~~ that storage when they are invalidated.

Garbage Collection :- It enables the runtime system to detect

Stack Allocation → Whenever a procedure/function/subroutine/method, space for its local variables are pushed onto the stack. When the procedure terminates, that space is popped off the stack.

Spiral

Date .....

Activation Trees

Stack allocation would not be feasible if procedure calls, or activations of procedures, did not nest in time.

If an activation of procedure p calls procedure q, then that activation of q must end before the activation of p can end. There are three common cases:-

→ The activation of q terminates normally. Then in essentially any language, control resumes just after the point p at which the call to q was made.

→ The activation of q, or some procedure q called, either directly or indirectly aborts, i.e. it becomes impossible for execution to continue. In that case, p ends simultaneously with q.

→ The activation of q terminates because of an exception that q can't handle. Procedure p may handle the exception, in which case the activation of q has terminated while activation of p continues, although not necessarily from the point at which the call to q was made. If p can't handle the exception, then this activation of p terminates at the same time as the activation of q.

Spiral

Date .....

Quicksort

```
int a[11];
void readArray(){
    int i;
    ...
}
int partition(int m, int n){
/* picks a separation value (pivot) v & partitions a[m...n]
so that a[m...p-1] < v ≥ a[p+1...n]. Returns p */
void quicksort(int m, int n){
int i;
if (n > m){
    i = partition(m,n);
    quicksort(m, i-1);
    quicksort(i+1, n);
}
}
main(){
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

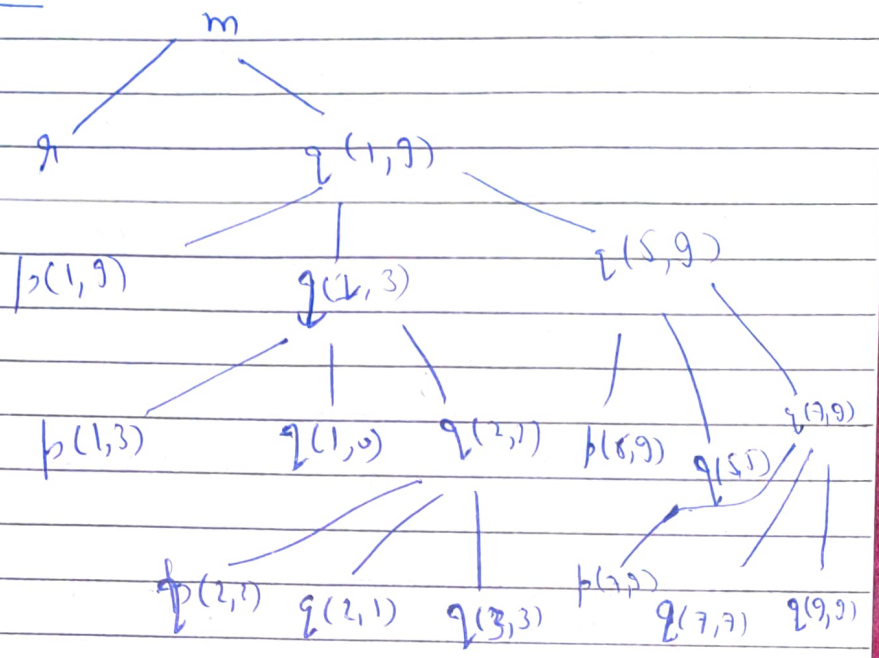
Activations →

enter main()
enter readArray()
leave readArray()
enter quicksort(1,9)
enter Partition(1,9)
leave Partition(1,9)
enter quicksort(1,3)
leave quicksort(1,3)
enter quicksort(5,9)
leave quicksort(5,9)
leave main()

Spiral

Date .....

Activation Tree



The use of run-time stacks is enabled by several useful relationships between the activation tree & the behaviour of the program

→ The sequence of procedure calls corresponds to a preorder traversal of activation tree.

→ The sequence of returns corresponds to a post order traversal of the activation tree

→ Suppose that control lies within particular activation of some procedure, corresponding to a node N of activation tree. Then the activations that are currently open are those that correspond to node N & its ancestors. The order in which these activations were called is the order in which they appear along the path to N, starting at the root & they will return in the reverse of that order.

Spiral

Date .....

Control Stack / Run-time Stack

Procedure calls & returns are usually managed by a run-time stack called the control stack.

Activation Record → Each live activation has an activation records (sometimes called frame) on the control block.

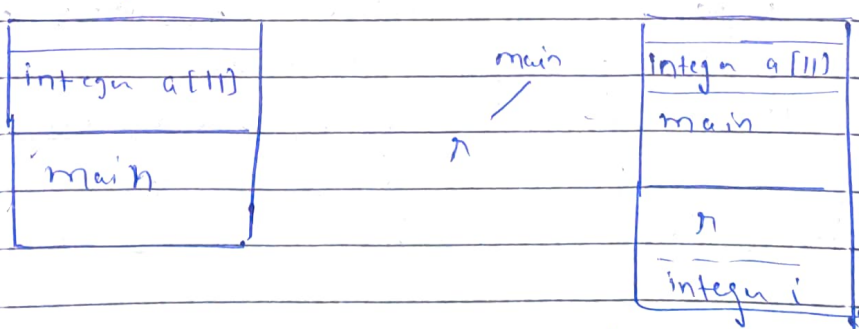
→ Root of activation trees at the bottom content of activation record.

Content of Activation Records

1) Actual Parameters → used by calling procedures
2) Returned Values → if procedure returns a value
3) Control link → pointing to activation of caller
4) Access link → used to locate data by called procedure
5) Saved Machine Status → status is stored just before calling another procedure.
6) Local data → belong to procedure
7) Temporaries → arising from evaluating expression.

Stack →

frame for main

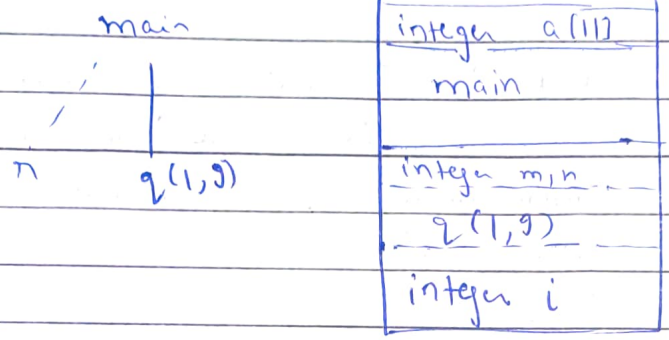


n is activated

Spiral

Date .....

c) n has been popped & q(1,9) pushed



Downward growing stack of activation records

Example → find the Output

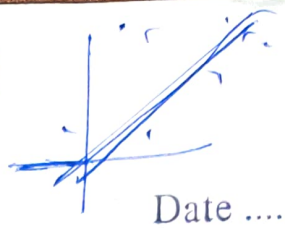
```
int f(int x, int *y, int **z) {
    **z += 1;
    *y += 2;
    x += 3;
    return x + *y + **z;
}
int main() {
    int x, c, *b, **a;
    c = 4, b = &c, a = &b;
    x = f(c, b, a);
    printf("%d", x);
    return 0;
}
```

Ans → 21

Spiral

$f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$ $[0\ \infty] \rightarrow [0\ 1]$

ReLu → Rectified Linear Unit



Date .....

Calling Sequence → Consists of code that allocates an activation record on the stack.

Return Stack → Consists of code to restore the state of machine so the calling procedure can continue its execution after call.