Compiler Design

Parse tree of the string 9-5+2 using

G = {list, digit}, {+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, P, list>

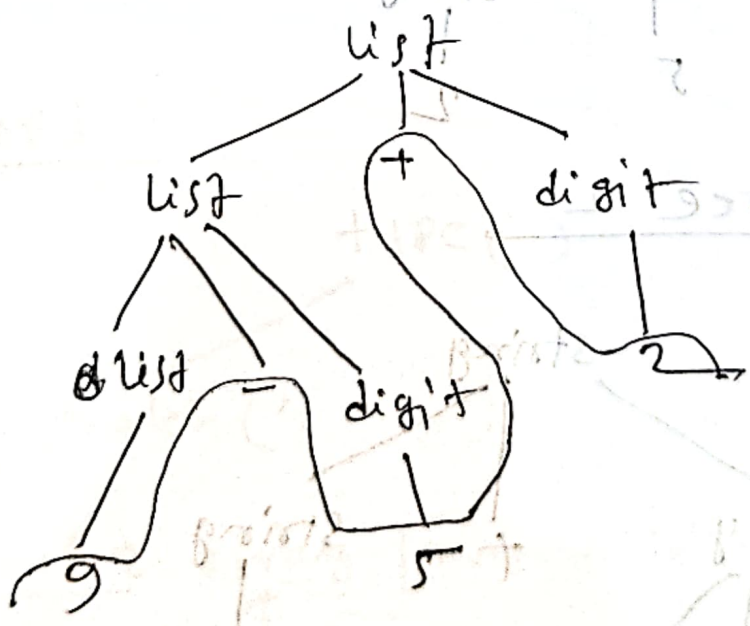
P =

list → list + digit

list → list - digit

list → digit

digit → 0 — 9

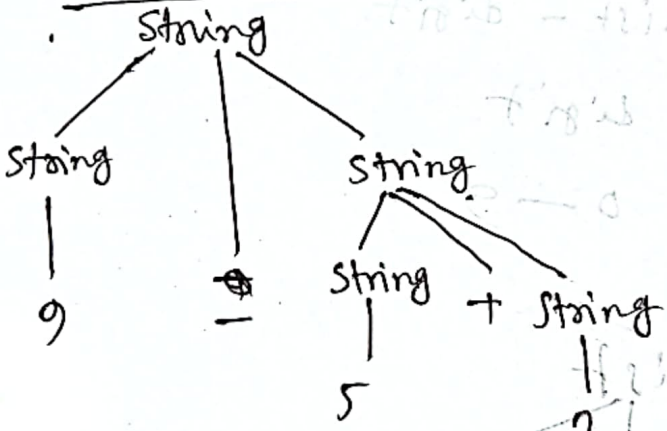


1. $G = \langle \{String\}, \{+, -, 0 \cdots 9\}, P, String \rangle$

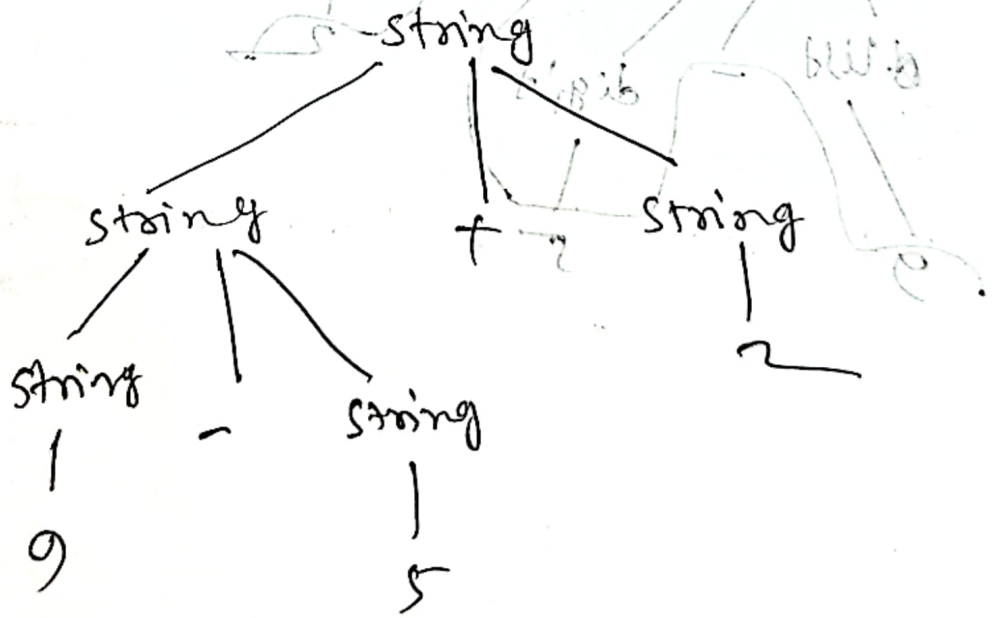
$String \rightarrow String + String \mid String - String \mid$

$0 \cdots 9$

parse tree 1



parse tree 2



This grammer is ambiguous, because more than one parse tree possible

18-19 (b)

predictive parser

type → simple
| ^ id
| array [simple] of type

simple → integer
| char
| num dot dot num

Input: array [num dotdot num] of integer

Step 1

type()

match('array')

Input: array [num dotdot num] of integer

lookahead

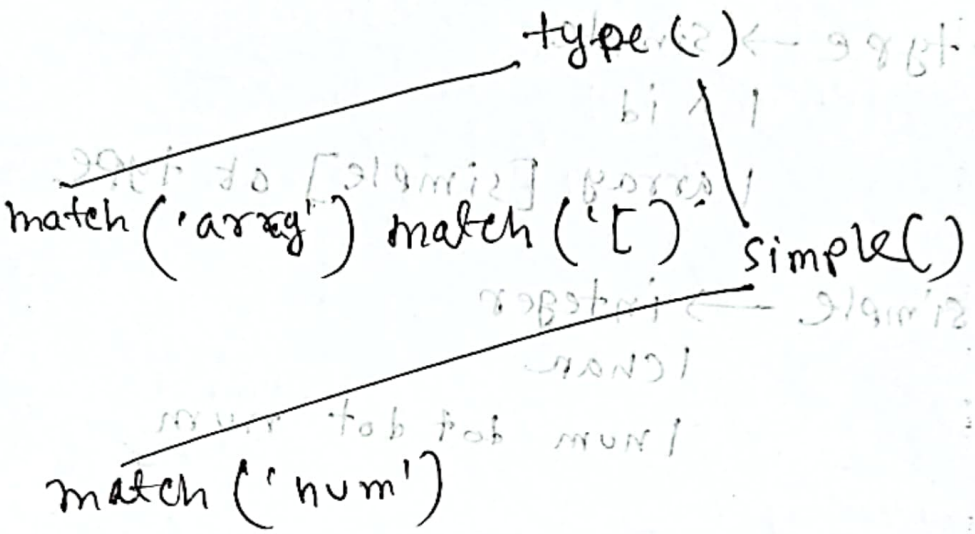
Step 2

type()

match('array') match('[')

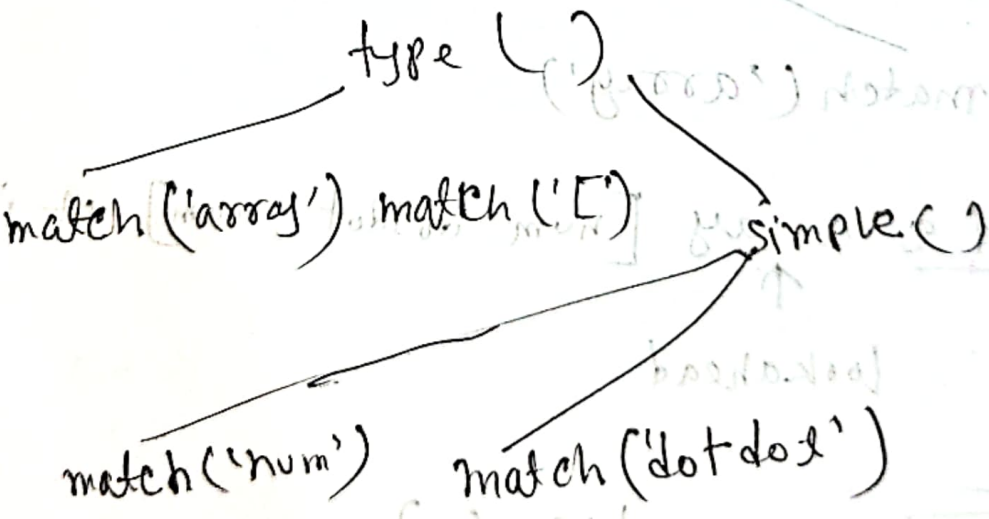
Input: array [

~~steps~~ Step 3



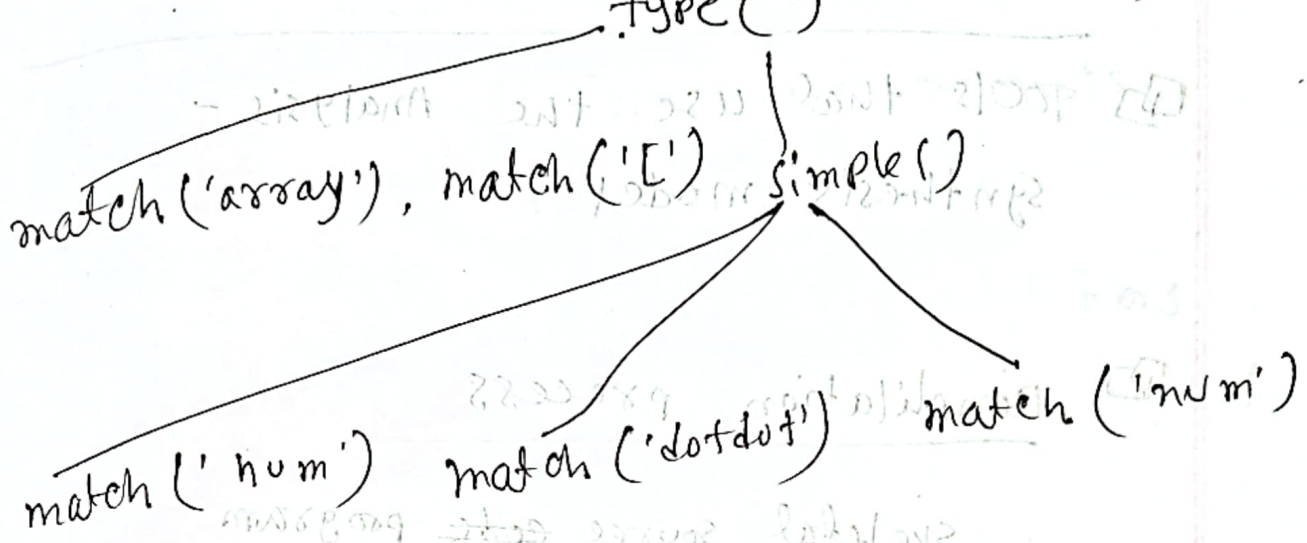
Inp: array [num - - - -
↑
lookahead

Step 4



Inp: array [num dot dot
↑
lookahead

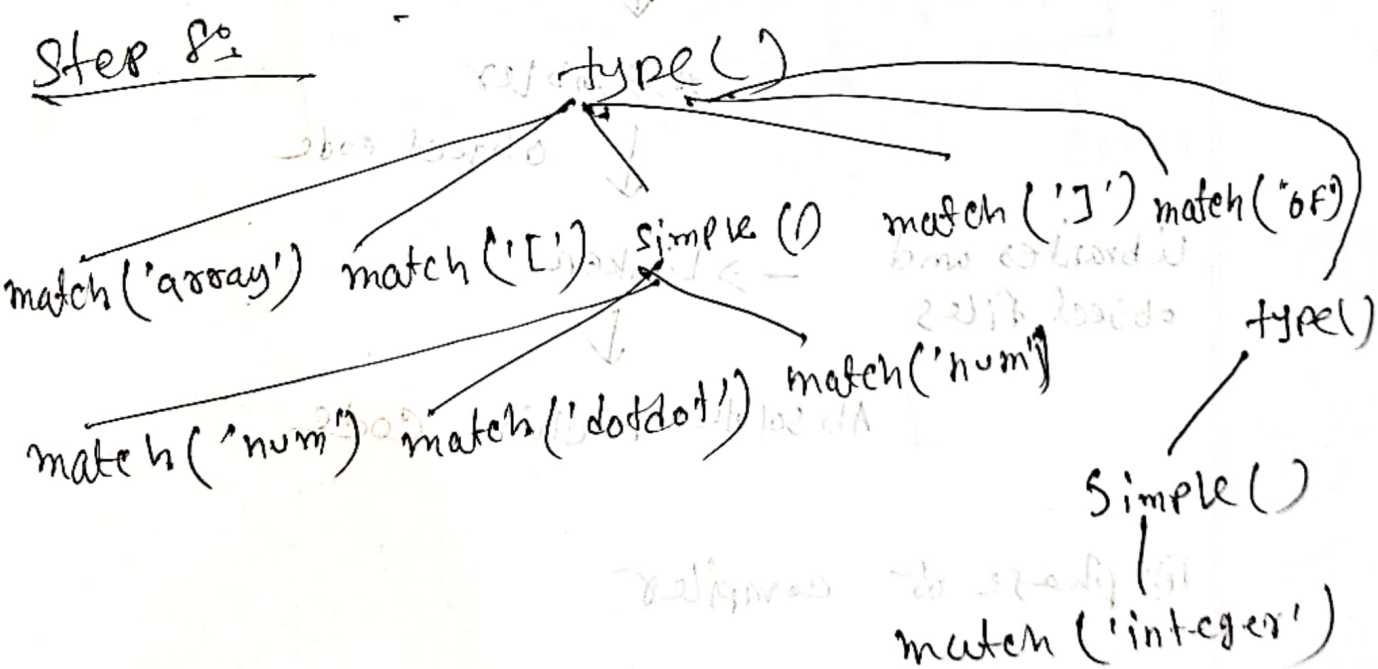
Step 5



Imp: array [num dotdot num] . — —

↑ lookahead

Step 8:



Imp: array [num dotdot num] of integer

↑ lookahead

Char-1

Analysis - Synthesis model of compilation

TOOls that use the Analysis - synthesis model

Compilation process

skeletal source ~~code~~ program

↓

Processor

↓ source program

compiler

↓ Target program

Assembler

↓ object code

Libraries and object files → Linker

↓

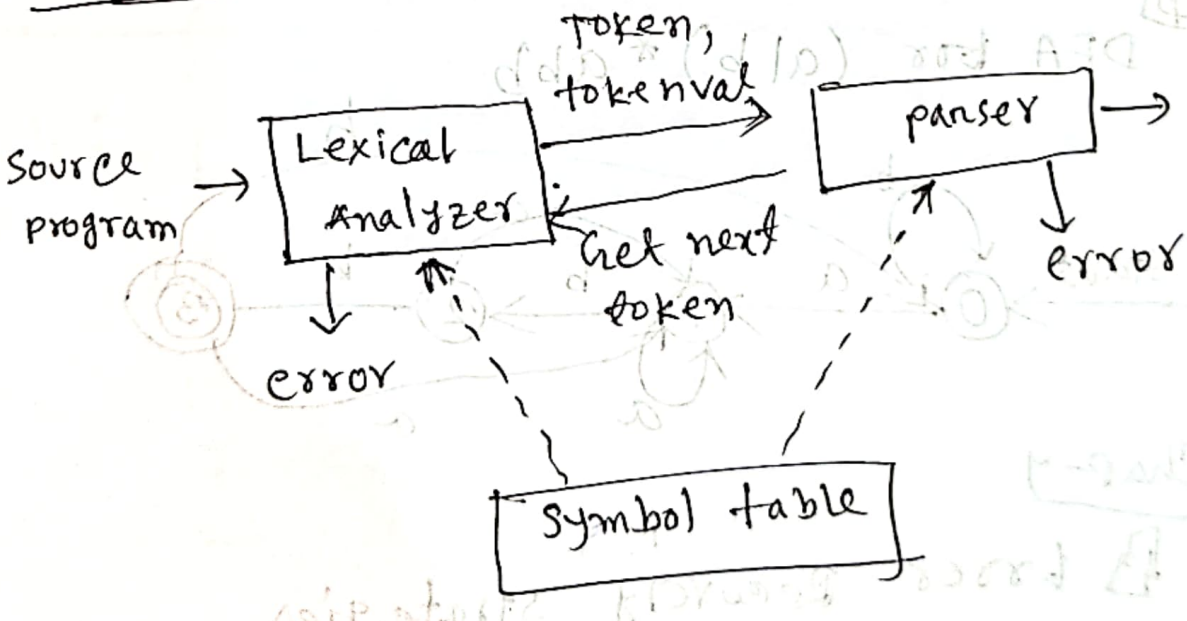
Absolute machine code

phase of compiler

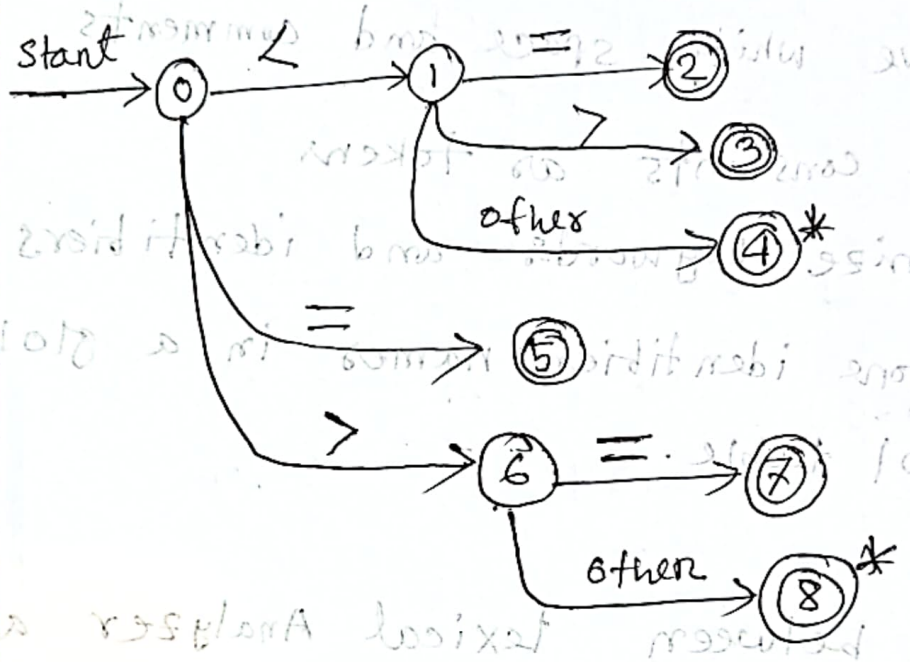
Lexical Analyzer:

1. Remove white space and comments
2. Encode constants as tokens
3. Recognize keywords and identifiers
4. Store identifier names in a global symbol table.

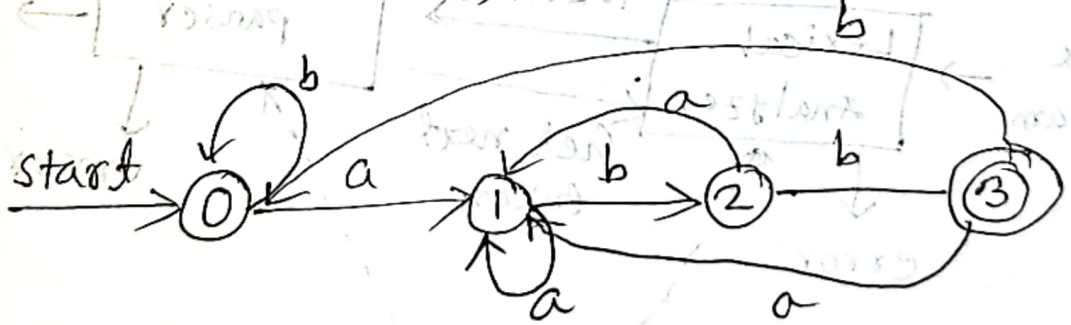
Interaction between lexical Analyzer and panser :



Relop → < | <= | <> | > | >= | =



DFA for (a|b)*abb



Chap-4

Error Recovery strategies

17-18) 4(b) see Chap 4(a) page 33

6(b) 18-19

LR(0)

1) $E \to E+T$
2) $E \to T$
3) $T \to T*F$
4) $T \to F$
5) $F \to (E)$
6) $F \to id$

Augmented Grammer

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

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

Canonical LR(0) collection

I_0 :
$E' \to .E$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

goto (I_0, E)

I_1 : $E' \to E.$
$E \to E. + T$

goto (I_0, T)

I_2 : $E \to T.$
$T \to T. * F$

goto (I_0, F)

I_3 : $T \to F.$

goto $(I_0, ()$

I_4 : $F \to (. E)$
$E \to . E + T$
$E \to . T$
$T \to . T * F$
$T \to . F$
$F \to . (E)$
$F \to . id$

goto (I_0, id)

I_5 : $F \to id.$

goto $(I_1, +)$

I_6 : $E \to E + . T$
$T \to . T * F$
$T \to . F$
$F \to . (E)$
$F \to . id$

goto $(I_2, *)$

I_7 : $T \to T * . F$
$F \to . (E)$
$F \to . id$

goto (I_4, E)

I_8 : $F \to (E.)$
$E \to E. + T$

goto (I_6, T)

I_9: $E \to E + T.$

$T \to T. * F$

goto (I_7, F)

I_{10}: $T \to T * F.$

goto (I_8,))

I_{11}: $F \to (E).$

SLR(1) parsing table

State	action						goto		
	+	*	(	)	id	$	E	T	F
0			s4		s5		1	2	3
1	s6					acc			
2	r2	s7		r2		r2			
3	r4	r4		r4		r4			
4			s4		s5		8	2	3
5	r6	r6		r6		r6			
6			s4		s5		9	9	3
2			s4		s5				10
8	s6			s11					
9	r1	s7		r1		r1			
10	r3	r3		r3		r3			
11	r5	r5		r5		r5			

$w = id * id + id$

stack	Input	Action
0	id*id+id$	Shift
0id5	*id+id$	reduce by 6 F→id
0F3	*id+id$	reduce by T→F
0T2	*id+id$	Shift
0T2*7	id+id$	Shift
0T2*7id5	+id$	reduce by F→id
0T2*7F10	+id$	reduce by T→T*F
0T2	+id$	reduce by E→T
0E1	+id$	Shift
0E1+6	id$	Shift
0E1+6id5	$	reduce by F→id
0E1+6F3	$	reduce by T→F
0E1+6T9	$	reduce by E→E+T
0E1	$	Accept.

LL(1)

	First	Follow
S → iEtS | iEtSes | a	{i, a}	{$, e}
E → b	{b}	{t}
S → iEtSS' | a S' → ε / es E → b	{ε, e}	{$, e}

Non terminal ↓	a	i	b	e	t	$
S	S→a	S→iEtSS'				
E			E→b			
S'				S'→es S'→ε		S'→ε

This is not LL(1) grammer because in the cell of terminal e there is two production rules satisfied.

LALR (1)

(1)
(2)
(3)
(4)

① $S \to CC$
$C \to cC$
$C \to c \mid d$

$S' \to S$
$S \to CC$
$C \to cC$
$C \to c \mid d$

I_0

$S' \to .S, \$$
$S \to .CC, \$$
$C \to .cC, c/d$
$C \to .c, c/d$
$C \to .d, c/d$

goto (I_0, S)

I_1 $S' \to S., \$$

goto (I_0, C)

I_2 $S \to C.C, \$$
$C \to .cC, \$$
$C \to .c, \$$
$C \to .d, \$$

goto (I_0, c)

I_3 $C \to c.C, c/d$
$C \to .cC, c/d$
$C \to .c, c/d$
$C \to .d, c/d$

$C \to c., c/d$

goto (I_0, d)

I_4 $C \to d., c/d$

goto (I_2, C)

I_5 $S \to CC., \$$

goto (I_2, c)

I_6 $C \to c.C, \$$
$C \to$

goto (I_2, C)

I_6

$C \to c.C, \$$

$C \to .cC, \$$

$C \to .c, \$$

$C \to .d, \$$

$C \to c., \$$

goto (I_2, d)

I_4 $C \to d., \$$

goto (I_3, C)

I_8 $C \to cC., \$/d$

goto (I_3, c)

I_3

goto (I_6, C)

I_9 $C \to cC., \$$

I	Action			Goto	
	c	d	$	S	C
0	S3	S4		1	2
1			Acc		
2	S6	S7			5
36	S36 R3	S47 R3	R3		89
47	R4	R4	R4		
5			R1		
89	S6	S7	R3		9
7			R4		
8	R2	R2	R2		
9			R2		

I	Action			Goto	
	c	d	$	S	C
36	S36 R3	S47 R3	R3		89
47	R4	R4	R4		
89	R2	R2	R2		

Three address code

Three address code এতে 3 টি variable এবং 1 টি operator use করা হয়।

$x = y * z + y * z$ এটি কে $y * z$ — $y * z$

Three address code for x :

$T_1 = y * z$

$T_2 = T_1 - T_1$

Triples representation

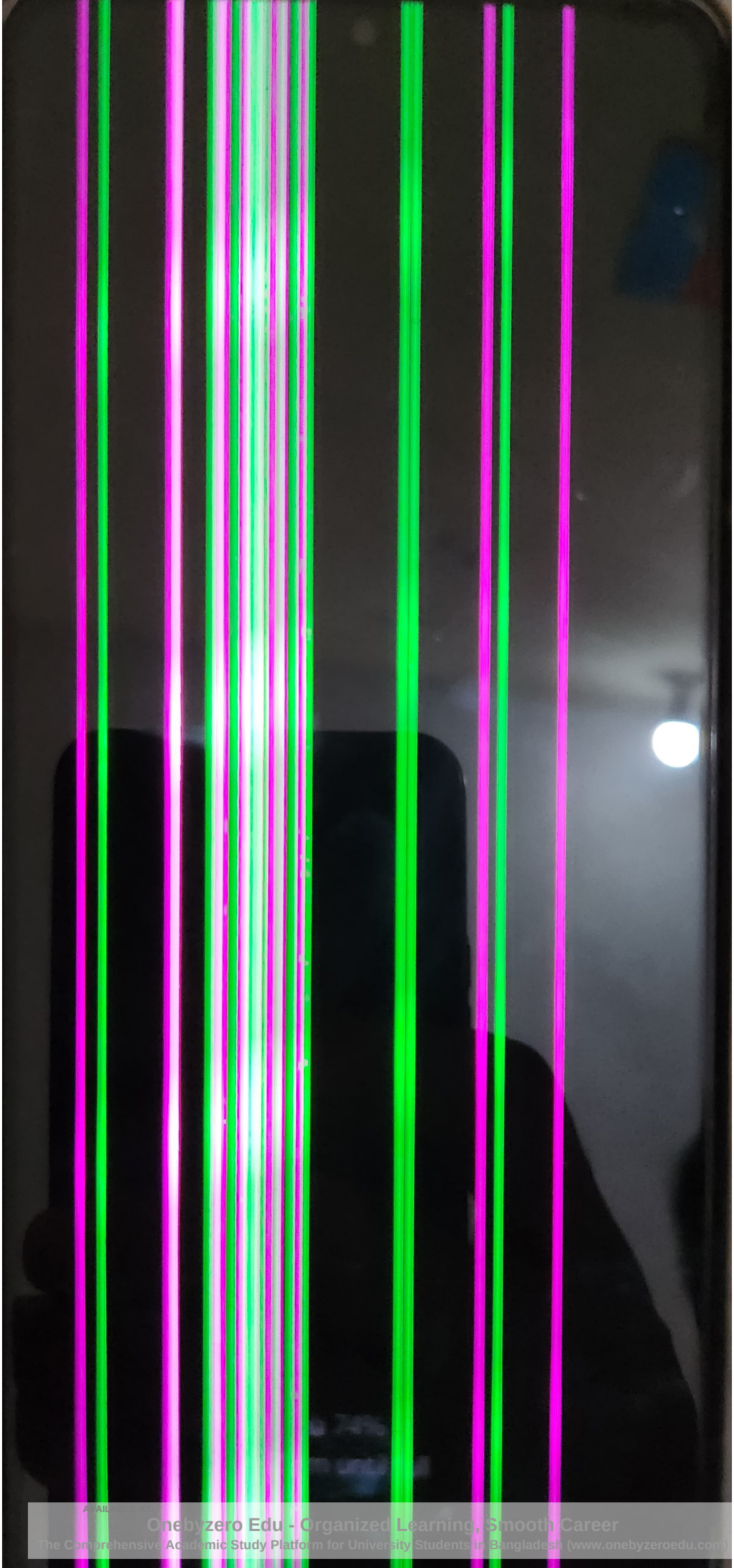
	Index	operator	Argument 1	Argument 2
t_1	0	*	y	z
t_2	1	−	0	0

Operator precedence Relation table

	+	*	ID	$
+	>	<	<	>
*	>	>	<	>
id	>	>		>
$	<	<	<	Acc

E → E+E | E*E | id

Stack	Relation	Input	comment
$	<	Id+Id*Id$	Shift → id
$id	>	+id*id $	R→E→id
$E	<	+id*id $	S→+
$E+	<	id*id $	S→id
$E+id	>	*id $	R→E→id
$E+E	<	*id $	Shift→*
$E+E*	<	id $	S→id
$E+E*id	>	$	R→E→id
$E+E*E	>	$	R→E→E*E
$E+E	>	$	R→E→E+E
$E	Acc	$	



18-19

(1) (a)

Type checking: is an essential process in programming where a compiler verifies and enforces constraints on data types within a program.

There are two primary types of type checking:

(i) static type checking:

→ performed at compile time.

→ static type checking verifies types of variables based on the programs source code.

→ It detects errors before the program runs, enhancing code reliability.

(ii) Dynamic checking :

→ Dynamic type checking occurs at runtime and verifies types as the program executes.

→ It is more flexible.

→ May leads to runtime errors if types don't match.

2(a)

Type equivalance :

In compiler design, type equivalance refers to the rules used to determine if two type expressions are considered the same. There are two main approaches:

1) Name equivalance

4) Structural equivalance

Intermediate code generator

The intermediate code generator acts as a crucial bridge in the compilation process, transforming source code into an intermediate representation (IR) that is independent of both the source language and the target machine. This IR simplifies further processing like optimization and translation to machine code. It enables compilers to be more portable and efficient by handling code analysis and optimization at an intermediate level.

2(b)

Given expression:

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

Three address code:

$T_1 = b - c$

$T_2 = T_1 * d$

$T_3 = T_1 * a$

$T_4 = T_2 + T_3$

$T_5 = a + T_4$

quadruple representation:

Operator	Argument 1	Argument 2	result
−	b	c	T_1
*	T_1	d	T_2
*	T_1	a	T_3
+	T_2	T_3	T_4
+	a	T_4	T_5

Triples

	Index	Argument1	Argument2	Operator
T_1	0	b	c	−
T_2	1	0	d	*
T_3	2	0	a	*
T_4	3	1	2	+
T_5	4	a	3	+

2(c)

peephole optimization :

peephole optimization is a compiler design technique that improves code by replacing small sequences of instructions with shorter, faster, or more efficient sequences.

Machine dependent optimization tailors code specifically for a particular machine architecture. It involves utilizing features like CPU registers, memory organization and instruction set of a particular processor to achieve optimal performance.

Examples: Register allocation, peephole optimization etc.

Machine independent optimization focuses on improving code efficiency regardless of the underlying hardware. It primarily works on intermediate code and targets general code improvements.

Examples: Dead code Elimination, code motion, Strength Reduction etc

1(a)

Dangling else

A dangling else problem is a problem where the 'else' doesn't know to which 'if' it belongs to.

It occurs when we use nested if. When there are multiple "if" statements, the "else" part doesn't get a clear view with which "if" it should combine.

For example:

```
if (condition) {
{
if (condition) {
{
else {
{
```

Resolving dangling else problem:

① Design non-ambiguous programming languages

② using braces and indentation.

For example:

```
if (condition) {
    if (condition) {
    }
    else {
    }
}
```

③ we can also use ~~the~~ the "if-else if-else" format.

4(b)

Given Grammer:

id − id / id

production rules:

$E \rightarrow E - E$

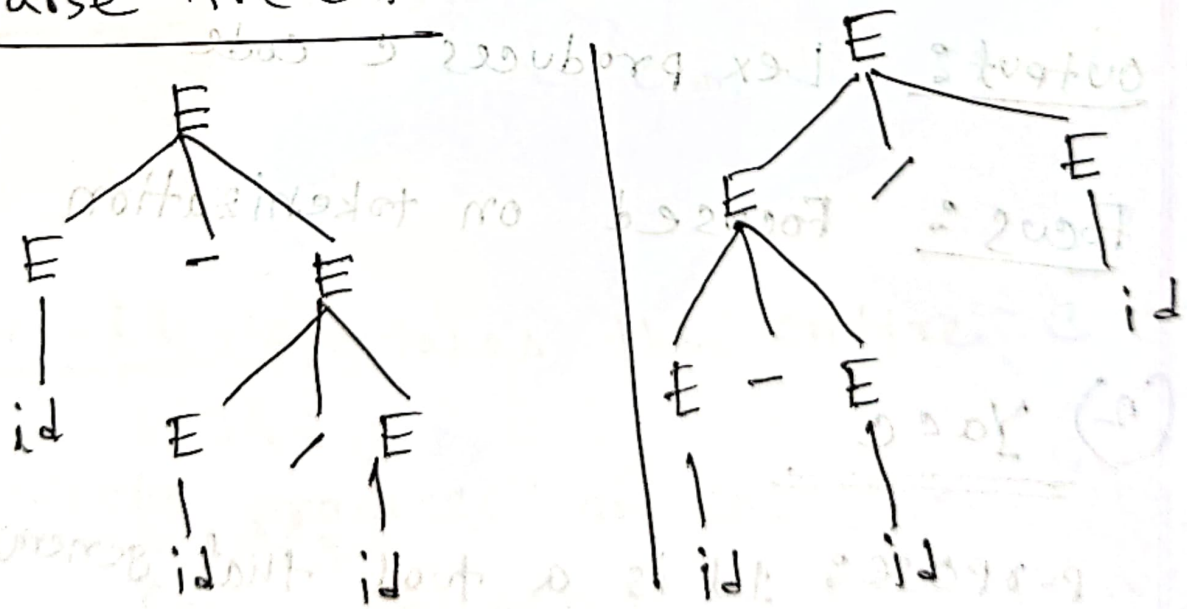
$E \rightarrow E / E$

$E \rightarrow - E$

$E \rightarrow (E)$

$E \rightarrow id$

parse tree:



Since for the grammer (id−id/id) there is two parse tree possible.

So the grammer is ambigious.

5(a)

(1) Lex

purpose: Lex is a tool that generates a lexical analyzer

Input: It takes a set of rules that specify how to break the input text into tokens.

output: Lex produces C code

Focus: Focused on tokenization

(2) Yacc

purpose: It is a tool that generates a parser for a programming language

Input: It takes a context-free-grammer that defines the structure of the language.

output: Yacc produces C code for a parser.

Focus: focused on parsing

(3) C compiler

purpose: Translates C codes into machine code that a computer can directly executes.

Input: It takes the entire C program

output: produces an executable file that contains machine code instructions.

Focus: Handles complex task.

5(b)

A handle is a substring that connects a right-hand side of the production rule in the grammar

Example

Grammar :

S → a ABe

A → Abc | b

B → d

abbcde

aAbcde

aAde

aABe

S

Handle

5(c)

Substring :

abbcde

Grammar :

S → aABe

A → Abc | b

B → d

Right sentential form	Handle	Replacing production
a b b c d e	b	A → b
a A b c d e	A b c	A → A b c
a A d e	d	B → d
a A B e	a A B e	S → a A B e
S		

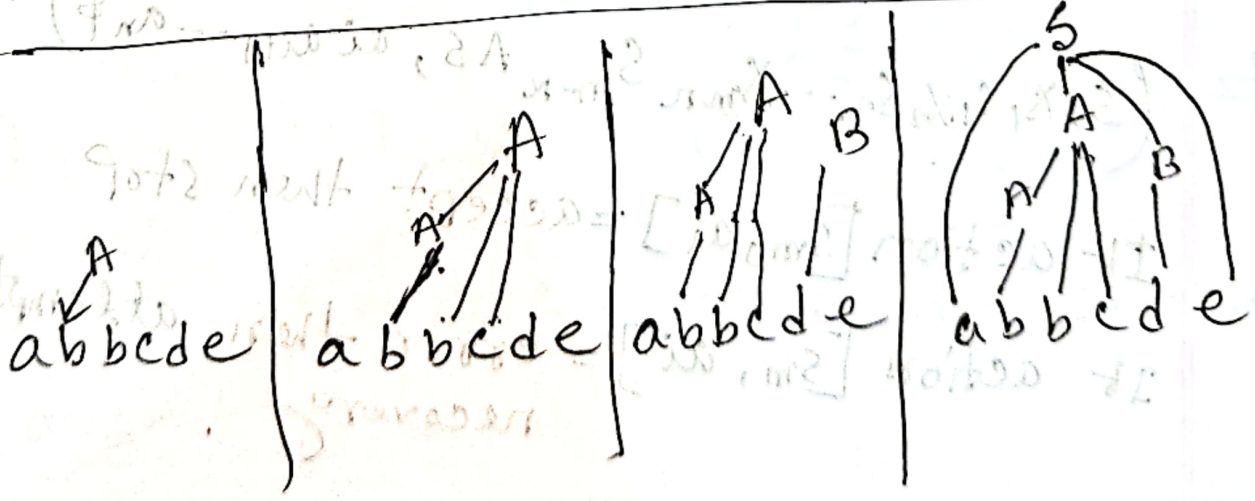
It follows that:

S → a A B e is a handle of a A B e in location 1

B → d is a handle of a A d e in location 3

A → A b c is a handle of a A b c d e in location 2

A → b is a handle of a b b c d e in location 4



6(a)

Algorithm for LR parser:

$(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m, a_i a_{i+1} \dots a_n \$)$

stack | input

If action $[S_m, a_i]$ = shift s then push a_i, push s, and andvance input:

$(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m a_i s, a_{i+1} \dots a_n \$)$

If action $[S_m, a_i]$ = reduce $A \to \beta$ and goto $[S_{m-r}, A] = s$ with $r = |\beta|$ then pop $2r$ symbols, push A, push s;

$(S_0 X_1 S_1 X_2 S_2 \dots X_{m-r} S_{m-r} A S, a_i a_{i+1} \dots a_n \$)$

If action $[S_m, a_i]$ = accept then stop

If action $[S_m, a_i]$ = error then attempt recovery.

7(a)

panic mode	phanse-level recovery
(1) A bassic error recovery technique that skips input tokens untill a synchronization token is found	(1) A method that corrects errors by modifying the input
(2) Simple to implement, but may skip over multiple error	(2) more sophisticated, can recover from more types of error
(3) Relatively simple to implement	(3) more complex to design and implement
(4) Less sophisticated	(4) more sophisticated
(5) may lose a large amount of content	(5) Minimal skipping

7(b)

Given grammer:

$S \rightarrow (L) / a$

$L \rightarrow L, S / S$ ⟶ Removing left recursion

$S \rightarrow (L) / a$

$L \Rightarrow S L'$

$L' \rightarrow , S L' / \varepsilon$

	First	Follow
S	{(, a}	{$, ,,)}
L	{(, a}	{)}
L'	{,, ε}	{)}

parsing table

	a	(	)	·	$
S	S→a	S→(L)			
L	L→SL'	L→SL'			
L'			L'→ε	L'→·SL'	

	First	Follow
S	{(, a}	{$}
L	{(, a}	{), $}

Non-terminal	(	)	*	/	a	$
S	S→(L)				S→a	
L						

8(a)

$E \rightarrow TE_R$

$E_R \rightarrow +TE_R \mid \epsilon$

$T \rightarrow FT_R$

$T_R \rightarrow *FT_R \mid \epsilon$

$F \rightarrow (E) \mid id$

Follow (A)

Follow (E) = {$,)}

Follow (E_R) = {$,)}

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

Follow (T_R) = {+, $,)}

Follow (F) = {*, +, $,)}

8(b)

(1) Overloading, coercion, and polymorphism

```
int op(int), OP(float);
int F(float);
int a, c[10], d;

d = c+d ;      // Fail
*d = a ;       // Fail
a = op(d);     // OK: overloading (c++)
a = F(d);      // OK: coersion of d to float
Vector <int> v ; // ok: template instantiation
```

(11) Flow-of-Control Checks

```
myfunc()
{
  ....
  break; //Error
}
```

```
myfunc()
{
  switch (a)
  { case 0:
      break; //OK
    case 1:
  }
}
```

```
myfunc()
{
  while(n)
  {
    if (i>10)
      break; //OK
  }
}
```