COMPILER DESIGN	CS4101	ASSIGNMENT

NAME: ANISH BANERJEE 14.11.2024

ENROLMENT NUMBER: 2021CSB001

(1) Given a grammar with the following production rules:

$S \rightarrow E\#$

$E \rightarrow T \mid E+T$

$T \rightarrow P \mid T*P$

$P \rightarrow F \mid F \wedge P$

$F \rightarrow i \mid (E)$

Where $V_T = \{+, \#, *, \wedge, i, (,)\}$

$V_N = \{S, E, T, P, F\}$

(a) Compute FIRST and FOLLOW sets to all non-terminal symbols.

Ans) $FIRST(S) = \{i, (\}$

$FIRST(E) = \{i, (\}$

$FIRST(T) = \{i, (\}$

$FIRST(P) = \{i, (\}$

$FIRST(F) = \{i, (\}$

$FOLLOW(S) = \{\$\}$

$FOLLOW(E) = \{\#, +,)\}$

$FOLLOW(T) = \{\#, +, *,)\}$

$FOLLOW(P) = \{\#, +, *,)\}$

$FOLLOW(F) = \{\#, +, *, \wedge,)\}$

(b) Construct LR(0) m/c for the grammar.

After augmenting the grammar by introducing a new start symbol S', we obtain:

$S' \to S$
① $S \to E\#$
② $E \to T$
③ $E \to E+T$
④ $T \to P$
⑤ $T \to T*P$
⑥ $P \to F$
⑦ $P \to F \wedge P$
⑧ $F \to i$
⑨ $F \to (E)$

Initial Item:

$I = \{S' \to \cdot S\}$

$I_0 = \text{closure}(I)$

$= \{S' \to \cdot S,$
$S \to \cdot E\#,$
$E \to \cdot T,$
$E \to \cdot E+T,$
$T \to \cdot P,$
$T \to \cdot T*P,$
$P \to \cdot F,$
$P \to \cdot F \wedge P,$
$F \to \cdot i,$
$F \to \cdot (E)\}$

$I_1 = goto(I_0, S)$
$= \{S' \to S\cdot\}$

$I_2 = goto(I_0, E)$
$= \{S \to E\cdot\#,$
$E \to E\cdot + T\}$

$I_3 = goto(I_0, T)$
$= \{E \to T\cdot,$
$T \to T\cdot * P\}$

$I_4 = goto(I_0, P)$
$= \{T \to P\cdot\}$

$I_5 = goto(I_0, F)$
$= \{P \to F\cdot,$
$P \to F\cdot \wedge P\}$

$I_6 = goto(I_0, i)$
$= \{F \to i\cdot\}$

$I_7 = goto(I_0, ()$
$= \{F \to (\cdot E),$
$E \to \cdot T,$
$E \to \cdot E + T,$
$T \to \cdot P,$
$T \to \cdot T * P,$
$P \to \cdot F,$
$P \to \cdot F \wedge P,$
$F \to \cdot i,$
$F \to \cdot (E)\}$

$I_8 = goto(I_2, \#)$
$= \{S \to E\#\cdot\}$

$I_9 = goto(I_2, +)$
$= \{E \to E+\cdot T,$
$T \to \cdot P,$
$T \to \cdot T * P,$
$P \to \cdot F,$
$P \to \cdot F \wedge P,$
$F \to \cdot i,$
$F \to \cdot (E)\}$

$I_{10} = goto(I_3, *)$
$= \{T \to T * \cdot P,$
$P \to \cdot F,$
$P \to \cdot F \wedge P,$
$F \to \cdot i,$
$F \to \cdot (E)\}$

$I_{11} = goto(I_5, \wedge)$
$= \{P \to F \wedge \cdot P,$
$P \to \cdot F,$
$P \to \cdot F \wedge P,$
$F \to \cdot i,$
$F \to \cdot (E)\}$

$I_{12} = goto(I_7, E)$
$= \{F \to (E\cdot),$
$E \to E\cdot + T\}$

$I_3 = goto(I_7, T)$
$I_4 = goto(I_7, P)$
$I_5 = goto(I_7, F)$
$I_6 = goto(I_7, i)$
$I_7 = goto(I_7, ()$

$I_{13} = goto(I_9, T)$
$= \{E \to E+T.,$
$T \to T.*P\}$

$I_4 = goto(I_9, P)$

$I_5 = goto(I_9, F)$

$I_6 = goto(I_9, i)$

$I_7 = goto(I_9, ()$

$I_{14} = goto(I_{10}, P)$
$= \{T \to T*P.\}$

$I_5 = goto(I_{10}, F)$

$I_6 = goto(I_{10}, i)$

$I_7 = goto(I_{10}, ()$

$I_{15} = goto(I_{11}, P)$
$= \{P \to F \wedge P.\}$

$I_5 = goto(I_{11}, F)$

$I_6 = goto(I_{11}, i)$

$I_7 = goto(I_{11}, ()$

$I_{16} = goto(I_{12},))$
$= \{F \to (E).\}$

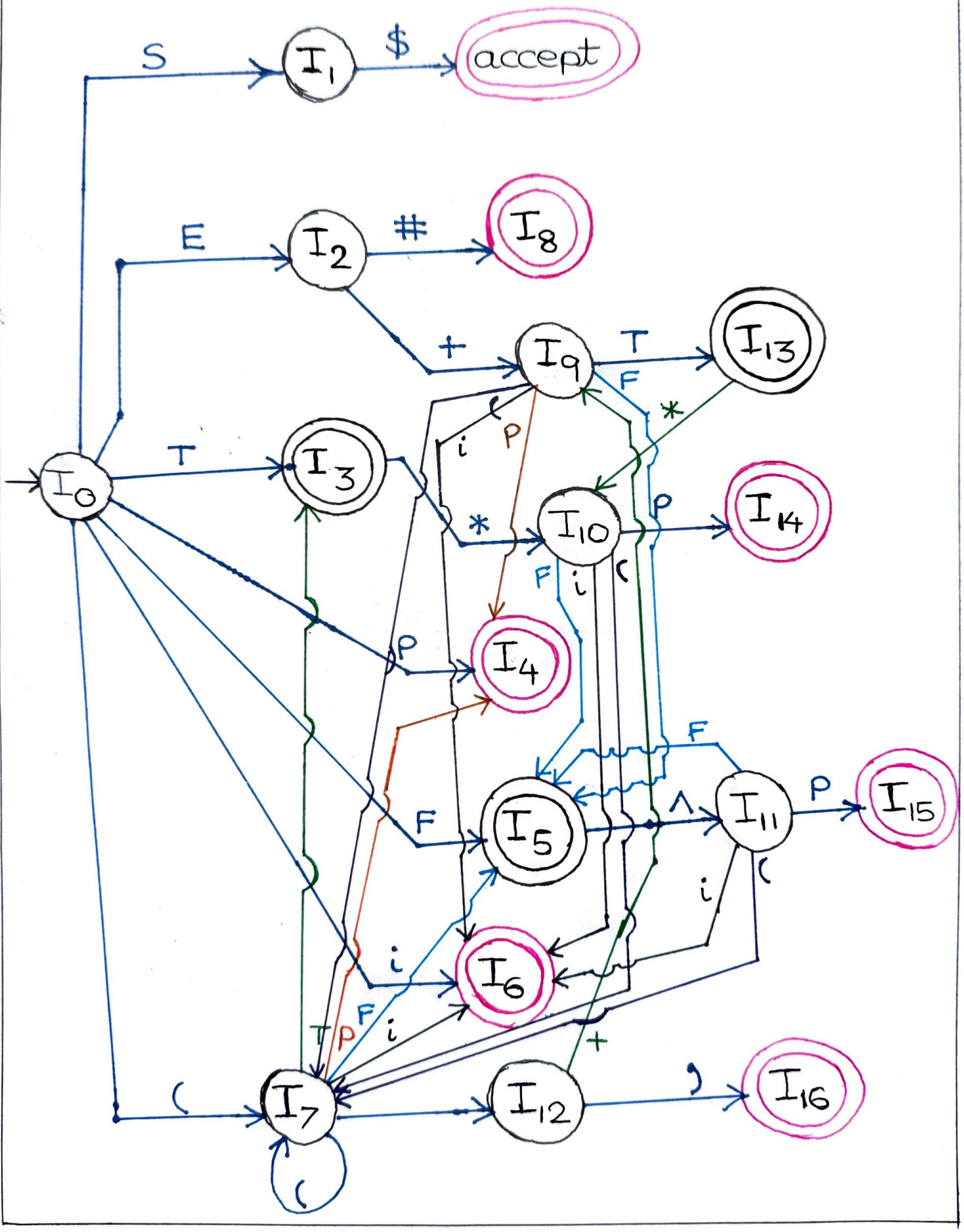
$I_9 = goto(I_{12}, +)$

$I_{10} = goto(I_{13}, *)$

The LR(0) machine is constructed as follows.

There are Shift - Reduce conflicts at three states, namely I_3, I_5 and I_{13}, which have been denoted in Brown. The other Reduce states are shown in Pink.

LR(0) m/c


I0
I1
I2
I3
I4
I5
I6
I7
I8
I9
I10
I11
I12
I13
I14
I15
I16
accept
S
$
E
#
+
T
F
*
(
)
i
P

(c) Construct SLR(1) parsing table for the grammar.

STATE	ACTION								GOTO				
	#	+	*	^	i	(	)	$	S	E	T	P	F
0					s6	s7			1	2	3	4	5
1								acc					
2	s8	s9											
3	r2	r2	s10				r2						
4	r4	r4	r4				r4						
5	r6	r6	r6	s11			r6						
6	r8	r8	r8	r8			r8						
7					s6	s7				12	3	4	5
8								r1					
9					s6	s7					13	4	5
10					s6	s7						14	5
11					s6	s7						15	5
12		s9					s16						
13	r3	r3	s10				r3						
14	r5	r5	r5				r5						
15	r7	r7	r7				r7						
16	r9	r9	r9	r9			r9						

(2) Consider the following grammar G for Boolean expressions:

$B \rightarrow B$ or $T \mid T$

$T \rightarrow T$ and $F \mid F$

$F \rightarrow$ not $B \mid (B) \mid$ true $\mid$ false

where B is the start symbol, set of non-terminals are {B, T, F} and set of terminals are {and, or, not, (,), true, false}.

(a) Compute FIRST and FOLLOW for each nonterminal in G.

Ans) First, we need to remove the left recursion from the given grammar.

$\Rightarrow B \rightarrow B$ or $T \mid T$

is converted to

$B \rightarrow TB'$

$B' \rightarrow$ or $TB' \mid \epsilon$

by introducing a new symbol B'

$\Rightarrow T \rightarrow T$ and $F \mid F$

is converted to

$T \rightarrow FT'$

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

by introducing a new symbol T'.

The transformed grammar is as follows:—

$B \rightarrow TB'$
$B' \rightarrow or\ TB' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow and\ FT' \mid \epsilon$
$F \rightarrow not\ B \mid (B) \mid true \mid false$

$FIRST(B) = \{true, false, not, (\}$
$FIRST(B') = \{or, \epsilon\}$
$FIRST(T) = \{true, false, not, (\}$
$FIRST(T') = \{and, \epsilon\}$
$FIRST(F) = \{true, false, not, (\}$

$FOLLOW(B) = \{\$, and, or,)\}$
$FOLLOW(B') = \{\$, and, or,)\}$
$FOLLOW(T) = \{or, \$, and,)\}$
$FOLLOW(T') = \{or, \$, and,)\}$
$FOLLOW(F) = \{and, or, \$,)\}$

(b) Construct a predictive parsing table for G.

① $B \rightarrow TB'$

$FIRST(TB') = FIRST(T)$
$= \{true, false, not, (\}$

② $B' \rightarrow or\ TB'$

$FIRST(or\ TB') = \{or\}$

③ $B' \rightarrow \epsilon$

$FOLLOW(B') = \{\$,), and, or\}$

④ $T \rightarrow FT'$

FIRST(FT') = FIRST(F)
= {true, false, not, (}

⑤ $T' \rightarrow$ and FT'

FIRST(and FT') = {and}

⑥ $T' \rightarrow \epsilon$

FOLLOW(T') = {or, $,), and}

⑦ $F \rightarrow$ not B

FIRST(not B) = {not}

⑧ $F \rightarrow (B)$

FIRST((B)) = {(}

⑨ $F \rightarrow$ true

FIRST(true) = {true}

⑩ $F \rightarrow$ false

FIRST(false) = {false}

There are 2 conflicts in the table, marked in Pink.

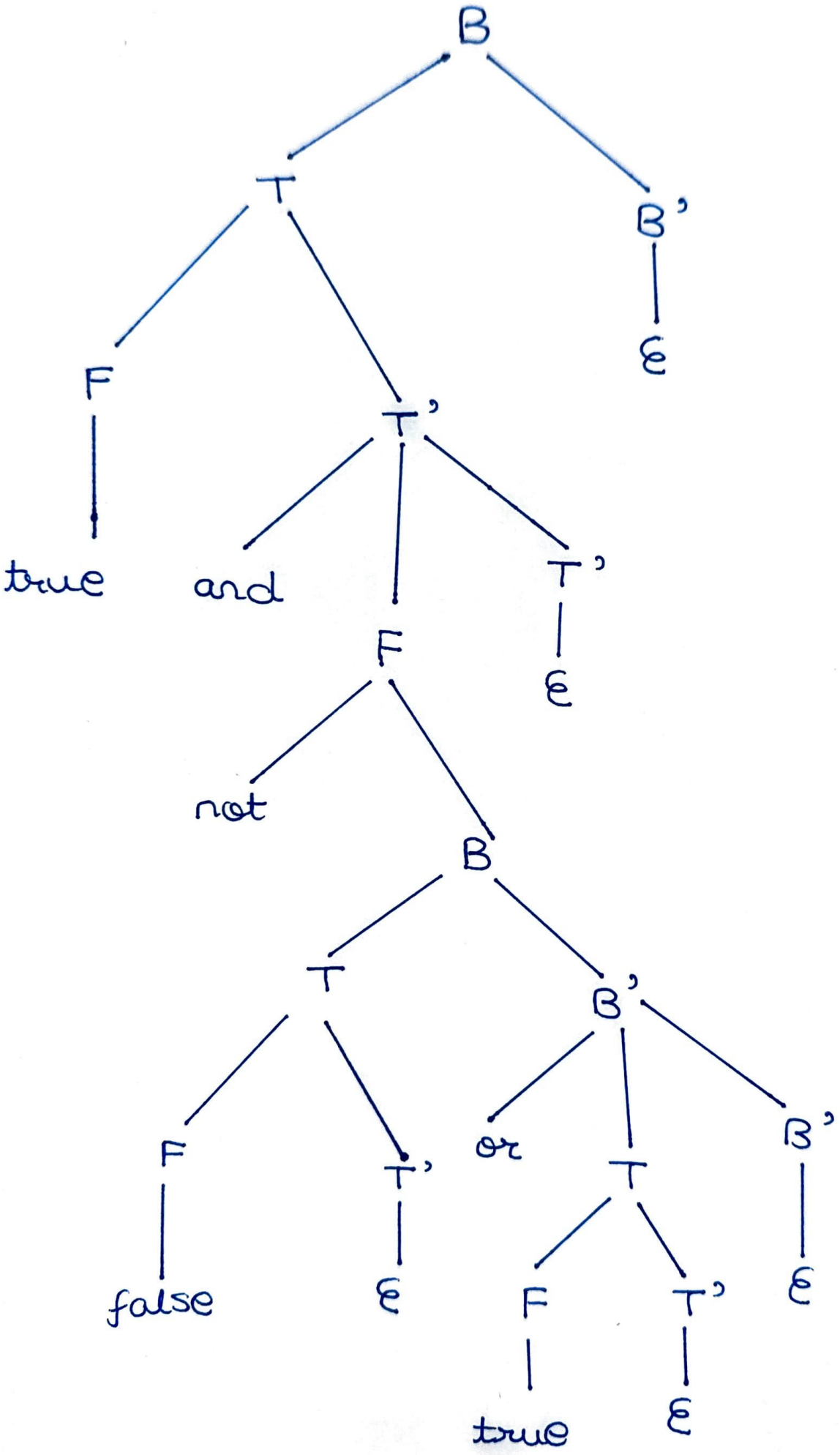
PREDICTIVE PARSING TABLE

NON-TERMINALS	and	or	not	(	)	true	false	$
B			B→TB'	B→TB'		B→TB'	B→TB'	
B'	B'→ϵ	B'→or TB' B'→ϵ			B'→ϵ			B'→ϵ
T			T→FT'	T→FT'		T→FT'	T→FT'	
T'	T'→and FT' T'→ϵ	T'→ϵ			T'→ϵ			T'→ϵ
F			F→notB	F→(B)		F→true	F→false	

(c) Show how your predictive parser processes the input string:
'true and not false or true'
Draw the parse tree traced out by your parser.

SL. NO.	STACK	INPUT	ACTION
1	$B	true and not false or true $	B → TB'
2	$B'T	true and not false or true $	T → FT'
3	$B'T'F	true and not false or true $	F → true
4	$B'T'true	true and not false or true $	Match 'true'
5	$B'T'	and not false or true $	T' → and FT'
6	$B'T'Fand	and not false or true $	Match 'and'
7	$B'T'F	not false or true $	F → not B
8	$B'T'Bnot	not false or true $	Match 'not'
9	$B'T'B	false or true $	B → TB'
10	$B'T'B'T	false or true $	T → FT'
11	$B'T'B'T'F	false or true $	F → false
12	$B'T'B'T'false	false or true $	Match 'false'
13	$B'T'B'T'	or true $	T' → ε
14	$B'T'B'	or true $	B' → or TB'
15	$B'T'B'T or	or true $	Match 'or'
16	$B'T'B'T	true $	T → FT'
17	$B'T'B'T'F	true $	F → true
18	$B'T'B'T'true	true $	Match 'true'
19	$B'T'B'T'	$	T' → ε
20	$B'T'B'	$	B' → ε
21	$B'T'	$	T' → ε
22	$B'	$	B' → ε
23	$	$	Accepted

PARSE TREE



(3) The following context-free grammar G defines the syntax of a simple programming language in which declarations and assignments can be arbitrarily mixed:

Pgm → Program Lines end
Lines → Line; Lines | Line
Line → varid | id := id

Extend G to an attributed grammar which checks that every identifier occurring in an assignment is declared somewhere in the program. To this aim specify
- the association of attributes to symbols,
- the domains of the attributes, and
- the attribute equations for each production.

Here you can assume that the terminal symbol id is equipped with a synthesised attribute 'name' whose value is the name of the respective identifier.

Ans) To extend the given context-free grammar G into an attributed grammar that checks whether every identifier used in an assignment is declared somewhere in the program, we need to define attributes, their domains, and equations for each production rule.

Grammar G:

Pgm ⇒ Program Lines end
Lines → Line; Lines | Line

Line → varid | id := id

Non Terminals in the Attributed Grammar :

→ Pgm: Represents the entire program
→ Lines: Represents a sequence of lines in the program
→ Line: Represents an individual line of code in the program.
→ id: Represents an identifier.

Terminals in the Attributed Grammar:

→ var: A keyword used for declarations
→ := : The assignment operator
→ id : An identifier (with the synthesised attribute 'name')

Attributes : (Association to symbols)

→ For Pgm
- Pgm.declared: A set of declared identifiers
- Pgm.valid: A boolean indicating whether the program is valid

→ For Lines
- Lines.declared: A set of declared identifiers upto this point.
- Lines.valid : A boolean indicating if all lines in the program are valid with respect to identifier declarations.

→ For Line
- Line.declared: A set of identifiers declared in this line.

- Line.valid : A boolean indicating whether this line is valid in terms of declared identifiers.

→ For id :

- id.name : The name of the identifier (synthesised attribute).

Domains of the attributes :

→ Pgm.declared : Set of strings
→ Lines.declared : Set of strings
→ Lines.valid : Boolean
→ Line.declared : Set of strings
→ Line.valid : Boolean
→ id.name : String

Attribute Equations for each production :

⇒ Pgm → Program Lines end

- Pgm.declared = Lines.declared
- Pgm.valid = Lines.valid

⇒ Lines → Line ; Lines

- $Lines_1$.declared = Line.declared ∪ $Lines_2$.declared
- $Lines_1$.valid = Line.valid ∧ $Lines_2$.valid

⇒ Lines → Line

- Lines.declared = Line.declared
- Lines.valid = Line.valid

⇒ Line → varid

- Line.declared = {id.name}
- Line.valid = true

⇒ Line → $id_1 := id_2$

- Line.declared = $\emptyset$
- Line.valid = (id_1.name ∈ Lines.declared) ∧ (id_2.name ∈ Lines.declared)

Explanation

⇒ Pgm accumulates the set of declared identifiers from Lines and checks the overall validity.

⇒ Lines recursively accumulates declared identifiers and checks the validity of assignments.

⇒ Line → varid adds the declared identifier to the set.

⇒ Line → $id_1 := id_2$ checks that both identifiers on the left and right sides of the assignment are in the declared set.

Synthesised attribute for id:

The attribute id.name for any terminal id represents the name of the identifier, which is directly assigned as the string value of the identifier. This synthesised attribute helps track which identifiers are declared and which are used in assignments. This ensures that any identifier used is declared earlier in the program, maintaining the consistency and correctness of the program's syntax.

(4)(a) What do you mean by machine dependent and machine independent optimisation?

Ans) Machine dependent optimisation

It refers to the techniques in compiler optimisation that take into consideration the architecture of the target machine, which includes instruction set, register set and memory hierarchy. It aims at generating code that runs as effectively as possible on a specific hardware architecture.

Some examples include:

→ Instruction scheduling: Involves reordering instructions to take advantage of the pipeline architecture of the CPU.

→ Register allocation: Efficient use of CPU registers to minimise memory access during execution.

→ Loop Unrolling: Adjusting the number of loop iterations to minimise loop overhead and take advantage of instruction-level parallelism.

Illustration: Suppose a compiler is generating code for a RISC-based architecture that has specific sets of registers and a limited number of

addressing modes. The compiler may choose instructions tailored for that architecture to make best use of available registers and instruction cycles.

Machine independent optimisation

It refers to the optimisation techniques that improve the performance of a program without being dependent on the underlying hardware architecture or operating system. It focusses on improving the code in a way that can be applied across all platforms.

Some examples include:

→ Constant folding: It evaluates constant expressions at compile time rather than runtime, thereby saving execution time.

→ Common subexpression elimination: It identifies expressions computed multiple times and computes them once, storing the result for reuse. It removes duplicate calculations in the code.

→ Dead code elimination: Removes code that never gets executed and does not affect the program's output.

Illustration:

```
int a = 3 * 4; // Constant folding replaces this with int a = 12;
```

(b) Define synthesised attribute and inherited attribute with examples using grammar rules and annotated parse tree.

Ans) **Synthesised attribute**

It is an attribute whose value is computed from the values of the child nodes in a parse tree. It is used in Bottom-Up parsing, where attributes are passed upwards.

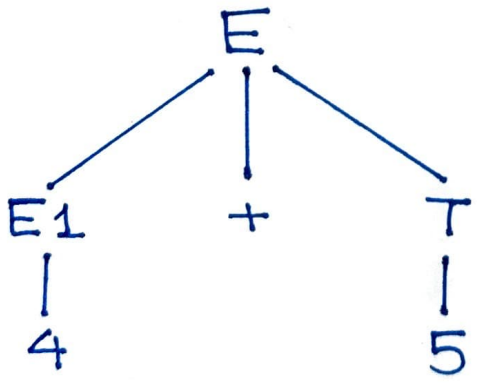
Example:

Let us consider the grammar rule

$E \rightarrow E1 + T \quad \{E.val = E1.val + T.val\}$

Here E.val is a synthesised attribute because its value is computed from its children E1.val and T.val.

Parse Tree: (for expression 4+5)



If E1.val = 4 and T.val = 5, then

E.val = 4+5 = 9

Inherited attribute

It is an attribute whose value is assigned from the parent or sibling nodes in a

parse tree. It is used in top-down parsing, where attributes are passed down or across the tree.

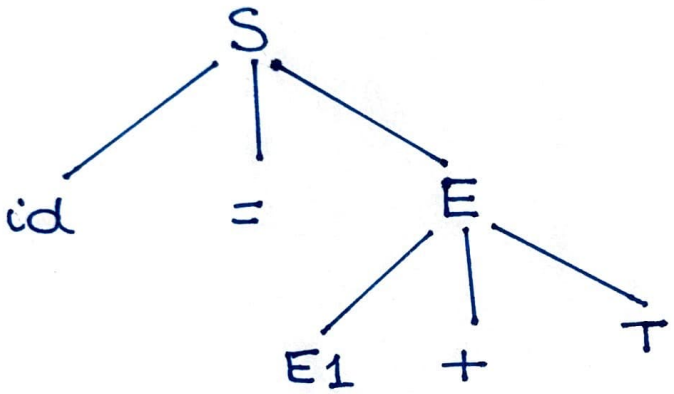
Example:

Let us consider the grammar rule:

$S \rightarrow id = E \quad \{E.in = id.type\}$

E.in is an inherited attribute passed down from S.

Parse Tree: (for assignment x=y+z)



If id.type is 'int', then E.in inherits 'int'.

(c) Define S-attributed definitions and L-attributed definitions. State the rule related to attribute dependencies.

Ans) In compiler design, syntax-directed definitions are used to specify the semantic rules for a context-free grammar (CFG).

S-attributed definitions

It is a type of syntax-directed definition where only synthesised attributes are used. A synthesised attribute is an

attribute whose value is computed from the attributes of its children in the parse tree and passed to its parent node. S-attributed definitions are evaluated using post-order traversal (bottom-up) of the parse tree, which aligns with the operations of an LR parser.

Example:

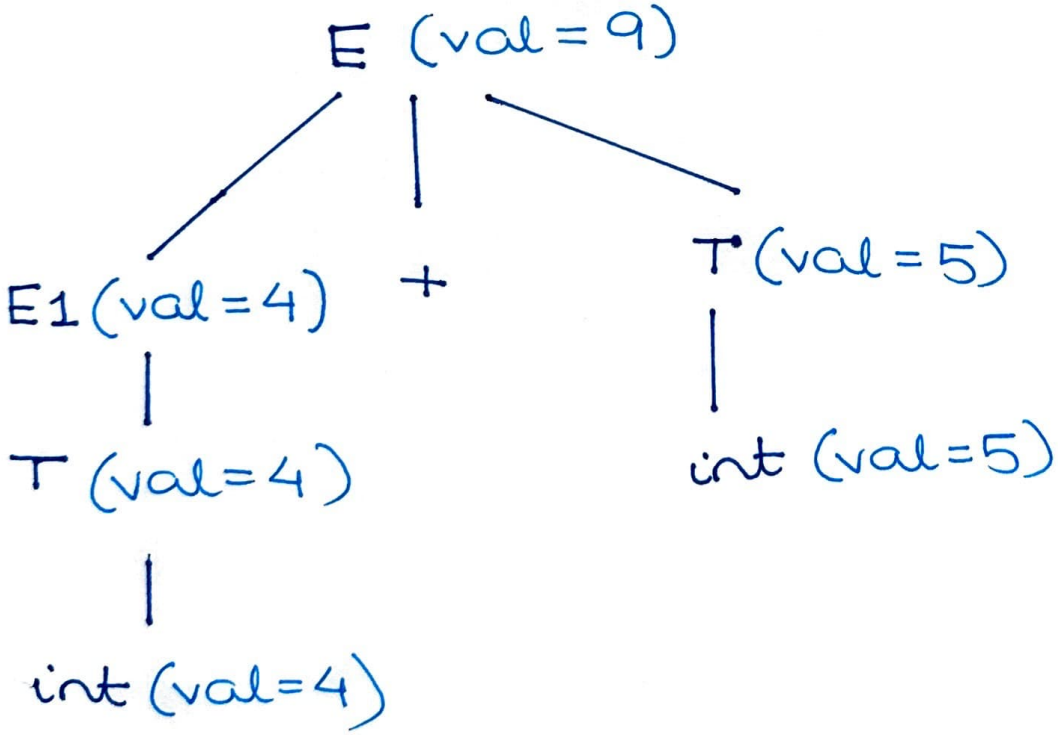
Let us consider the grammar:

$E \rightarrow E1 + T$ {E.val = E1.val + T.val}

$E \rightarrow T$ {E.val = T.val}

$T \rightarrow int$ {T.val = int.value}

Parse Tree:



The synthesised attribute 'val' for the non-terminal E is computed bottom up.

L-attributed definitions

An L-attributed definition is more flexible than S-attributed definitions, allowing both synthesised and inherited attributes. Inherited attributes are passed from a parent node to its children or between siblings. This type of attribute is useful for operations where the value of a node depends on the context set by its ancestors or siblings.

L-attributed definitions can be evaluated during a single left-to-right traversal of the parse tree, making them compatible with top-down parsing (LL parsing).

Rule: In L-attributed definitions, an attribute $X_i.a$ of a grammar symbol X_i in the production $A \rightarrow X_1 X_2 \dots X_n$ can depend on:

→ The inherited attributes of A (i.e. A.i)
→ The inherited attributes of any X_j to the left of X_i (i.e. $X_1.i, X_2.i, \dots, X_{i-1}.i$)
→ The synthesised attributes of X_j to the left of X_i (i.e. $X_1.S, X_2.S, \dots X_{i-1}.S$)

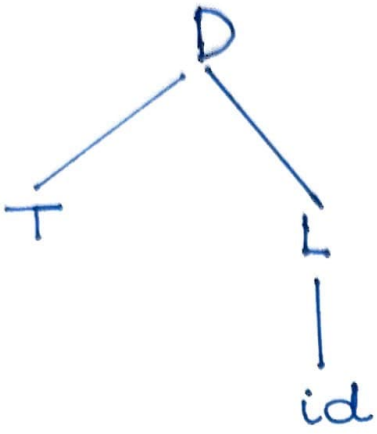
Example:

Let us consider the following grammar:

$D \rightarrow TL$ { L.type = T.type }

$L \rightarrow id$ { id.type = L.type }

Parse Tree:



The T.type attribute is inherited by L.type and passed to id.type.

Rule related to Attribute Dependencies

The attribute dependency rule states that attributes can be evaluated if:

→ Synthesised attributes: The value of a synthesised attribute of a non-terminal X depends only on the attribute values of its children.

→ Inherited attributes: The value of an inherited attribute of a non-terminal X depends on:

- Attributes of its parent node or ancestors
- Attributes of its siblings that appear to the left of X in the derivation.