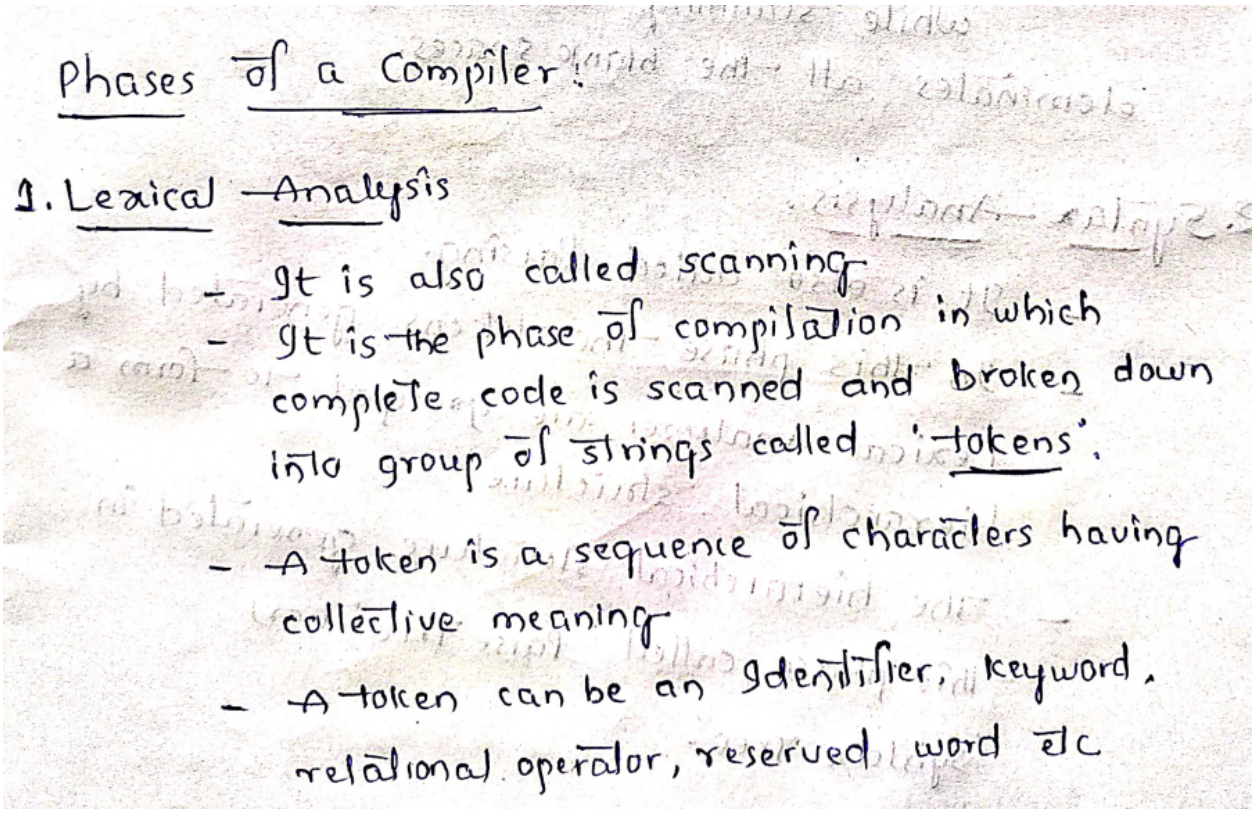
CD LONG ANSWERS

- Explain the various phases of a compiler in detail. Also Write down the output for the
  following expression after each phase total = count + rate * 60.

ANS:



eg:- total := count + rate * 60

The Lexical Analysis phase scans the given string from left to right and broken down into series of tokens.

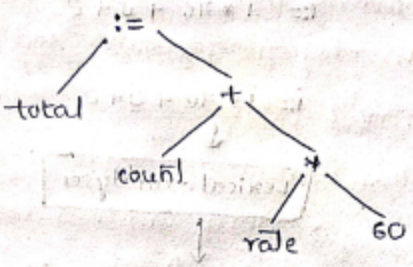
total	–	identifier
:=	–	operator
count	–	identifier
+	–	operator
rate	–	identifier
*	–	operator
60	–	constant

While scanning the string, this phase eleminates all the blank spaces.

2. Syntax Analysis:

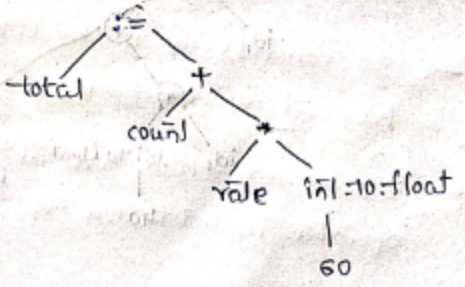
- It is also called Parsing
- In this phase, the tokens generated by Lexical Analyzer are grouped to form a hierarchical structure.
- The hierarchical structure generated in this phase called "Parse Tree" (or) "Syntax Tree".

For the expression total := count + rate * 60
the parse tree can be generated as follows:



3. Semantic Analysis:

- Once the syntax is checked and parse tree is generated.
- The semantic analysis determines the meaning of the source string
- For example, meaning of source string means matching of "parenthesis" in the expression, (or) matching of "if .... else" statements (or) performing arithmetic operations of the expr. that are type compatible (or) checking the scope of operation.



4. Intermediate Code Generation:

- Intermediate code is a kind of code which is easy to generate. and this code is easily converted to target code.
- This code is in variety of forms such as
  - * three address code
  - * quadraple
  - * triple
  - * posix
- Here, we will consider 'three address code' as intermediate code.

eg:-
$t_1 :=$ int to float (60)
$t_2 :=$ rate * t_1
$t_3 :=$ count + t_2
total = t_3

There are certain properties which should be possessed by three address code.

- It consists of instructions each of which has at the most three operands.
- Each instruction has at the most one operator in addition to assignment.

- The compiler must generate a temporary name to hold the value computed by each instruction.
- Some instructions may have fewer than three operands for example first and last instructions of above given three address code.

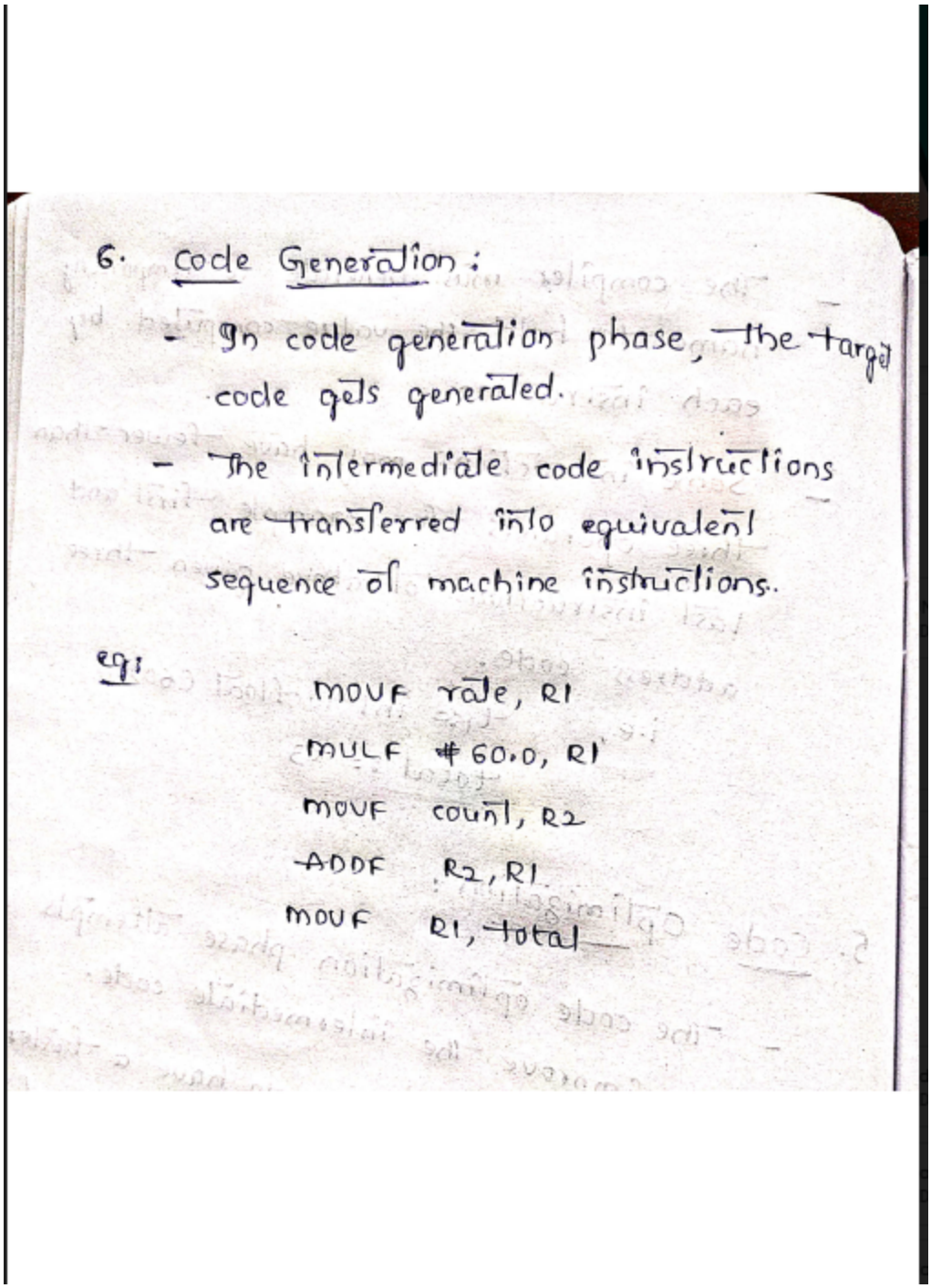
i.e., $t_1 :=$ int to float (60)
total $:= t_3$

5. Code Optimization:

- The code optimization phase attempts to improve the intermediate code.
- Thus, it is necessary to have a faster executing code (or) less consumption of memory
- Optimizing the code means, overall running time of target program can be improved.

eg:

$t_1 :=$ rate * 60.0
$t_2 :=$ count + t_1
total $:= t_2$



6. Code Generation :

- In code generation phase, the target code gets generated.
- The intermediate code instructions are transferred into equivalent sequence of machine instructions.

eg:

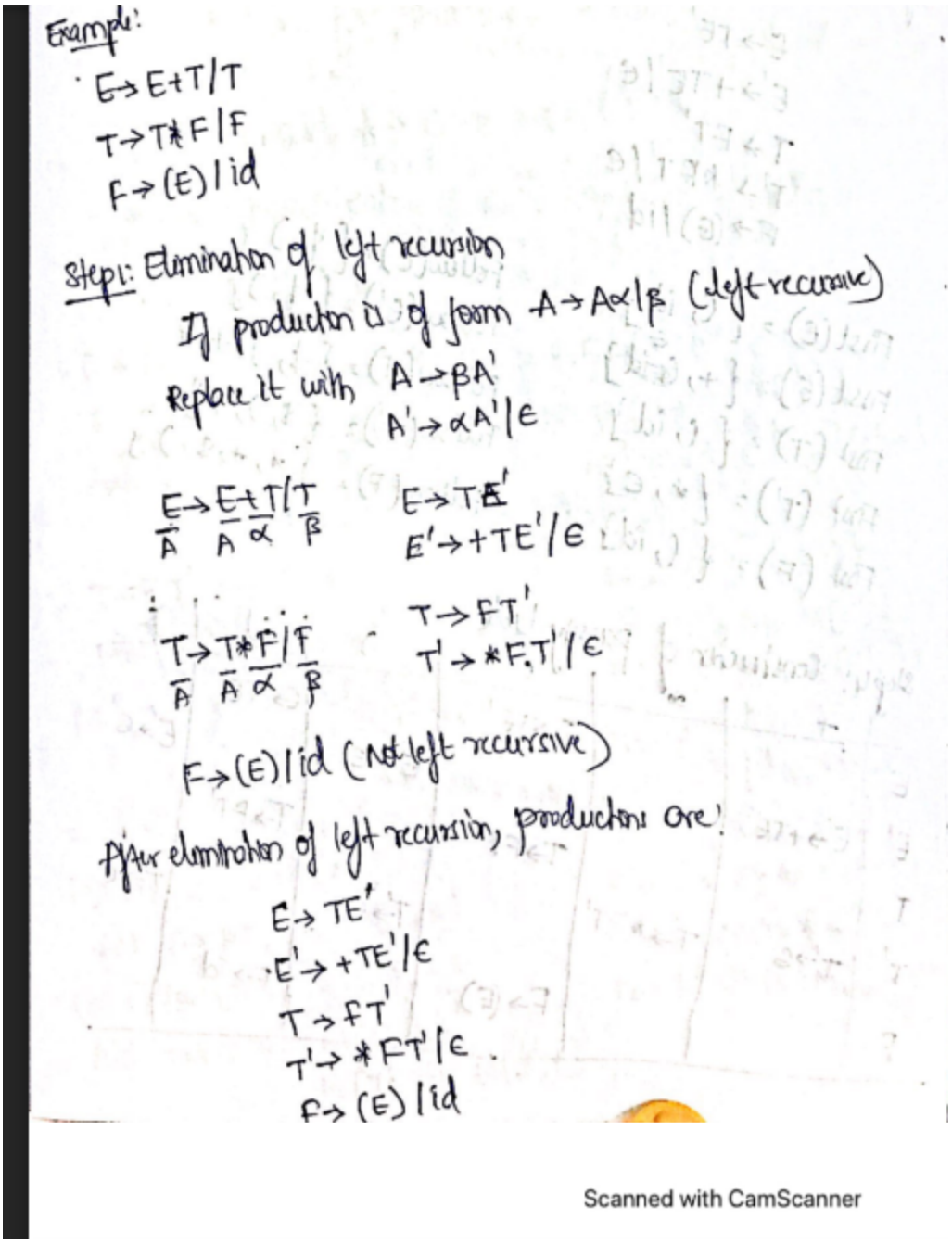
```
MOVF rate, R1
MULF #60.0, R1
MOVF count, R2
ADDF R2, R1
MOVF R1, total
```

• Write down the steps to compute CLOSURE and GOTO Functions.

ANS:

- Construct Predictive Parsing Table for the following Grammar
  E - > E + T | T
  T - > T * F | F
  F - > (E) | id

ANS:



Example:

- $E \to E+T/T$
  $T \to T*F/F$
  $F \to (E)/id$

Step1: Elimination of left recursion

If production is of form $A \to A\alpha/\beta$ (left recursive)

Replace it with $A \to \beta A'$
$A' \to \alpha A'/\epsilon$

$E \to \underset{A}{E}\underset{A}{+}\underset{\alpha}{T}/\underset{\beta}{T}$ $\quad E \to TE'$
$E' \to +TE'/\epsilon$

$T \to \underset{A}{T}\underset{A}{*}\underset{\alpha}{F}/\underset{\beta}{F}$ $\quad T \to FT'$
$T' \to *FT'/\epsilon$

$F \to (E)/id$ (Not left recursive)

After elimination of left recursion, productions are:

$E \to TE'$
$E' \to +TE'/\epsilon$
$T \to FT'$
$T' \to *FT'/\epsilon$
$F \to (E)/id$

Scanned with CamScanner

Step 2: Elimination of left factory

$A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 | \ldots$ (Left factory)

$A \to \alpha A'$

$A' \to \beta_1 | \beta_2 | \beta_3$

There is no left factory in the productions.

Step 3: Calculation of first and follow:

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) | id$

First(E) = { (, id }

First(E') = { +, ~~ϵ~~ ϵ }

First(T) = { (, id }

First(T') = { *, ϵ }

First(F) = { (, id }

Follow(E) = { $,) }

Follow(E') = { $,) }

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

Follow(T') = { $,), + }

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

Step 4: Construction of parsing table

	+	*	(	)	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			$F \to (E)$		$F \to id$	

Rules for construction of parsing table:

$A \to \alpha$

1. If the production is of $A \to \alpha$
   Calculate First(α), ↳ it produces terminal symbol a
   then, add $A \to \alpha$ to $M[A, a]$
2. If First(α) prod contains ϵ (or) $A \to \epsilon$ (epsilon)
   then, we have to calculate Follow(A) ↳ produces terminal b.
   then, add $A \to \epsilon$ to $M[A, b]$
3. The remaining entries of the parsing table are filled with ~~error~~ error.

1. $E \to TE'$
   First(TE') = {(, id}
   Add $E \to TE'$ to M[E, (], M[E, id]

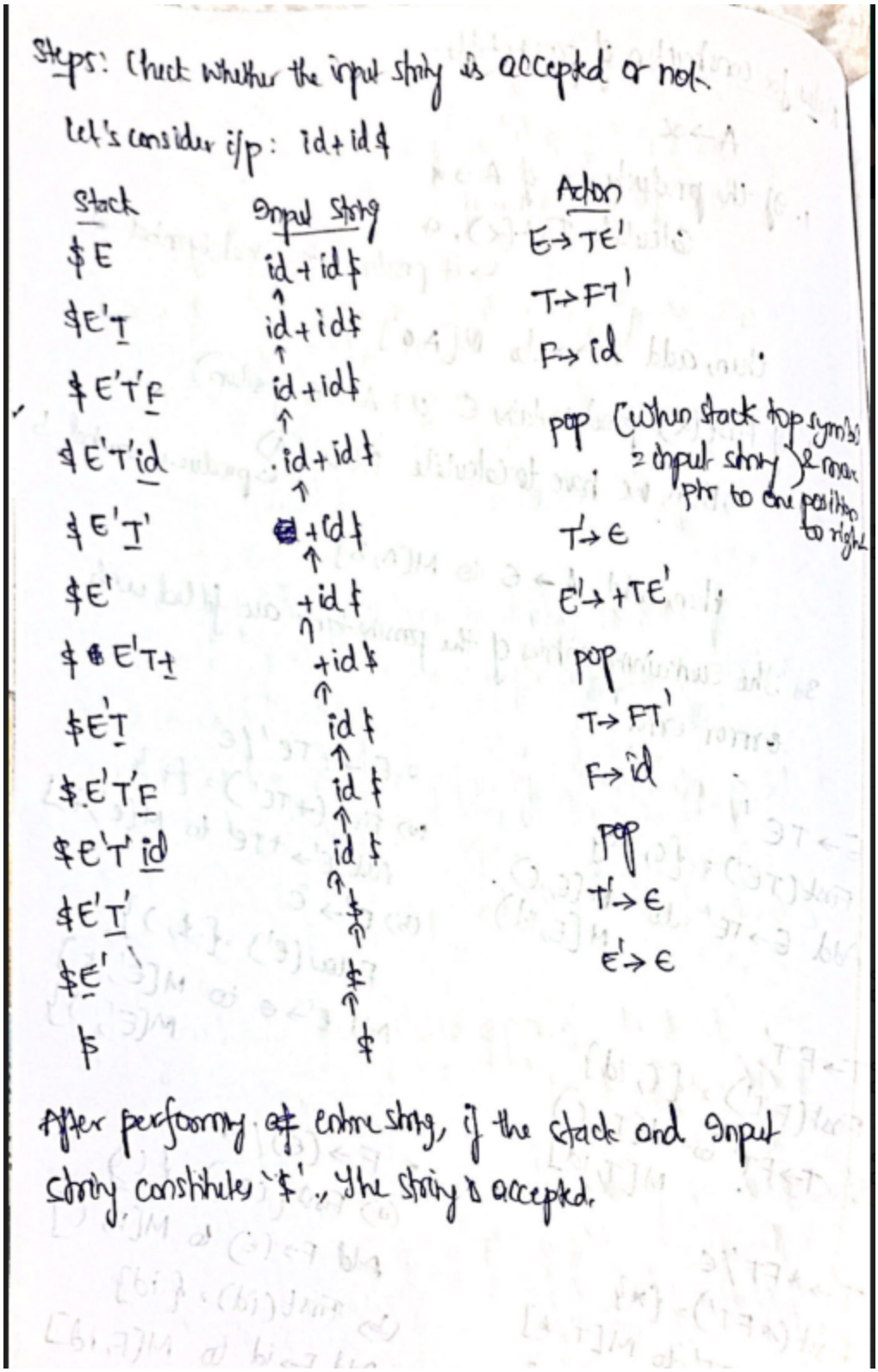
2. $E' \to +TE' / \epsilon$
   (a) First($+TE'$) = {+}
   Add $E' \to +TE'$ to M[E', +]
   (b) $E' \to \epsilon$
   Follow(E') = {$,)}
   Add $E' \to \epsilon$ to M[E', $], M[E',)]

3. $T \to FT'$
   First(FT') = {(, id}
   Add $T \to FT'$ to M[T, (], M[T, id]

4. $T' \to *FT' / \epsilon$
   (a) First($*FT'$) = {*}
   Add $T \to *FT'$ to M[T', *]
   (b) Follow(T') = {$,), +}
   Add $T \to \epsilon$ to M[T', $], M[T',)], M[T', +]

5. $F \to (E) / id$
   (a) First((E)) = {(}
   Add $F \to (E)$ to M[F, (]
   (b) First(id) = {id}
   Add $F \to id$ to M[F, id]

Scanned with CamScanner



Steps: Check whether the input string is accepted or not.

Let's consider i/p: id+id$

Stack	Input String	Action
$E	id+id$	E→TE'
$E'T	id+id$	T→FT'
$E'T'F	id+id$	F→id
$E'T'id	id+id$	pop (when stack top symbol = input string) & move ptr to one position to right
$E'T'	+id$	T'→ϵ
$E'	+id$	E'→+TE'
$E'T+	+id$	pop
$E'T	id$	T→FT'
$E'T'F	id$	F→id
$E'T'id	id$	pop
$E'T'	$	T'→ϵ
$E'	$	E'→ϵ
$	$	

After performing entire string, if the stack and input string constitutes '$', the string is accepted.

• Explain the differences between L-attribute and S-attribute definitions with example.

ANS:

S-attributes (Synthesized attributes):

- S-attributes are evaluated and synthesized during a bottom-up traversal of the parse tree.
- They depend only on attributes of the node's children.
- S-attributes can be evaluated in any order as long as all child attributes are computed before the parent attribute.
- Example:

  E → E + T { E.val = E.val + T.val }
  E → T { E.val = T.val }
  T → T * F { T.val = T.val * F.val }
  T → F { T.val = F.val }
  F → (E) { F.val = E.val }
  F → id { F.val = id.lexval }

  L-attributes (Inherited attributes):

  - L-attributes are evaluated and synthesized during a top-down traversal of the parse tree.
  - They depend on attributes of both the node's parent and its children.
  - L-attributes must be evaluated in a specific order, typically from the root down to the leaves.
  - Example

E → E + T { E.val = E.val + T.val }
E → T { E.val = T.val }
T → T * F { T.val = T.val * F.val }
T → F { T.val = F.val }
F → (E) { F.val = E.val }
F → id { F.val = id.lexval }

• Explain the role of Lexical Analyzer and also list out issues of Lexical Analyzer

ANS:

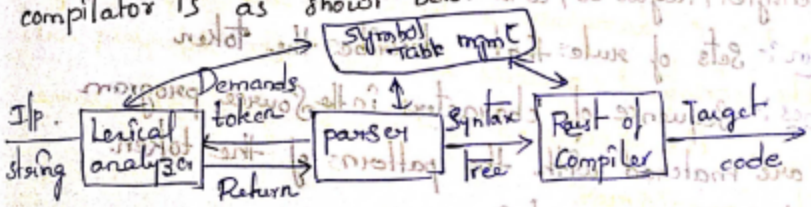
recovery.

=> Role of lexical analyzer:-

Lexical analyzer is the first phase of compiler. The lexical analyzer reads the i/p source program from left to right one character at a time & generates the sequence of tokens.

* Each token is a single logical choose unit such as identifier ; keyword , operators & punctionation marks. and so on.
* The parser we can use these tokens to determine the syntax of the source program.

The role of lexical analyzer in the process of compilator is as shown below:



Rule of lexical analyzer.

As the lexical analyzer scans the source program to recognise the tokens it is also called as Scanner.

* Apart from token identification, lexical analyzer also performs following functions) lb.

Functions of lexical analyzer:-

* It produces stream of tokens.
* It eliminates blanks & comments
* It generates symbol table which stores the information.

about identifier, constants encountered in the i/p.
* It keeps track of line numbers
* It reports the error encountered while generating the tokens
* The lexical analyzer works in 2 phases.
In 1st phase it performs scan & in the 2nd phase it does lexical analysis.
Means it generates the series of tokens.

Token, pattern, lexems:- terminology
Let us learn some technologies which are frequently used when we talk about the activity of lexical analysis

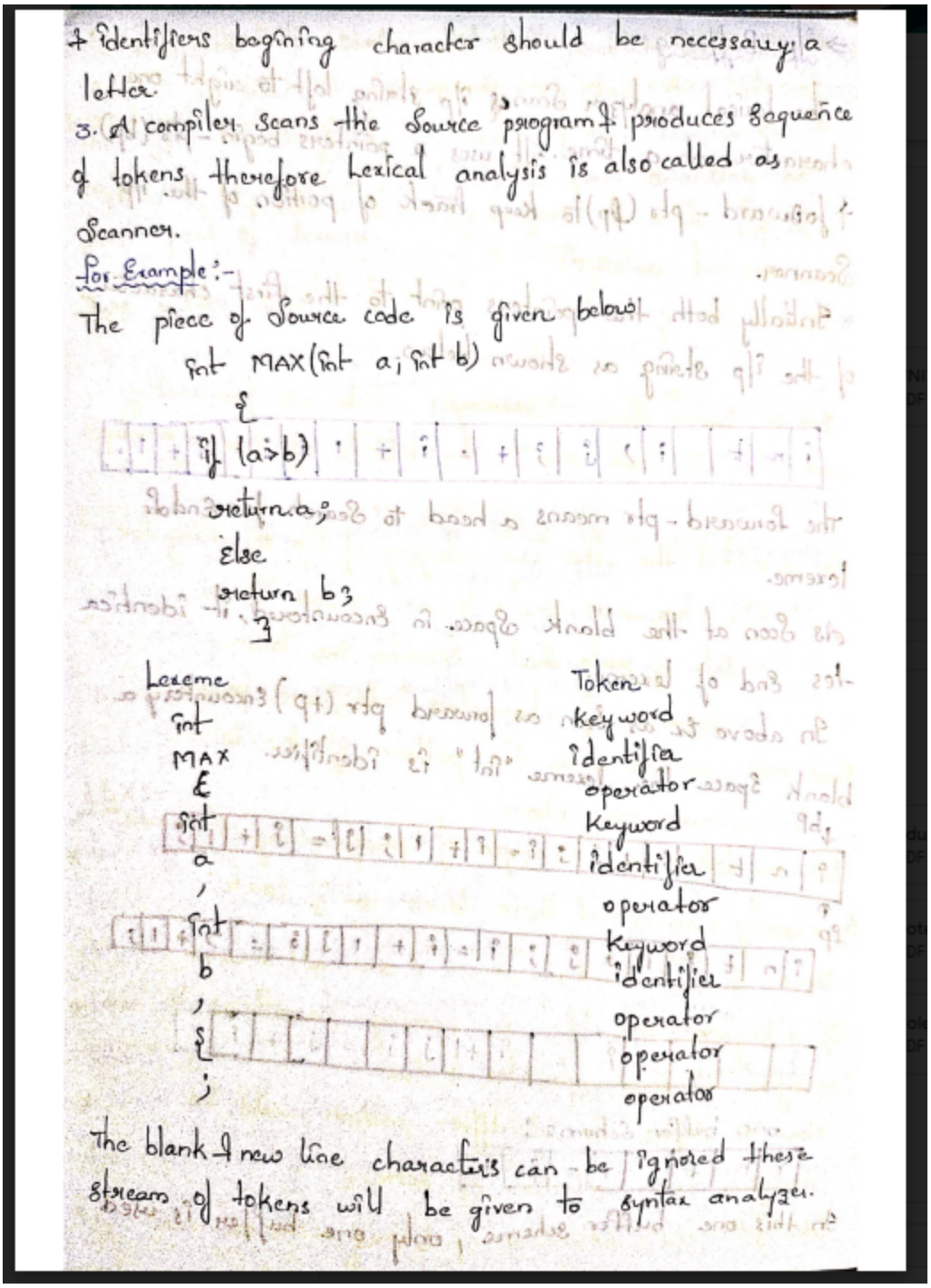
Tokens:-
It describes the class or category of i/p string.
Ex: identifier, keywords, constants are called tokens

pattern:- sets of rules that describes the token

Lexemes:- Sequence of characters in the source program that are matched with the patterns of the token.
Ex:- int i, num, ans, choice.

Let us taken one example of programing statement to clearly understand these terms per if (a<b).
Here "if", "(", "a", "<", "b", ")" are all lexemes.
* If a keyword, '(' is opening parathesis, "a" is identifier "<" is a operator & soon.

Now to define the identifier pattern could be
1. identifier is a collection of letters.
2. identifier is a collection of letters & alpha numeric character



Issues of Lexical Analyzer:

1. **Ambiguity**: Ambiguous languages can lead to difficulties in tokenization. For example, in some languages, whether a sequence of characters should be considered a single token or multiple tokens may not be clear.

2. **Efficiency**: Lexical analysis can be computationally intensive, especially for large programs. Efficient algorithms and data structures are needed to handle this.
3. **Error Reporting**: Properly reporting errors, especially when tokens are not recognized, is crucial for developers. Clear and concise error messages are essential.

• Explain Specification of LEX with example.

ANS:

2.17 Lex specifications:

A Lex program (the .l file) consists of three parts:

declarations
%%
translation rules
%%
auxiliary procedures

1. The *declarations* section includes declarations of variables,manifest constants(A manifest constant is an identifier that is declared to represent a constant e.g. *# define PIE 3.14*), and regular definitions.
2. The *translation rules* of a Lex program are statements of the form :

p1	*{action 1}*
p2	*{action 2}*
p3	*{action 3}*
...	...
...	...

where each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern *p* matches a lexeme. In Lex the actions are written in C.

3. The third section holds whatever *auxiliary procedures* are needed by the *actions.*Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

- Construct Canonical set of items LR(0) for the following grammar and design DFA for
  the items.
  E - > E + T | T

T - > T * F | F
F - > (E) | id

ANS: