1. **Why is buffering used in lexical analysis? What are the commonly used buffering methods?**

   The input character is read from secondary storage, but reading in this way from secondary storage is costly. hence buffering technique is used.A block of data is first read into a buffer, and then second by lexical analyzer.
   There are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme.

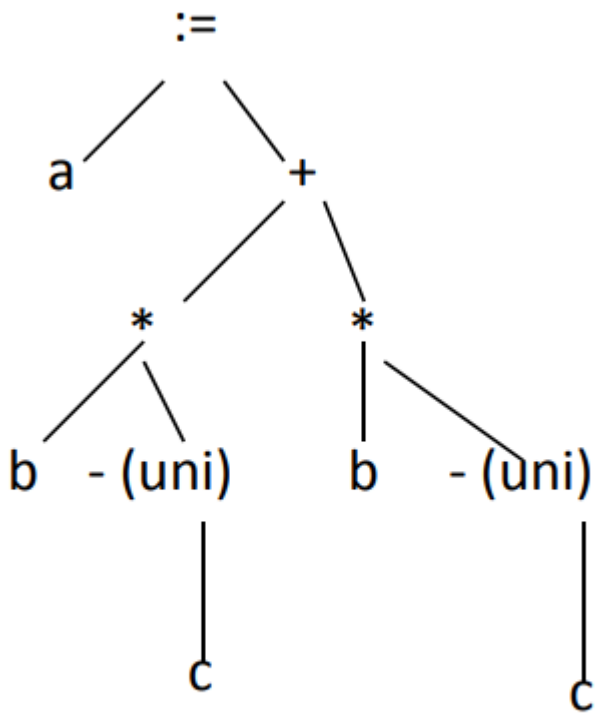
2. **What are the various representations of Intermediate languages?**
   - Post fix notation
   - Syntax tree
   - 3 address code

3. **An arithmetic expression with unbalanced parenthesis is lexical or syntax error. Comment on it.**

   Lexical error
   Errors mentioned below are detected during Lexical phase:
   - Exceeding length of identifier or numeric constants.
   - The appearance of illegal characters
   - Unmatched string

4. **Describe the language denoted by the regular expressions b*ab*ab*ab*.**
   Language containing exactly 3 a's with any number of b's.

5. **Construct the syntax tree for the following assignment statement: a:=b*-c+b*-c.**



6. Obtain the regular expressions for the following sets:

a) The set of all strings over {a, b} beginning and ending with 'a'.

b) The set of all strings over {b} such that string belong to {b2 , b5 , b8 , …}.

a) a(a+b)*a

b) bb(bbb)*

7. Identify the type of error in the following statements.

y = 3 + * 5; --> syntax error(2 operators are used by violating the syntax)

int a = "hello"; --> semantic error(type mismatch)

Justify this with the suitable points.

8. Comment on the efficiency of the compiler if the number of passes in compilation is increased.

9. What are the various representations of Intermediate languages?

- Post fix notation
- Syntax tree
- 3 address code

10. A compiler that translates a high-level language into another high-level language is called a source- to-source translator. What advantages are there to use C as a target language for a compiler?

- low level,
- easy to generate,
- can be written in an architecture-independent manner,
- highly available,
- has good optimisers.

11. Comment on the efficiency of the compiler if the number of passes in compilation is increased

12. Would it be better if the phases of the compiler are combined into a single phase?

13. Differentiate compilers from interpreters.

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Command languages use interpreters. For instance BASIC uses interpreter and Java uses both interpreter and compiler.	Programming language like C and C++ use compilers.

14. Explain how a language is processed with the suitable diagram and assume the c/c++ language

Language processing system

15. Obtain the regular expressions for the following having Σ={0,1}.

a) Strings of 0's and 1's beginning with 0 and ending with 1.

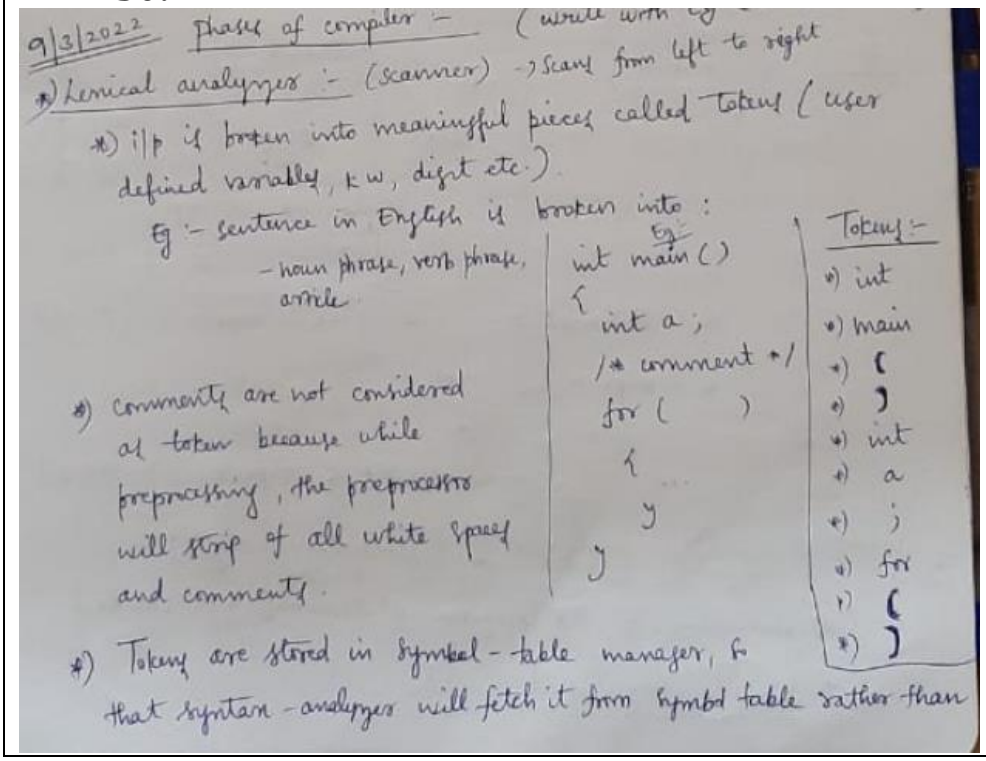
b) {x | x contains an even number of 0's or an odd number of 1's}.

a) 0(0+1)*1

b) (00)*+(11)*1

16. Generate the target code for the following code fragment. Assume a=(b+c) * (b+c) * as input to the lexical analyzer and try to show the intermediate outputs of all the phases from lexical analyzer to code generator.

17. Discuss the working of various phases of Compiler. Interpret the output for each phase for the following assignment statement a: = b + c * 50.



depending on lexical analyzer.

*) Syntax analyzer :- (parser)

process → (Syntax analysis | hierarchical)

*) Tokens are grouped to form a valid sequence.

*) CFG specifies the rules | procedures for the constructs.

*) identifies or verifies whether it is valid or not.

*) tokens are formed to hierarchical struct - parse tree.

< sentence > := < NP >< VP >

< NP > := article < noun >

< article > := a | an | the

< VP > := < verb >< NP >

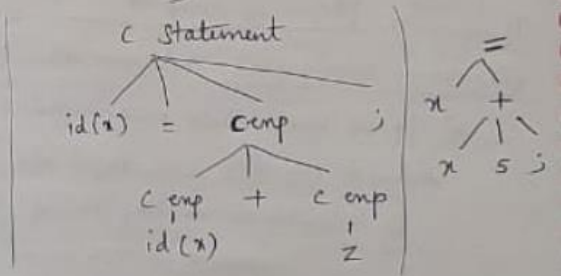
article, a, an, the → non-terminals

Remaining → non-terminals

→ production rules.

Eg:- x = x + 5 ;

x, =, x, +, 5, ;



< C Statement > := id "=" < C expr > ; (x = a;)

< C expr > := const (x = 0)

:= id (x = a)

:= < c expr > + < c expr > (x = x + y)

5/2022 *) Semantic analyzer :- (semantic analysis)

*) Semantics of the program statements are checked.

*) procedures invoked.

*) i/p received from Symbol table ( ), and lexical analyzer ( ).

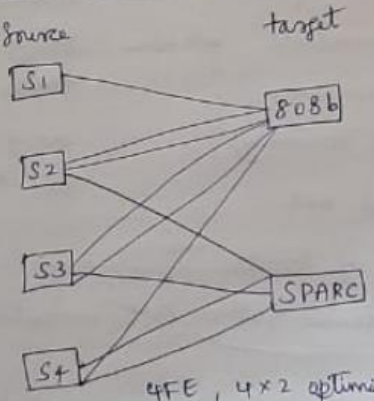
*) functions :-

→ Type checking
→ definite assignment.
→ object binding.
→ flow of control checks.

*) Eg :-

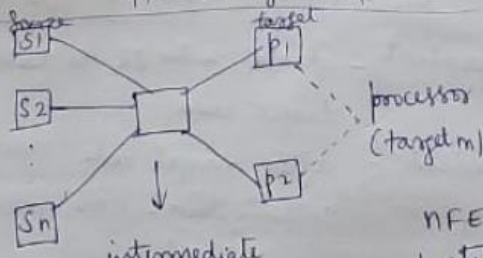
```
int i;
i + 1;
int s = "hello";
int a = 5 - s;
```

*) Intermediate code generation :-



*) It generates intermediate code, which is a form that can be readily executed by machine.

4FE, 4×2 optimizer
4×2 code generators



intermediate code

nFE
1 optimizer
2 code generators

Advantages :- *) portability (can be transferred from one system to another).
*) Retargetting.

3 representations of intermediate code generation :-

*) post fix notation (eg, a b +).
*) Syntax tree
*) 3- address code.
(2 → arguments, 1 → result)

Eg :- $x = a + b$ $\quad t_1 = a$
$x = t_3$ $\quad t_2 = b$
$t_3 = a + b$.

*) Code optimization :-

Why ?
→ It is a program transformation to improve the intermediate code by making it consume fewer resources (CPU, memory).
→ Optimization should not change the meaning of source.
→ no delay is entertained rather faster execution is expected.

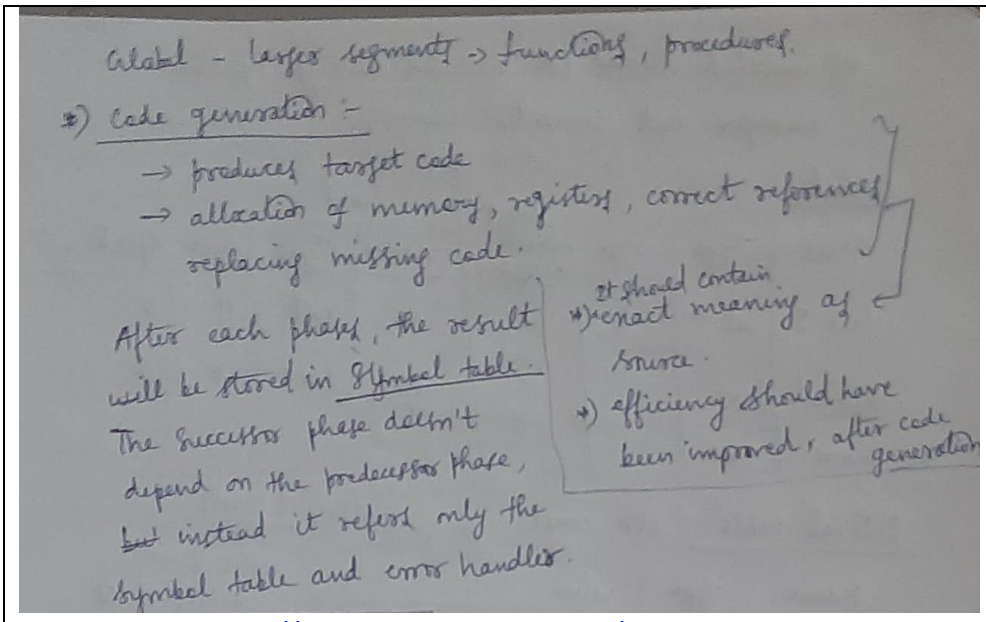
*) pass - reading i/p (entire source) program once.
*) phase - getting the part of target compilation. one of the process to get target program.

Types :- M/c → machine ; IR → intermediate code.

M/c independent - deals with IR to obtain better target code.

M/c dependent - after target code optimization - CPU, memory, registe

Local - small segments performed (prior to global).



For example : https://www.geeksforgeeks.org/working-of-compiler-phases-with-example/?ref=rp

In web: x = a+b*50

a:=b+c*50(given question)

Chanage the variables accordingly.

18. A compiler can choose one of the two options.

a)Translate the input source into intermediate code and then convert it to final machine code.

b) Directly generate the final machine code from the input source.

What is the preferred option and why? Justify this with the suitable claims.

Option a is correct.

(Language processing system / cousins of compiler)

19. a.Compare tokens, patterns and lexemes.

Token	Pattern	Lexeme
id	A letter followed by letters or digits	Salary, name, age, var1, a
const	Letters coming in exact sequence of "const"	const
Integer_num	Sequence of digits with at least one digit	1234, 500, 3
Floating_num	Sequence of digits with embedded period (.) at one digit on the either side	5.2, 23.45. 567.22
Relational_op	String >, <, >=, <=, !=, ==	>, <, >=, <=, !=, ==
literal	Any sequence of characters enclosed in double qutations	"core dumped"

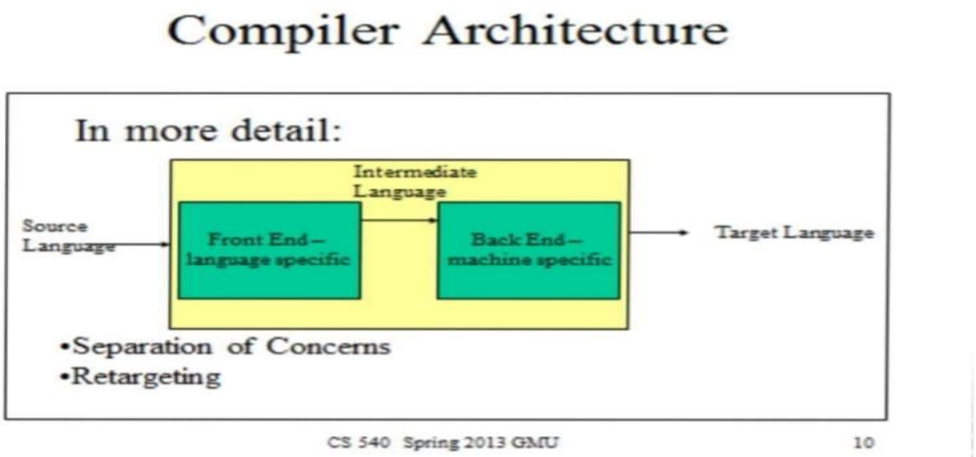
b.Count the number of tokens in the following code snippet.

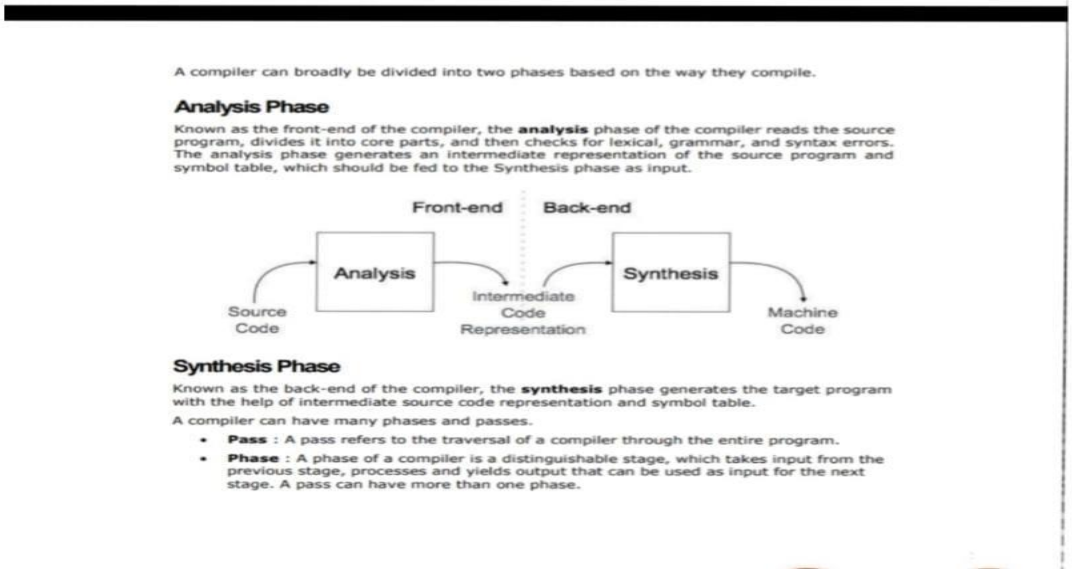
int main()

```
{
a=10,b=20;
printf(" sum is : %d", a+b;
return 0;
}
```

Ans: 25

20. Explain in detail the structure of the compilers. Compare language dependent and independent phases.





The compiler has two modules namely the front end and the back end. Front-end constitutes the Lexical analyzer, semantic analyzer, syntax analyzer, and intermediate code generator. And the rest are assembled to form the back end.

Language dependent:

Lexical analysis
Syntax analysis
Semantic analysis
Intermediate code generation

Language independent:

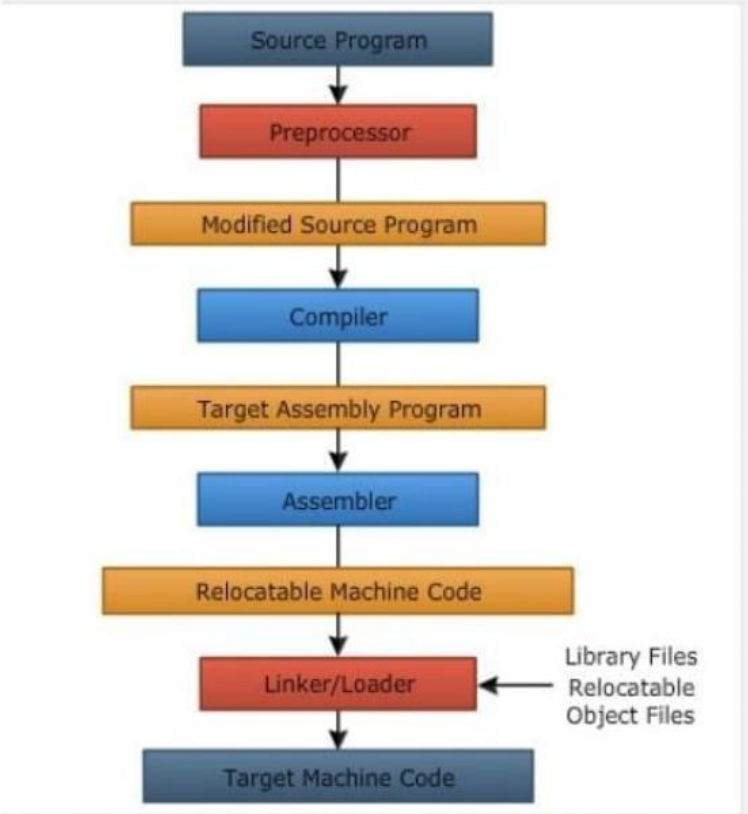
Code optimization
Target code generation

✕ **Add points from qn.17**

21. How to solve the source program to target machine code by using language processing system?

Introduction :

The computer is an intelligent combination of software and hardware. Hardware is simply a piece of mechanical equipment and its functions are being compiled by the relevant software. The hardware considers instructions as electronic charge, which is equivalent to the binary language in software programming. The binary language has only 0s and 1s. To enlighten, the hardware code has to be written in binary format, which is just a series of 0s and 1s. Writing such code would be an inconvenient and complicated task for computer programmers, so we write programs in a high-level language, which is Convenient for us to comprehend and memorize. These programs are then fed into a series of devices and operating system (OS) components to obtain the desired code that can be used by the machine. This is known as a language processing system.



Language Processing System

Components of Language processing system :

High-Level Language − If a program includes #define or #include directives, it is known as HLL.

Preprocessor

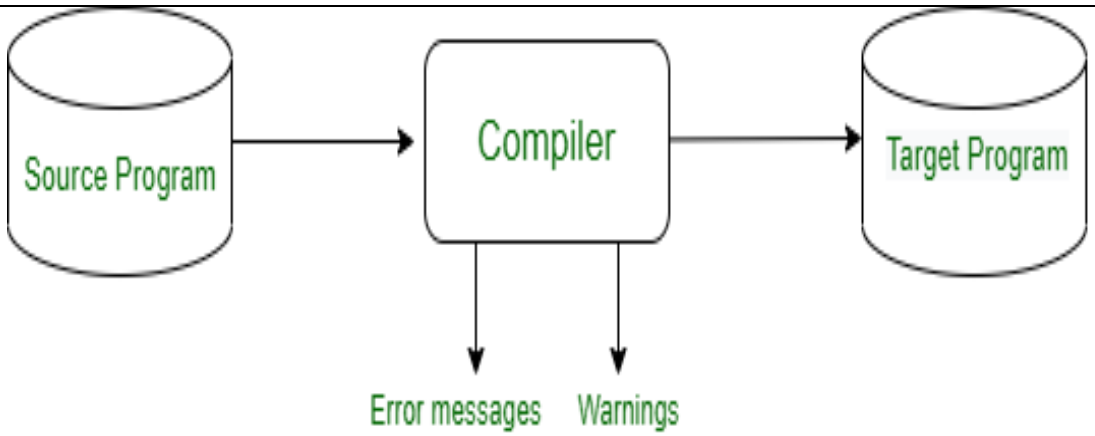
The pre-processor terminates all the #include directives by containing the files named file inclusion and all the #define directives using macro expansion. A pre-processor can implement the following functions −

- **Macro processing** − A preprocessor can enable a user to define macros that are shorthands for higher constructs.
- **File inclusion** − A preprocessor can include header files into the program text.
- **Rational preprocessor** − These preprocessors augment earlier languages with additional current flow-of-control and data structuring facilities.
- **Language Extensions** − These preprocessors try to insert capabilities to the language by specific amounts to construct in macro.

Pure HLL − It means that the program will not contain any # tags. These # tags are also known as preprocessor directives.

Compiler –

The compiler takes the modified code as input and produces the target code as output.



Assembler − Assembler is a program that takes as input an assembly language program and changes it into its similar machine language code.
Assembly Language − It is an intermediate state that is a sequence of machine instructions and some other beneficial record needed for implementation. It neither in the form of 0's and 1's.

Advantages of Assembly Language

- Reading is easier.
- Addresses are symbolic & programmers need not worry about addresses.
- It is mnemonics. An example we use ST instead of 01010000 for store instruction in assembly language.
- It is easy to find and correct errors.

Relocatable Machine Code − It means that you can load that machine code at any point in the computer and it can run. The address inside the code will be so that it will maintain the code movement.
Loader / Linker −

- **Linker**: A linker or link editor is a program that takes a collection of objects (created by assemblers and compilers) and combines them into an executable program.
- **Loader**: The loader keeps the linked program in the main memory.

Differences between Linker/Loader:
The differences between linker and loader as follows.

Linker	Loader
The linker is part of the library files.	The loader is part of an operating system.
The linker performs the linking operation.	The loader loads the program for execution.
It also connects user-defined functions to user-defined libraries.	Loading a program involves reading the contents of an executable file in memory.

Functions of loader :
Allocation –
It is used to allocate space for memory in an object program. A translator cannot allocate space because there may be overlap or large wastage of memory.
Linking –
It combines two or more different object programs and resolves the symbolic context between object decks. It also provides the necessary information to allow reference between them. Linking is of two types

as follows.

Static Linking :

It copies all the library routines used in the program into an executable image. This requires more disk space and memory.

Dynamic Linking :

It resolves undefined symbols while a program is running. This means that executable code still has undefined symbols and a list of objects or libraries that will provide definitions for the same.

Reallocation –

This object modifies the program so that it can be loaded to an address different from the originally specified location, and to accommodate all addresses between dependent locations.

Loading –

Physically, it keeps machine instructions and data in memory for execution.

22.Why do we separate the analysis phase into lexical and parsing phases?

Explain role of lexical analysis (same as qn.no: 29)

23.int fact;

```
int factorial (int n)
{
    int val;
if(n>1){
  val=n*factorial (n-1);
return (val);
}
else
{
return(1);
}
}
int main()
{ printf ("factorial program");
Fact 5=factorial(5);
Printf("fact=5=%d \n", fact5);
}
```

Consider the above program, what are the data structures used to store the information, how the symbol table management is handled?

Global Symbol Table

Name	Type	Size	Offset
fact5	INT	4	0
factorial	FUNC		
main	FUNC		

Local Symbol Table for Factorial ()

Name	Type	Size	Offset
n	INT	4	0
val	INT	4	4

```
int fact5;
int factorial (int n)
{
   int val;
   if (n>1){
     val=n*factorial (n-1);
     return (val);
   }else{
      return(1);
   }
}
int main ()
{
    print ("factorial program\n");
    fact5=factorial (5);
    printf ("fact5=%d \n",fact5);
}
```

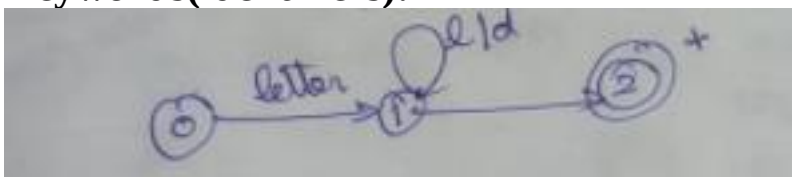
Literal Table

Name	Type	Value
lit1	STRING	"Factorial Program\n"
lit2	CONST	5
lit3	STRING	"fact5=%d \n"

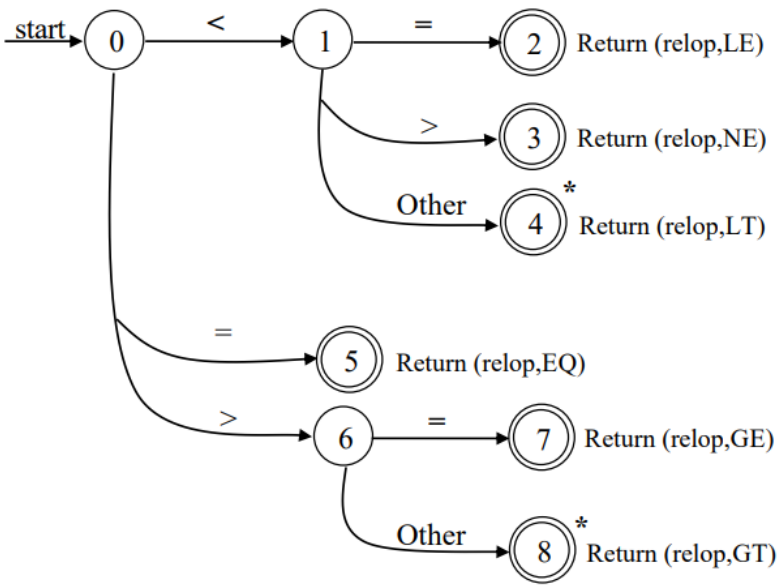
Fig. 1.11 *Symbol table and literal table for a sample input source*

24. Construct transition diagram for keyword, identifiers and relational operators.

Keywords(identifiers):



Relational operators:



Transition Diagram for relation operators

25. Obtain the required DFA from the regular expression (a/b)* a (a/b) using Minimization of DFA using Π_{new} constructions. Write the suitable algorithm.

26. Explain in detail the structure of the compilers. Compare language dependent and independent phases

Refer qn.no: 20

27. Obtain the NFA from the following regular expression (a+b) abb (a/b)* using Thompson construction. Convert it into DFA and then minimize the DFA. Describe the sequence of moves made by each in processing the input string ababba.

28. What are the advantages and disadvantages of intermediate language?

29. Explain the role performed by lexical analysis of the compiler

30. Write down the possible error recovery actions taken by lexical analyzer

Refer qn.no: 34

31. Define the purpose of gettoken( ) function

- The GetToken function recognizes explicit spaces, tabs, or newline characters as the parameter delimiters.
- gettoken() examines the lexeme and returns the token name, either id or a name corresponding to

a reserved keyword.

32.Define the purpose of installid( ) function

- installID() checks if the lexeme is already in the table.
- If it is not present, the lexeme is installed as an id token. In either case a pointer to the entry is returned.

33.State the difference between linear, hierarchical and syntax analysis.

✕ Refer qn.18

34.Explain Symbol table management and error handling

Symbol Table Management

• Symbol table is used to store all the information about identifiers used in the program.

• It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.

• It allows finding the record for each identifier quickly and to store or retrieve data from that record.

• Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

Example

int a, b; float c; char z;

Symbol name	Type	Address
a	Int	1000
b	Int	1002
c	Float	1004
z	char	1008

Error handling:

1. Panic Mode Recovery

- In this method, successive characters from the input are removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are deli-meters such as; or }
- The advantage is that it's easy to implement and guarantees not to go into an infinite loop
- The disadvantage is that a considerable amount of input is skipped without checking it for additional errors

2. Statement Mode recovery

- In this method, when a parser encounters an error, it performs the necessary correction on the remaining input so that the rest of the input statement allows the parser to parse ahead.
- The correction can be deletion of extra semicolons, replacing the comma with semicolons, or

inserting a missing semicolon.

- While performing correction, utmost care should be taken for not going in an infinite loop.
- A disadvantage is that it finds it difficult to handle situations where the actual error occurred before pointing of detection.

3. Error production

- If a user has knowledge of common errors that can be encountered then, these errors can be incorporated by augmenting the grammar with error productions that generate erroneous constructs.
- If this is used then, during parsing appropriate error messages can be generated and parsing can be continued.
- The disadvantage is that it's difficult to maintain.

4. Global Correction

- The parser examines the whole program and tries to find out the closest match for it which is error-free.
- The closest match program has less number of insertions, deletions, and changes of tokens to recover from erroneous input.
- Due to high time and space complexity, this method is not implemented practically.

35.Explain construction tools /cousins of compiler.

✕ Refer qn.21