M T W T F S S

Page No.:

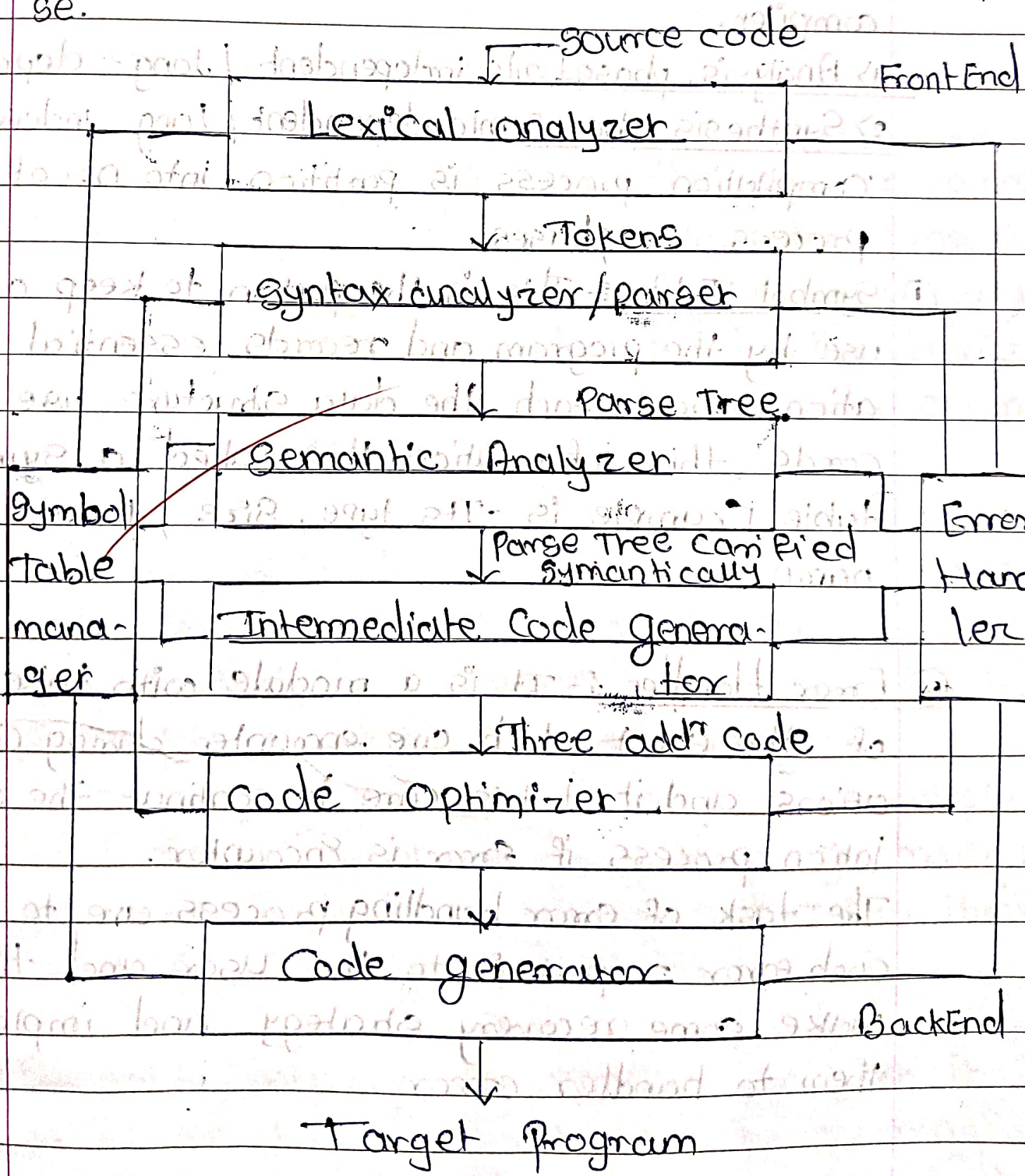
Date:

YOUVA

Assignment No : 01

(a) Draw and neat diagram show in various phases of compiler. explain in brief each phase.

⟹



A Compiler operates in phases a phase is logically interrelated operation that takes source prog. in one presentation and produces o/p in another presentation. There are two phases of compile user phases of compiler.

1) Analysis phase [mlc independent / lang - dependant.
2) Synthesis phase [mlc dependent / Lang. independent.

Compilation process is partition into no. of sub-process or phases.

① Symbol Table :- This is the portion to keep a name use by the program and records essential information about Each the data structure use to records this information this called a symbol table. Example is - Its type, Size, location, name.

② Error Handler :- It is a module with take care of the event which are encounter during compilations. and it takes care to continue the compilation process if Error is Encounter.

The task of error handling process are to detect each error report it to the user and then make some recovery strategy and implement them to handler error.

- phases of Compiler :

① Analysis phase / Front end :

① Lexical Analyzer
② Syntaxe Analyzer / Parser
③ Semantic Analyzer
④ Intermediate code generator

1) Lexical Analyzer : It read the prog. or source code and convert it into tokens by using tool called "LEX". Tokens are define by regular Expression which are understood by the lexical analyzer. It removes white spaces ' ', comments " // ", & tabs " \t ".

2) Syntax Analyzer / Parser : It will construct the parse tree take the token one by one and uses CFG [context free Grammer] to construct the parse tree. the i/p has to check whether is in the desire format or not. Syntax error can be detected by if the i/p is not according to the grammer given. so it takes i/p as tokens and checks their synthetic connection.

3) Semantic Analyzer : In Semantic analyzer it verify the parse tree whether its meaningful or not. it uses the parse tree and into in the

Symbol table to check the source prog. For Semantic consistancy with the program language definition it will check type checking difinitions of variable, use flow control checking i.e., it will be check program should be meaningful.

4) Intermediate code generator : An intermediate representation of the final m/c language code is produce. this phase bridges the analysis and synthesis phases of translator.

② Synthesis phase / Back-End :
① Code optimizer
② Code generator

1) Code optimizer : This is a optional phase describe to into the intermediate code so, that o/p ran fasters and takes less space.

2) Code generator : The last phase of translation is code generation. A number of optimization tools reduce the length of m/c lang. prog. are carried out during this space. the o/p of the code generation is m/c lang. prog. of the specified computer.

(02) List and explain various compiler Construction tools.

→ compiler Construction tools are :
1) Scanner Generator
2) parser Generator
3) Syntax directed translation Engine
4) Data flow Engine
5) automatic code Generator

1) Scanner Generator : This tool automatically generate lexical analyzer.

2) Parser Generator : This tools produce syntax analyzer. this parser generator normally take i/p that is base on a context free grammer [CFG] for example. YACC is a LALR parser generator an it available as a command on the UNIX System.

3) Syntax directed translation Engine : This produce collections of routines are walk the parse tree generating in a intermediate tools.

4) Data flow Engine : much of the information needed to perform good code optimization

involves "Data flow analysis". Here, the gathering of information above how values are transmitted from one part of the prog. is done. different task of this kind can be perform by the same kind of routine. the users supplies detail of the relationship bet intermediate code statements or info being gather.

5) automatic code generator: This tools takes a collection of tools that define the translation of each operation of the intermediate lang and changes then into the m/c lang. For the target m/c. the rules must includes sufficient details that we can handle for different possible access methods for data.

The basic technique is template matching the intermediate code stmts are replace by templates that represents sequences of m/c instructions, in such a way that the assumption above storage of variable match from template to template.

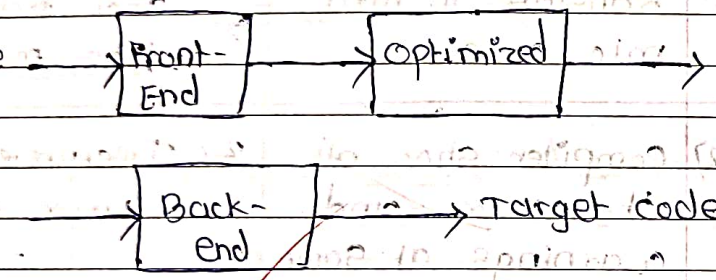
Q3) Difference between compiler & interpreter.

→

	Compiler		Interpreter
1)	It scan entire program first and then translate in into m/c code.	1)	It scan to program line by line and translate in into m/c code.
2)	Compiler show all the errors and warnings at same times.	2)	Interpreter show one error at a time.
3)	Errors occures after scanning the whole program.	3)	Errors occures after scanning the each line.
4)	Debugging is slow	4)	Debugging is high
5)	execution time is less	5)	execution time is more
6)	Compiler are large m/m size	6)	interpreter are smaller m/m size

Q4) Write a note on front-end and back-end models of compilation.

⟹ The phases of compilation are classified into front-end and back-end. this compiler orgnization is shown in fig.



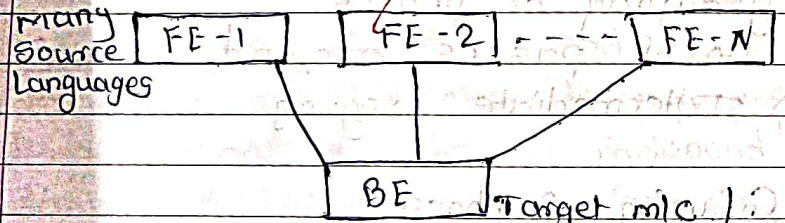
1) Front-End : The front-end consists of those phases that depends primarily source language and are largely independ of the target mlc. these normally include :-
① Lexical analyzer.
② Syntax analyzer / parser
③ Semantic analyzer
④ Creation of symbol table
⑤ Generation of intermediate code
⑥ Certain amount of code optimization
⑦ Error handling associated with this phase.

2) Back-End : The Backend includes those phase's of compilen that depend on target mlc the phases do not depend on a source language ! they used only the intermediate language. back-end normally includes:
① code optimization
② code generation
③ necessary symbol table management
④ required error handling.

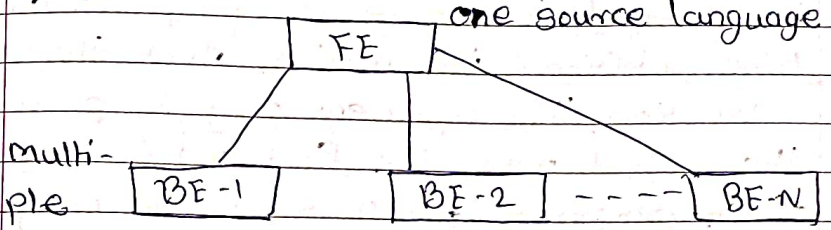
Q5) Explain following point with respect to compiler.
i) grouping of phases
ii) no. of passes required
iii) Intermediate language

⟹ i) Grouping of phases :
The different types of phases deals with the logical orgnization of the compiler in an implementation activities from several phases may be group together into a pass that reads an input files and writes o/p files. For example :- The front end phases of lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator. might be group together into one pass.

Code optimization may be an not obligatore in then there could be a backend pass consisting of code generation for a particular target m/c. design intermediate representation that allow the front-end for particular lang. to interface with the backend for a certain target m/c.
with this collections we can produce compilers for different source languages for one target m/c by combine diff. frontend with the backend for that target m/c. as shown in following fig.



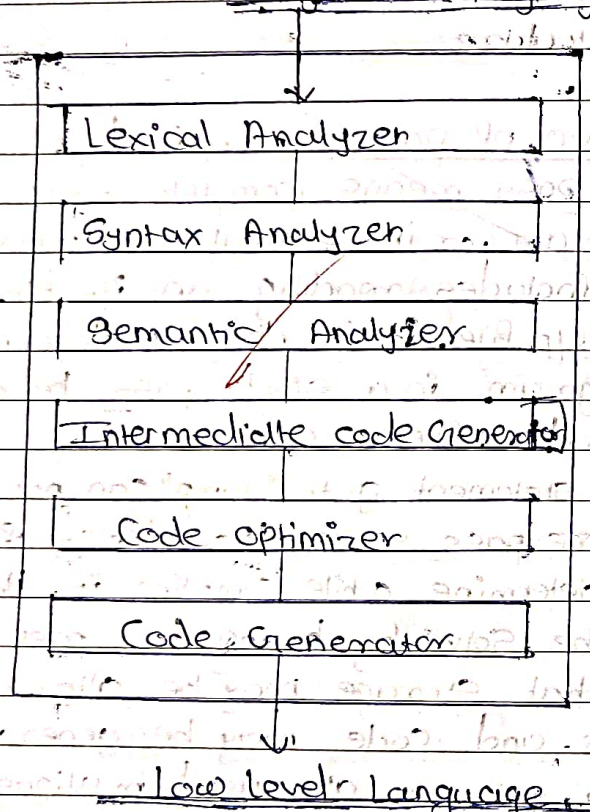
In the some way we can produce compilers for different target m/c by combining a front-end with back end for different target m/c. as shown in following fig.



- ii) No. of passes required :-

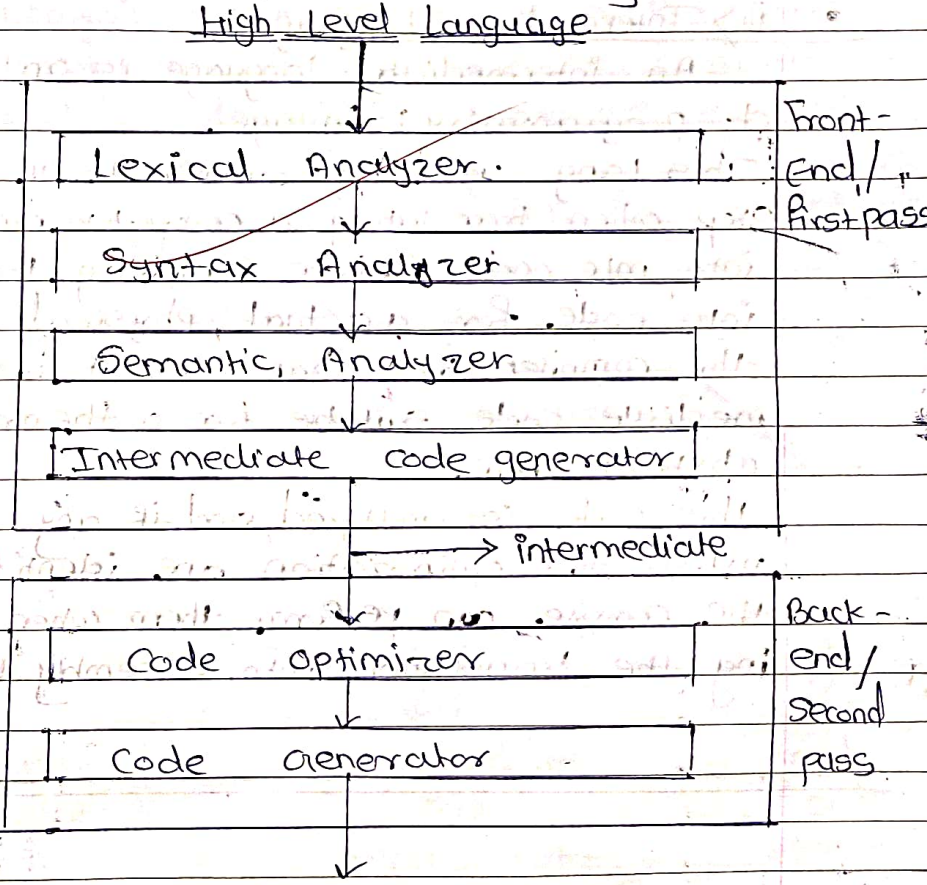
A pass means complete scan of the source program or its equivalent representation. it includes reading an i/p file & writting an o/p file. it is difficult to compiler source program in a single pass because of forward differences.
The statement goto "jump" can not be compile in one pass sence, the address of label cannot be determine while compiling it. this problem can be solve by having two pass compiler, so, that storage may be allocated in first pass and code may be generated it second pass, we can also used multipass compiler for reasons for storage limitation and optimization.
The first pass process the source program and produces and easy to used form which

is equivalent to the source program. and combine additional information, about the storage allocation this equivalent prog. is process by the second pass. the front end & backend Each comprises one pass of the compiler.



• Fig: Single/one pass compiler.

If we combine or group all the phases of compiler design in a single module know as single/one pass compiler. in above fig. there are all six phases are group in a single module. some important point of single pass compiler is an. A one-pass compiler is that type of compiler that passes through the part of each compilation unit exactly one.



• Fig: Multipass compiler.

It divided a large program into multiple small programs and process them. It develops multiple intermediate codes. So it required less memory it is known as 'wide compiler'.

iii) Intermediate Language:-

An intermediate language is an abstract programming language.
This lang. used by compiler as an in bet^n step when translating a computer program into m/c code. Before compiling the prog. into code for a actual, physical m/c, the compiler first translates it into intermediate code suitable for a theoretical, abstract m/c.
This code is analyzed, and if any opportunities for optimization are identified, the compiler can perform them when performing the translation into assembly language.

Q6) Describe Analysis and synthesis model of compiler.

⇒

i) Analysis model [Front-End]:-
- Analysis phase consists of starting it is m/c independent but language dependent.
- Analysis phase also known as front-End.
- Breaks the source program into constituent pieces and creates intermediate representation.
- Analysis phase divided into four sub phases:
① Lexical Analyzer
② Syntax Analyzer
③ Semantic Analyzer
④ Intermediate code generator.

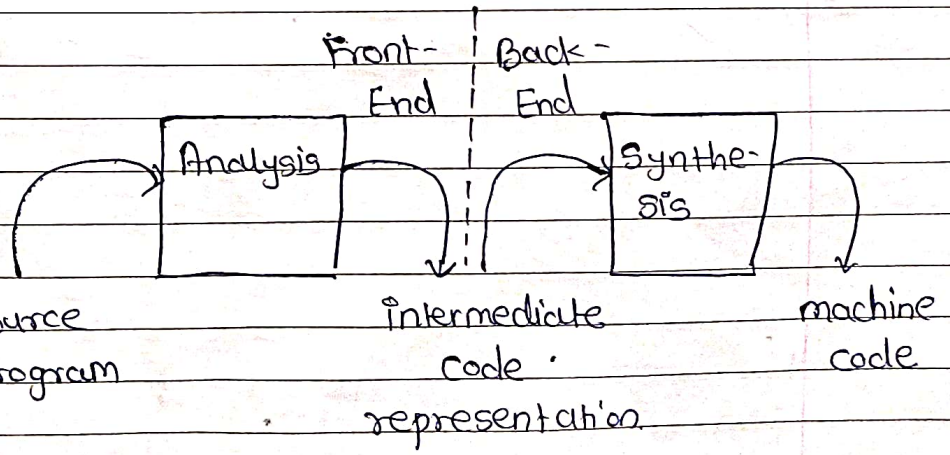


Fig. Analysis and synthesis model

ii) Synthesis model [Back-end]

- Synthesis phase generates the target program from the intermediate representation.
- Synthesis phase is mlc dependent and language independent.
- Synthesis phase also known as Back-End.
- Synthesis phase consists are two sub-phases.
  i) Code optimizer
  ii) code generator.

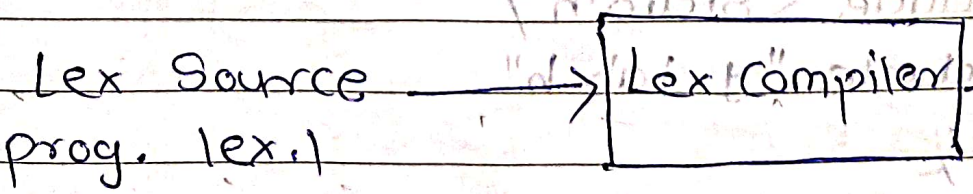
Assignment No : 2

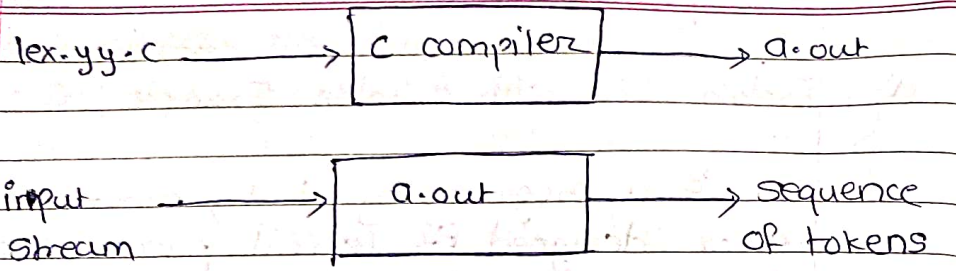
Q) Explain LEX with suitable Example?

⇒

LEX is a programming tool, program generator designed for lexical processing of character input / output stream. Anything from simple text search program that looks for pattern in its input - output file to a C - compiler that transforms a program into optimized code.

It is used with YACC parser generator. The lexical analyzer is a program that transforms an input stream into a sequence of tokens. Firstly lexical analyzer creates a program lex.l in the Lex language. Then Lex compiler runs the lex.l program and produces a C program lex.yy.c. Finally C compiler runs the lex.yy.c program and produces an object program a.out. a.out is lexical analyzer that transforms an input stream into a sequence of tokens.





- Example of LEX Source code :

This file contains include statements for standard input and output, as well as for the y.tab.h file. If you use the -d flag with the yacc command, the yacc program generates that file from the yacc grammer file information. The y.tab.h file contains definitions for the tokens that the parser program uses. In addition, the calc.lex file contains the rules to generate these tokens from the input stream. The following are the contents of the calc.lex file

```
%{
# include <stdio.h>
# include "y.tab.h"
int c;
%}
```

```
%%
" " ;
[a-z] {
        c = yytext[0];
        yylval.a = c - 'a';
        return (LETTER);
      }
[0-9] {
        c = yytext[0];
        yylval.a = c - '0';
        return (DIGIT);
      }
[^a-z0-9\b] {
        c = yytext[0];
        return (c);
      }
%%
```

(Q2) Define the term lexeme, token and pattern with Example?

⟹

① Lexeme :

It is a sequence of characters in the source code that are matched by given predefined language rules for every lexeme to be specified as a valid token.

Example :-
main is lexeme of type identifier (token)
(,), {, } are lexemes of type punctuation (token)

② Token :-
It is basically a sequence of characters that are treated as a unit as it cannot be further broken down. In programming languages like C language keywords (int, char, float, const, goto, continue, etc). identifiers (user-defined names), operators (+, -, *, %), delimiters / punctuators like comma (,), semicolon (;), braces ({}), etc. strings can be considered as tokens. This phase/ lexical analyzer phase recognizes three types of tokens:

i) Terminal symbols [TRM] - keywords and operators
ii) Literals [LIT]
iii) Identifiers [IDN]

Example :-
int a = 10; // input source code

Tokens - int (keyword), a (identifier), = (operator), 10 (constant) and ; (punctuation - semicolon)

③ pattern :-
It specifies a set of rules that a scanner follows to create a token.

Example of programming lan C, C++.
For a keyword to be identified as a valid token, the pattern is the sequence of characters that make the keyword.

Q3) Explain the role of regular Expression and DFA in lexical analyzer.

⇒

① Regular Expressions :-
Regular expressions have the capability to express finite lang. by defining a pattern for finite string of symbols.
The grammar defined by regular expressions is known as regular grammer.
The language defined by regular grammar is known as regular language.
Regular expression is an important notation for specifying patterns.

Each pattern matches a set of strings, so regular expressions serve as a names for a set of strings.

The specification of regular expression is an example of a recursive definition.

If r and s are regular expressions denoting the language L(r) and L(s), then.

i) Union : (r) | (s) is a regular expression denoting L(r) ∪ L(s)

ii) Concatenation : (r)(s) is a regular expression denoting L(r) L(s)

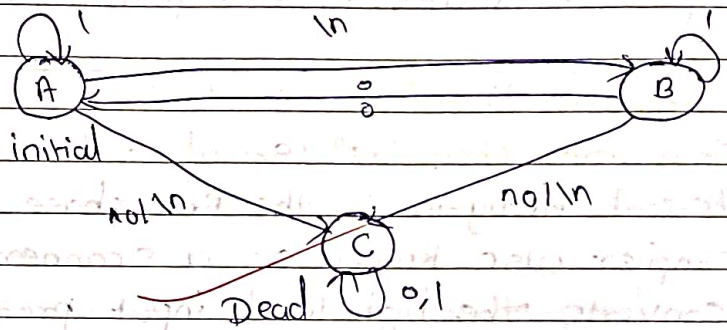
iii) Kleene closure :- (r)* is a regular expression denoting (L(r))*

(e) DFA [Deterministic Finite automata] :-

DFA is the simplest of all automata & capable of processing regular languages [Regular expressions]

In DFA, there is only one path for specific input from the current state to next state. DFA does not accept the null move, i.e., the DFA cannot change state without any input character. DFA can contain multiple final states. It is used in lexical analysis phase.

Example :-



(Q4) why white space characters are generally not allow in identifiers names, Justify?

⟹

white space characters are generally not allow in identifier name because of white space breaks up a programming language into a stream of symbols to be interpreted by a compiler or parser to make sense of it. Just like this paragraph.

It makes parsing easier if white space always terminates a symbol. otherwise you have to have some other rule,

which will be less comprehensible both to the compiler and, more importantly the maintainer.

Spaces where a space separating two identifiers may be meaningful. white space in names is not permitted.

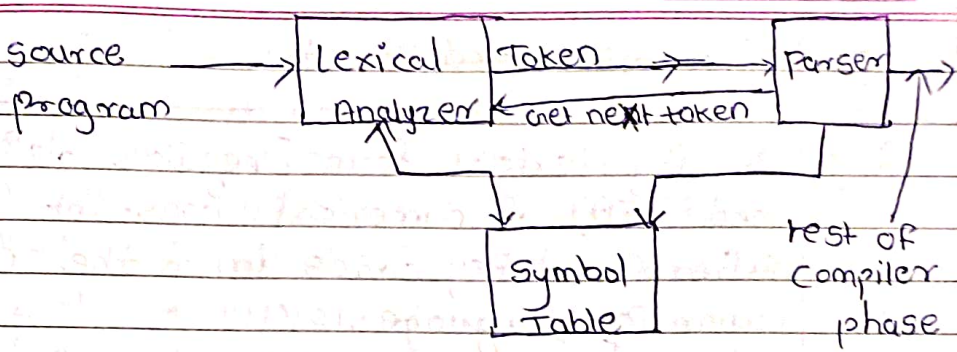
Q) Explain the role of lexical analyzer? Explain interaction bet" lexical and parser phase?

⇒ Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the high level input program into a sequence of Tokens.

Lexical analysis can be implemented with the Deterministic finite Automata. The o/p is a sequence of tokens that is sent to the parser for syntax analysis.

- How Lexical Analyzer functions:

1. Tokenization
2. Remove white space characters.
3. Remove comments.
4. It also provides help in generating error messages by providing row numbers and column numbers.



The above fig is the block semantic of lexical analyzer an shown the intraction between the lexical analyzer and the parser.

Lexical analyzer phase divided into a cascade of two parts.

① Scanning ② Lexical analysis

- Basic concept of lexical analysis are:
  i) Lexeme
  ii) Alphabets and tokens
  iii) pattern
  iv) lexical errors.

Lexical Analyzer phase perform operation on above concept and give will input next parser / syntax analyzer phase. Form tokens.

Q5) ii) ~~Explain interaction~~ ✗

Q6) Specify lexical form [regular definition and DFA] of numerical constant, identifiers and key words in in the C programming language.

⟹

~~a) numerical constant :-~~

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}
/* regular definitions */
delim    [ \t\n]
ws       {delim}+
letter   [A-Za-z]
digit    [0-9]
id       {letter}({letter}|{digit})*
number   {digit}+(\.{digit}+)?(E[+\-]?
                                (digit}+)?
%%
{ws}     {/* no action and no return */}
if       {return(IF);}
then     {return(THEN);}
else     {return(ELSE);}
{id}     {yylval = install_id(); return(ID);}
{number} {yylval = install_num(); return(NUMBER);}
"<"      {yylval = LT; return(RELOP);}
"<="     {yylval = LE; return(RELOP);}
"="      {yylval = EQ; return(RELOP);}
"<>"     {yylval = NE; return(RELOP);}
">"      {yylval = GT; return(RELOP);}
">="     {yylval = GE; return(RELOP);}
%%
install_id(){
  /* procedure to install the lexeme, whose
  first character is pointed to by yytext
  and whose length is yyleng, into the
  symbol table and return a pointer
  thereto */
}
install_num(){
  /* similar procedure to install a
  lexeme that is a number */
}
```

Q7) Explain the reference for lex Specifications

⟹ A Lex program consists of three parts:

i) The definition section
ii) The rule section
iii) The user subroutines.

The parts are separated by lines consisting of two percent signs. [%%]

1) The definition section:-

The definition section can includes the literal block, definitions, internal table declarations, start conditions, and translations. Lines that start with whitespace are copied verbatim to the C file. Typically this is used to include comments enclosed in "/*" to "*/", preceded by whitespace.

2) Rules section:-

The rules section contains pattern lines and C code. A line that starts with white space, or enclosed in "%{" & "%}" is C code. A line that starts with anything else is a pattern line.

3) User Subroutines :-

User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled Separately.

Assignment No :- 3

Q1) what is the necessary and sufficient condition for a grammar to be LL(1)? with respect to parsing explain the following terms.

a. Ambiguous grammar
b. Left Recursion
c. Shift - Reduce conflict
d. Precedence ~~Associativity~~

⇒ The necessary and sufficient condition for a grammar to be LL(1) is given below:

- condition 1 :-
  For the productions associated with each nonterminal X

  X → \alpha_1 pr(r_1)
  | ---
  | \alpha_n pr(r_n)

First (\alpha_i) \intersection First (\alpha_j) is empty whenever i =\= j. That is the bodies of

the productions must have disjoint First sets.

- Condition 2:
  For each nonterminal x and the production get associated with the non-terminal.

  $x \longrightarrow \alpha_1 \quad pr(x_1)$
  $| \ldots$
  $| \alpha_n \quad pr(x_n)$

  at most one α_i is nullable.

- Condition 3:
  If x is a nullable nonterminal then Follow (x) is disjoint from First (x).

a) **Ambiguous Grammar :-**
A CFG is said to ambiguous if there exists more than one derivation tree for the given input string i.e., more than one LeftMostDerivationTree [LMDT] or RightMost-Derivation Tree [RMDT].

b) **Left Recursion :-**
A grammar becomes left-recursive if it has any non-terminal 'A' whose derivation contains 'A' itself as the left-most symbol. Left-recursive grammar is considered to be a problematic situation for top-down parsers. Top-down parsers start parsing from the start symbol, which in itself is non-terminal. So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.
Example: $A \longrightarrow A\alpha | \beta$
is an example of immediate left recursion where A is any non-terminal symbol and α represents a string of non-terminals.

c) **Shift - Reduce conflict**
The Shift-Reduce conflict is the most common type of conflict found in grammars. It is caused when the grammar allows a rule to be reduced for a particular token, but, at the same time, allowing another rule to be shifted for that same token.

M T W T F S S
Page No.:
Date:
YOUVA

(d) Precedence, Associativity :

- precedence :-

If two different operators share a common operand, the precedence of operators decides which will take the operand. That is, 2+3*4 can have two different parse trees, one corresponding to (2+3)*4 and another corresponding to 2+(3*4).

- Associativity :-

If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.

(02) Differentiate bet" LR(0), canonical LR & LALR items.

→ The difference bet" LR(0), LALR, and LR is the following :

- An LALR parser is an "upgraded" version of an LR(0) parser that keeps track of more precise information to disambiguate the grammar. An LR parser is a significantly more powerful parser that keeps track of even more precise information than an LALR parser.

- LALR parsers are a constant factor larger than LR(0) parsers, and LR parsers are usually exponentially larger than LALR parsers.

- Any grammar that can be parsed with an LR(0) parser can be parsed with an LALR parser and any grammar that can be parsed with an LALR parser can be parsed with an LR parser. There are grammars that are LALR but not LR(0) and LR but not LALR.

Q3) Compare the top-down parsing and bottom-up parsing.

⟹

	Top-down parsing	Bottom-up parsing
1)	It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar.	It is a parsing strategy that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar.
2)	Top-down parsing attempts to find the left most derivations for an input string.	Bottom-up parsing can be defined as an attempt to reduce the input string to the start symbol of a grammar.
3)	In this parsing technique we start parsing from the top to down in a top-down manner.	In this parsing technique we start parsing from the bottom to up in a bottom-up manner.
4)	This parsing technique uses left most Derivation.	This parsing technique uses Right most Derivation.
5)	The main leftmost decision is to select what production rule to use in order to construct the string.	The main decision is to select when to use a production rule to reduce the string to get the starting symbol.
6)	Example: Recursive Descent parser.	Example: Its shift Reduce parser.

Q4) what is an operator precedence grammar? Explain how to construct precedence relations.

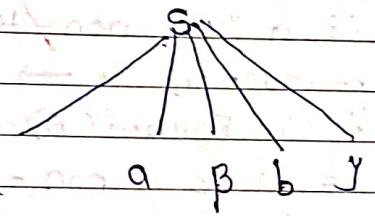
⟹

An operator precedence grammar is a kind of grammar for formal languages. Technically, an operator precedence grammar is a context-free grammar that has the property that no production has either an empty right-hand side or two adjacent non-terminals in its right-hand side. These properties allow precedence relations to be defined betⁿ the terminals of the grammar.

- Algorithm for precedence relations
- begin
- For each production $A \rightarrow B_1, B_2, \dots, B_n$.
  For $i = 1$ to $n-1$
  . If B_i and B_{i+1} are both terminals then
  set $B_i = B_{i+1}$
  IF $i < = n-2$ and B_i and B_{i+2} are both terminals and B_{i+1} is non-terminal then.
  set $B_i = B_{i+2}$
  IF B_i is terminal and B_{i+1} is non-terminal then for all a in LEADING(B_{i+1}).
  set $B_i \lessdot a$
  IF B_i is non-terminal and B_{i+1} is terminal then for all a in TRAILING(B_i).
  set $a \gtrdot B_{i+1}$
- end.

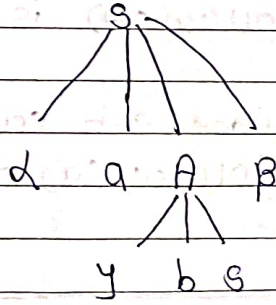
The precedence relations betⁿ terminals Symbols can also be shown by parse-Tree -

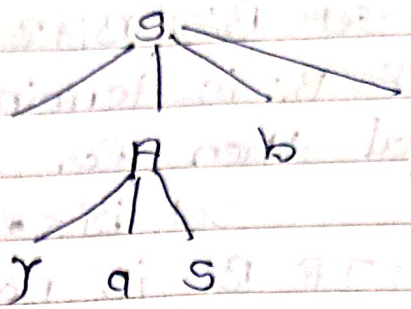
$a \doteq b$
a has the same precedence as b



$a \lessdot b$

a yields precedence to b



a ⇒ b a takes precedence over b	

Q5) write a algorithm for constructing FOLLOW set of a grammar.

⇒

- begin
- if a is a start symbol, then FOLLOW() = $
- if a is a non-terminal and has a production a ⟶ AB, then FIRST(B) is in Follow(A) except ∈.
- if a is a non-terminal and has a production a ⟶ AB, where B ∈, then FOLLOW(A) is in FOLLOW(a).

Follow set can be seen as:

FOLLOW(a) = {t | S* at* }