Name: B. Praneetha
RollNo: 22981A0513
Section: 2nd year (CSE-A)
Subject: Compiler design

Assignment-01

1) What is the role of lexical Analyzer? And explain about the lex tool with an example program.

A) Role of lexical analyser:

1) lexical analyser functions.
2) Separation of lexical Analyser from syntax analyser.
3) Tokens, lexemes and patterns.
4) lexical forms.

Lexical analyser:

→ It produces a stream of tokens and removes comments from the source program.

→ It provides error messages with corresponding Line no's and column no's.

→ Interaction blw lexical analyser and syntax analyser.

Separation of lexical analyser from syntax Analyser:

→ To simplify the design of a compiler.

→ To increase the efficiency of a compiler.

→ To enhance the portability of a computer.

Tokens, lexemes and patterns:

→ Tokens is a pair of which is represented in the form of <tokenname, attributevalue>

→ Token name may be an identifier, keyword, operator or constant.

→ Patterns are mainly useful to define regular expression.

Lexical Errors:

→ These are generated during lexical analyser process:

→ Spelling errors (Ex: di in place of do).

→ Appearance of illegal characters (Ex: printf("hai"); @#)

→ Exceeding length of (the length of variable in c is 32).

Program:

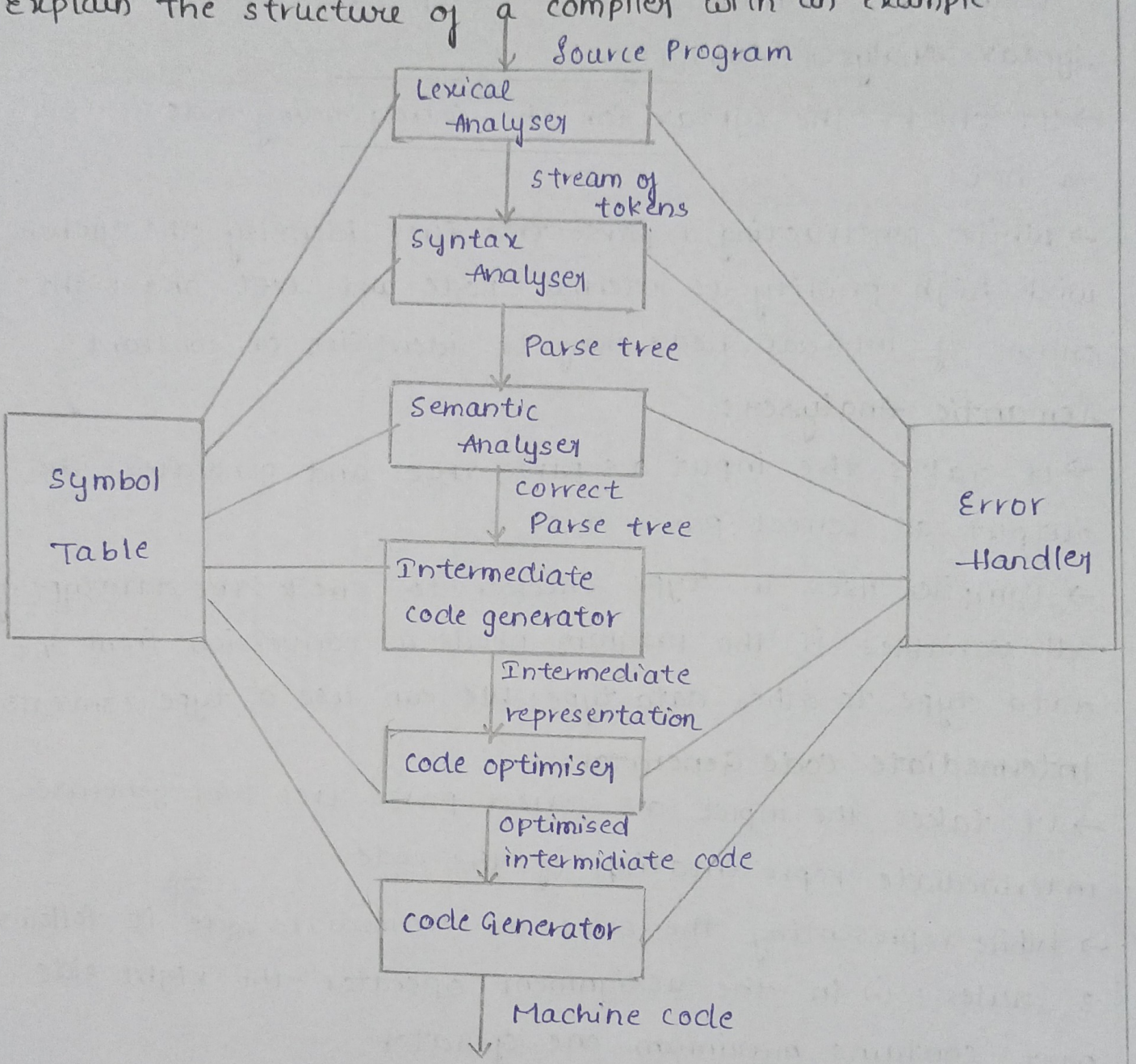
```
%{
    /* lex program for recognise the tokens */
%}
letter [a-z A-Z]
digit [0-9]
id{letter} ({letter/digit})*
numbers {digit}+ (.{digit}+)? ([E [+ -]]?{digit}+)?
%%
{id} {printf("%s is an identifier", yytext);}
if   {printf("%s is a keyword", yytext);}
else {printf("%s is a keyword", yytext);}
"<"  {printf("%s is less than operator", yytext);}
">"  {printf("%s is greater than operator", yytext);}
"==" {printf("%s is double equal operator", yytext);}
">=" {printf("%s is greater than or equal operator", yytext);}
"<=" {printf("%s is less than or equal operator", yytext);}
"!=" {printf("%s is not equal operator", yytext);}
```

{number} {printf("%s is a number", yytext);}

%% %%

2) Explain the structure of a compiler with an example.

A)



Lexical analyser / Scanner:

→ It reads the program character by character and converts it into meaningful schemes.

Lexeme is a sequence of characters.

Each lexeme is represented in the form of token i.e identifier, keyword, operator and constant.

Each token is represented in the form of
<tokenname, attribute value>

→ Transition diagram is used to recognise the tokens.

Syntax Analyser / Parser:

→ It checks the syntax for the corresponding code is correct or not.

→ While constructing a parse tree first identify the symbol with high priority as internal node and left and right chains of internal node may be identifier or constant.

Semantic Analyser;

→ It takes the input as parse tree and generates the output as correct parse tree.

→ Compiler uses a type checker to check the datatype of all variables if the program needs a conversion from one data type to other data type. We can use a type converter.

Intermediate Code Generator;

→ It takes the input as correct parse tree and generates intermediate representation of the code.

→ While representing the code as intermediate code it follows 3 rules: (i) In the assignment operator the right side part contains maximum one operator.

(ii) Compiler uses a temporary variable for storing temporary result for further calculation.

(iii) Fewer than the operands / addresses we can use.

Code optimiser;

→ Some of the stmts which are not necessary to get the output to remove or eliminate those stmts by codeoptimiser.

Code Generator :

→ It uses some mnemonics like LD, ST, ADD, MUL.

→ LD AC/R, MM

ST MM, R

ADD R1, R1, R2

MUL R3, R3, R4

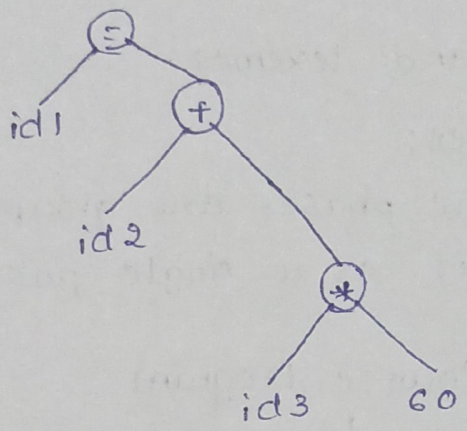
Example: x = y + z * 60

x + y = x = y + z * 60

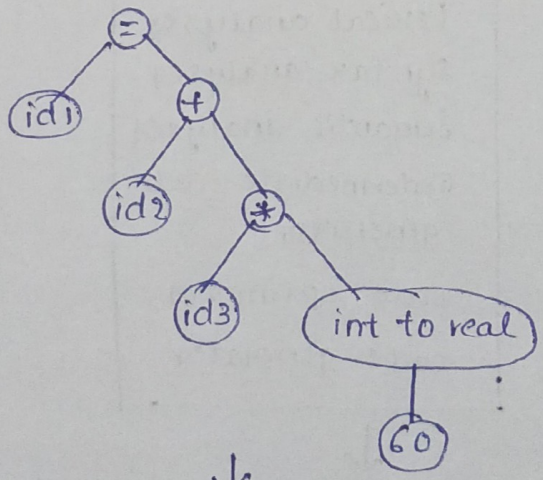
↓

id1 = id2 + id3 * 60

↓



↓



↓

t1 = int to real(60)

t2 = id3 * t1

t3 = id2 + t2

id1 = t3

↓

t1 = id3 * 60.0

id1 = id2 + t1

↓

```
LDF    R1, id3
MULF   R1, R1, #60.0
LDF    R2, id2
ADDF   R1, R1, R2
STF    id1, R1.
```

3) Write a short note on;

(i) Single Pass compiler

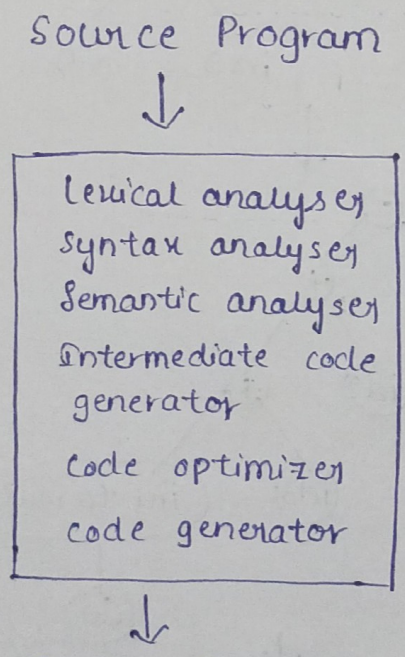
(ii) Multi pass compiler

(iii) lex compiler

(iv) Tokens, patterns and lexemes.

A) (i) Single Pass compiler;

If all the phases are grouping into one path then you can call it as a single pass compiler.



Multipass compiler:

If we divide all the 6 phases into 2 parts then we can call it as multipass compiler. There are two parts:

1) Analysis part / front end of the compiler.
2) Synthesis part / Back end of the compiler.

Analysis part:

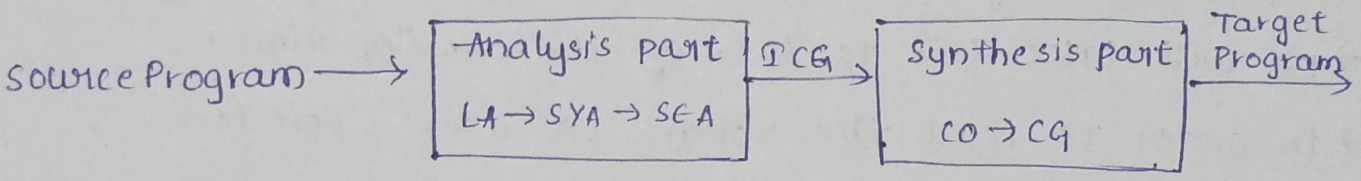
→ It depends on source program of the code but and not depends on target program.

→ This part contains the phases of lexical analyser, syntax analyser and semantic analyser.

Synthesis part:

→ It depends on the target program and independent on source program.

→ This part contains two phases of the compiler i.e., code optimiser and code generator.



Lex compiler:

Lex helps in generating lexical analyzers, which are used to break input text into a sequence of tokens for further processing. lex reads an input file that specifies regular expressions along with corresponding actions, and generates a c program as output. The generated c program can then be compiled to create a custom lexer for a particular language or application. lex is often used in combination with the Yaac parser generator to create complete compilers or interpreters for programming languages.

(iv) Tokens, patterns and lexeme:

→ Tokens is a pair of which is represented in the form of <tokenname, attribute value>

→ Token name may be an identifier, keyword, operator or constant.

→ Attribute value is a pointer which points to an entry of the symbol table for token information.

→ lexeme is a sequence of characters.

→ Patterns are mainly useful to define regular expression.

4) Explain input buffering in detail.

A) Input Buffering:

→ lexical Analyser scans the input from left to right one character at a time and produces tokens.

→ Input buffering will read the program character by character.

→ LA will take 25% to 30% of the compiler time.

→ In order to read the tokens, LA uses 2 pointers:

* lexeme Begin Pointer (LBP)
* Forward pointer (FP).

lexeme Begin Pointer: It points to the beginning character of the current lexeme.

Forward Pointer: It placed at the beginning point of the lexeme. It reads the first character and moves ahead right in order to read next character.

→ The use of buffering is block of characters are to be read into the buffer using only one system call.

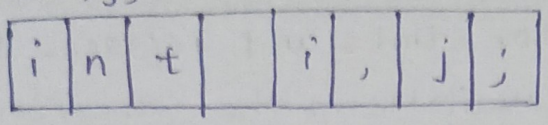
→ Buffering can be implemented in 2 ways:

1. One Buffer Scheme 2. Two Buffer Scheme

One Buffer scheme:

→ It uses only one buffer to read the input string.

→ Ex: int i,j;



→ If the input string size is larger than the capacity of the buffer size then the buffer has to be overridden in order to store the remaining input string.

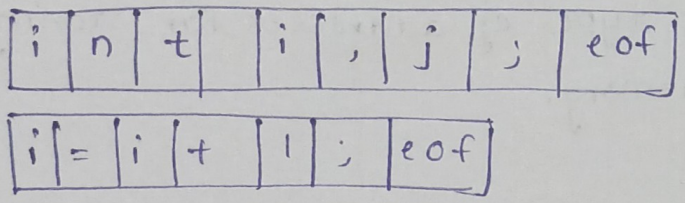
→ We can overcome this problem with the help of 2 buffer scheme.

Two Buffer scheme:

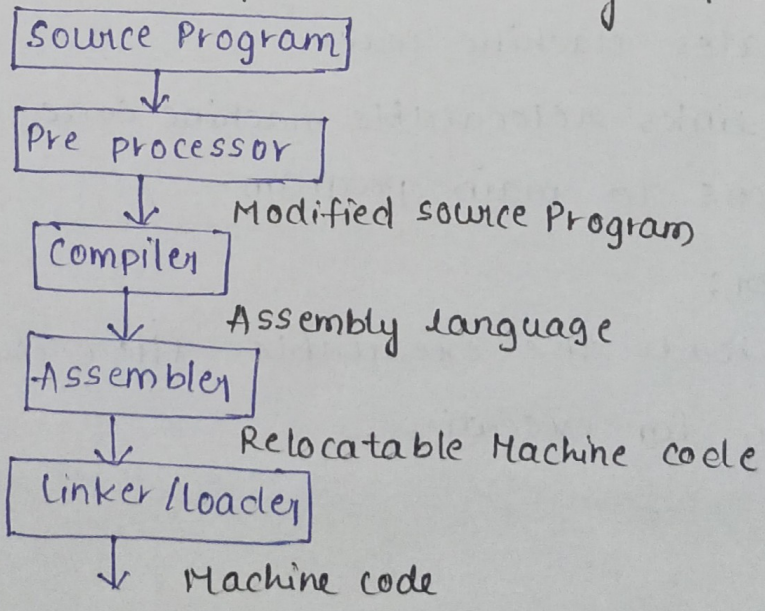
→ It uses 2 buffers to read input string and both the buffers are scanned alternatively.

→ When the end of the current buffer is reached then the other buffer is to be filled.

Ex: ie



5) Briefly explain about the steps for executing a program?



Source Program: Initially we can write the program in high-level languages (HLL) i.e, C, C++, Java --- etc.

Pre Processor:

→ It takes source program as input and generates modified source code.

→ It removes the preprocessor stmts and replaces the definition of those which are available in the corresponding file inclusion of preprocessor stmts.

Compiler:

→ It takes modified source program as input and generates target program in assembly language.

→ It checks whether the given program follows the language syntax rules or not.

Assembler:

→ It takes assembly language as input and generates relocatable Machine code.

→ It uses different types of addresses for storing the program in main memory.

Linker:

→ It takes the relocatable Machine code as input and generates Machine code.

→ It links relocatable machine code of various library functions to main program.

Loader:

→ It loads the executable file (.exe) into the main memory for execution.