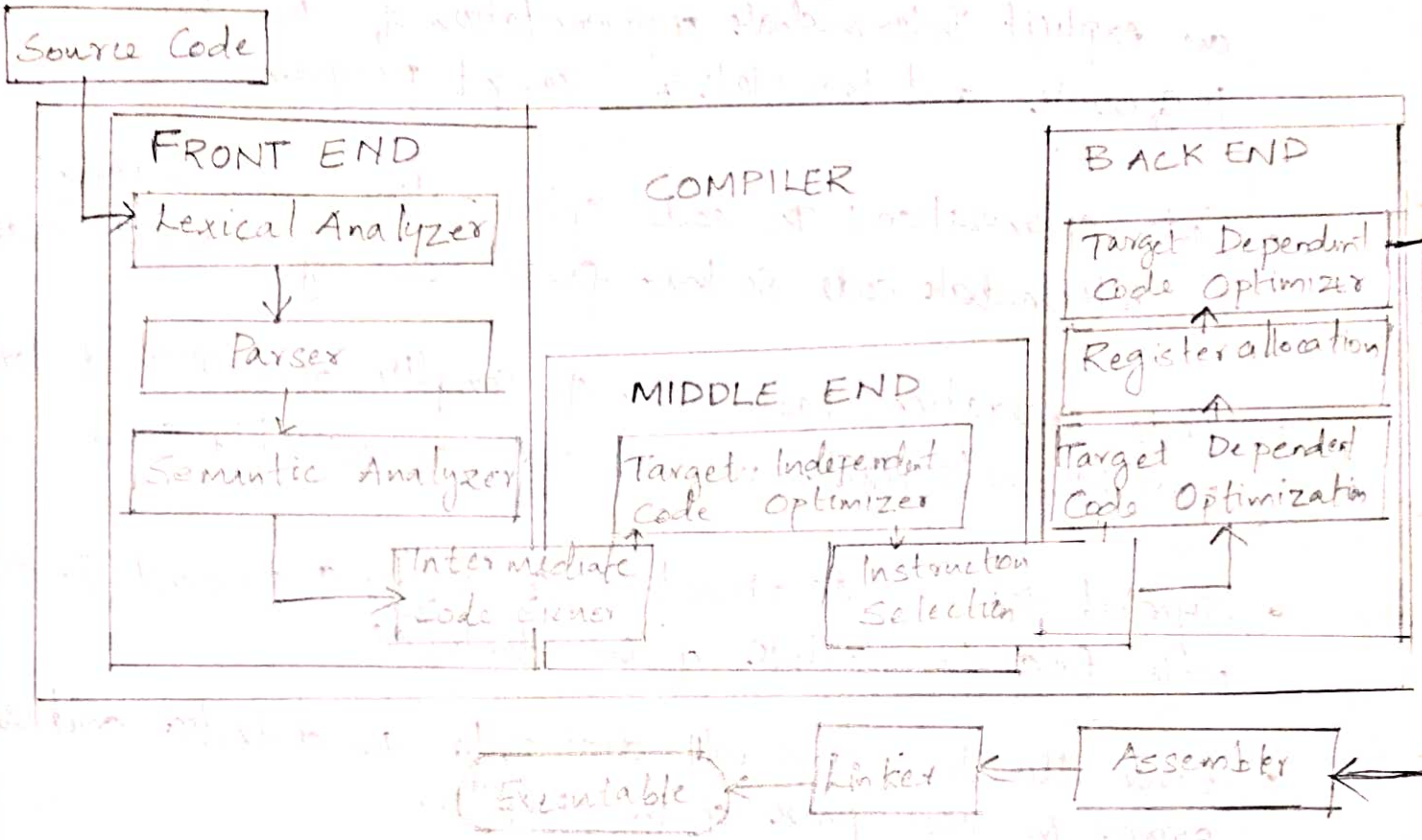
INTRODUCTION TO COMPILERS

A compiler is a program that translates a high level language program into a functionally equivalent low-level language program. A typical compiler is broken down into phases as shown below



- Lexical Analysis : A phase reads the characters in the source program and groups them into streams of tokens; each token represents a logically cohesive sequence of characters, suchas identifiers, operators and keywords. The character sequence that forms a token is called "lexemes"

- Parser : Parser imposes a hierarchical structure of the token stream which is called as syntax tree. Parsers check the given sentence in grammatically and well formed and checking the sentence belongs to the language of the grammar

- Semantic analysis : In this phase the compiler connect variable definitions to their uses and checks each expression has a correct type and translate the abstract syntax into simpler representation for generating machine code

- Intermediate code generation: In this phase a compiler generate an explicit intermediate representation of the source code. Its easy to generate and translate to target program.

- Code optimization : The code optimization phase attempt to impro the intermediate code so that faster scanning machine code will re

- Code generation: Last phase of compiler generation of target code, may be relocatable machine code or assembly code

- Symbol table : A structure containing a record for each identifie with field for attribute of the identifier

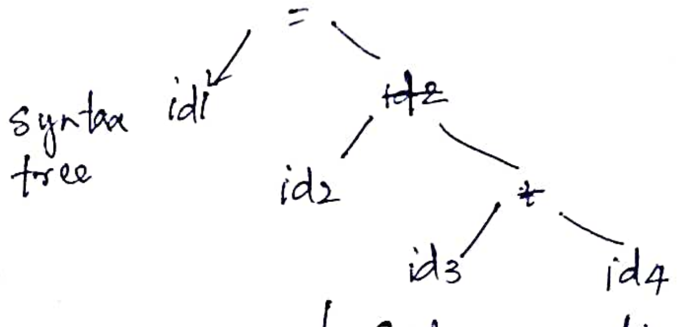
- Error Handler : This will deal with the detected and reported error in each phase of the compiler.

source code a = b + c * d

↓ Lexical analyzer ← Lexemes

token id1 = id2 + id3 * id4

↓ Syntax analyzer.



↓ Code generation

Generated Code.

```
load  id3
mul   id4
add   id2
store id1
```

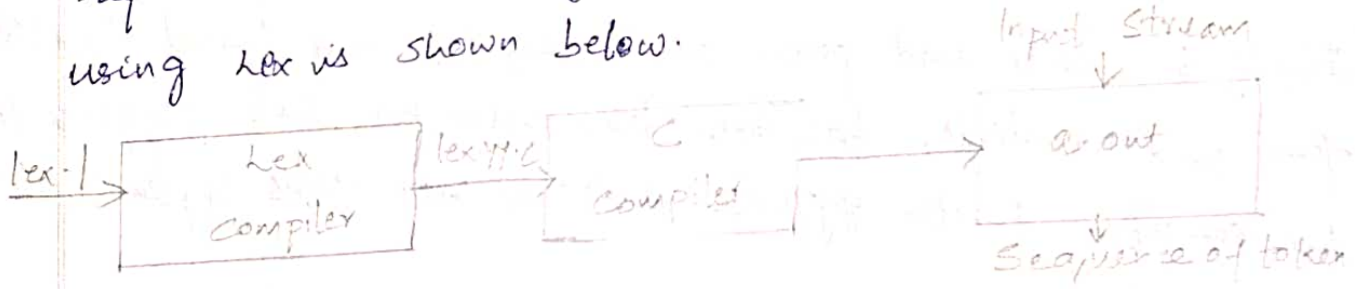
Compiler Construction tools

Tools are software tools used by the compiler writes to constru a compiler. these tools are specialized languago for specifying and implementation the component. Classified as

1) Scanner generators - LEX
2) Parse Generators - YACC
3) Syntax-directed translation engins
4) Automatic code generators
5) Data Flow engine.

INTRODUCTION TO LEX & YACC

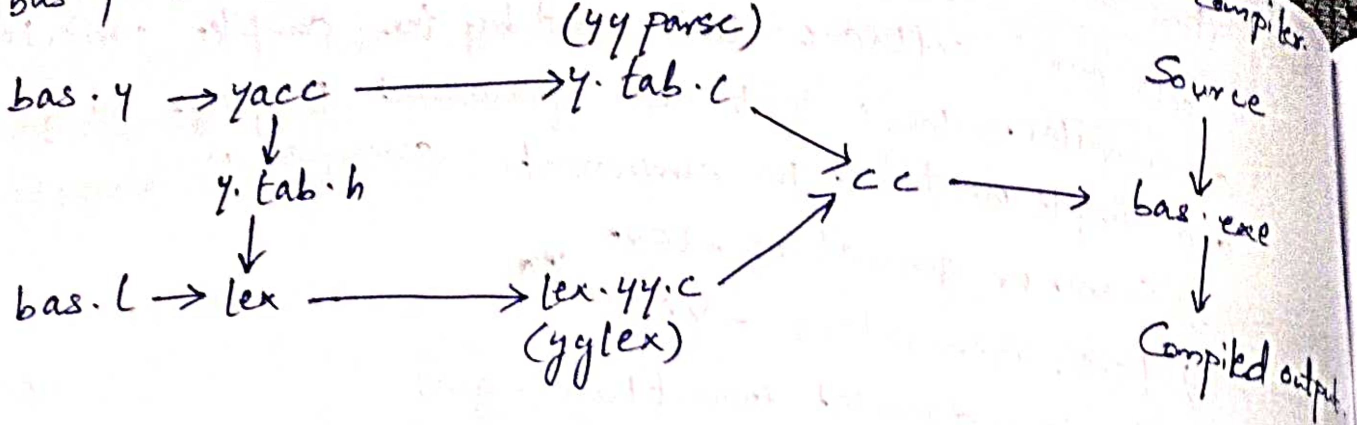
We code pattens and input to lex. It will read the patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on patterns written in the input file and convert the strings to tokens. The tokens are numerical representation of strings and simplify processing. The translation using Lex is shown below.



The symtable will contain other information like data type and location of the variable in memory.

We code a grammarand input it to YACC. The yacc will read the grammar and generate C code for a syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze tokens from lexical analyzer and create syntax tree.

'bas.y' for YACC compiler and 'bas.l' for lex compiler.



COMMANDS

```
yacc -d  bas.y      (creates y.tab.h, y.tab.c)
lex  bas.l          (creates lex.yy.c)
gcc -o bas y.tab.c lex.yy.c -ll
                    (creates bas.exe)
```

-d → causes yacc to generate definition for tokens and place them in file y.tab.h.

bas.l includes file y.tab.h and generates a lexecical analyzer that Includs yylex function. in file lex.yy.c

Finally the lexer and parser are compiled and linked together to form the executable, bas.exe From main we call yyparse to run the compiler. Function yyparse automaticaly calls yylex to obtain each token.

LEX FORMAT

...definition

%%

...rules...

%%

...sub routies...

Lex predefined variables

int yylex (void)	call to invoke lexer, returns token
char * yytext	pointer to matched string
yyleng	length of matched string.
yyval	value assosiated with token
int yywrap (void)	wrapup returns 1 if done 0 if not
FILE * yyout	output file
FILE * yyin	input file
INITIAL	initial start condition.
BEGIN	condition switch start condition
ECHO	write matched string

YACC (Yet Anotuer Compiler Compiler)

It is a LALR parser generator.

Format

Declarations
%%
Translation rules
%%
Support C-routines.

Declaration

Any Cdeclarations, delimited by %{ and %}.

%union It is defined stack type

%token These are return terminals.

%type type of NT

%nonassoc No assosiativity.

%left %right Assosiativities.

%start LHS of NT %prec precedence

- $$ refers to the attribute value assosiated with NT on the left.
  $i refers the value assosiated with the ith grammar symbol on the right.