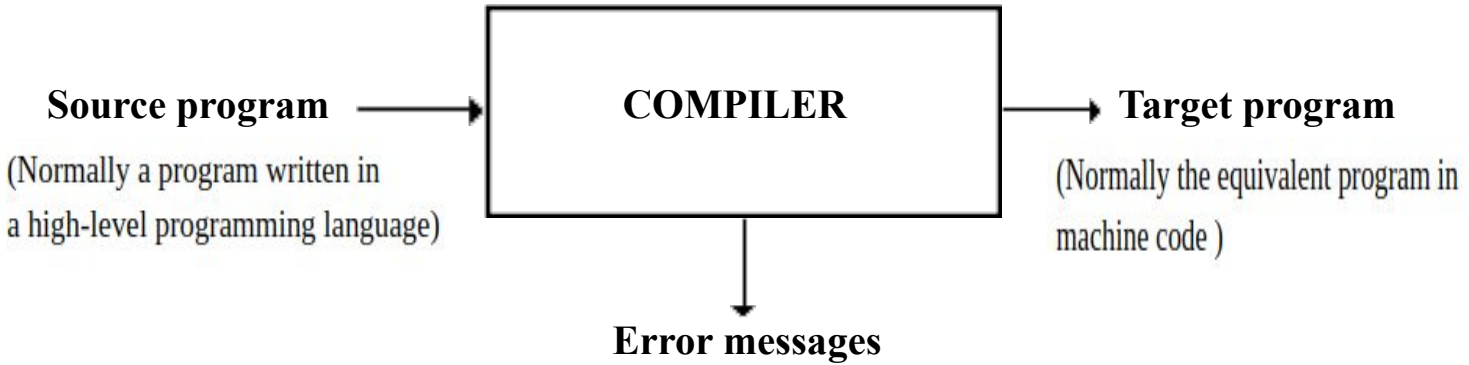
COMPILERS

A compiler is a program takes a program written in a source language and translates it into an equivalent program in a target language.



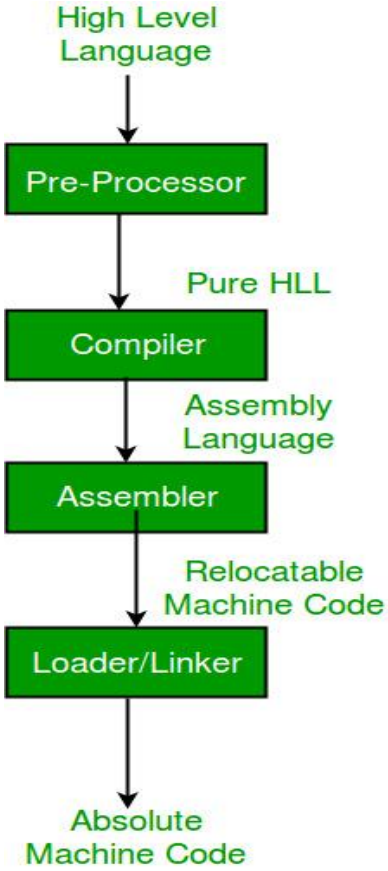
COMPILERS

Compilers translate from a source language (a high level language) to a functionally equivalent target language (the machine code of a particular machine or a machine-independent virtual machine).

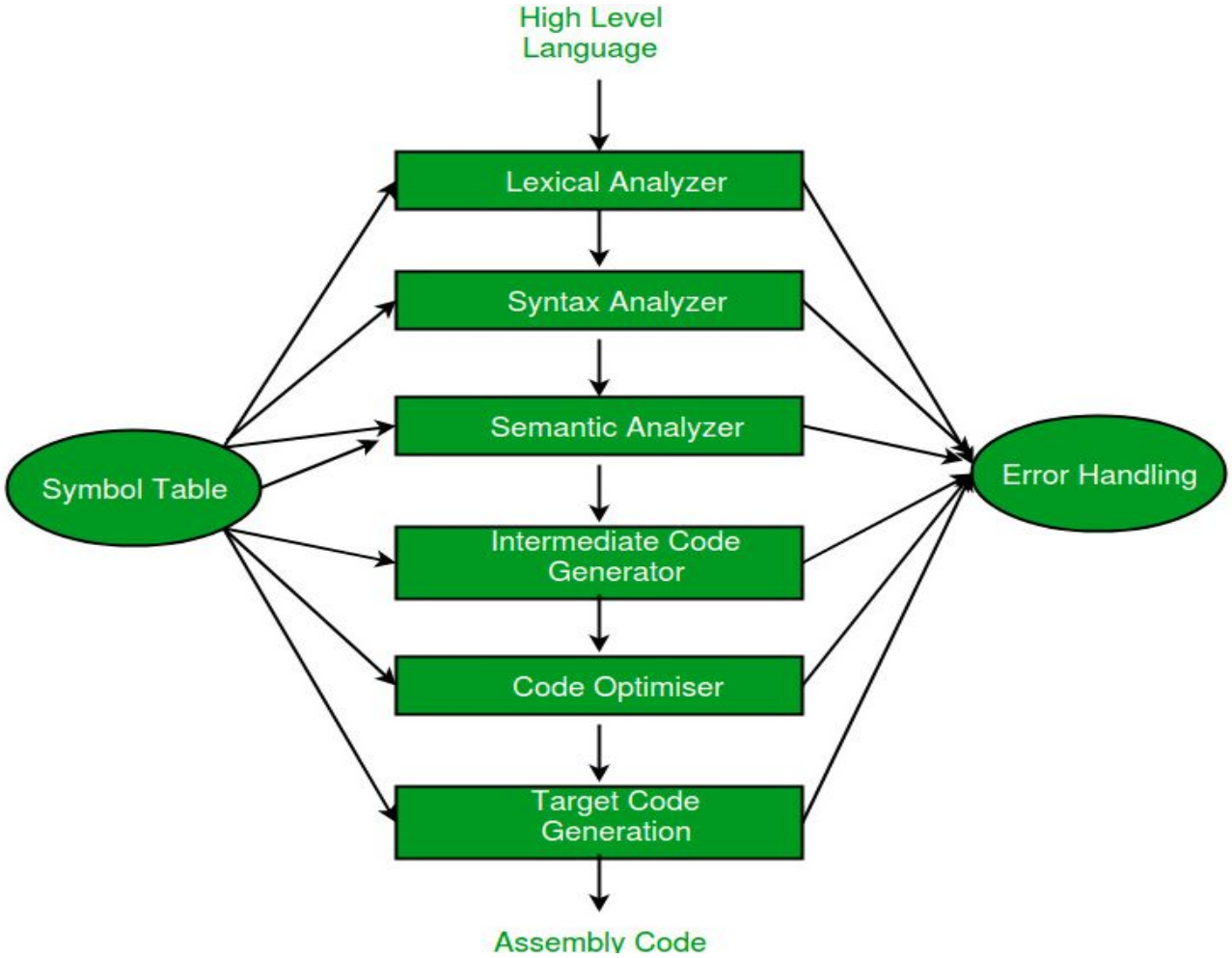
Major Parts of Compilers

- There are two major parts of a compiler: **Analysis** and **Synthesis**
- In analysis phase, an intermediate representation is created from the given source program.
  - ➔ **Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.**
- In synthesis phase, the equivalent target program is created from this intermediate representation.
  - ➔ **Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.**

Language processing system



Phases of a Compiler



Phases of a Compiler

- Each phase transforms the source program from one representation into another representation.
- They communicate with error handlers.
- They communicate with the symbol table.

1.Lexical Analyzer

- Lexical Analyzer reads the source program character by character and returns the tokens of the source program.
- A token describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimiters and so on)
- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

2.Syntax Analyzer

- A Syntax Analyzer creates the syntactic structure (generally a parse tree) of the given program.
- A syntax analyzer is also called as a parser.
- A parse tree describes a syntactic structure.
- Takes tokens one by one and uses CFG to construct parse Tree
- It takes input as tokens and checks their syntactic correctness

3.Semantic Analyzer

- Verifies the parse Tree
- It will check for the Semantic consistency
  - Label checking, Type checking etc
- It uses the parse Tree and information in the symbol table

4.Intermediate Code Generation

- Generates Intermediate Code
- To build a new compiler we don’t need to build it from scratch. We can take the intermediate code from the already existing compiler

5.Code Optimization

- It removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory)
- Code optimization is an optional phase.
- Optimization can be categorized into two types:
  - Machine-dependent
  - Machine-independent.

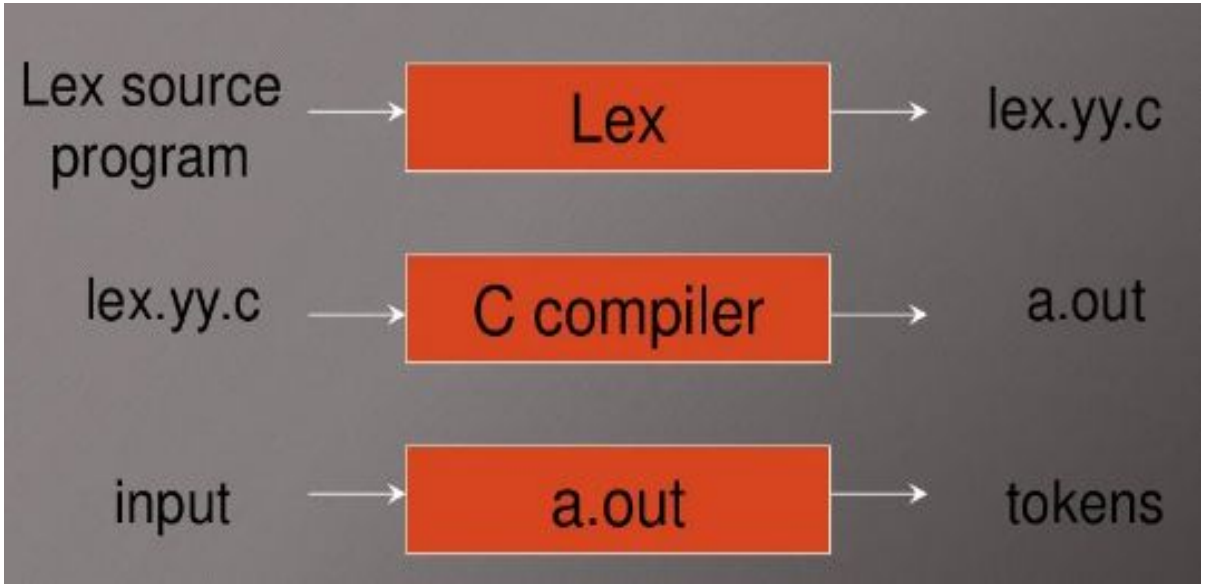
6.Target Code Generator

- The main purpose of the Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection, etc.
- The output is dependent on the type of assembler.
- This is the final stage of compilation.
- The optimized code is converted into relocatable machine code which then forms the input to the linker and loader.

LEX

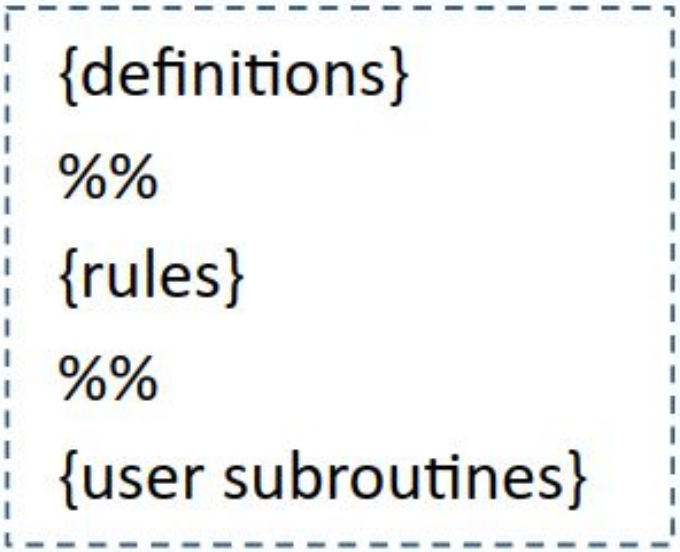
- Lex written by Mike Lesk and Eric Schmidt
- Lex is a tool used to generate a **lexical analyzer.**
- Lex translates a set of regular expression specifications (given as input in input_file.l) into a C implementation.
- Lex will read patterns then produces C code for a lexical analyzer that scans for the identifiers

LEX Compiler



Structure of LEX Program

1. Declarations / Definitions
2. Rules
3. Auxiliary Functions / User subroutines



Regular Expressions

RE CHARACTERS	DESCRIPTION
.	Matches any single character except newline character
*	Matches zero or more copies of preceding expression
+	Matches one or more occurrences of preceding expression
^	Matches the beginning of a line as first character of a RE
$	Matches the end of a line as the last character of a RE
{ }	Indicates how many times the previous pattern is allowed to match
\	Used to escape meta-characters, as part of C escape sequences
?	Matches zero or one occurrence of the preceding expression
|	Matches either the preceding RE or the following expression
[ ]	Character class, matches any character within the brackets
"....."	Interprets everything within quotation marks literally
/	Matches the preceding RE but only if followed by the following RE
( )	Groups a series of RE's together into a new RE

Examples of Regular Expressions

Expression	Matches
`abc`	`abc`
`abc*`	`ab abc abcc abccc ...`
`abc+`	`abc abcc abccc ...`
`a(bc)+`	`abc abcbc abcbcbc ...`
`a(bc)?`	`a abc`
`[abc]`	one of: `a`, `b`, `c`
`[a-z]`	any letter, a-z
`[a\-z]`	one of: `a`, `-`, `z`
`[-az]`	one of: `-`, `a`, `z`
`[A-Za-z0-9]+`	one or more alphanumeric characters
`[ \t\n]+`	whitespace
`[^ab]`	anything except: `a`, `b`
`[a^b]`	one of: `a`, `^`, `b`
`[a|b]`	one of: `a`, `|`, `b`
`a|b`	one of: `a`, `b`

Lex Actions

- An action is executed when a RE is matched
- Default action : copying input to output
- Blank, tab and newline ignored

  [\t \n]+ ;

- **yytext**
  - Array containing the text that matched the RE
  - Rule : [a-z]+ printf("%s",yytext);
- **ECHO**
  - Places matched string on the output

Lex Actions

- Why require rules with the default action?
  - Required to avoid matching some other rule
  - 'read' will also match instances of 'read' in 'bread' and 'readjust'
- **yyleng**
  - Count of the number of characters matched
  - Count number of words and characters in input
    - [a-zA-Z]+ {words++ ; chars += yyleng;}
  - Last character in the string matched

yytext [yyleng-1]

Lex Actions

- yymore()
  - Next time a rule is matched, the token should be appended to current value of yytext

```
%%
mega-      ECHO; yymore();
kludge     ECHO;
```

Input : mega-kludge
Output: mega-mega-kludge

- yyless(n)
  - Returns all but the first ‘n’ characters of the current token back to input stream

```
%%
foobar     ECHO; yyless(3);
[a-z]+     ECHO;
```

Input : foobar
Output: foobarbar

Lex Actions

- Access to I/O routines
  - Input() : returns next input character
  - Output (c) : writes character ‘c’ on the output
  - Unput (c) : pushes character ‘c’ back onto the input stream to be read later by input()
- yywrap()
  - Called whenever lex reaches end-of-file
  - Returns 1 : Lex continues with normal wrapup on end of input

All the Lex Actions

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

Commands to Run the Lex Code

★ To Create Lex file

➔ **gedit filename.l**

★ To run Lex on a source file

➔ **lex filename.l**

- It produces file named lex.yy.c which is C program for the lexical analyzer

★ To compile lex.yy.c

➔ **cc lex.yy.c -ll**

★ To run lexical analyzer program

➔ **./a.out**

Lex Program

```
%{
    #include<stdio.h>
     int c=0;
%}
%%
Pattern    {action}
 %%
 main()
 {
 }
```

Commands to download Lex and Yacc

- **sudo apt-get update**
- **sudo apt-get -f install flex** (if you are using local account, otherwise if it is with root exclude sudo i.e. **apt-get -f install flex**)
- **apt install flex** (to install lex packages)