CST302 Compiler Principles

COMPILER PRINCIPLES LAB

IMPORTANCE OF COMPILER DESIGN LAB

- ➢Compiler is a software which takes as input a program written in a High-Level language and translates it into its equivalent program in Low Level program.
- ➢Compilers teaches us how real-world applications are working and how to design them.
- ➢Learning Compilers gives us with both theoretical and practical knowledge that is crucial in order to implement a programming language.
- ➢It gives you a new level of understanding of a language in order to make better use of the language (optimization is just one example). Sometimes just using a compiler is not enough. You need to optimize the compiler itself for your application.
- ➢Compilers have a general structure that can be applied in many other applications, from debuggers to simulators to 3D applications to a browser and even a cmd / shell.
- ➢Understanding compilers and how they work makes it super simple to understand all the rest. a bit like a deep understanding of math will help you to understand geometry or physics.

We cannot do physics without the math. not on the same level. Just using something (read: tool, device, software, programming language) is usually enough when everything goes as expected. But if something goes wrong, only a true understanding of the inner workings and details will help to fix it.

Even more specifically, Compilers are super elaborated / sophisticated systems (architecturally speaking). If you will say that can or have written a compiler by yourself - there will be no doubt as to your capabilities as a programmer.

So, better be a pilot who have the knowledge and mechanics of an airplane than the one who just know how to fly.

Every computer scientist can do much better if have knowledge of compilers apart from the domain and technical knowledge.

Compiler design lab provides deep understanding of how programming language Syntax, Semantics are used in translation into machine equivalents apart from the knowledge of various compiler generation tools like **LEX,YACC** etc.

A compiler or interpreter for a programming language is often decomposed into two parts:

1. Read the source program and discover its structure.
2. Process this structure, e.g. to generate the target program.

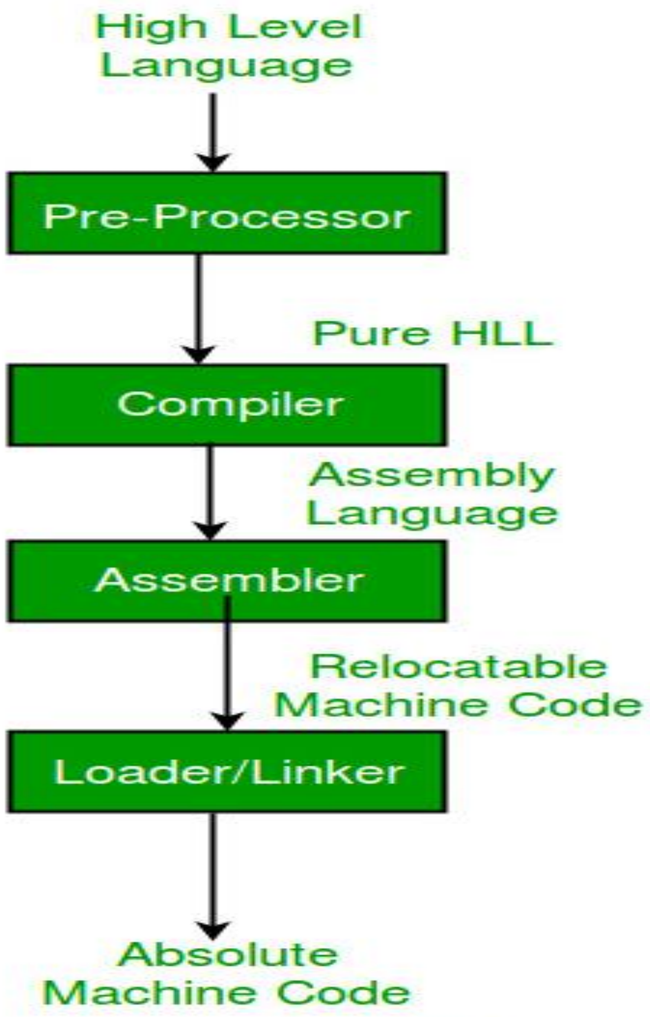
Lex and Yacc can generate program fragments that solve the first task.

The task of discovering the source structure again is decomposed into subtasks:

1. Split the source file into tokens (Lex).
2. Find the hierarchical structure of the program (Yacc).

Language Processing Systems

- A computer is a logical assembly of Software and Hardware. The hardware knows a language, that is hard for us to grasp, consequently, we tend to write programs in a high-level language, that is much less complicated for us to comprehend and maintain in our thoughts.
- Now, these programs go through a series of transformations so that they can readily be used by machines. This is where language procedure systems come in handy.



High-Level Language to Machine Code

- **High-Level Language:** If a program contains pre-processor directives such as #include or #define it is called HLL. They are closer to humans but far from machines. These (#) tags are called preprocessor directives. They direct the pre-processor about what to do.
- **Pre-Processor:** The pre-processor removes all the #include directives by including the files called file inclusion and all the #define directives using macro expansion. It performs file inclusion, augmentation, macro-processing, etc.
- **Assembly Language:** It’s neither in binary form nor high level. It is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.
- **Assembler:** For every platform (Hardware + OS) we will have an assembler. They are not universal since for each platform we have one. The output of the assembler is called an object file. Its translates assembly language to machine code.

- **Interpreter:** An interpreter converts high-level language into low-level machine language, just like a compiler. But they are different in the way they read the input. The Compiler in **one go** reads the inputs, does the processing, and executes the source code whereas the **interpreter does the same line by line.** A compiler scans the entire program and translates it as a whole into machine code whereas an interpreter translates the program one statement at a time. Interpreted programs are usually slower concerning compiled ones.
- For example: Let in the source program, it is written #include "Stdio. h". Pre-Processor replaces this file with its contents in the produced output. The basic work of a linker is to merge object codes (that have not even been connected), produced by the compiler, assembler, standard library function, and operating system resources. The codes generated by the compiler, assembler, and linker are generally re-located by their nature, which means to say, the starting location of these codes is not determined, which means they can be anywhere in the computer memory, Thus the basic task of loaders to find/calculate the exact address of these memory locations.
- **Relocatable Machine Code:** It can be loaded at any point and can be run. The address within the program will be in such a way that it will cooperate with the program movement.
- **Loader/Linker:** Loader/Linker converts the relocatable code into absolute code and tries to run the program resulting in a running program or an error message (or sometimes both can happen). Linker loads a variety of object files into a single file to make it executable. Then loader loads it in memory and executes it.

Types of Compiler

There are mainly three types of compilers.

- Single Pass Compilers
- Two Pass Compilers
- Multipass Compilers

Single Pass Compiler

- When all the phases of the compiler are present inside **a single module**, it is simply called a single-pass compiler. It performs the work of converting source code to machine code.

Two Pass Compiler

- Two-pass compiler is a compiler in which the program is translated twice, once from the **front end and the back from the back end** known as Two Pass Compiler.

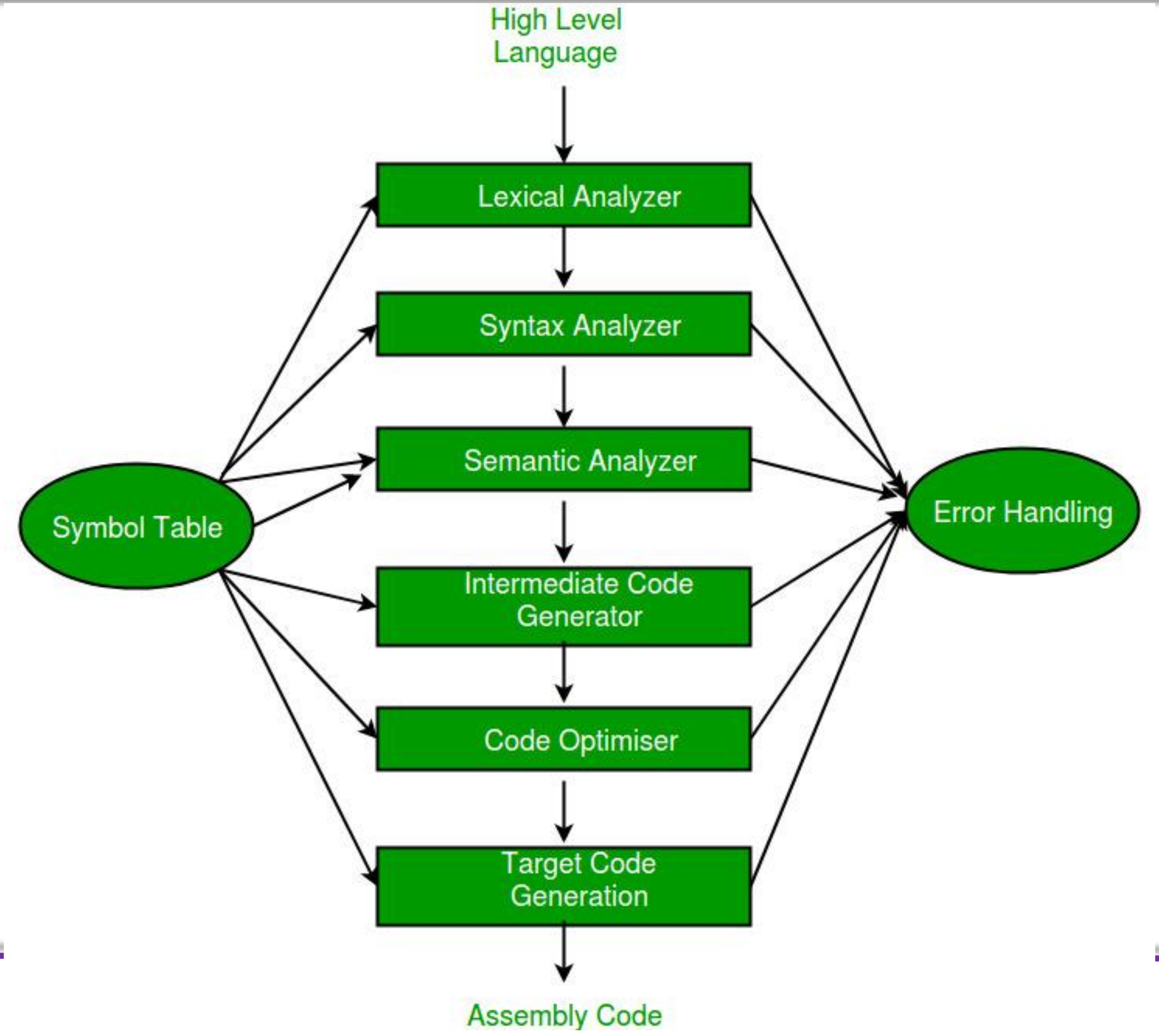
Multipass Compiler

- When several intermediate codes are created in a program and a **syntax tree** is processed many times, it is called Multi pass Compiler. It breaks codes into smaller programs.

PHASES OF A COMPILER

A compiler is a software program that converts the **high-level source code written in a programming language into low-level machine code** that can be executed by the computer hardware. The process of converting the source code into machine code involves several phases or stages, which are collectively known as the phases of a compiler. The typical phases of a compiler are:

1. **Lexical Analysis:** The first phase of a compiler is lexical analysis, also known as scanning. This phase reads the source code and breaks it into a stream of **tokens**, which are the basic units of the programming language. The tokens are then passed on to the next phase for further processing.
2. **Syntax Analysis:** The second phase of a compiler is **syntax analysis, also known as parsing**. This phase takes the stream of tokens generated by the lexical analysis phase and checks whether they conform to the grammar of the programming language. The output of this phase is usually an **Abstract Syntax Tree (AST).**
3. **Semantic Analysis:** The third phase of a compiler is **semantic analysis**. This phase checks whether the code is semantically correct, i.e., whether it conforms to the **language's type system and other semantic rules.**
4. **Intermediate Code Generation**: The fourth phase of a compiler is intermediate code generation. This phase generates an intermediate representation of the source code that can be easily **translated into machine code.**
5. **Optimization:** The fifth phase of a compiler is optimization. This phase applies various optimization techniques to the intermediate code to improve the **performance of the generated machine code**.
6. **Code Generation:** The final phase of a compiler is code generation. This phase takes the optimized intermediate code and generates the **actual machine code that can be executed by the target hardware**.


High Level Language
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Code Optimiser
Target Code Generation
Assembly Code
Symbol Table
Error Handling

Symbol Table – It is a data structure being used and maintained by the compiler, consisting of all the **identifier's names along with their types**. It helps the compiler to function smoothly by finding the identifiers quickly.

The analysis of a source program is divided into mainly three phases. They are:

1. **Linear Analysis**
   This involves a scanning phase where the stream of characters is read from left to right. It is then grouped into various tokens having a collective meaning.
2. **Hierarchical Analysis**
   In this analysis phase, based on a collective meaning, the tokens are categorized hierarchically into nested groups.
3. **Semantic Analysis**
   This phase is used to check whether the components of the source program are meaningful or not.

- 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.

1. **<u>Lexical Analyzer</u>**
   It is also called a scanner. It takes the output of the preprocessor (which performs file inclusion and macro expansion) as the input which is in a pure high-level language.

It reads the characters from the source program and groups them into lexemes (sequence of characters that “go together”). Each lexeme corresponds to a token.

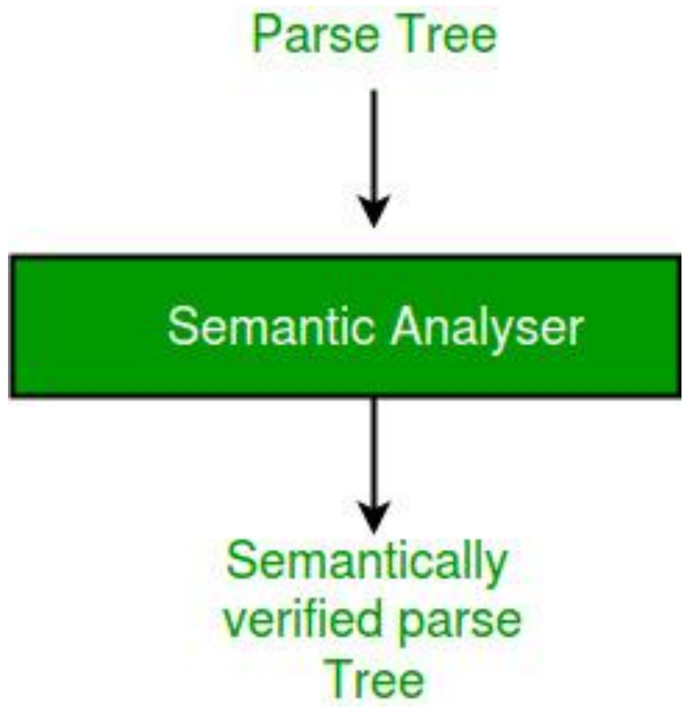
Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes lexical errors (e.g., erroneous characters), comments, and white space.

1. **Syntax Analyzer** It is sometimes called a parser. It constructs the parse tree. It takes all the tokens one by one and uses Context-Free Grammar to construct the parse tree.

Why Grammar?
The rules of programming can be entirely represented in a few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not.
The parse tree is also called the derivation tree. Parse trees are generally constructed to check for ambiguity in the given grammar. There are certain rules associated with the derivation tree.

1. Any identifier is an expression
2. Any number can be called an expression
3. Performing any operations in the given expression will always result in an expression. For example, the sum of two expressions is also an expression.
4. The parse tree can be compressed to form a syntax tree

Syntax error can be detected at this level if the input is not in accordance with the grammar.



- **Semantic Analyzer** – It verifies the parse tree, whether it's meaningful or not. It furthermore produces a verified parse tree. It also does type checking, Label checking, and Flow control checking.
- **Intermediate Code Generator** – It generates intermediate code, which is a form that can be readily executed by a machine We have many popular intermediate codes. Example – Three address codes etc. Intermediate code is converted to machine language using the last two phases which are platform dependent. Till intermediate code, it is the same for every compiler out there, but after that, it depends on the platform. 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 and build the last two parts.
- **Code Optimizer** – It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered. Optimization can be categorized into two types: machine-dependent and machine-independent.
- **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.

The advantages of using a compiler to translate high-level programming languages into machine code are:

1. Portability: Compilers allow programs to be written in a high-level programming language, which can be executed on different hardware platforms without the need for modification. This means that programs can be written once and run on multiple platforms, making them more portable.
2. Optimization: Compilers can apply various optimization techniques to the code, such as loop unrolling, dead code elimination, and constant propagation, which can significantly improve the performance of the generated machine code.
3. Error Checking: Compilers perform a thorough check of the source code, which can detect syntax and semantic errors at compile-time, thereby reducing the likelihood of runtime errors.
4. Maintainability: Programs written in high-level languages are easier to understand and maintain than programs written in low-level assembly language. Compilers help in translating high-level code into machine code, making programs easier to maintain and modify.
5. Productivity: High-level programming languages and compilers help in increasing the productivity of developers. Developers can write code faster in high-level languages, which can be compiled into efficient machine code.

- In summary, compilers provide advantages such as portability, optimization, error checking, maintainability, and productivity.

Overall, compiler design is a complex process that involves multiple stages and requires a deep understanding of both the programming language and the target platform. A well-designed compiler can greatly improve the efficiency and performance of software programs, making them more useful and valuable for users.

<u>**Definition**</u>

- Automata Theory is a branch of computer science that deals with designing abstract self-propelled computing devices that follow a predetermined sequence of operations automatically. An automaton with a finite number of states is called a Finite Automaton.

‘Theory of Computation’ or ‘Theory of Automata’ is the core area of computer science and engineering; it is the branch that aims to attempts the deep understanding of computational processes by means of effectively solving the problems via mathematical models, tools, and techniques.

Finite Automata

- Finite Automata, also known as the Finite State Machine, is a simple machine that is able to recognize patterns. It is an abstract machine with five components or tuples. It contains a set of states and rules for going from one state to the next, but it is dependent on the input symbol used. It is essentially an abstract representation of a digital computer.

- Finite automata has two states: Accept State or Reject State.

- **Types of Finite Automata**

- Two types of Finite Automata are there:

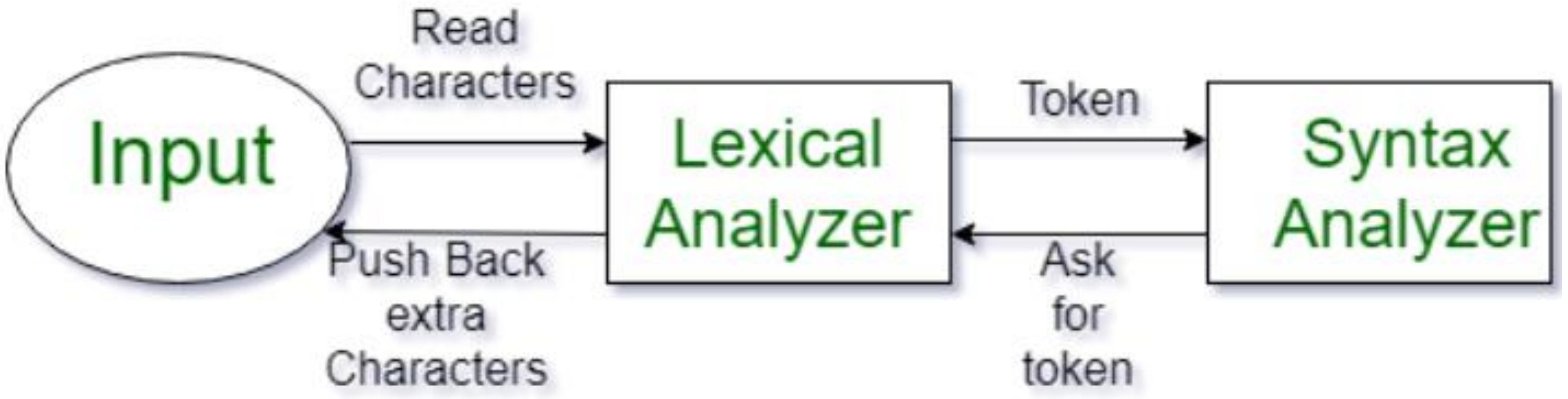
Deterministic Finite Automata (DFA): In DFA, the computer only travels to one state for each input character.

- Here, the uniqueness of the calculation is referred to as deterministic. The null move is not accepted by DFA.

Non-deterministic Finite Automata (NFA): NFA is used to send any number of states for a certain input. It is capable of accepting the null move.

INTRODUCTION OF LEXICAL ANALYSIS

- 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 output is a sequence of tokens that is sent to the parser for syntax analysis


Read Characters
Input
Lexical Analyzer
Token
Syntax Analyzer
Push Back extra Characters
Ask for token

What is a token?

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

Type token (id, number, real, . . .)

Punctuation tokens (IF, void, return, . . .)

Alphabetic tokens (keywords)

Keywords; Examples-for, while, if etc.

Identifier; Examples-Variable name, function name, etc.

Operators; Examples '+', '++', '-' etc.

Separators; Examples ',' ';' etc

Example of Non-Tokens:

Comments, preprocessor directive, macros, blanks, tabs, newline, etc.

Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme.

Eg: “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

How Lexical Analyzer works

Input preprocessing: This stage involves cleaning up the input text and preparing it for lexical analysis. This may include removing comments, whitespace, and other non-essential characters from the input text.

Tokenization: This is the process of breaking the input text into a sequence of tokens. This is usually done by matching the characters in the input text against a set of patterns or regular expressions that define the different types of tokens.

Token classification: In this stage, the lexer determines the type of each token. For example, in a programming language, the lexer might classify keywords, identifiers, operators, and punctuation symbols as separate token types.

Token validation: In this stage, the lexer checks that each token is valid according to the rules of the programming language. For example, it might check that a variable name is a valid identifier, or that an operator has the correct syntax.

Output generation: In this final stage, the lexer generates the output of the lexical analysis process, which is typically a list of tokens. This list of tokens can then be passed to the next stage of compilation or interpretation.

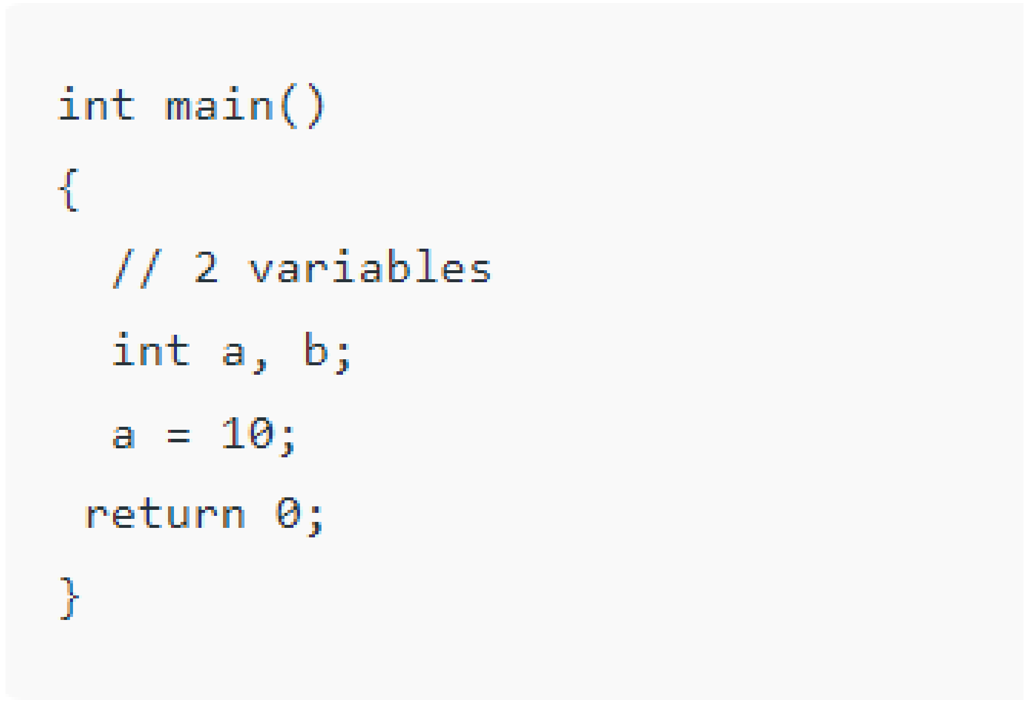
The lexical analyzer identifies the error with the help of the automation machine and the grammar of the given language on which it is based like C, C++, and gives row number and column number of the error.

Suppose we pass a statement through lexical analyzer **a = b + c** ;

It will generate token sequence like this: **id=id+id**;

Where each id refers to it’s variable in the symbol table referencing all details

For example, consider the program



```
int main()
{
  // 2 variables
  int a, b;
  a = 10;
 return 0;
}
```

All the valid tokens are:

```
'int'  'main'  '('  ')'  '{'  'int'  'a' ','  'b'  ';'
 'a'  '='  '10'  ';' 'return'  '0'  ';'  '}'
```

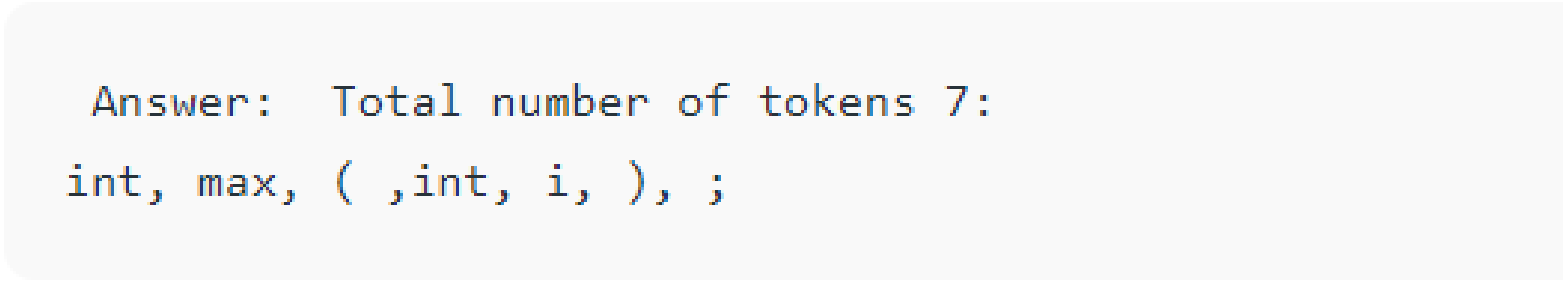
Above are the valid tokens. You can observe that we have omitted comments.

Exercise 1: Count number of tokens :

```
int main()
{
  int a = 10, b = 20;
  printf("sum is :%d",a+b);
  return 0;
}
Answer: Total number of token: 27.
```

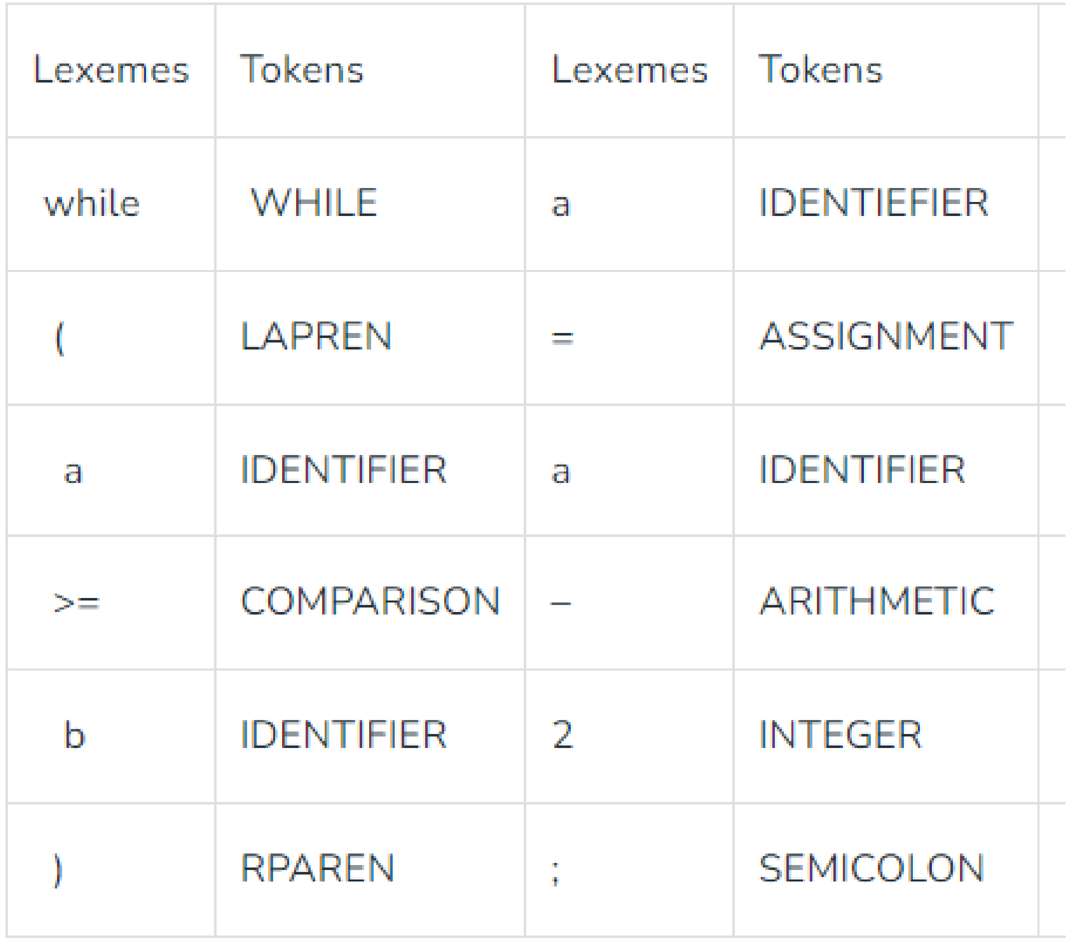
Exercise 2: Count number of tokens : int max(int i);

- Lexical analyzer first read **int** and finds it to be valid and accepts as token
- **max** is read by it and found to be a valid function name after reading **(**
- **int** is also a token , then again **i** as another token and finally ;



```
 Answer:  Total number of tokens 7:
int, max, ( ,int, i, ), ;
```

We can represent in the form of lexemes and tokens as under



Lexemes	Tokens	Lexemes	Tokens
while	WHILE	a	IDENTIEFIER
(	LAPREN	=	ASSIGNMENT
a	IDENTIFIER	a	IDENTIFIER
>=	COMPARISON	–	ARITHMETIC
b	IDENTIFIER	2	INTEGER
)	RPAREN	;	SEMICOLON

Advantages:

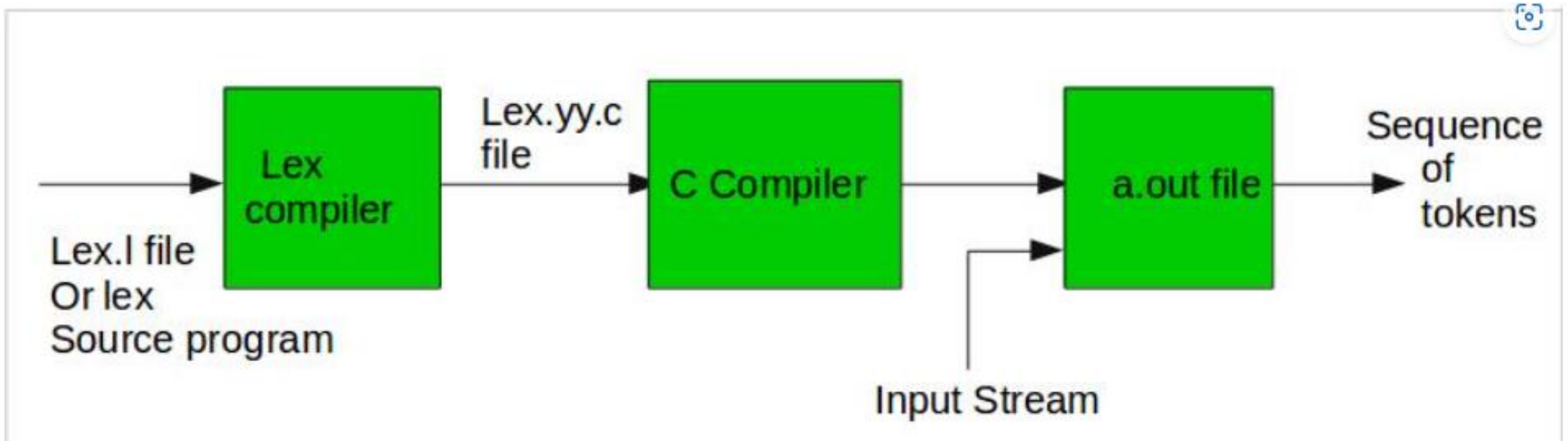
- **Efficiency:** Lexical analysis improves the efficiency of the parsing process because it breaks down the input into smaller, more manageable chunks. This allows the parser to focus on the structure of the code, rather than the individual characters.
- **Flexibility:** Lexical analysis allows for the use of keywords and reserved words in programming languages. This makes it easier to create new programming languages and to modify existing ones.
- **Error Detection:** The lexical analyzer can detect errors such as misspelled words, missing semicolons, and undefined variables. This can save a lot of time in the debugging process.
- **Code Optimization:** Lexical analysis can help optimize code by identifying common patterns and replacing them with more efficient code. This can improve the performance of the program.

Disadvantages:

- **Complexity:** Lexical analysis can be complex and require a lot of computational power. This can make it difficult to implement in some programming languages.
- **Limited Error Detection:** While lexical analysis can detect certain types of errors, it cannot detect all errors. For example, it may not be able to detect logic errors or type errors.
- **Increased Code Size:** The addition of keywords and reserved words can increase the size of the code, making it more difficult to read and understand.
- **Reduced Flexibility:** The use of keywords and reserved words can also reduce the flexibility of a programming language. It may not be possible to use certain words or phrases in a way that is intuitive to the programmer.

FLEX (FAST LEXICAL ANALYZER GENERATOR)

- **FLEX (fast lexical analyzer generator)** is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987. It is used together with Berkeley Yacc parser generator or GNU Bison parser generator. Flex and Bison both are more flexible than Lex and Yacc and produces faster code.
  Bison produces parser from the input file provided by the user. The function **yylex()** is automatically generated by the flex when it is provided with a **.l file** and this yylex() function is expected by parser to call to retrieve tokens from current/this token stream.
- **Note:** The function yylex() is the main flex function that runs the Rule Section and extension (.l) is the extension used to save the programs.



Step 1: An input file describes the lexical analyzer to be generated named lex.l is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c.
Step 2: The C compiler compile lex.yy.c file into an executable file called a.out.
Step 3: The output file a.out take a stream of input characters and produce a stream of tokens.

- **Program Structure:**
- In the input file, there are 3 sections:

1. **Definition Section**
2. **Rules Section**
3. **User Code Section:**

1. Definition Section: The definition section contains the declaration of variables, regular definitions, manifest constants.

In the definition section, text is enclosed in “%{ %}” brackets. Anything written in this brackets is copied directly to the file lex.yy.c

Syntax:

```
%{
   // Definitions
%}
```

2. Rules Section: The rules section contains a series of rules in the form: pattern action and pattern must be unintended and action begin on the same line in {} brackets. The rule section is enclosed in “%% %%”.

Syntax:

```
%%
pattern  action
%%
```

Examples: Table below shows some of the pattern matches.

Pattern	It can match with
[0-9]	all the digits between 0 and 9
[0+9]	either 0, + or 9
[0, 9]	either 0, ‘, ‘ or 9
[0 9]	either 0, ‘ ‘ or 9
[-09]	either -, 0 or 9
[-0-9]	either – or all digit between 0 and 9
[0-9]+	one or more digit between 0 and 9
[^a]	all the other characters except a
[^A-Z]	all the other characters except the upper case letters
a{2, 4}	either aa, aaa or aaaa
a{2, }	two or more occurrences of a
a{4}	exactly 4 a’s i.e, aaaa
.	any character except newline
a*	0 or more occurrences of a
a+	1 or more occurrences of a
[a-z]	all lower case letters
[a-zA-Z]	any alphabetic letter
w(x | y)z	wxz or wyz

3. User Code Section: This section contains C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer.

Basic Program Structure:

```
%{
// Definitions
%}

%%
Rules
%%

User code section
```

- **How to run the program:**
  To run the program, it should be first saved with the extension **.l or .lex**. Run the below commands on terminal in order to run the program file.
- **Step 1:** flex filename.l or flex filename.lex depending on the extension file is saved with
  Step 2: gcc lex.yy.c
  Step 3: ./a.out
  Step 4: Provide the input to program in case it is required

LEX

What is Lex? What is Yacc?

What is Lex?

Lex is officially known as a "Lexical Analyzer".

It's main job is to break up an input stream into more usable elements.

Or in, other words, to identify the "interesting bits" in a text file.

For example, if you are writing a compiler for the C programming language, the symbols { } ( ) ; all have significance on their own. The letter a usually appears as part of a keyword or variable name, and is not interesting on it's own. Instead, we are interested in the whole word. Spaces and newlines are completely uninteresting, and we want to ignore them completely, unless they appear within quotes "like this"

All of these things are handled by the Lexical Analyser.

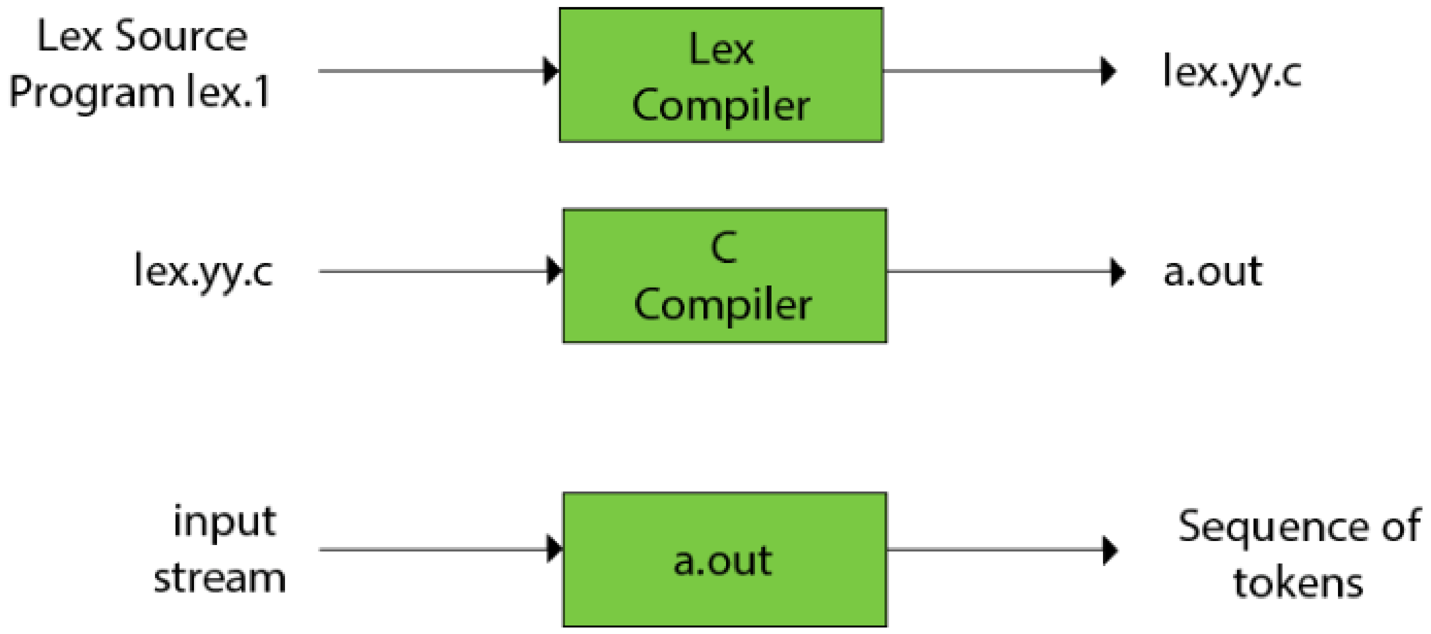
LEX

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 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.

LEX


Lex Source Program lex.1
Lex Compiler
lex.yy.c
lex.yy.c
C Compiler
a.out
input stream
a.out
Sequence of tokens

Lex file format

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

Definitions include declarations of constant, variable and regular definitions.

- **Rules** define the statement of form p1 {action1} p2 {action2}....pn {action}.
- Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.
- **User subroutines** are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

```
{ definitions }
%%
 { rules }
%%
{ user subroutines }
```

LEX

what is Lex in Compiler Design but before understanding Lex we have to understand what is Lexical Analysis.

Lexical Analysis

It is the first step of compiler design, it takes the input as a stream of characters and gives the output as tokens also known as tokenization. The tokens can be classified into identifiers, Sperators, Keywords, Operators, Constant and Special Characters.

It has three phases:

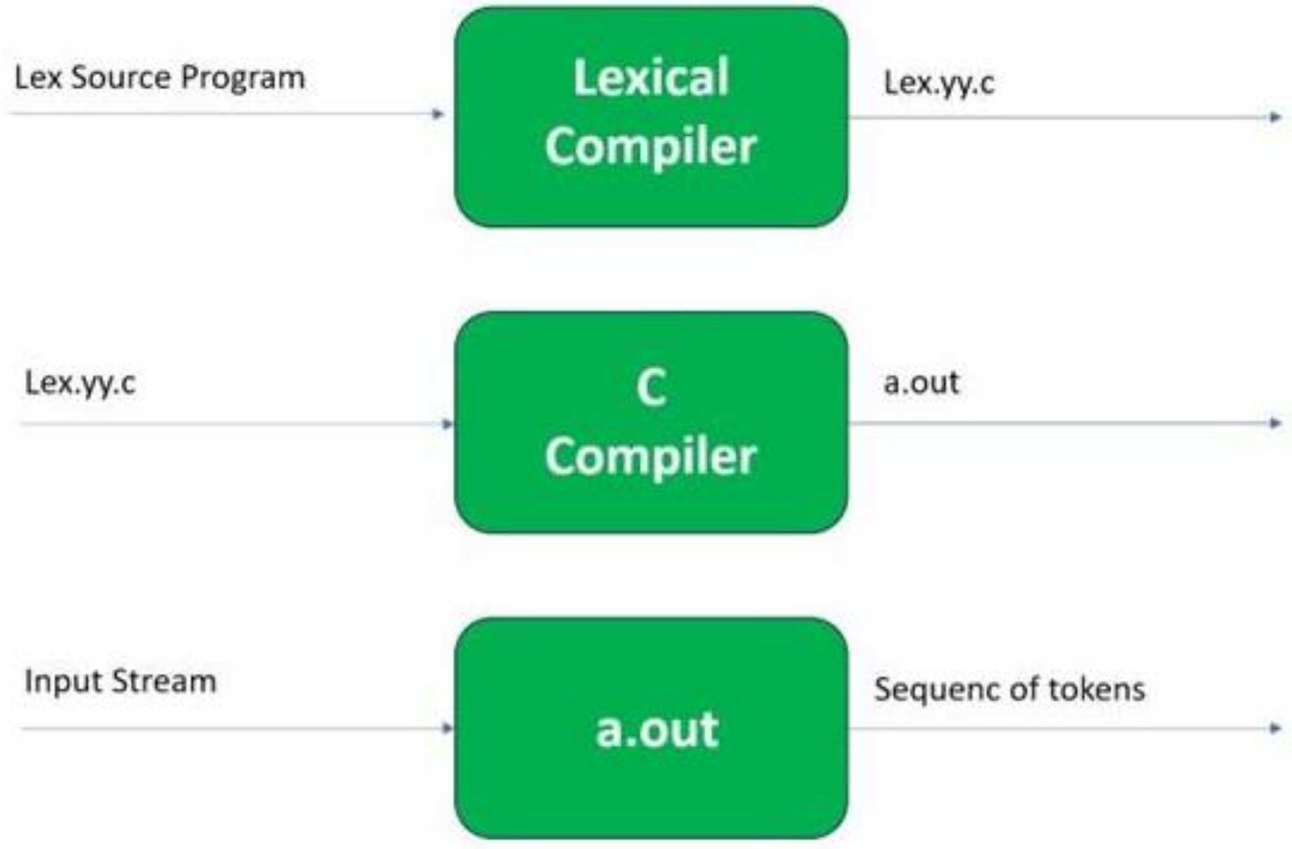
1. **Tokenization:** It takes the stream of characters and converts it into tokens.
2. **Error Messages:** It gives errors related to lexical analysis such as exceeding length, unmatched string, etc.
3. **Eliminate Comments:** Eliminates all the spaces, blank spaces, new lines, and indentations.

LEX

Lex
Lex is a tool or a computer program that generates Lexical Analyzers (converts the stream of characters into tokens). The Lex tool itself is a compiler. The Lex compiler takes the input and transforms that input into input patterns. It is commonly used with YACC(Yet Another Compiler Compiler). It was written by Mike Lesk and Eric Schmidt.
Function of Lex
1. In the first step the source code which is in the Lex language having the file name 'File.l' gives as input to the Lex Compiler commonly known as Lex to get the output as lex.yy.c.
2. After that, the output lex.yy.c will be used as input to the C compiler which gives the output in the form of an 'a.out' file, and finally, the output file a.out will take the stream of character and generates tokens as output.

```
lex.yy.c: It is a C program.
File.l: It is a Lex source program
a.out: It is a Lexical analyzer
```

LEX



Block Diagram of Lex

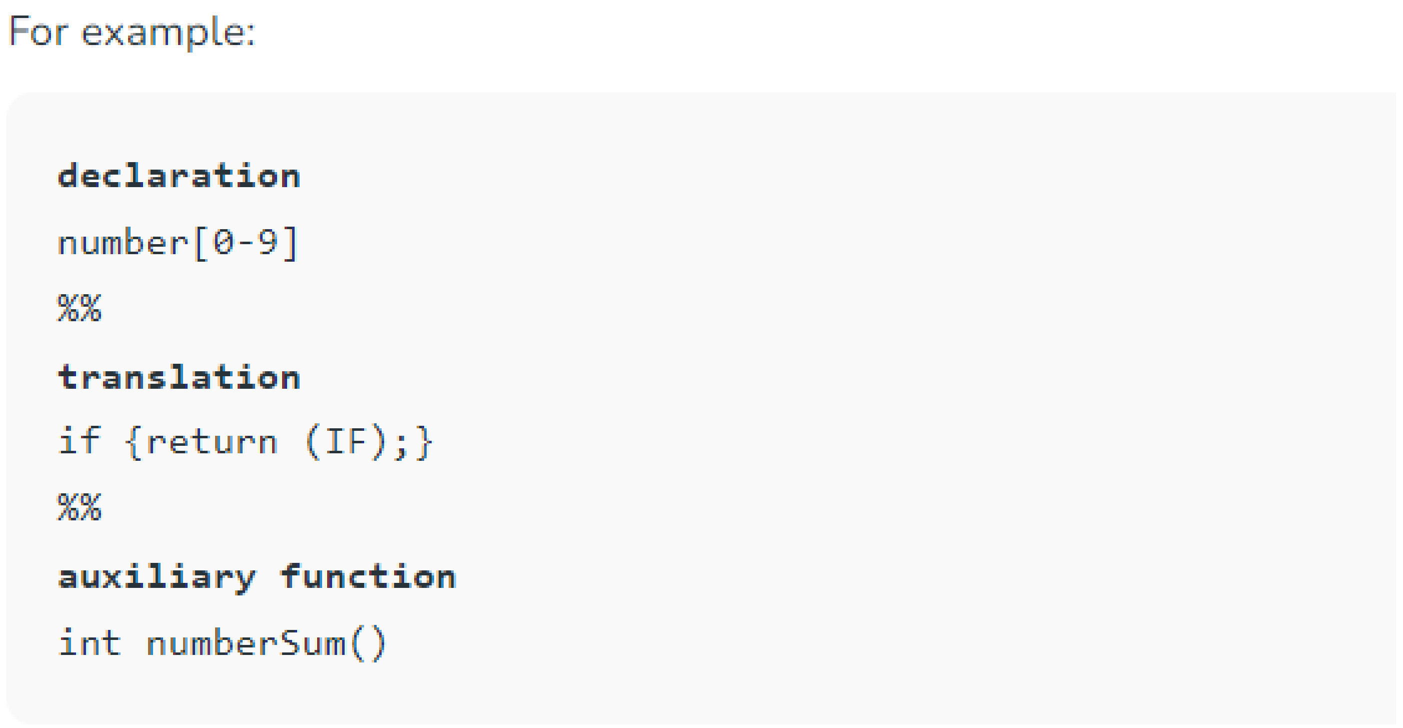
Lex File Format

A Lex program consists of three parts and is separated by %% delimiters:-

```
Declarations
%%
Translation rules
%%
Auxiliary procedures
```

Declarations: The declarations include declarations of variables.
Transition rules: These rules consist of Pattern and Action.
Auxiliary procedures: The Auxilary section holds auxiliary functions used in the actions.

LEX



For example:

```
declaration
number[0-9]
%%
translation
if {return (IF);}
%%
auxiliary function
int numberSum()
```

YAAC

What is Yacc?

Yacc is officially known as a "parser".

It's job is to analyse the structure of the input stream, and operate of the "big picture".

In the course of it's normal work, the parser also verifies that the input is syntactically sound.

Consider again the example of a C-compiler. In the C-language, a word can be a function name or a variable, depending on whether it is followed by a (or a = There should be exactly one } for each { in the program.

YACC stands for "Yet Another Compiler Compiler". This is because this kind of analysis of text files is normally associated with writing compilers.

However, as we will see, it can be applied to almost any situation where text-based input is being used.

For example, a C program may contain something like:

In this case, the lexical analyser would have broken the input sream into a series of "tokens", like this:

```
{
        int int;
        int = 33;
        printf("int: %d\n",int);
}
```

In this case, the lexical analyser would have broken the input sream into a series of "tokens", like this:

```
{
int
int
;
int
=
33
;
printf
(
"int: %d\n"
,
int
)
;
}
```

- Note that the lexical analyser has already determined that where the keyword int appears within quotes, it is really just part of a litteral string.
- It is up to the parser to decide if the token int is being used as a keyword or variable. Or it may choose to reject the use of the name int as a variable name.
- The parser also ensures that each statement ends with a ; and that the brackets balance.

- Download Flex 2.5.4a: http://gnuwin32.sourceforge.net/packages/flex.htm
- Download Bison 2.4.1: http://gnuwin32.sourceforge.net/packages/bison.htm
- Download DevC++: https://www.bloodshed.net/
- **Installing the Software and setting up the environment variable (path):**
- Install Flex at “C:\GnuWin32”
- Install Bison at “C:\GnuWin32”
- Install DevC++ at “C:\Dev-Cpp”

Reference:How to Compile Run LEX YACC Programs on Windows - VTUPulse

- Download Flex 2.5.4a:

http://gnuwin32.sourceforge.net/packages/flex.htm

- Download Bison 2.4.1:

http://gnuwin32.sourceforge.net/packages/bison.htm

- Dev-C++ download

https://sourceforge.net/projects/orwelldevcpp/files/latest/download

SET THE ENVIRONMENT VARIABLES

```
C:\Dev-Cpp\MinGW64\bin
C:\GnuWin32\bin
```

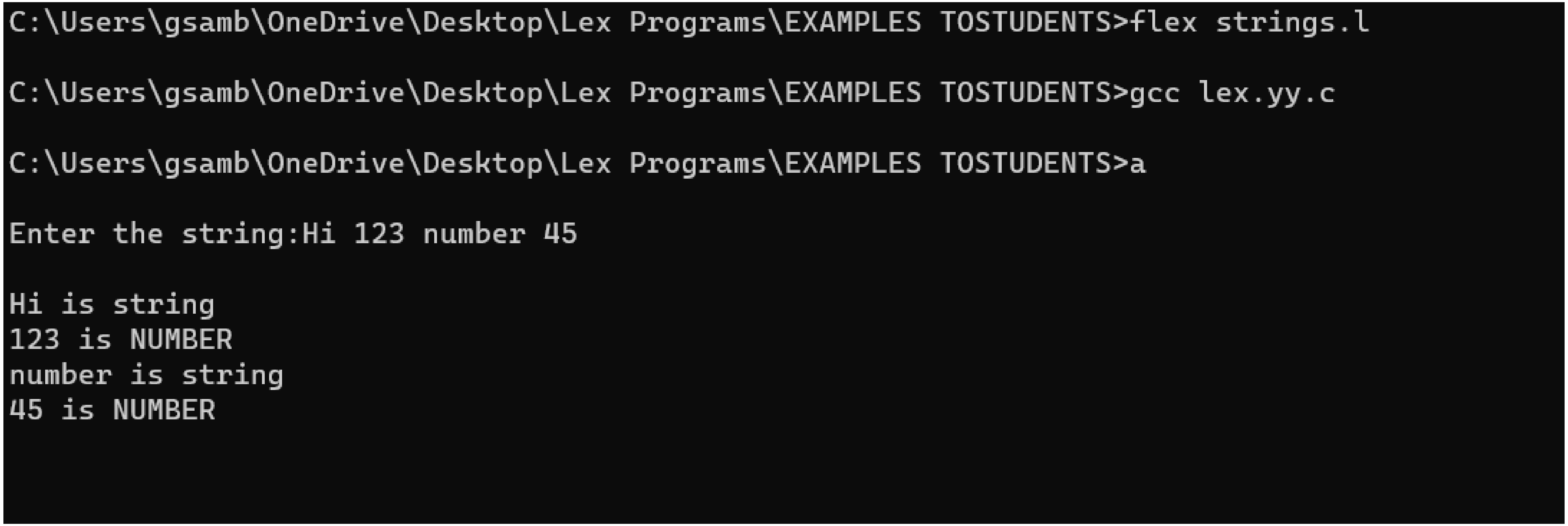
Lex Program to recognize and display keywords, numbers, and words in a given statement

```
%{
#include<stdio.h>
%}
%%
if |
else |
printf {printf("%s is a keyword",yytext);}
[a-zA-Z]+ { printf("\n%s is string", yytext);}
[0-9]+ { printf("\n%s is NUMBER", yytext);}
.|\n {ECHO;}
%%


int main()
{
printf("\nEnter the string:");
```

Lex Program to recognize and display keywords, numbers, and words in a given statement

```
int main()
{
printf("\nEnter the string:");
yylex();
}

int yywrap()
{
  return 0;
}
```

Lex Program to recognize and display keywords, numbers, and words in a given statement



```
C:\Users\gsamb\OneDrive\Desktop\Lex Programs\EXAMPLES TOSTUDENTS>flex strings.l

C:\Users\gsamb\OneDrive\Desktop\Lex Programs\EXAMPLES TOSTUDENTS>gcc lex.yy.c

C:\Users\gsamb\OneDrive\Desktop\Lex Programs\EXAMPLES TOSTUDENTS>a

Enter the string:Hi 123 number 45

Hi is string
123 is NUMBER
number is string
45 is NUMBER
```

LEX

1. LEX PROGRAM TO COUNT NUMBER OF WORDS.
2. LEX PROGRAM TO COUNT THE NUMBER OF LINES, SPACES AND TABS
3. LEX PROGRAM TO COUNT THE NUMBER OF VOWELS AND CONSONANTS IN A GIVEN STRING.
4. LEX PROGRAM TO COUNT THE POSITIVE NUMBERS, NEGATIVE NUMBERS AND FRACTIONS