UNIT-I

Syllabus:
Introduction: The Structure of a Compiler, Phases of Compilation, The Translation Process, Major Data Structures in a Compiler, Bootstrapping and Porting.
Lexical Analysis (Scanner): The Role of the Lexical Analyzer, Input Buffering, Specification of Tokens, Recognition of Tokens, The Lexical Analyzer Generator Lex.
Objective: Basic ideas in concepts of designing and implementing translators for various languages and system building tools like LEX is introduced for lexical analysis.
Outcome:

Introduction

In this new era, all are using software application in their daily lives. Software applications are written in some programming languages. These programs must be converted into a form which can be executed by a computer. Software that does this conversion is called as translators.

Types of Translators

1. Interpreter
2. Assembler
3. Compiler

1) Interpreter is one of the translators that translate high level language to low level language. An interpreter reads the source code one instruction or line at a time, converts this line into machine code and executes it. The machine code is then discarded and the next line is read.

2) Assembler is a software or a tool that translates Assembly language to machine code. So, an assembler is a type of a compiler, and the source code is written in Assembly language. Assembly is a human readable language, but it typically has a one to one relationship with the corresponding machine code.

3)Compiler is a program that translates one language(source code) to another language (target code). During this translation, reporting errors to user.

Difference between Compiler and Interpreter

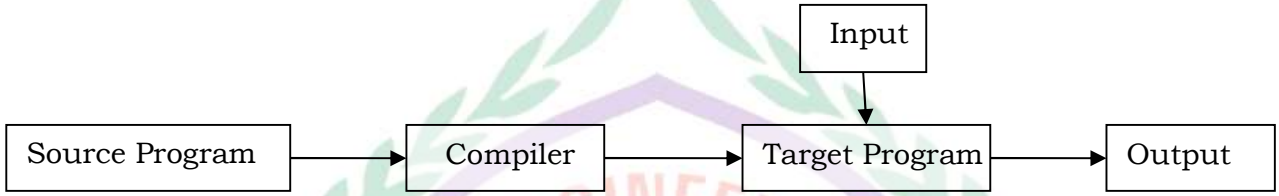
Compiler scans the entire program once and then converts it into machine language which can then be executed by computer's processor. In short compiler translates the entire program in one go and then executes it. Interpreter on the other hand first converts high level language into an intermediate code and then executes it line by line.

The execution of program is faster in compiler than interpreter as in interpreter code is executed line by line.

Compiler generates error report after translation of entire code whereas in case of interpreter once an error is encountered it is notified and no further code is scanned.

Compiler: A compiler is a system software which is used to translate the program written in one programming language into machine understandable language to be executed by a computer. Compiler should make the target code efficient and optimized in terms of time and space.

A compiler reads a program written in one language called as the source language and translate it into an equivalent program in another language called as target language. In this conversion process compiler detects and reports any syntactical errors in the source program.



Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.

First computers of late 1940s were programmed in machine language. They are soon replaced by assembly language instructions and memory locations are specified with symbolic names. An assembler translates the symbolic assembly code into equivalent machine code. Assembly language is an improved programming language but still it is machine dependent. Later high level languages are introduced, where programs are written in English related statements.

An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user. The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs . An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Brief History

• The term “compiler” was coined in the early 1950s by Grace Murray Hopper. Translation was then viewed as the “compilation” of a sequence of routines selected from a library

• The first compiler of the high-level language FORTRAN was developed between 1954 and 1957 at IBM by a group led by John Backus

• The study of the scanning and parsing problems was pursued in the 1960s and 1970s and led fairly to a complete solution

• The development of methods for generating efficient target code, known as optimization techniques, is still ongoing research

• Compiler technology was also applied in rather unexpected areas:

Classification of Compilers

Compilers are sometimes classified as single-pass, multi-pass, load-and-go, debugging, or optimizing, depending on how they have been constructed or on what function they are supposed to perform.

The processes of translation of program to machine understandable code is divided into two parts.

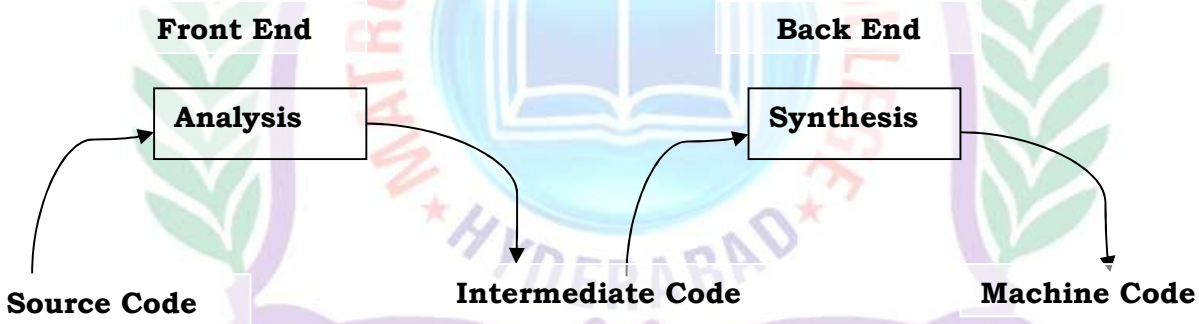
1) Analysis part
2) Synthesis part

1) Analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The analysis phase contains the following stages of compiler.

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis

2) Synthesis part constructs the desired target program from the intermediate representation of the source program.

- Intermediate Code Generator
- Code Optimizer
- Code Generator

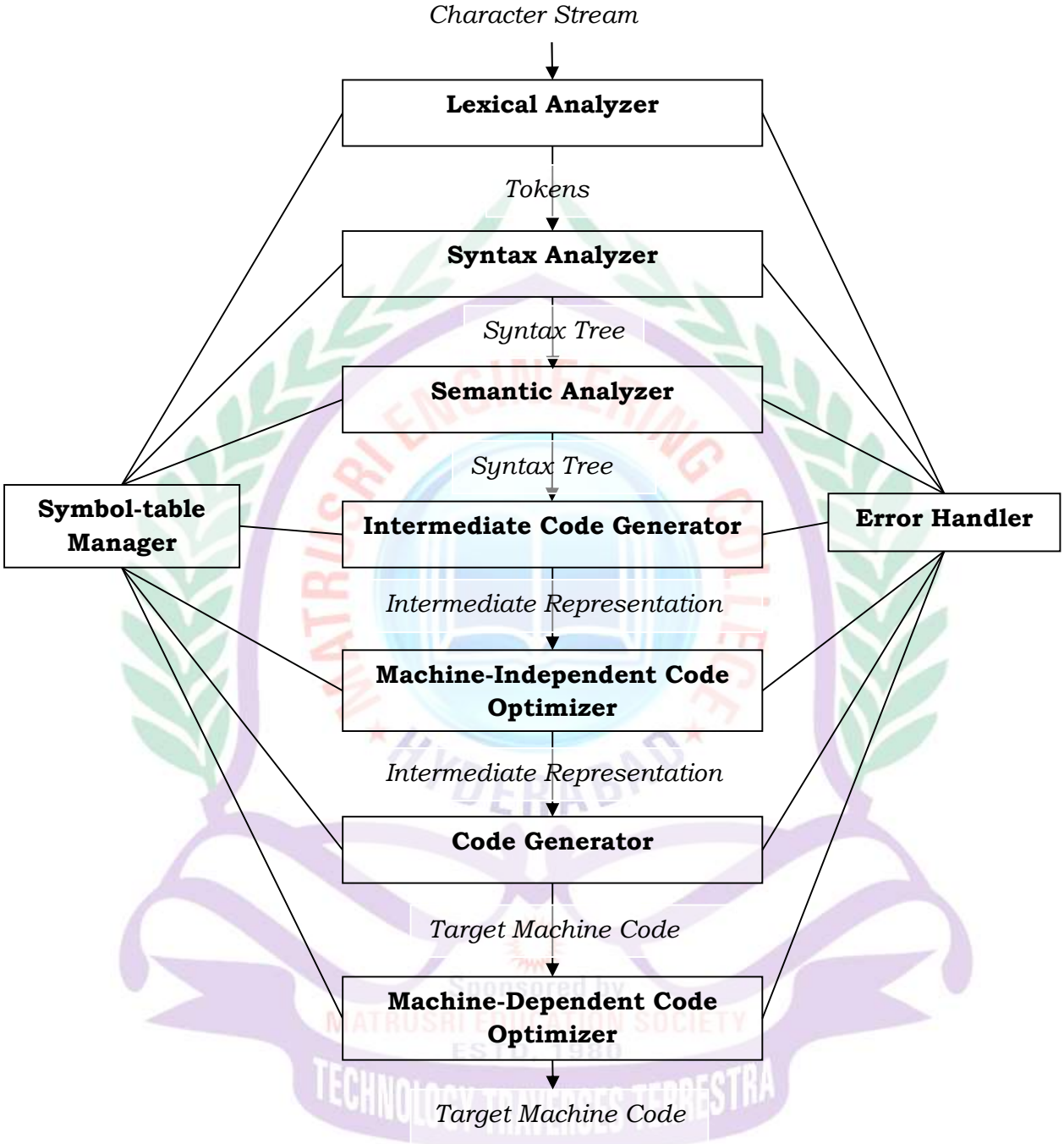


Properties of a good compiler

- Reliability: It must generate correct machine code.
- Faithfulness: Compiler itself must be a bug-free.
- Compilation Speed: Compiler mut run fast(Compilation time must be proportional to program size)
- Diagnostics: It must give good diagnostics and error messages.
- Error handling: must report error to user.
- Implementation and Maintenance: portable and must work well with existing debuggers.
- Good human interface: Easy to use

The Structure of a Compiler

Compiler operates as a sequence of phases, each phase transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in the below figure.



Phases of compiler

The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target code. Since optimization is optional, one or the other of the two optimization phases shown in the above figure may be ignored.

Phases of Compilation

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Code Generation
- Code Optimization
- Code Generation
- Symbol Table Management
- Error Handling

Lexical Analysis:

The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters that make up the source program starting from left to right and groups the characters into meaningful sequences called *lexemes*. For each lexeme, the lexical analyzer produces a *token* of the form

<token-name, attribute-value>

- token-name is an abstract symbol that is used during syntax analysis.
- attribute-value points to an entry in the symbol table

These tokens are passed to syntax analyzer.

Token: It represents a logically cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc.

Example: p = i + r * 60

- Here p, =, i, +, r, *, 60 are all separate lexemes.
- <id,1> <=> <id,2> <+> <id,3> <*> <60> are the tokens generated.

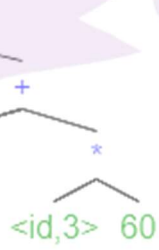
Syntax Analysis:

The second phase of the compiler is syntax analysis or parsing. The parser uses the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments(operands) of the operation.

Example:

For p = i + r * 60, the syntax tree is

<id,1> = <id,2> + <id,3> * 60

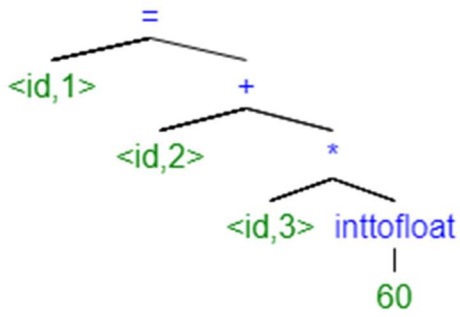


Semantic Analysis:

It is the third phase of the compiler. The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It performs type conversion of all the data types into real data types.

An important part of semantic analysis is type checking. Some language specification may permit some type conversions called coercions. Example ***inttofloat.***

Example:
For p = i + r * 60, the syntax tree is
<id,1> = <id,2> + <id,3> * 60



Intermediate Code Generation
It is the fourth phase of the compiler. In the process of translating a source program into target code, a compiler may construct one or more intermediate representations.

After semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation of the source program.

- three-address code is one of the intermediate code representations.

three-address code consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register.

Example:
For p = i + r * 60, the syntax tree is
<id,1> = <id,2> + <id,3> * 60

Three-address code for the above code is

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Code Optimization:
It is the fifth phase of the compiler. It gets the intermediate code as input and produces optimized intermediate code as output.

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code is generated. Better means faster, shorter code, or target code that consumes less power.

Example:
For p = i + r * 60, the syntax tree is
<id,1> = <id,2> + <id,3> * 60

Optimized code is:

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generation:
It is the sixth phase of the compiler. The code generator takes intermediate representation/optimized machine independent representation as input and maps

it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used in the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

Example:

For p = i + r * 60, the syntax tree is

<id,1> = <id,2> + <id,3> * 60

using registers R1 and R2, the machine code is:

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

Symbol Table Manager:

- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- It allows to find the record for each identifier quickly and to store or retrieve data from that record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

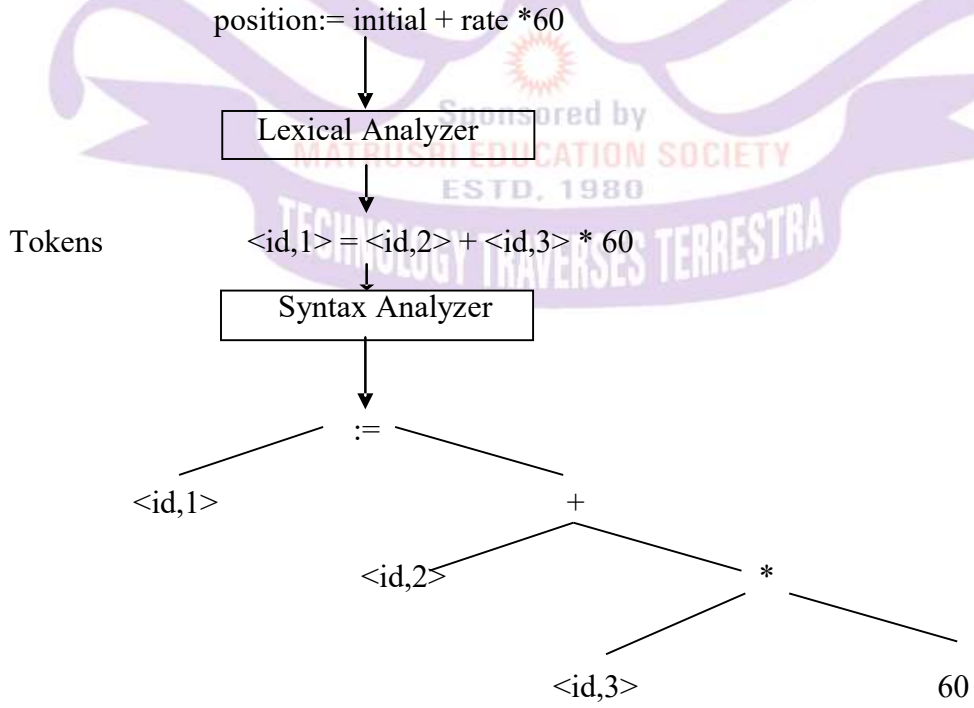
Error Handling:

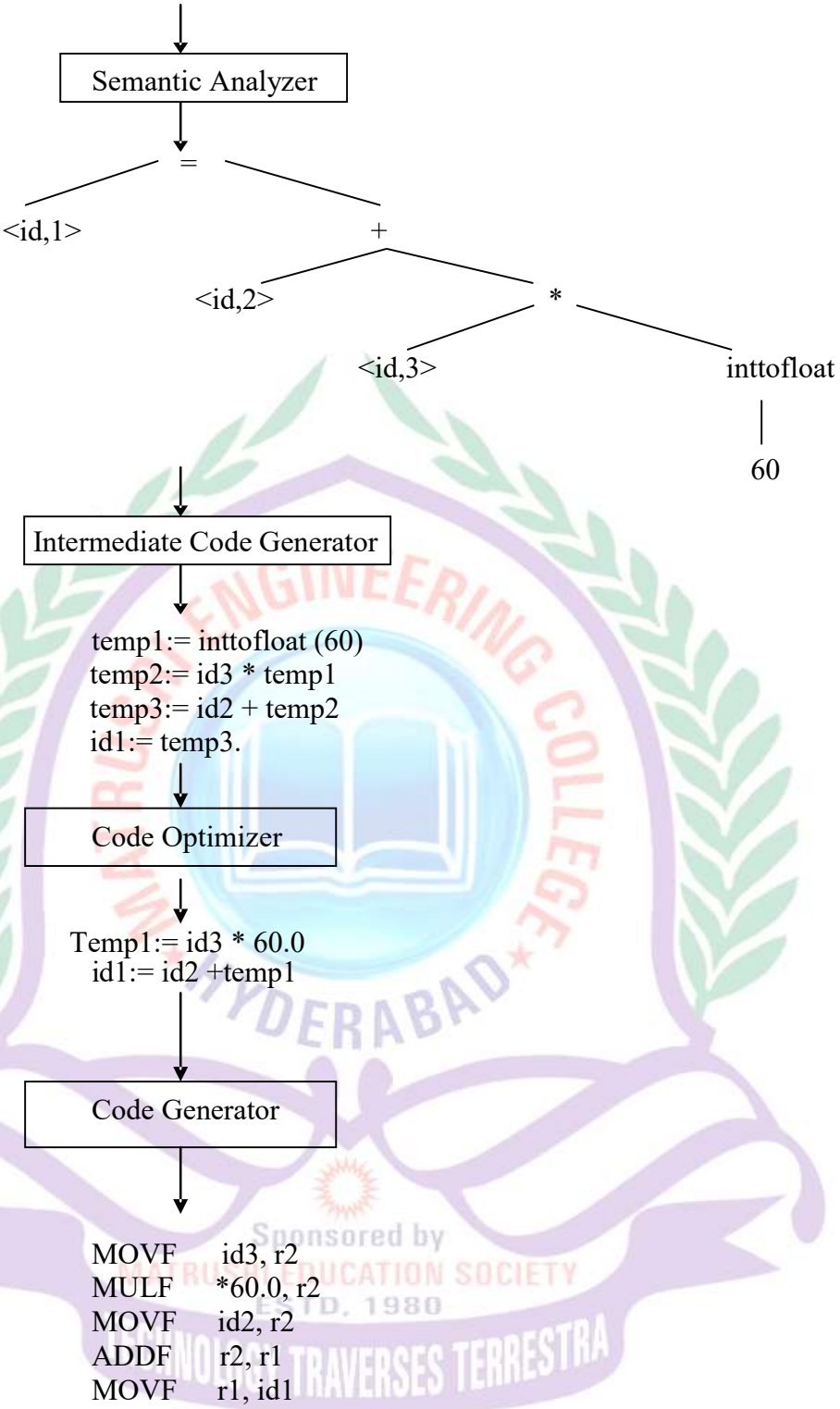
One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error message should allow the programmer to determine exactly where the errors have occurred.

- Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.

The Translation Process

To illustrate the translation of source code through each phase, consider the statement ***position = initial + rate * 60***





Errors Encountered in different Phases:

Each of the six phases (but mainly the front end) of a compiler detect errors. On detecting an error, the compiler must:

- report the error in a helpful way for better understanding,
- correct the error if possible, and
- continue processing (if possible) after the error to look for further errors.

A program may have the following kinds of errors at various stages:

- Lexical : name of some identifier typed incorrectly
- Syntactical : missing semicolon or unbalanced parenthesis

- Semantical : incompatible value assignment
- Logical : code not reachable, infinite loop

Types of Error: Two types of error usually encountered in programs.

Syntax errors are errors in the program text. they may be either lexical or grammatical.

- A lexical error is a mistake in a lexeme, for examples, typing *tehn* instead of *then*, or missing off one of the quotes in a literal.
- A grammatical error is a one that violates the (grammatical) rules of the language, for example if x = 7 y := 4 (missing semicolon).

Syntax errors must be detected by a compiler and at least reported to the user (in a helpful way). If possible, the compiler should make the appropriate correction(s).

Semantic errors are mistakes concerning the meaning of a program construct; they may be either type errors, logical errors, or run-time errors.

- Type errors occur when an operator is applied to an argument of the wrong type, or to the wrong number of arguments.
- Logical errors occur when a badly conceived program is executed, for example:
  while x = y do …
  when x and y initially have the same value and the body of loop need not change the value of either x or y.
- Run-time errors are errors that can be detected only when the program is executed.
  for example: var x : real; readln(x); writeln(1/x)
  which would produce a run time error if the user input 0.

Semantic errors are much harder and sometimes impossible for a computer to detect.

Compiler-Construction Tools

Software development environments contains tools like language editors, debuggers, version managers, profilers, test harnesses, and so on are used to construct compilers. In addition to general software-development tools, other more specialized tools are created to implement various phases of a compiler. They are

1.Parser generators: It automatically produces syntax analyzers from a grammatical description of a programming language.
2. Scanner generators: It produces lexical analyzers from a regular-expression description of the tokens of a language.
3. Syntax-directed translation engines: They produce collections of routines for walking a parse tree and generating intermediate code.
4. Code-generator: It produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
5. Dataflow analysis engines: It facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Dataflow analysis is a key part of code optimization.
6. Compiler-construction toolkits: They provide an integrated set of routines for constructing various phases of a compiler.

Major Data Structures in a Compiler:

Algorithms used in individual phases of a compiler frequently interacts with data structures for their efficient implementation. so that, a compiler compiles a program in O(n) time irrespective of the size of the program. Few data structures that are used in different phases are:

Tokens: Scanner collects characters/lexeme into a token. It represents the token symbolically as a value of an enumerated data type representing a set of tokens of the source language. Sometimes, it is necessary to preserve the character string itself or other information derived from it. In most languages the scanner needs to generate one token at a time (single symbol lookahead). So, a single global variable can be used to hold the token information. In other cases, an array is required to hold the lexeme or token.

Syntax Tree: Parser generates syntax tree. The syntax tree is constructed as a standard pointer-based structure that is dynamically allocated. Entire tree can be kept as a single variable pointing to the root. Each node is a record. Its fields represent the information collected by the parser and the semantic analyzer.

Symbol Table: Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.

Symbol table is used by both the analysis and the synthesis parts of a compiler.

- A symbol table may serve the following purposes:-
- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

Scanner, parser may enter identifiers into table, semantic analyzer will add data type and other information to identifiers.

Literal Table: The Literal Table Stores constants and strings used in the program.
One literal table applies globally to the entire program
Used by the code generator to:

- Assign addresses for literals
- Enter data definitions in the target code file

Avoids the replication of constants and strings. Quick insertion and lookup are essential. Deletion is not allowed.

Intermediate Code:
Depending on the kind of intermediate code, it may be kept in an array of text strings, a temporary text file, Linked list of structures.

Temporary Files:
Computers did not have enough memory for the entire program to be kept in memory during compilation. This was solved by using temporary files to hold the products of intermediate steps.

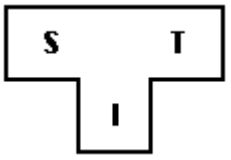
Now a days, memory constrains are not at all a problem. Occasionally, compilers generate intermediate files during some of the steps.

Bootstrapping and Porting:

A compiler is characterized by three languages:

- Source Language **(S)**: Language in which programs are written
- Target Language **(T)**: Language understood by the machine. i.e., **S** is to be translated into **T** with the help of **I**
- Implementation Language **(I)**: Language used to write translator/compiler

Compilers are represented in the form of a T-diagram or S_I^T



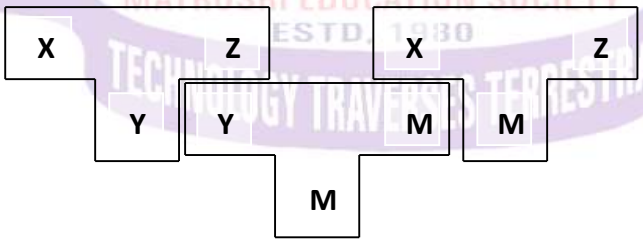
Developing translator for a new language from scratch is a non-trivial exercise (because, we have to use machine language for writing the compiler).

So, when a translator is required for an old language on a new machine, or a new language on an old machine, use of existing compilers on either machine is the best choice for developing. i.e., write the compiler in another language for which a compiler already exists.

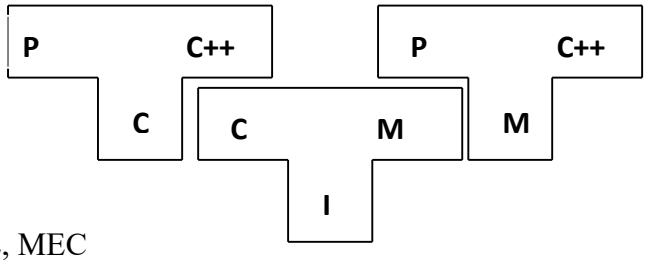
Developing a compiler for a new language by using an existing compiler is called ***bootstrapping.*** Bootstrapping is the process by which a simple language is used to translate more complicated programs, which intern may handle an even more complicated program.

Bootstrapping is used to create compilers and to move them from one machine to another by modifying the back end.

For example: To write a compiler for new language X and the implementation language of this compiler is say Y and the target code being generated is in language Z. That is, we create XYZ. Now if Y runs on machine M and generates code for M then it is denoted as YMM. Now if we run XYZ using YMM then we get a compiler XMZ. That means a compiler for source language X that generates a target code in language Z and which runs on machine M.



Development of Pascal translator for C++ language is done by bootstrapping Pascal translator for C language with C language translator for M.



Porting:
The process of modifying an existing compiler to work on a new machine is often known as porting the compiler.

To develop a compiler for new hardware machine from an existing compiler, change the synthesis part of the compiler because, synthesis part is machine dependent part. This is called porting.

Quick and dirty Compiler:

Native Compiler: Native compiler are compilers that generates code for the same Platform on which it runs. It converts high language into computer's native language.

Cross Compiler: A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.
Bootstrap Compiler**.**

Lexical Analysis (Scanner)

In the first phase of compilation process, the source program is divided into characters or sequence of characters called as tokens. Tokens are like character or words of natural language. Each token represents a unit of information in the source program.
In the source program the following elements are considered as tokens.

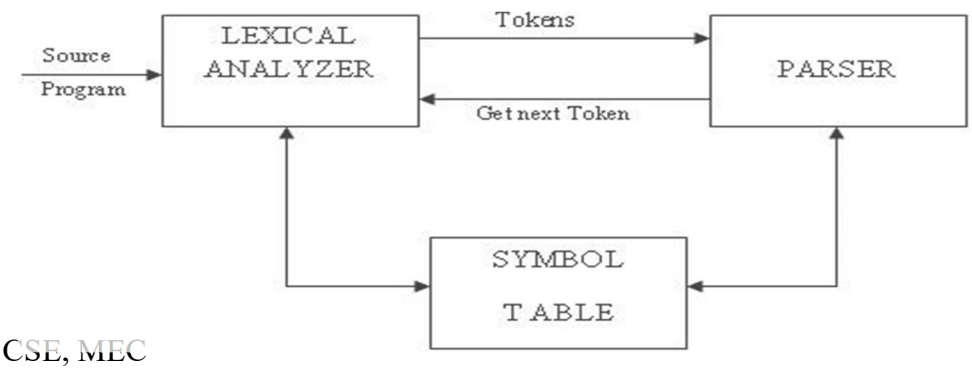
- Keywords/Reserve words: predefined words of the language (Lexemes)
- Identifiers: use defined strings (ID)
- Special Symbols

Any value associated to a token is called as an attribute of the token. Attribute may be a string or numerical.

The ***getNextToken*** command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

The Role of the Lexical Analyzer
The job of the lexical analyzer (scanner) is to read the source program character by character and form them into logical units called as tokens. These tokens are given as inputs to next phase of compiler. Scanner rarely converts the entire program into tokens at once, but conversion always depends on the parser.

Lexical analyzer interacts with the symbol table when it discovers a lexeme constituting an identifier and enters that lexeme into the symbol table.



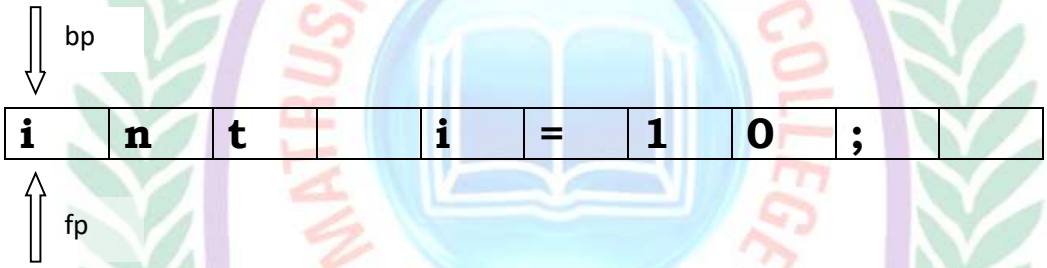
Scanner performs the following tasks:

- identification of lexemes
- stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
- correlating error messages generated by the compiler with the source program

Input Buffering:

The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(bp) and forward ptr(fp) to keep track of the pointer of the input scanned.

The forward ptr(fp) moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. Once a blank space(whitespace) is encountered, lexeme is identified and whitespace is ignored, them both pointers are placed at the next character.



Input character are always read from secondary storage, but this reading is costly. To speed up the scanning process, buffering technique is used. A block of data is first read into a buffer and lexical analysis process id continued on buffer.

Buffering techniques:

1. Buffer (one or two buffers are used)
2. Sentinels

Buffering:

One Buffer Scheme: In this scheme, only one buffer is used to store the input string. The problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled.

Two Buffer Scheme: To overcome the problem of one buffer scheme, two buffers are used to store the input string. The first buffer and second buffer are scanned alternately.

Initially both the ***bp*** and ***fp*** are pointing to the first character of first buffer. Then the ***fp*** moves towards right in search of end of lexeme. As soon as blank character is recognized, the string between bp and ***fp*** is identified as corresponding token. To identify the boundary of first buffer, end of buffer character should be placed at the end first buffer.

Sentinel:
Special character introduced at the end of the buffer is called as Sentinel which is not a part of program. ***eof*** is the natural choice for sentinel.

Specification of Tokens:

Regular expressions are an important notation for specifying lexeme patterns.

Strings and Languages:

An alphabet is any finite set of symbols. Examples of symbols are letters, digits, and punctuation. The set {0, 1} is the binary alphabet. ASCII and Unicode are important examples of an alphabet. A string over an alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms "sentence" and "word" are often used as synonyms for "string".

A language is any countable set of strings over some fixed alphabet including null set “ø” and set with empty character {ε}.

1. ***Alphabets***: Any finite set of symbols

{0,1} is a set of binary alphabets,
{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets,
{a-z, A-Z} is a set of English language alphabets.
2. Strings: Any finite sequence of alphabets is called a string.

3. ***Special symbols***: A typical high-level language contains the following symbols:

Arithmetic Symbols	Addition(+), Subtraction(-), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.)
Assignment	=
Special assignment	+=, -=, *=, /=
Comparison	==, !=. <. <=. >, >=
Preprocessor	#

4. ***Language***: A language is considered as a finite set of strings over some finite set of alphabets.

5. ***Longest match rule***: When the lexical analyzer read the source-code, it scans the code letter by letter and when it encounters a whitespace, operator symbol, or special symbols it decides that a word is completed.

6. ***Operations***: The various operations on languages are:

Union of two languages L and M is written as, L U M = {s | s is in L or s is in M}
Concatenation of two languages L and M is written as, LM = {st | s is in L and t is in M}

- The Kleene Closure of a language L is written as, L* = Zero or more occurrence of language L.

7. ***Notations***: If r and s are regular expressions denoting the languages L(r) and L(s), then

Union : L(r) U L(s)
Concatenation : L(r) L(s)
Kleene closure : (L(r))*

8. ***Representing valid tokens of a language in regular expression***: If x is a regular expression, then:

- x* means zero or more occurrence of x.
- x+ means one or more occurrence of x.

9. ***Finite automata***: Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly. If the input string is successfully processed and the automata reaches its final state, it is accepted. The mathematical model of finite automata consists of:

- Finite set of states (Q)
- Finite set of input symbols (Σ)
- One Start state (q0)
- Set of final states (qf)
- Transition function (δ)

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), $Q \times \Sigma \rightarrow Q$

Recognition of Tokens:
Patterns are created by using regular expression. These patterns are used to build a piece of code that examines the input strings to find a prefix that matches the required lexemes.

Consider the below grammar for branching statements and recognize the tokens from it.

```
stmt -> if expr then  stmt
      |  If expr then else stmt
      | ϵ
expr -> term relop term
      | term
term -> id
      |number
```

The terminals of the grammar are if, then, else, relop, id, and number, which are the names of tokens. The patterns for these tokens are described using the following regular definitions.

```
digit      -> [0-9]
digits     -> digit+
number     -> digits (. digits)? ( E [+-]? digits )?
letter     -> [A-Za-z]
id         -> letter ( letter j digit )
if         -> if
```

```
then    -> then
else    -> else
relop   -> < | > | <= | >= | = | <>
```

with the help of above definitions, lexical analyzer will recognize, if, then, else as keywords, and relop, id, number as lexemes

Lexical analyzer also strips out white space, by recognizing the "token" with the following regular expression:

```
ws   -> (blank/tab/newline)+
```

The below table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what is the attribute value of token.

Lexeme	Token Name	Attribute Value
Any ws	_	_
if	if	_
then	then	_
else	else	_
Any id	id	Pointer to table entry
Any number	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	ET
< >	relop	NE

The Lexical Analyzer Generator Lex.

We can also produce a lexical analyzer automatically by specifying the lexeme patterns to a lexical-analyzer generator and compiling those patterns into code that functions as a lexical analyzer.

- Modification of lexical analyzer is easy, since we have only to rewrite the affected patterns, not the entire program.
- It also speeds up the process of implementing the lexical analyzer.

A lexical-analyzer generator called Lex (flex)

A Lex input file consists of three parts.

1. A collection of definitions.
2. A collection of rules.
3. A collection of auxiliary routines or user routines.

All the sections are separated by double percent signs. Default layout of a Lex file is:

{definitions}
%%
{rules}
%%
{auxiliary routines}

Declaration Section: The declarations section includes declarations of variables, identifiers (which are declared to stand for a constant, e.g., the name of a token), and regular definitions.
The following syntax is used to include declaration section in LEX specification

```
%{
        Declarations
%}
```

Rules Section: The translation rules each have the form **$Rule_i$ { $Action_i$ }.**

Each rule is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C. The following syntax is used to include rules section in LEX specification

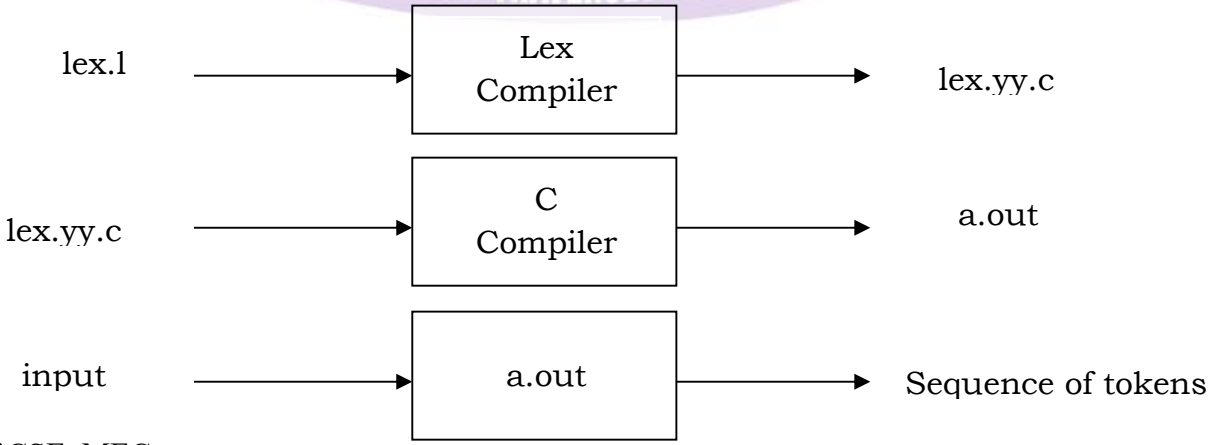
```
%%
        Rule1 { Action1 }
        Rule2 { Action2 }
%%
```

When lexical analyzer starts reading the input character by character. If a character/set of characters is matched with one of the regular expressions, then the corresponding action part will be executed.

Auxiliary Routines: The third section contains additional functions that are required. These functions may be compiled separately and loaded with the lexical analyzer.
Some procedures are required by actions in rule section.
yylex(), yywrap() are the predefined procedures of LEX.

Lex program or Lex files are saved with .l extension(dot l).

The input notation for the Lex tool is referred as the Lex language and the tool itself is the Lex compiler. The Lex compiler transforms ***lex.l*** to a C program, in a file that is always named ***lex.yy.c***. The latter file is compiled by the C compiler into a file called ***a.out***.



Predefined functions and variables

Lex Predefined functions and Variables	
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(input reaches to end), 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

The below program appends line number to the lines of the loaded file:

```
                /* Declaration section */
%{
   int yylineno;
%}
                /* Rules section */
%%
^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
%%
                /* Auxiliary Procedures */

int main(int argc, char **argv)
{
    yyin = fopen(argv[1], "r");
    yylex();
    fclose(yyin);
}
```

Word count without using files

```
%{
#include<stdio.h>
int wc=0,lc=0,cc=0;
%}
word [^ \n\t]+
eol \n
%%
```

```
{word} {wc++;cc+=yyleng;}
{eol} {lc++;}
%%
void main()
{
printf(“Enter a line of text:\n”);
scanf(“%s”,yytext);
yylex();
printf("\nno. of characters:%d\nno. of words:%d\nno. of lines:%d\n",cc,wc,lc);
}
int yywrap()
{
return 1;
}
```