Unit-1: Introduction to Compilers

1.1 Compilers and Translators

If you are unable to speak French and yet wish to communicate a French speaker then you need someone to translate English into French. The same happens with computers languages. We would like to communicate in English but the computer only understands binary - so a translator is required. Thus the basic function of a translator is to convert a SOURCE (or original) program into an Object (or binary) program.

There are three main categories of translator:

- Assemblers,
- Interpreters and
- Compilers.

Under normal circumstances (ie unless great speed or compactness is required) the source program will be written in a high level language which is either interpreted or compiled.

A translator can be defined as: "**A device that changes a sentence from one language to another without change of meaning.**"

1.1.1 Assemblers

In the early days of programming, machine code (binary) was the only option. Unfortunately this was laborious, prone to error and difficult. A slight improvement on this was the use of hexadecimal or octal which reduced the number of errors and the time to enter the program. Eventually assembly languages were developed which were easier and more productive to use whilst preserving the speed and compactness of machine code.

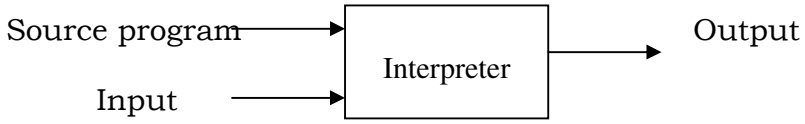
Assembly languages vary from one type of computer to another (or more correctly from processor to processor) which results in a difficulty in transporting programs from one computer to another.

Assemblers are the simplest of all the translators to understand since the majority of the statements in the source code are mnemonics (short words that help you to remember something) representing specific binary patterns - the others being labels, directives (or pseudo-ops) which give instructions to the assembler.

So, for instance, rather than enter the binary pattern 01011100 which might mean "Increment the contents of the Accumulator by 1" we could type in the mnemonic "INC" which the assembler would translate into the appropriate binary pattern.

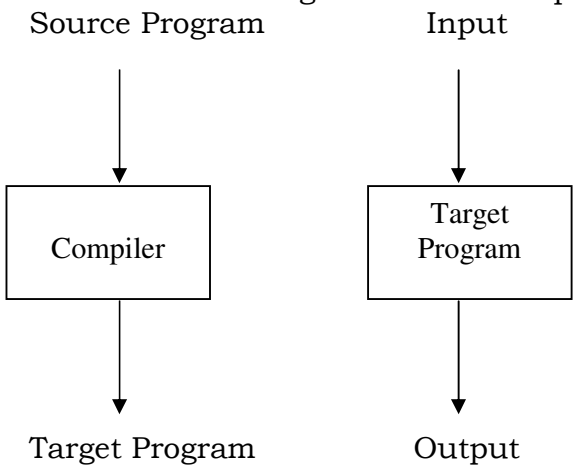
1.1.2 Interpreters

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, as shown in Fig.



1.1.3 Compilers

A compiler is a program that can read a program in one language - the *source* language - and translate it into an equivalent program in another language - the *target* language. An important role of the compiler is to report any errors in the source program that it detects during the translation process.



If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

1.2 Need of Translators

A computer system can understand only a machine language. With machine language, the user must communicate directly with computer in terms of bits, registers, and very primitive machine operations. Since a machine language program is nothing more than a sequence of 0's and 1's, programming a complex algorithm in such a language is terribly tedious and fraught with opportunities for mistakes. Perhaps the most serious disadvantage of machine – language coding is that all operations and operands must be specified in a numeric code.

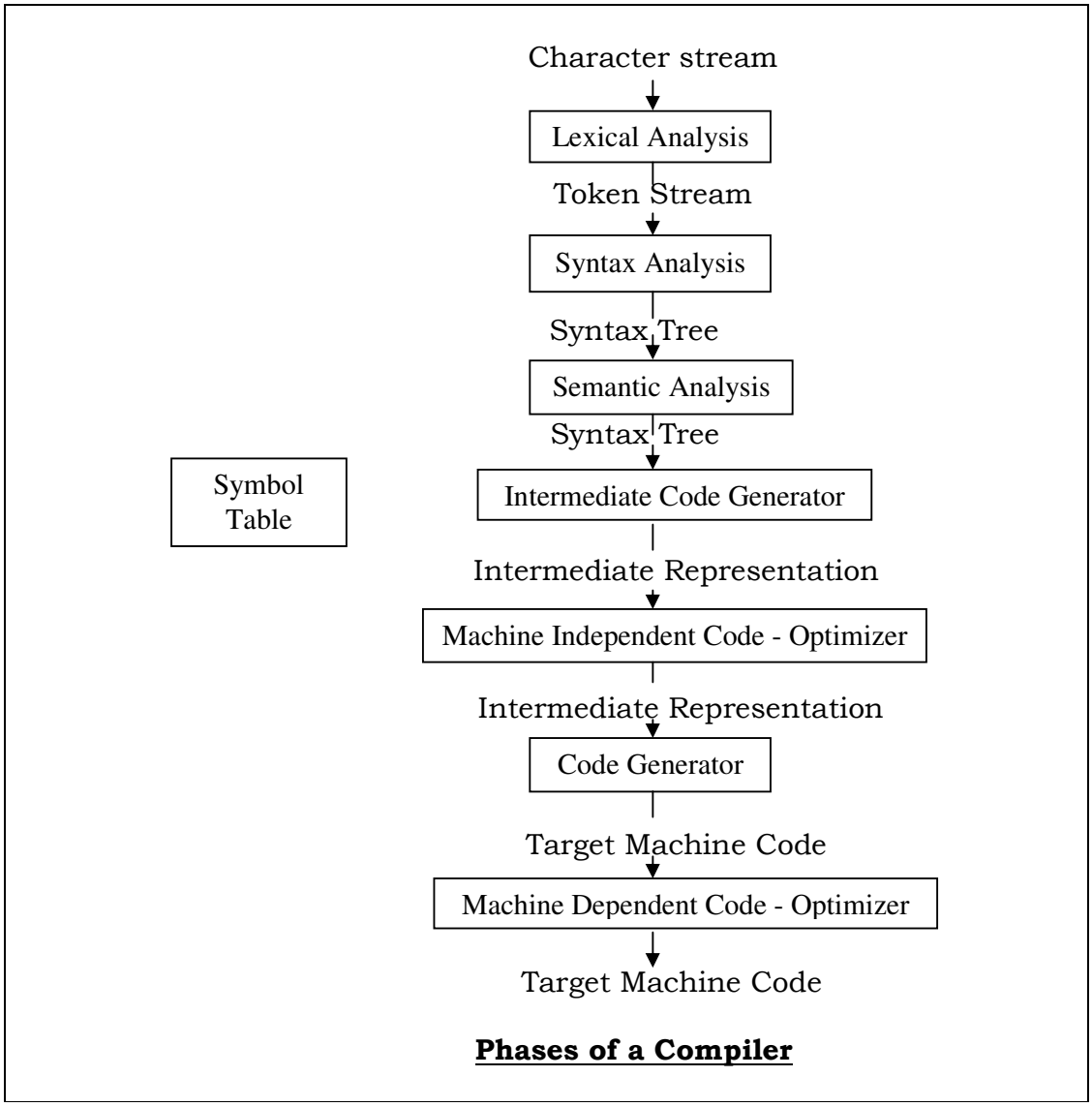
So it is required to convert the machine language into user – understandable language and user – understandable language into machine language.

A computer cannot execute a program written in assembly language. That program has to be first translated into machine language, which the computer can understand.

Thus arises the need of translators.

1.3 Structure of Compilers

Mapping of a source program into a semantically equivalent target program is divided into two parts: analysis and synthesis.



Phases of a Compiler

The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.
The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.

The analysis part is often called the front end of the compiler; the synthesis part is the back end.

Compiler operates as a sequence of phases, each of which transforms one representation of the source program to another. These phases are:

1.3.1 Lexical Analysis

The lexical analyzer is the interface between the source program and the compiler. It reads the source program one character at a time, carving the source program into a sequence of atomic units called tokens. In other words, the main function of the lexical analyzer is to determine the tokens, that may be identifiers, keywords, constants, operations, and punctuation symbols such as commas and parentheses.

For example, suppose a source program contains the following statement:

IF (5 eq MAX) GOTO 100

The characters in this statement are mapped into the following eight tokens passed on to the syntax analyzer:

“IF”, “(”, “5”, “eq”, “MAX”, “)”, “GOTO”, and “100”

Blanks separating the lexemes would be discarded by the lexical analyzer.

1.3.2 Syntax Analysis

The second phase of the compiler is syntax analysis or parsing. A parser has two functions as:

(i) To check whether the tokens occurring in the input are permitted by the specification of the source language, and
(ii) To give the sequence of tokens, a tree like structure also called as ***parse tree***.

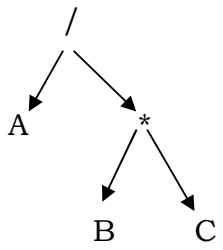
The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a

syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

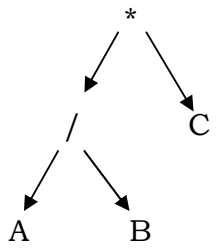
For example, consider an operation

A / B * C

Parser will generate two parse trees for the statement as:



(a)



(b)

PARSER checks whether the output of lexical analyzer satisfies the context free grammar (CFG).

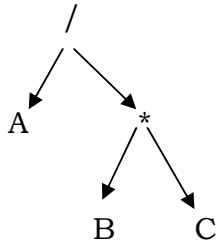
1.3.3 Semantic Analysis

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 also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation. An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

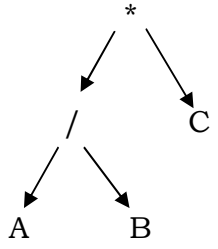
1.3.4 Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

For example, the parse tree generated for the statement: A/B*C



(a)



(b)

$T_1 = B * C$	$T_1 = A/B$
$T_2 = T_1/A$	$T_2 = T_1*C$
(a)	(b)

1.3.5 Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

Code optimization can be done in two ways:

Local Optimization: There are local transformations that can be applied to a program to attempt an improvement.

For example, consider the following statements:

```
            if A > B GOTO L2
            GOTO L3
L2:
            ----------
            ----------
L3:
            ----------
            ----------
```

This sequence could be replaced by the single statement

```
            if A <=B GOTO L3
L3:
            ----------
            ----------
```

Loop Optimization: A typical improvement is to move a computation that produces the same result, each time around the loop to a point in the program just before the loop is entered.

1.3.6 Code Generation

The code generator takes as input an intermediate representation of the source program 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 by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.

For example, A statement A = B + C is mapped into the target language code sequence as:

```
LOAD B
ADD C
STORE A
```

1.3.7 Symbol Table Management (BOOK KEEPING)

A compiler need to collect information about all the data objects that appear in the source program. The information is collected by early phases of the compiler – lexical and syntactic analysis – and entered into the symbol table. The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

1.3.8 Grouping of Phases into Phases

Grouping of phases deals with the logical organization of a compiler. In an implementation, activities from several phases may be grouped together into a pass that reads an input file and writes an output file. For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass. Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine.

1.4 Compiler Construction Tools

The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on. In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler.

The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler. Some commonly used compiler-construction tools include:

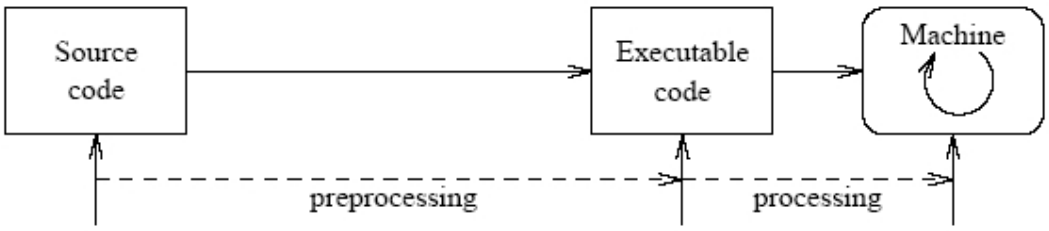
1. Parser generators that automatically produce syntax analyzers from a grammatical description of a programming language.
2. Scanner generators that produce lexical analyzers from a regular-expression description of the tokens of a language.
3. Syntax-directed translation engines that produce collections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

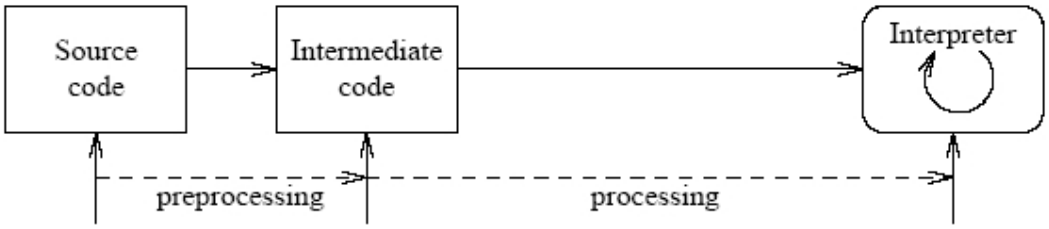
1.4. Compiler Vs Interpreters

Compiler and Interpreter can be distinguished as follows:

Compiler	Interpreter
1. Spends a lot of time analyzing and processing the program.	1. Relatively little time is spent analyzing and processing the program.
2. The resulting executable is some form of machine – specific binary code.	2. The resulting code is some sort of intermediate code.
3. The computer hardware interprets (executes) the resulting code.	3 The resulting code is interpreted by another program.
4 Program execution is fast.	4. Program execution is relatively slow.



Compilation



Interpretation

Representation of Differences