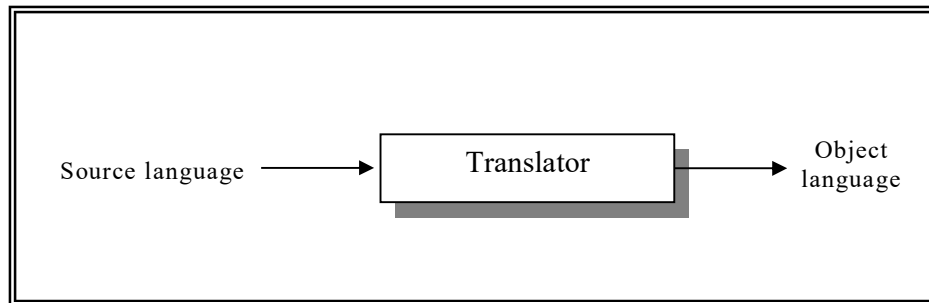


Compilers and Translators

1.1 Translators:

A translator is a program that takes as input a program written in one programming language (the *source* language) and produces as output a program in another language (the *object* or *target* language).



If the source language is a high –level language (HLL) such as C, C++, FORTRAN, etc and the target language is a machine language, then the translator is called a **compiler**.

1.2 INTERPRETER:

An interpreter is a program that translates the source code (instructions in a high level language) into object code instructions (machine code) and the computer *immediately* executes that instruction before translating the next instruction. The interpreter **does not** produce an object code for the entire program. Certain instructions that have to be repeated (e.g., instructions in a loop) may be interpreted time and again.

1.2.1 Characteristics of interpreter:

- Machine code is not stored.
- Source code is essential for repeated execution of statements.
- Statement is analyzed during its interpretation.
- Useful for testing and debugging as overhead of storage is not incurred.

1.2.2 Advantages of using an interpreter:

1. Interpreter programs are smaller in size and save memory space.
2. It is simpler to develop an interpreter than to develop a compiler since interpretation does not involve code generation.
3. It eliminates the need to store program's translated object-code in the computer.
4. They are easy to use because they are totally interactive.
5. Interpreters are more useful where commands are not executed repeatedly e.g., commands for an operating system.
6. Interpreters can cope with languages in which the size and type of variables can depend on input data.

7. The overhead of compilation is reduced (code optimization, code generation phases).
8. Interpreters are advantageous during program development because a program may be modified between every two executions and its output can be seen immediately.

1.2.3 Disadvantages of using an interpreter:

1. Slow speed.
2. Interpretation is expensive in terms of CPU time because each statement is subjected to the interpretation cycle.
3. Statements that are used multiple times (e.g. in a loop) must be translated each time they are executed.

1.3 ASSEMBLER:

If the source language is assembly language and the target language is machine language then the translator is called an assembler.

Some compilers produce assembly code that is subsequently passed to an assembler for further processing.

Assembly code is a *mnemonic* version of machine code. Names are used instead of numeric codes for operations. A typical sequence of assembly instructions is shown below:

```
MOVE A, R1
ADD #2, R1
MOVE R1, B
```

1.4 THE NEED FOR TRANSLATORS:

The computer can only follow instructions given to it in 0's and 1's (machine language). However, writing programs in machine language is not only extremely tedious, bug-prone, but also the rich data structures supported by HLL cannot be used if programs are written in machine language. In machine code, all operations must be specified in a numeric code, e.g., the numeric code for addition of two numbers (ADD) may be, say, 2. Machine language programs are cryptic and impossible to debug and maintain. Hence, we need translators that can convert programs written in HLL to machine language.

1.5 MACROS

Many programming languages provide a macro facility whereby a macro statement will translate into a sequence of assembly language statements before being translated into machine code. *A macro facility is a text replacement facility.* Macros are handled by *macro pre-processors*.

A macro must be defined before it is used. Keywords like `define` and `macro` are used for macro definitions. Examples of macros are the `#define` macros in C language.

1.5 HIGH LEVEL LANGUAGES

1. HLL allows a programmer to express algorithms in a more natural notation. E.g., to add two numbers and store the result in a variable `c`, we may write the statement `c = a + b` in a HLL such as C language. However, the same instruction in machine code will be more than 3 to 4 instructions, and even in assembly language, the statement may be written in its simplest form as:

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

2. HLL permit the use of rich data structures such as arrays, structures, linked list, trees, etc which make data abstraction possible.
3. HLL do not require the programmer to be thoroughly familiar with the architecture of the computer such as details of the CPU registers, etc. Programming task is thus made simpler.
4. HLL also introduce certain complexities - e.g., the need for a translator to convert the HLL code into machine code. This is where compilers and interpreters come in!
5. Examples of HLL are C, C++, Java, FORTRAN (FORmula TRANslation), Visual Basic, Pascal (No full form, named after a French scientist), COBOL (COmmon Business Oriented Language)

1.5.1 Advantages of HLL:

High-level languages possess several advantages over low-level ones:

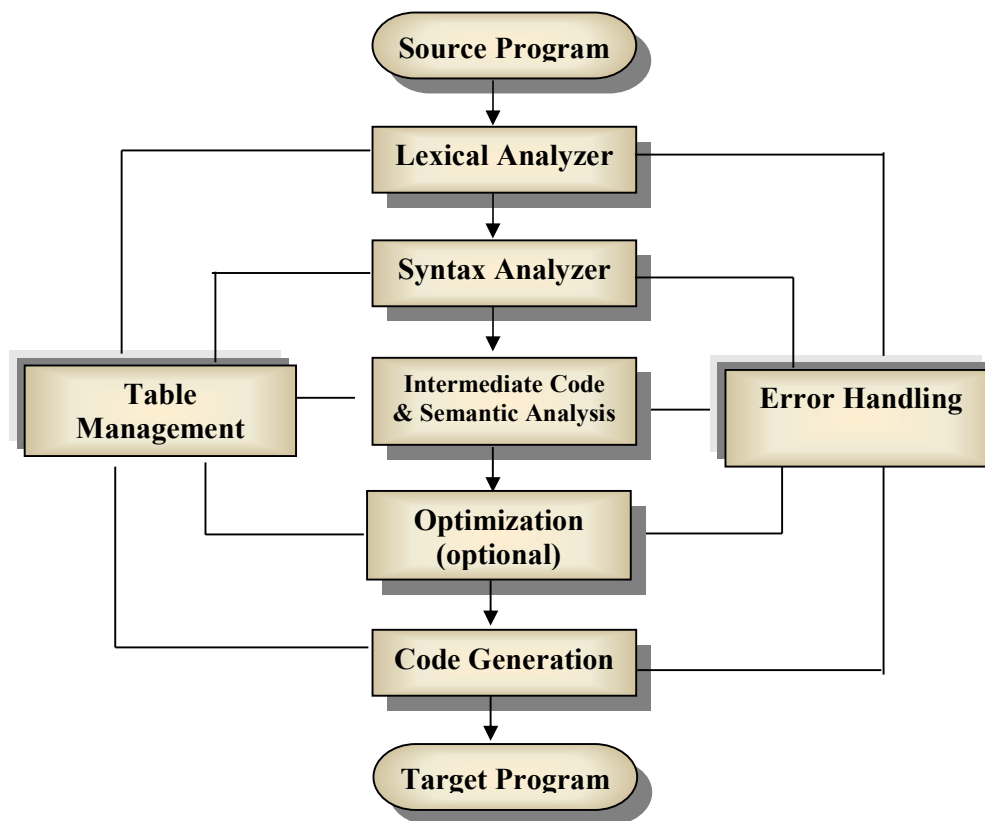
1. **Readability:** A good high-level language will allow programs to be written that in some ways resemble a quasi-English description of the underlying algorithms. If care is taken, the coding may be done in a way that is essentially self-documenting.
2. **Portability:** High-level languages, being essentially machine independent, can be used to develop portable software. This is software that can run unchanged on a variety of different machines - provided only that the source code is recompiled as it moves from machine to machine.
3. **Structure and object orientation:** The structured programming movement of the 1960's and the object-oriented movement of the 1990's have resulted in a great improvement in the quality and reliability of code. High-level languages can be designed so as to encourage these programming paradigms.
4. **Generality:** Most high-level languages allow the writing of a wide variety of programs, thus relieving the programmer of the need to become expert in many diverse languages.

5. **Brevity:** Programs expressed in high-level languages are often considerably shorter (in terms of their number of source lines) than their low-level equivalents.

6. **Error checking:** A programmer is likely to make many mistakes in the development of a computer program. Many high-level languages can enforce a great deal of error checking both at compile-time and at run-time.

1.6 THE STRUCTURE OF A COMPILER

A compiler is a software that accepts a program written in a high-level language and produces its machine language equivalent. The process of compilation takes place in several phases, which are shown below.



Phases of a Compiler

The tasks of the various phases of a compiler are explained below:

1.6.1 Lexical Analysis Phase:

This is the first phase of a compiler. This phase is also called the *scanning* phase. The compiler scans the source code from left to right, character by character, and groups these characters into *tokens*. Each token represents a sequence of characters such as variables, keywords, multi-character operators (such as :=, >=, ==, etc). The main functions of this phase are summarized below:

- (i) Identify the lexical units in a source statement.
- (ii) Classify units into different lexical classes e.g., constants, reserved words, etc., and enter them in different tables.
- (iii) build a descriptor (called a *token*) for each lexical unit.
- (iv) ignore comments in the source program.
- (v) detect tokens that are not a part of the language.

Consider the following program:

```
main()
{
    int m, x, c, y;
    m = 2;
    c = 1;
    scanf("%d", &x);
    Y = m * x + c;
    printf("%d", y);
}
```

In this example, the tokens are:

main	()	int	m	,	x	,	c	,	y	;	m	=
2	;	c	=	1	;	scanf	("	%	d	"	,	&
x)	;	y	=	m	*	x	+	C	;	printf	("
%	d	"	,	y)	;	}						

1.6.2 Syntax Analysis Phase:

This phase is also called the *parsing* phase. Sometimes this phase is grouped along with the lexical analysis phase in the same pass. The following operations are performed in this phase:

- (i) Obtain tokens from lexical analyzer.
- (ii) check whether the expression is syntactically correct.
- (iii) report syntax errors, if any.

If a program contains a statement such as

$$A + / B$$

After the lexical analysis, the syntax analysis phase should detect an error since the presence of two adjacent binary operators violates the rules of C language.

- (iv) determine the statement class, i.e., is it an assignment statement, a condition statement (if statement), etc.
- (v) group tokens into statements,
- (vi) construct hierarchical structures called *parse trees*. These parse trees represent the syntactic structure of the program.

E.g., consider the following statement:

$$y = m * x + c$$

This expression could have two possible meanings:

- a) Multiply m by x and then add the result to c . or
- b) Add x and c and then multiply the result by m .

A *parse tree* can represent these two interpretations. A parse tree represents the syntactic structure of an expression. The parse tree representing the first meaning is shown below in fig (a). Draw the parse tree for the second meaning in the box below (Fig (b)).

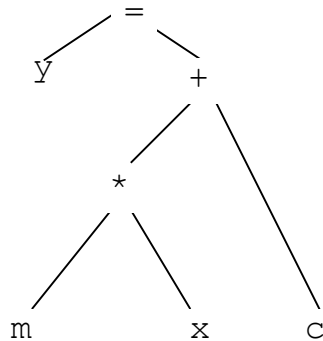


Fig (a)

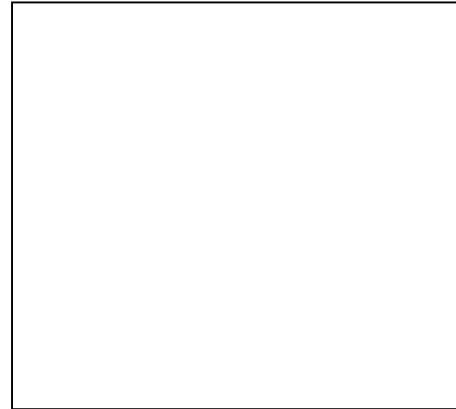


Fig (b)

Which of the two interpretations is to be used is decided by the language specification. These rules are specified in the syntactic specification of the language.

1.6.3 Intermediate Code Generation Phase:

After the syntax and the semantic analysis phase, compilers generate an intermediate representation of the source program. This intermediate representation should satisfy the following two properties:

- a) it should be easy to produce, and
- b) it should be easy to translate into another language.

Three-Address code

One of the intermediate forms produced is called the “three address code”. It is like an assembly language for a machine in which every memory location can act like a register. Three-address code consists of a sequence of instructions, each of which has at most three operands.

Consider a typical three-address code statement such as:

$$A = B \text{ op } C$$

where A , B and C are operands and op is a binary operator.

The following sequence of statements can be used instead of the parse tree shown above:

$$\begin{aligned} T1 &= m * x \\ T2 &= T1 + c \end{aligned}$$

Where $T1$ and $T2$ are temporary variables.

Thus we see that each instruction has at most three operands, and only one operator in addition to assignment.

An intermediate language also needs unconditional and simple conditional branching statements in which at most one relation is tested to determine whether or not a branch is to be made.

Control statements such as `if-then-else`, `while-do`, etc are translated into lower-level conditional three address statements.

Consider the statement

```
while A > B && A <= 2 * B - 5 do
    A = A + B
```

Its intermediate code can be shown to be:

```
L1:  if A > B goto L2
      goto L3
L2:  T1 = 2 * B
      T2 = T1 - 5
      If A <= T2 goto L4
      goto L3
L4:  A = A + B
      goto L1
L3:
```

Compilers do not generally produce a parse tree but rather go to the intermediate code directly as syntax analysis takes place.

The generation of an intermediate code offers the following advantages:

- (i) Flexibility: a single lexical analyzer / parser can be used to generate code for several different machines by providing separate *back-ends* that translate a common intermediate language to a machine-specific assembly language.
- (ii) Intermediate code is used in interpretation. The intermediate code is executed directly rather than translating it into binary code and storing it.

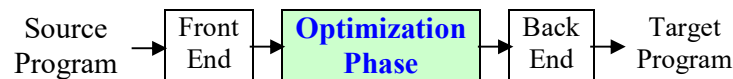
1.6.4 Semantic Phase:

The semantic phase has the following functions:

- (i) check phrases for semantic errors e.g., type-checking. In a C program, `int x = 10.5` should be detected as a semantic error.
- (ii) semantic analyzer keeps track of types of identifiers and expressions, to verify their consistent usage.
- (iii) semantic analyzer maintains the *symbol table*. The symbol table contains information about each identifier in a program. This information includes identifier type, scope of identifier, etc.
- (iv) using the symbol table, the semantic analyzer enforces a large number of rules such as:
 - a. every identifier is declared before it is used.
 - b. no identifier is used in an inappropriate context (e.g., adding a string to an integer);

- c. subroutine or function calls have a correct number and type of arguments,
 - d. every function contains at least one statement that specifies a return value.
- These values are checked at compile time, hence they are called *static semantics*.
- (v) Certain semantic rules are checked at run time; these are called *dynamic semantics*.
Examples of these are:
 - a. array subscript expression lie within the bounds of the array.
 - b. variables are never used in an expression unless they have been given a value.

1.6.5 Code Optimization Phase:



Optimization improves programs by making them smaller or faster or both. The **goal** of code optimization is to translate a program into a new version that computes the same result more efficiently – by taking less time, memory, and other system resources. For example, the C compiler “Turbo C” permits the programmer to optimize code for speed or for size.

Code optimization is achieved in two ways:

- a) rearranging computations in a program to make them execute more efficiently,
- b) eliminating redundancies in a program.

Note the following points about code optimization:

- i. Code optimization should not change the meaning of the program.
- ii. Code optimization tries to improve the program; the underlying algorithm is not affected.
- iii. Code optimization cannot replace an inefficient algorithm with an algorithm that is more efficient.
- iv. Code optimization also cannot fully utilize the instruction set of a particular architecture. Thus, code optimization is independent of the target machine and the PL.

Types of Optimizing Transformations:

The two types of optimizing transformations are:

- a) local transformations (applied over small segments of a program) , and
- b) global transformations (applied over larger segments consisting of loops or function bodies).

Local Optimization:

Local optimizations are those that can be performed by looking only at the statements in a basic block.

Some of the optimizing techniques are:

a) **Dead Code elimination:**


Code that can be omitted from a program without affecting its results is called *dead code*. If a variable is assigned a value that is never used subsequently in the program, then that statement is a dead code.

E.g., $j = 30$;

Turbo C can point out such instances by way of a warning message.

b) Elimination of common sub-expressions:

Expressions that yield the same value are called common sub-expressions or equivalent expressions. Consider the following code segments:

<pre> a := b * c; : : x := b * c + 3.5; </pre>		<pre> t := b*c; a := t; : x := t + 3.5; </pre>
--	---	--

Here the two occurrences of $b * c$ were eliminated by using a variable t .

c) Frequency Reduction:

Execution time of a program can be reduced by moving code from a part of a program which is executed very frequently to another part of the program which is executed fewer times. E.g., the transformation of loop optimization involves moving loop invariant code out of a loop.

d) Strength reduction:

This optimization technique reduces operation with a more efficient operation or a series of operations that yield the same result in fewer machine clock cycles.

For example, multiplication by a power of two is replaced by a left shift, which executes faster on most machines.

$a = b * 4;$ becomes $a = b + b + b + b;$ or $a = b \ll 2;$

Similarly, division by powers of two is expensive operations and these can be replaced with right shift.

$c = d / 2;$ becomes $c = d \gg 1;$

e) Loop optimization:

Most of the programs spend a lot of time inside the loops and hence loops are prime targets for optimization. The running time of a program may be improved if we reduce the number of instructions inside a loop. There are three main techniques for loop optimization:

- (1) **code motion** – move loop-invariant computations outside the loop (e.g. no need to have $a = 22 / 7 * r * r$ inside a loop. Instead put the statement $pi = 22 / 7$ outside the loop and then use the statement $a = pi * r * r$ inside a loop.
- (2) **Induction variable elimination**, and
- (3) **Strength reduction** – (discussed earlier)

1.6.6 Code Generation Phase:

This is the final phase of compilation process. It takes as input the intermediate representation (optimized) of the source program and produces as output an equivalent target code. The code generated depends upon the architecture of the target machine. Knowledge of the instructions and addressing modes in target computer is necessary for code generation process.

One of the important aspects of code generation is the efficient initialization of machine resources.

A number of assumptions may be made such as:

- a) correct code (more important than optimized code!)

- b) instruction types in target machines
- c) Proper usage of registers (register allocation)

1.6.7 Difference between a phase and a pass of a compilation

Compilation proceeds through a set of well-defined phases; these are the lexical analysis, syntax analysis, semantic analysis, intermediate code generation, machine independent code optimization, code generation, and machine dependent code generation. Each phase discovers information of use to later phases, or transforms the program into a form that is more useful to the subsequent phases.

A pass is a phase or a set of phases that is serialized with respect to the rest of the compilation. A pass does not start until the previous phases have been completed and it finishes before any subsequent phases start.

The structure of the source language decides the number of passes. A multi-pass compiler is slower than a single pass compiler. But a multi-pass compiler occupies more space than a single pass compiler.

1.7 BOOKKEEPING (Databases Maintained By The Compiler)

The compiler must maintain a complete database about all the data objects appearing in a source program. E.g.,

- a) data type of a variable (`int`, `float`, `char`, etc),
- b) type and size of an array (e.g., `int a[1]` and `float *p[10]`),
- c) the number and type of arguments required by a function and the return type of a function.

The information about data types may be explicit (as through declarations) or implicit (as through the first character of a variable name).

The **symbol table** is used to store information about various data objects. This table can be thought of as a dictionary. A typical structure of a symbol table can be as follows:

Index	Symbol	Type
1	m	float
2	x	float
3	c	int
4	y	float

This information collected about an identifier has many **uses**: it may be used for example to decide the data type of result in case of mixed-mode arithmetic. In case the language does not permit mixed mode arithmetic, then the compiler may issue an appropriate error message.

1.8.1 ERROR HANDLING

Detection and reporting errors is one of the most important tasks of a compiler. The error messages should allow the programmer to determine the location and type of error. A compiler should produce helpful error messages (such as `statement missing ; in line 30`)

rather than a cryptic message (such as Error No 2437 in line 30), since the later requires having to look up a manual or online help.

Errors can be encountered in various phases of compilation such as:

- i) lexical analyzer is unable to proceed because the next token in the source code is misspelled.
- ii) Syntax error such as missing parenthesis, or a missing ; symbol,
- iii) Intermediate code generator has detected an operator whose operands have incompatible types (e.g. "CPU" + 4)
- iv) Code optimizer may detect that certain code is unreachable
- v) Code generator phase may find a compiler-created constant that is too large to fit in a word of the target machine
- vi) Multiple entries in symbol table for same identifier but with different attributes (declared first time as `int` and then subsequently as `float`).

The **error handling routine** of the compiler issues appropriate messages.

1.8.2 Syntax Errors vs Logical Errors

Syntax Error	Logical Error
1. These are errors in the use of a programming language. The word " <i>syntax</i> " means grammar.	1. These are errors in the logic of the program.
2. The compiler or interpreter detects these errors.	2. The compiler or interpreter cannot detect such errors.
3. The user must correct the error and recompile or re-run the program in order to generate the correct object code.	3. The object code can be generated for a program with logical errors.
4. Since these errors are detected by the compiler / interpreter, they are less likely to cause a program crash - that is, a malfunctioning program.	4. Since a logical error cannot be detected by the compiler / interpreter, such an error may go unnoticed for a very long time and then may cause program/system crash.
5. <u>Examples</u> of such errors are : use of PRIN statement in BASIC instead of PRINT statement, or using = sign instead of == sign in C language.	5. <u>Example</u> : deducting a tax of 1.5% instead of 15%, or deducting HRA from Gross salary instead of adding it to gross salary.

REVIEW QUESTIONS

1. Define the term interpreter. What are the characteristics of an interpreter?
2. State the advantages and disadvantages of using an interpreter. Under what circumstances is the use of an interpreter preferred over a compiler. Elaborate.
3. What is the need for translators?
4. What is a macro? What are the advantages of a macro?
5. State the main features of high-level languages. What are the advantages of using HLL?
6. In what ways are high-level languages an improvement over assembly language? In what circumstances does it still make sense to program in assembly language?

7. List the principal phases of compilation and describe the work performed by each phase.
8. What is meant by code optimization? Is this the same as code improvement? Discuss any two code optimization techniques.
9. What is the difference between a phase and a pass of a compilation?
10. Explain the term "bookkeeping" in the context of compilation.
11. Explain the various type of errors that can be encountered in the compilation process.
12. What is(are) the difference(s) between syntax errors and logical errors?

ASSIGNMENT

1. Programming languages can be classified into "**declarative**" and "**imperative**". Examples of the former are languages such as Lisp, Scheme, Prolog, etc. The later type is more familiar to you (C, C++, BASIC, Java, Smalltalk, Pascal, etc). Use the Internet (or other resources available to you) to find out more about characteristics of each of these types and their relative merits and demerits. Under what circumstances are these various languages used ?
2. Study in detail the various options provided by a compiler such as Turbo C 2.0 or Turbo C 3.0 (also known as Turbo C ++). In particular study the different options available under "Options" menu in Turbo C 2.0 or Turbo C 3.0.
3. Approximately how many programming languages are there ? Why are so many languages needed ?

* * * * *