# Part – 9: Complier Design

## 9.1 Introduction to Compilers

**Translator:** A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator performs another role, the error-detection. Any violation of the HLL specification would be detected and reported to the programmers. Important role of translator are: translating the HLL program input into an equivalent ML program and providing diagnostic messages wherever the programmer violates specification of the HLL.

**Type of Translators**: - Interpreter, Compiler, Preprossessor, Etc

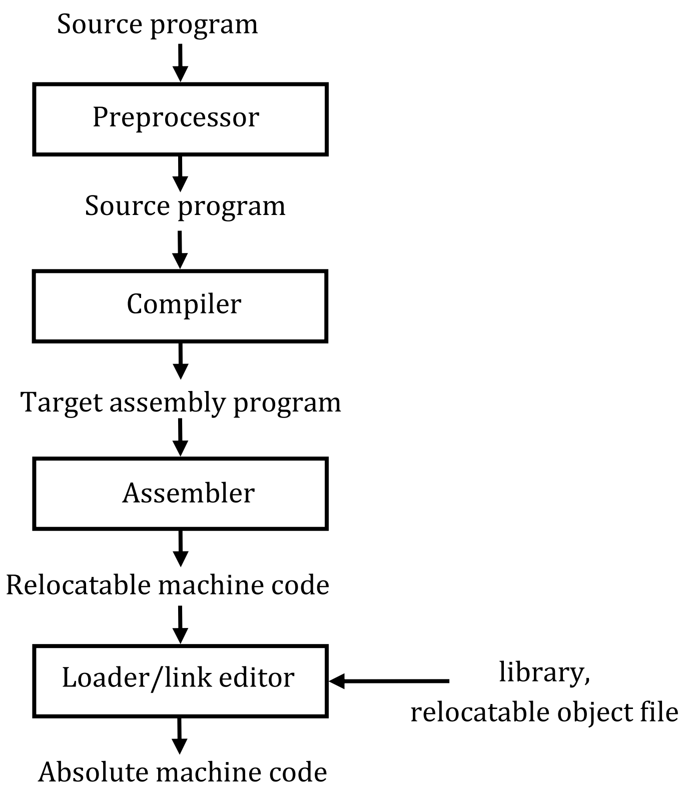


**Fig. 9.1.1.A language-processing system.**

In addition to a compiler, several other programs may be required to create an executable target program. Figure 9.1.1 shows a typical language processing process along with the compilation".

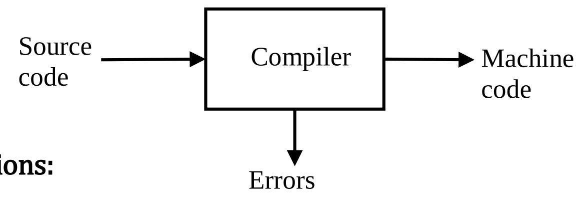
**Interpreter:** Unlike compiler, interpreter takes single instruction & executes.

**Advantage of Interpreter:-**

- Certain language features supported by Interpreter rather than compiler.
- "Portability"

## ➢ COMPILERS

- A compiler is a program that reads a program written in one language – the source language – and translates it into an equivalent program in another language – the target language
- As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.



**Applications:**

- Design of Interfaces
- Design of language migration tools
- Design of Re – engineering Tools

**Two-Pass Assembly:**

The simplest form of assembler makes two passes over the input, where a pass consists of reading an input file once. In the first pass, all the identifiers that denote storage locations are found and stored in a symbol table.

In the second pass, the assembler scans the input again. This time, it translates each operation code into the sequence of bits representing that operation in machine language, and it translates each identifier representing a location into the address given for that identifier in the symbol table.

The output of the second pass is usually relocatable machine code, meaning that it can be loaded starting at any location L in memory; i.e., If L is added to all addresses in the code, then all references will be correct. Thus, the out- put of the assembler must distinguish those portions of instructions that refer to addresses that can be relocated.

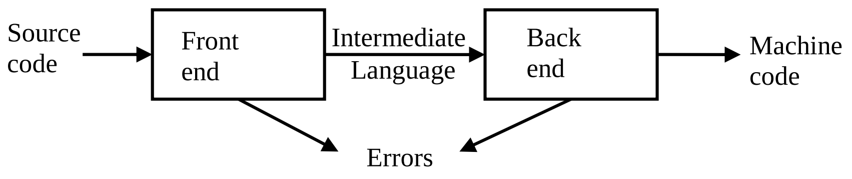
**Loaders and Link-Editors:**

- A program called loader performs the two functions of loading and link-editing.
- The process of loading consists of taking relocatable machine code, altering the relocatable addresses and placing the altered instructions and data in memory at the proper locations.
- The link-editor makes a single program from several files of relocatable machine code.

**The Phases of a Compiler**

A compiler includes the different phases, each of which transforms the source program from one representation to another. From figure 9.1.2 the compiler structure has following:

- Lexical analysis
- Syntax Analysis
- Semantic analysis
- Intermediate code generation
- Code optimization
- Target code generation



Front end is **0(n) or 0(n logn)**

Back end is **NP-complete**

The six phases divided into 2 Groups

1. Front End: Depends on stream of tokens and parse tree ( also called analysis phase)
2. Back End: Dependent on Target, Independent of source code ( also called synthesis phase)

**The Compilation Model**

There are two parts to compilation: **analysis** and **synthesis.**

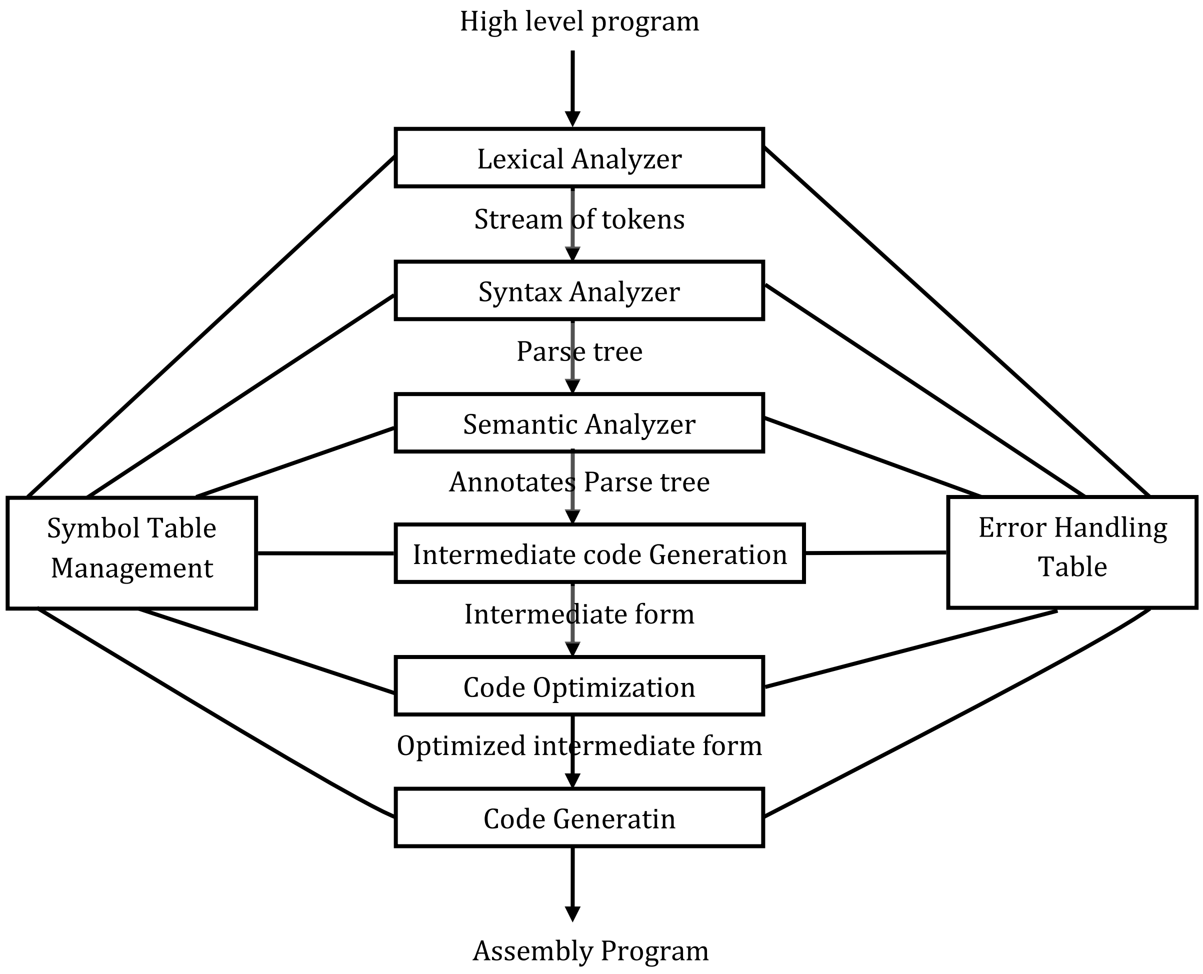


**Fig. 9.1.2. Compiler structure**

## Symbol-Table Management

- A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- Symbol table is a data Structure in a compiler used for managing information about variables & their attributes.

## Error Detection and Reporting

- The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler.
- The lexical phase can detect errors where the characters remaining in the input do not form any token of the language.
- Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase.

## ANALYSIS PHASE OF THE SOURCE PROGRAM

1. Linear or Lexical analysis, in which stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.
2. Hierarchical or Syntax analysis, in which characters or tokens are grouped hierarchically into nested collections with collective meaning.
3. Semantic analysis, in which certain checks are performed to ensure that the components of a program fit together meaningfully.

## Lexical Analysis:

- The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.
- Sometimes, lexical analyzers are divided into a cascade of two phases, the first called “scanning” and the second "lexical analysis."
- The scanner is responsible for doing simple tasks, while the lexical analyzer does the more complex operations.

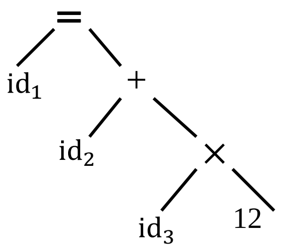
  Consider the expression
  $t = t_1 + t_2 \times 12$
  where $t$,$t_1$&$t_2$ are floats
  Lexical analyzer will generate $id_1 = id_2 + id_3 \times 12$

## Syntax Analysis:

- Hierarchical analysis is called **parsing** or **syntax analysis.**
- It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.
- The hierarchical structure of a program is usually expressed by recursive rules. For example, we might have the following rules as part of the definition of expressions:
  1. Any identifier is an expression.
  2. Any number is an expression.

3. If expression$_1$ and expression$_2$ are expression, then so are
   expression$_1$ + expression $_2$
   expression$_1$ * expression $_2$
   (expression$_1$)

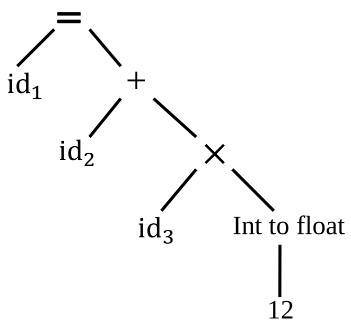
Ex. Parser will generate



**Semantic Analysis:**

- The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.
- It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.
- An important component of semantic analysis is type checking.

  Ex. Now as $t_1$ +1 &$t_2$ are float. 12 is also converted to float



**Intermediate Code Generation (or) ICG:**

- After syntax and semantic analysis compiler generate an explicit intermediate representation of the source program.
- This intermediate representation should have two important properties; **easy to produce**, and **easy to translate** into the target program.

  Ex. Intermediate code will be
  $\text{temp}^1 = \text{id}_3 \times 12.0$
  $$\text{temp}^2 = \text{id}_2 + \text{temp}^1$$
  $$\text{id}_1 = \text{temp}^2$$

## Code Optimization:

The code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result. Some optimizations are trivial.

## Advantages of Code Optimization:-

- Improves Efficiency
- Occupies less memory
- Executes fast

Ex. Optimized code will be

$\text{temp}^1 = \text{id}_3 \times 12.0$
$\text{id}_1 = \text{id}_2 + \text{temp}^1$

## Code Generation:

The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code. Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task. A crucial aspect is the assignment of variables to registers.

Machine code will look like

MUL $R_1$, 12.0
ADD $R_0$, $R_1$
MOV $\text{id}_1$, $R_0$
Where $R_1$ contains $\text{id}_3$&$R_0$ contains $\text{id}_2$.

### ➢ Lexical Analysis

- A **token** is a string of characters, categorized according to the rules as a symbol (e.g. IDENTIFIERS, KEYWORDS, OPERATORS, CONSTANTS, LITERAL STRINGS and PUNCTUATION SYMBOLS such as parenthesis, commas, and semicolons.).

- The process of forming tokens from an input stream of characters is called **tokenization** and the lexer categorizes them according to a symbol type.

- A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
- For example, the following statement

  **Int** number **;**

  The substring 'number' is a lexeme for the token "identifier" or "ID", '**int**' is a lexeme for the token "keyword", and '**;**' is a lexeme for the token";"