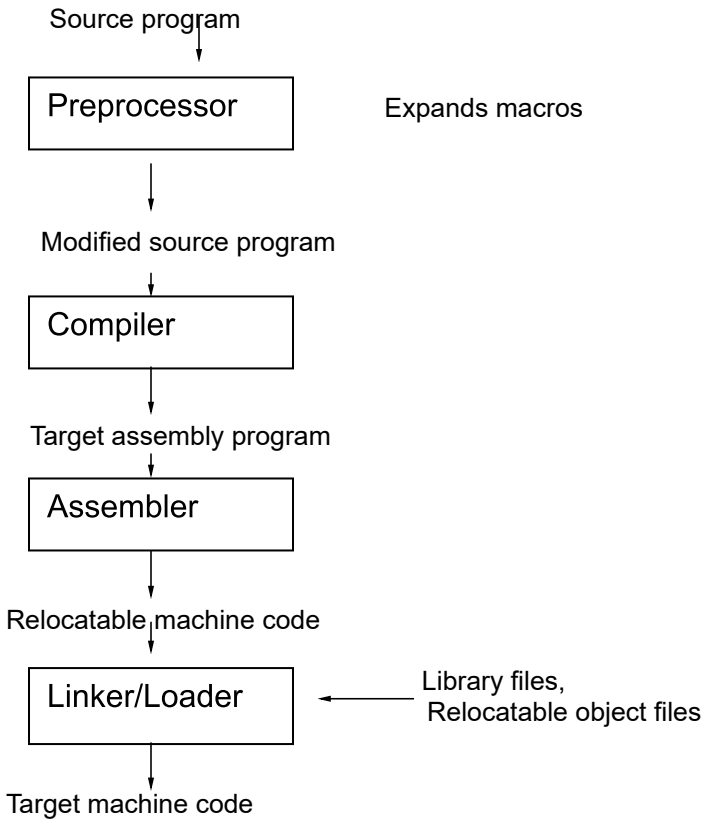
1. Introduction

Compiler is a translator program that translates a program written in (HLL) the source program and translates it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer. Executing a program written in HLL programming language is basically of two parts. The source program must first be compiled translated into an object program. Then the results object program is loaded into a memory executed.

Who developed the first compiler for a computer programming language?
The first compiler was written by Grace Hopper, in 1952, for the A-0 System language. The term compiler was coined by Hopper. The A-0 functioned more as a loader or linker than the modern notion of a compiler. The FORTRAN team led by John Backus at IBM is generally credited as having introduced the first complete compiler in 1957.

Language Processing System



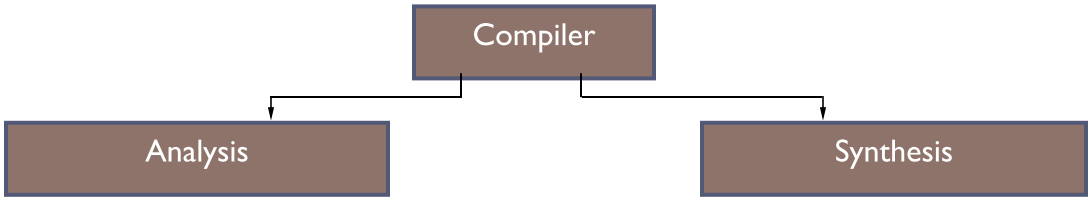
Pre-processor, Assembler, Loader and linker are also known as cousins of compiler.

Compilers and Interpreters

Compilers generate machine code, whereas interpreters interpret intermediate code Interpreters are easier to write and can provide better error messages (symbol table is still available) Interpreters are at least 5 times slower than machine code generated by compilers Interpreters also require much more memory than machine code generated by compilers Examples: Perl, Python, Unix Shell, Java, BASIC, LISP.

The Structure of a compiler

A compiler is a huge program that can consists of 10,000 to 1,000,000 lines of code. Since compiler is a huge program it is difficult to understand the entire compilation process. So the process is divided into number of modules called as PHASES. There are two major phases of compiler.



Analysis of the source program
Synthesis of a machine-language program

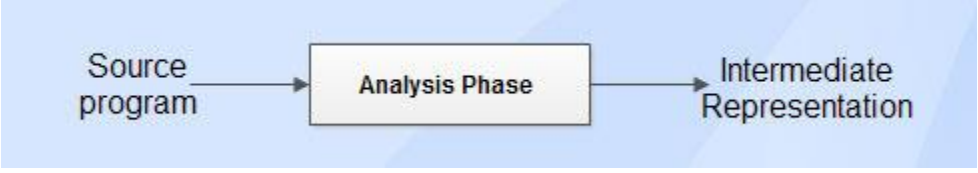
The structure of compiler consists of two parts:

Analysis part

• Analysis part breaks the source program into constituent pieces and imposes a grammatical structure on them which further uses this structure to create an intermediate representation of the source program.

• It is also termed as front end of compiler.

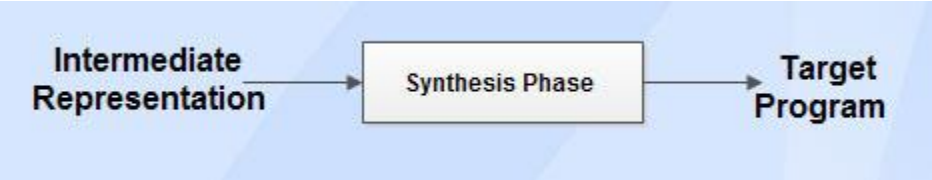
• Information about the source program is collected and stored in a data structure called symbol table.



Synthesis part

• Synthesis part takes the intermediate representation as input and transforms it to the target program.

• It is also termed as back end of compiler.



The design of compiler can be decomposed into several phases, each of which converts one form of source program into another.

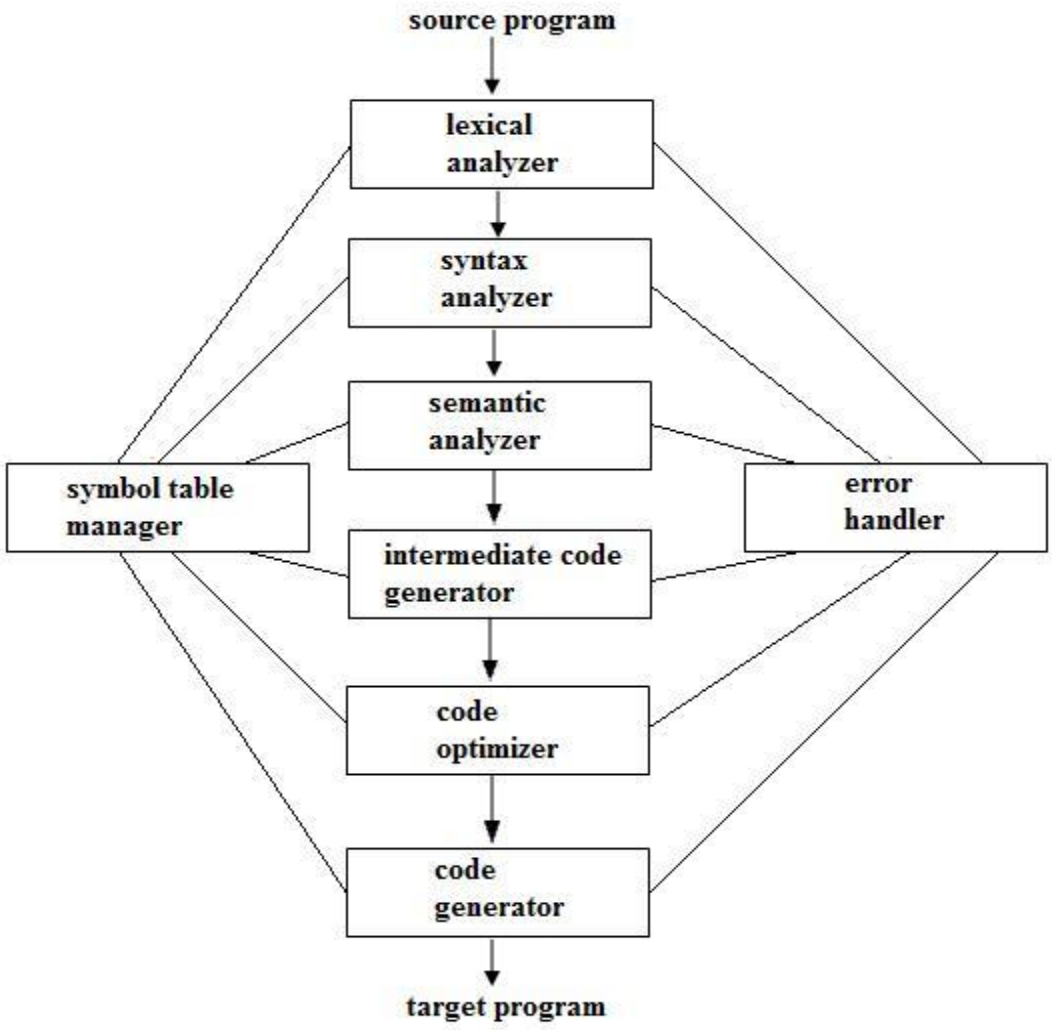


Fig 1.1 Structure of Compiler

The phases of compiler are as follows:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generation
5. Code optimization
6. Code generation

All of the aforementioned phases involve the following tasks:

• Symbol table management.

• Error handling.

Lexical Analysis

• Lexical analysis is the first phase of compiler which is also termed as scanning.

• Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called lexemes which produces token as output.

• **Token:** Token is a sequence of characters that represent lexical unit, which matches with the pattern, such as keywords, operators, identifiers etc.

• **Lexeme:** Lexeme is instance of a token i.e., group of characters forming a token. ,

• **Pattern:** Pattern describes the rule that the lexemes of a token takes. It is the structure that must be matched by strings.

• Once a token is generated the corresponding entry is made in the symbol table.

Input: stream of characters

Output: Token

Token Template: <token-name, attribute-value>

(eg.) c=a+b*5;

Lexemes and tokens

Lexemes	**Tokens**
c	identifier
=	assignment symbol

a	identifier
+	+ (addition symbol)
b	identifier
*	* (multiplication symbol)
5	5 (number)

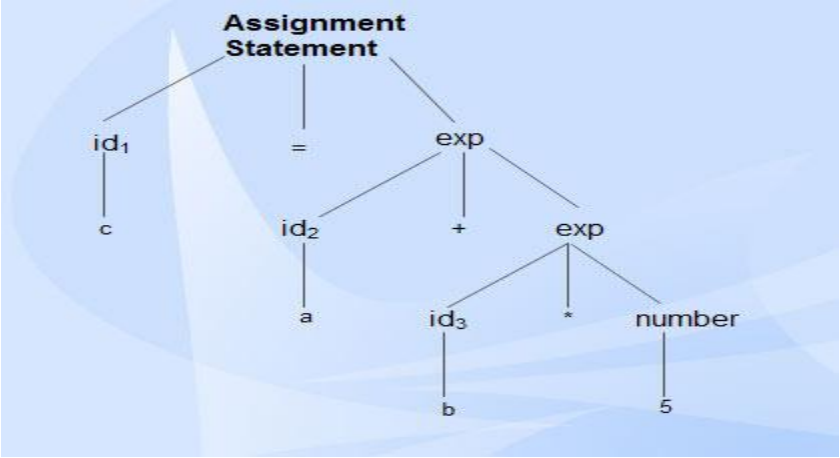
Hence, <id, 1><=>< id, 2>< +><id, 3 >< * >< 5>

Syntax Analysis

• Syntax analysis is the second phase of compiler which is also called as parsing.

• Parser converts the tokens produced by lexical analyzer into a tree like representation called parse tree.

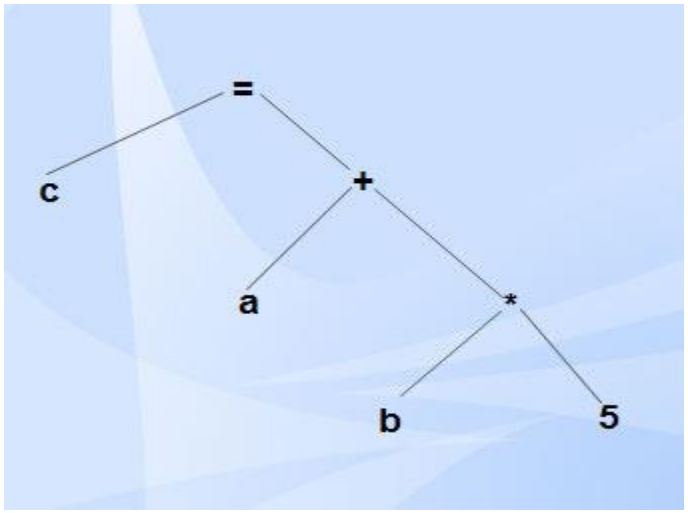
• A parse tree describes the syntactic structure of the input.



• Syntax tree is a compressed representation of the parse tree in which the operators appear as interior nodes and the operands of the operator are the children of the node for that operator.

Input: Tokens

Output: Syntax tree



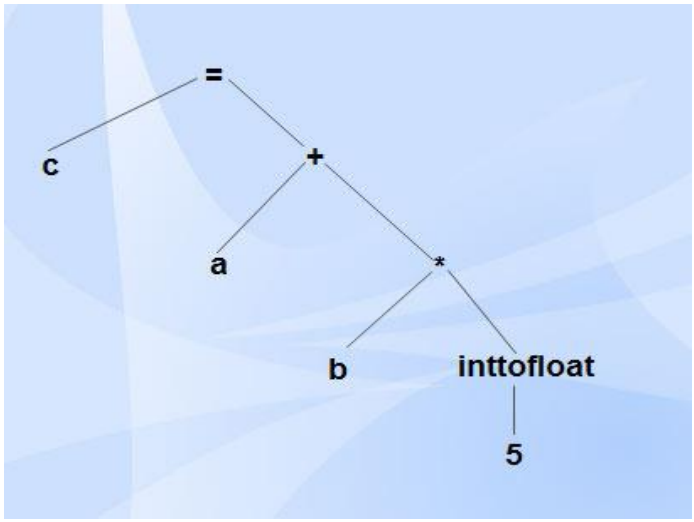
Semantic Analysis

• Semantic analysis is the third phase of compiler.

• It checks for the semantic consistency.

• Type information is gathered and stored in symbol table or in syntax tree.

• Performs type checking.



Intermediate Code Generation

• Intermediate code generation produces intermediate representations for the source program which are of the following forms:

o Postfix notation

o Three address code

o Syntax tree

Most commonly used form is the three address code.

t_1 = **inttofloat (5)**

t_2 = id_3* **tl**

t_3 = id_2 + t_2

id_1 = t_3

Properties of intermediate code

• It should be easy to produce.

• It should be easy to translate into target program.

Code Optimization

• Code optimization phase gets the intermediate code as input and produces optimized intermediate code as output.

• It results in faster running machine code.

• It can be done by reducing the number of lines of code for a program.

• This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.

• During the code optimization, the result of the program is not affected.

• To improve the code generation, the optimization involves

  o Deduction and removal of dead code (unreachable code).

  o Calculation of constants in expressions and terms.

  o Collapsing of repeated expression into temporary string.

  o Loop unrolling.

  o Moving code outside the loop.

  o Removal of unwanted temporary variables.

t_1 = id_3* **5.0**

id_1 = id_2 + t_1

Code Generation

• Code generation is the final phase of a compiler.

• It gets input from code optimization phase and produces the target code or object code as result.

• Intermediate instructions are translated into a sequence of machine instructions that perform the same task.

• The code generation involves

  o Allocation of register and memory.

  o Generation of correct references.

o Generation of correct data types.

o Generation of missing code.

LDF R_2, id_3

MULF R_2, # 5.0

LDF R_1, id_2

ADDF R_1, R_2

STF id_1, R_1

Symbol Table Management

• 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 finding 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.

Example

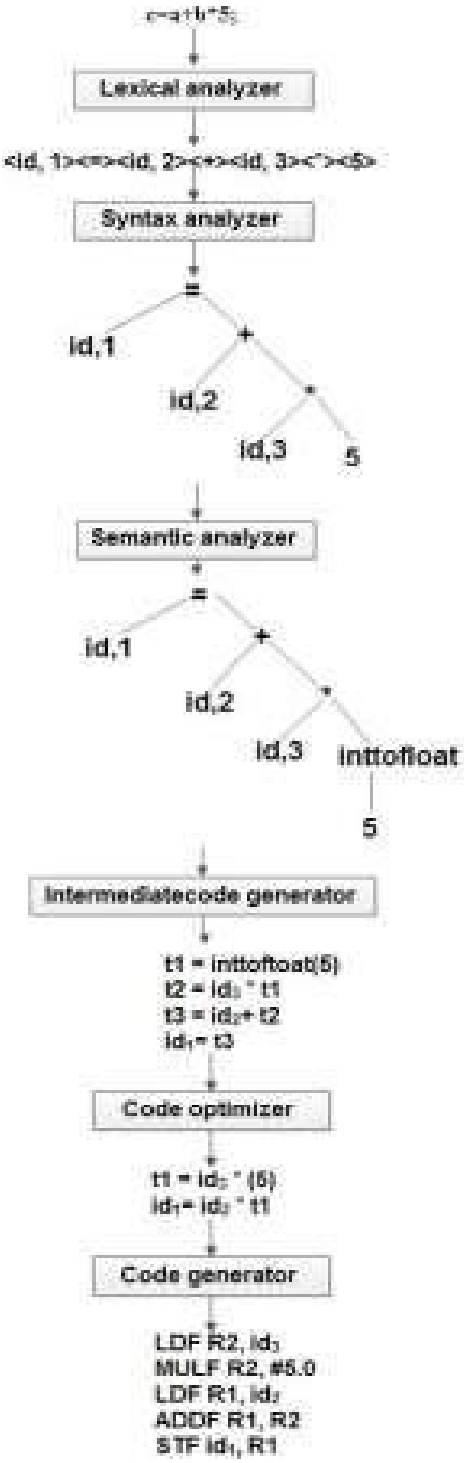
int a, b; float c; char z;

Symbol name	**Type**	**Address**
a	Int	1000
b	Int	1002
c	Float	1004
z	char	1008

Example

```
extern double test (double x);
  double sample (int count)
{
  double sum= 0.0;
    for (int i = 1; i < = count; i++)
      sum+= test((double) i);
      return sum;
  }
```

Symbol name	Type	Scope
test	function, double	extern
x	double	function parameter
sample	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement



Error Handling

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

• In lexical analysis, errors occur in separation of tokens.

• In syntax analysis, errors occur during construction of syntax tree.

• In semantic analysis, errors may occur at the following cases:

(i) When the compiler detects constructs that have right syntactic structure but no meaning

(ii) During type conversion.

• In code optimization, errors occur when the result is affected by the optimization. In code generation, it shows error when code is missing etc.

Figure illustrates the translation of source code through each phase, considering the statement

c =a+ b * 5.