Compiler

Translation

High Level Language <Compiler> Low Level Language

Compiler

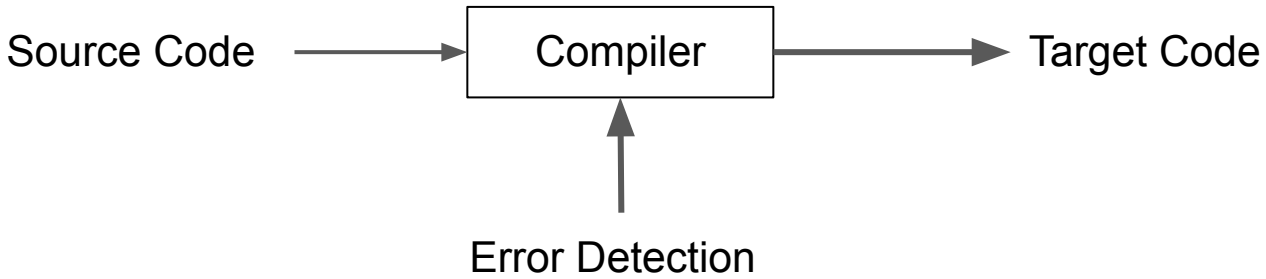
A software used to convert High Level Language into Low Level Language and from Low Level Language back to High Level Language.

Translation of source code into target code.

In general the language understood by Humans / Users are considered as High Level Language and language understood by Computer /Hardware is considered as Low Level Language.

Compiler

A special program that converts source code into target code and also finds the errors (Syntax Error) in the source code.

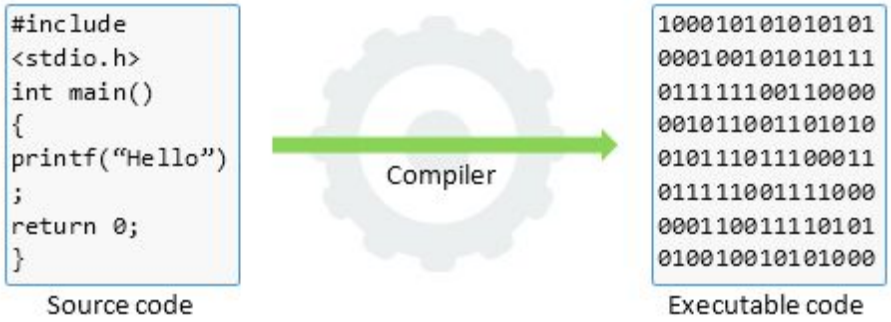


Compiler

C compilers:

1. Turbo C++
2. MinGW (GCC)
3. Clang

And more...



Compiler

Native Compiler
The compiler which generates executable for the same operating system on which it is compiling means for native system is known as Native Compiler.

Cross Compiler
The compiler which generates executable for a different operating system instead of the operating system it is running.

More reading:
https://www.geeksforgeeks.org/difference-between-native-compiler-and-cross-compiler/

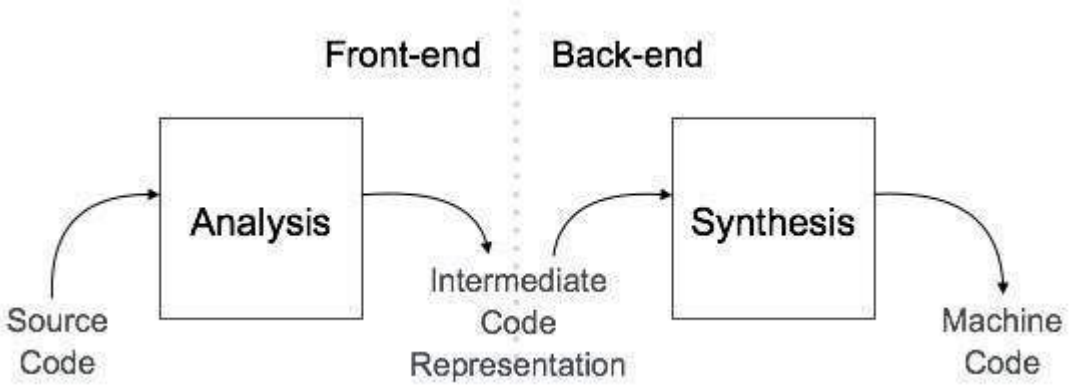
Why Study Compilers?

- General background information for good software engineer
- Increases understanding of language semantics
- Seeing the machine code generated for language constructs helps understand performance issues for languages
- Teaches good language design
- New devices may need device-specific languages
- New business fields may need domain-specific languages

Compiler

A compiler can broadly be divided into two phases based on the way they compile.

1. Analysis Phase (Front End)
2. Synthesis Phase (Back End)



More reading:
https://www.tutorialspoint.com/compiler_design/compiler_design_architecture.htm

Compiler

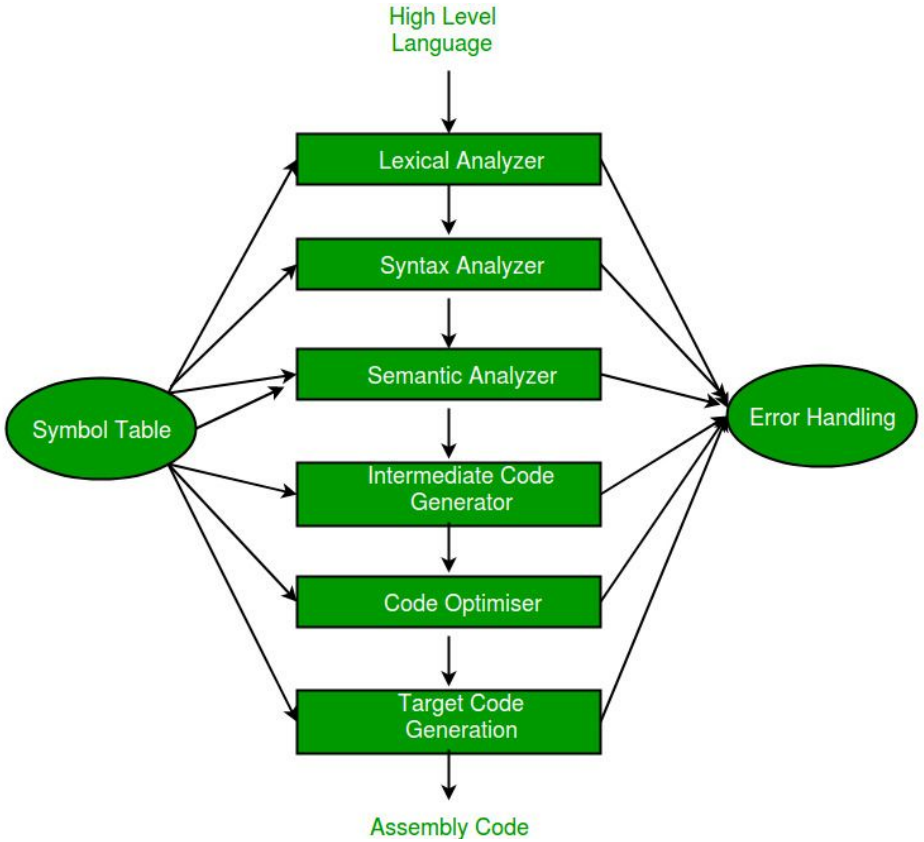
Compilation	Analysis Phase (Front End)	The analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors.The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.
	Synthesis Phase (Back End)	The synthesis phase generates the target program with the help of intermediate source code representation and symbol table.

Compilation

Compilation the process of converting source code into target code has several phases in it.

In general

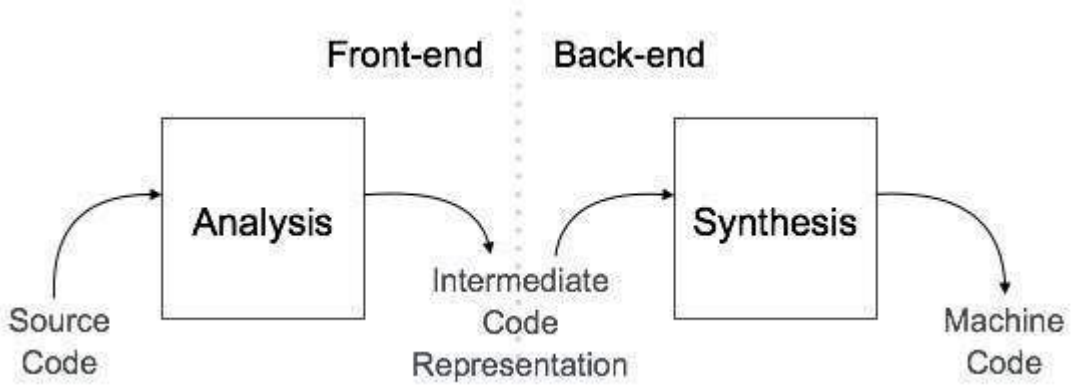
1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation



Compiler

A compiler can broadly be divided into two phases based on the way they compile.

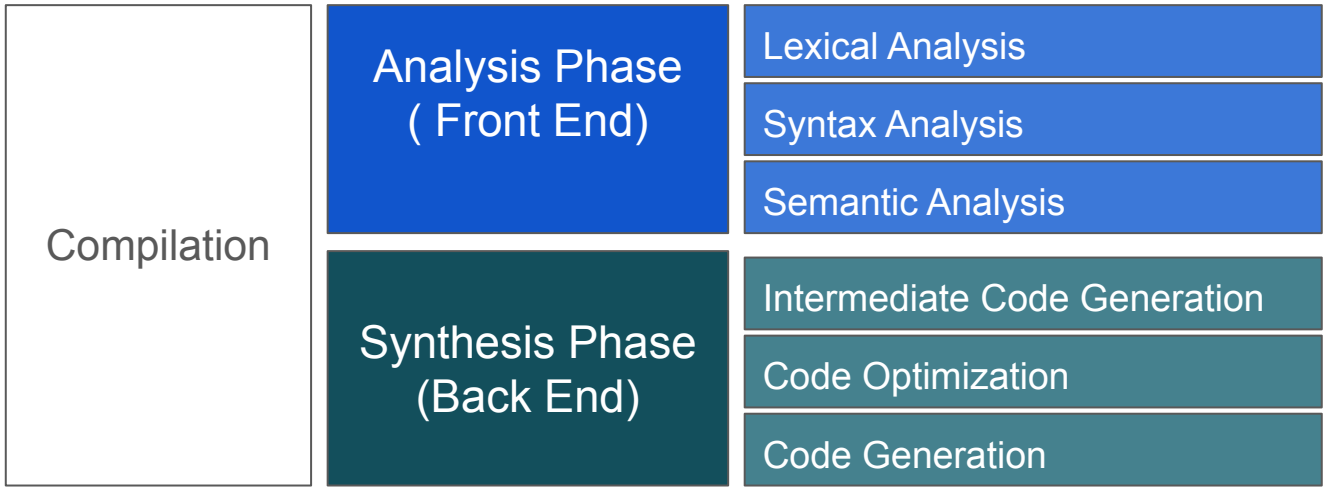
1. Analysis Phase (Front End) (Lexical Analysis, Syntax Analysis, Semantic Analysis)
2. Synthesis Phase (Back End) (Intermediate Code Generation, Code Optimization, Code Generation)



(Lexical Analysis, Syntax Analysis, Semantic Analysis)

(Intermediate Code Generation, Code Optimization, Code Generation)

Compiler


Compilation
Analysis Phase
(Front End)
Lexical Analysis
Syntax Analysis
Semantic Analysis
Synthesis Phase
(Back End)
Intermediate Code Generation
Code Optimization
Code Generation

Compilation

Symbol Table

an important **data structure** created and maintained by the compiler in order to keep track of semantics of variable i.e. it stores information about scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

When identifiers are found, they will be entered into a symbol table, which will hold all relevant information about identifiers and other symbols, variables, constants, procedures statements.

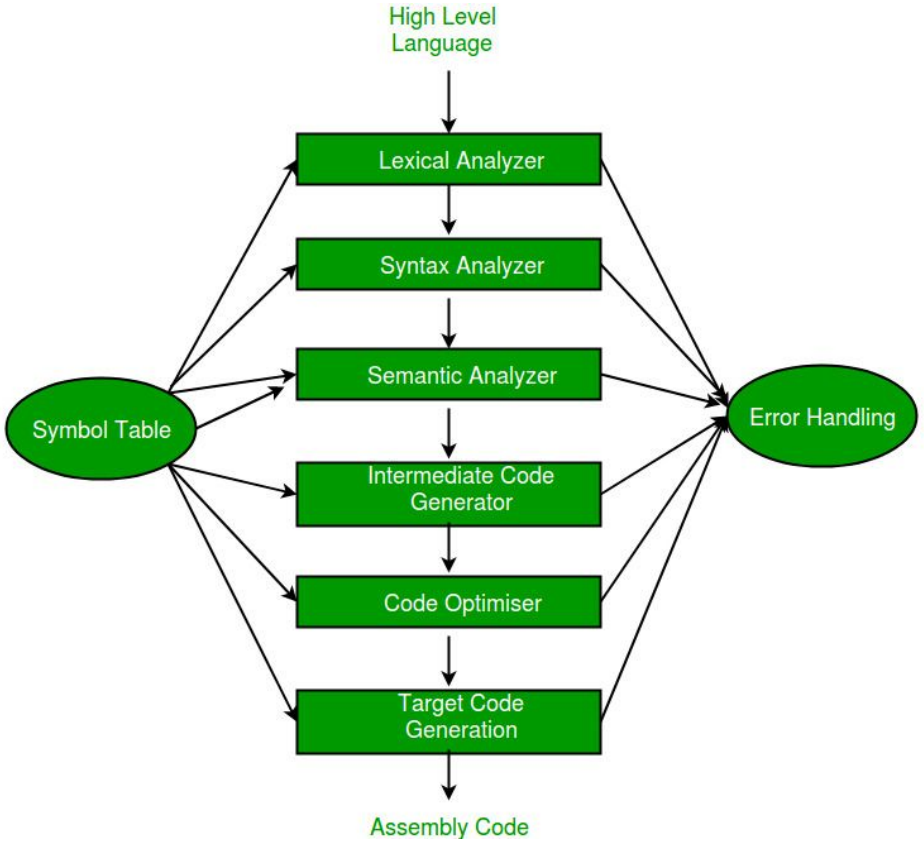
Symbol table is created during Lexical Analysis and Syntax Analysis which is then used during Semantic Analysis and Code Generation.

Compilation

Symbol Table

A symbol table can be implemented in one of the following ways:

1. Linear (sorted or unsorted) list
2. Binary Search Tree
3. Hash table



Compilation

Symbol Table at different phase

Lexical Analysis: Creates new table entries in the table, example like entries about token.

Syntax Analysis: Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.

Semantic Analysis: Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.

Intermediate Code generation: Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.

Code Optimization: Uses information present in symbol table for machine dependent optimization.

Target Code generation: Generates code by using address information of identifier present in the table.

Compilation

A compiler can have many **phases** and **passes**.

Pass : A pass refers to the traversal of a compiler through the entire program.

Phase : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

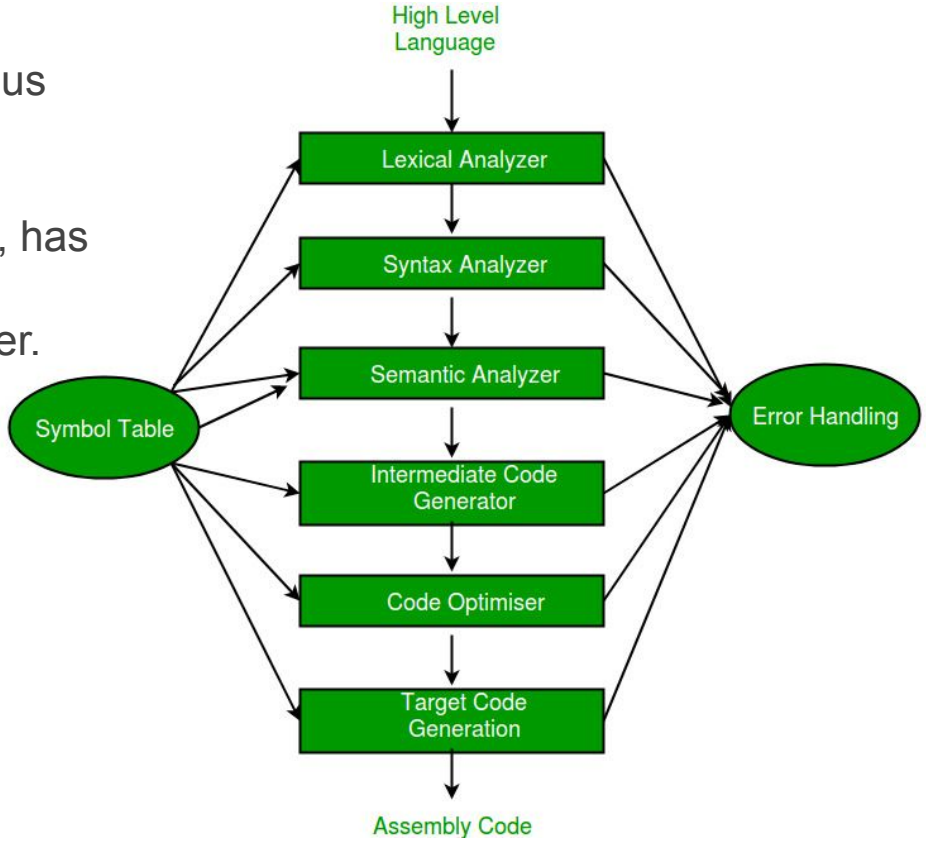
Phases of Compiler

The compilation process is a sequence of various phases.

Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.

Phases of Compiler

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation



Phases of Compiler

Phase-1

Lexical Analysis

The first phase of scanner works as a text scanner.
This phase scans the source code as **a stream of characters** and converts it into meaningful **lexemes**.

Lexical analyzer represents these **lexemes** in the form of tokens as:

<**token-name**, attribute-value>

A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimiters and so on).

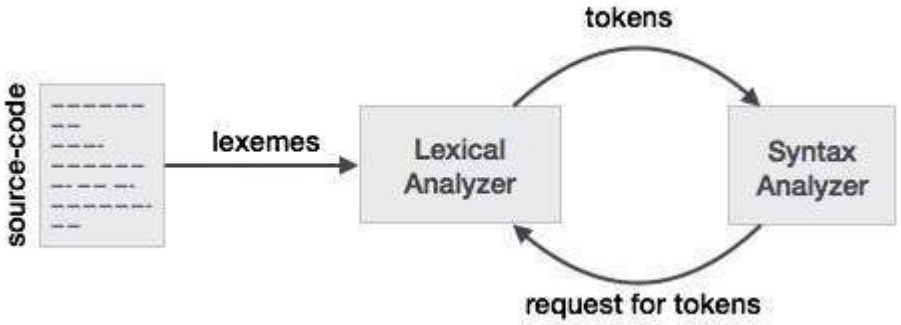
Phases of Compiler

Phase-1

Lexical Analysis

The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Phases of Compiler

Phase-1

Lexical Analysis

Ex: for expression **newval := oldval + 12**

=> tokens: **newval** identifier
:= assignment operator
oldval identifier
+ add operator
12 a number

- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

Phases of Compiler

Phase-2

Syntax Analysis

The next phase is called the **syntax analysis or parsing**.
It takes the token produced by lexical analysis as input and generates a parse tree (or **syntax tree**).
In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is **syntactically correct**.

Phases of Compiler

Phase-2

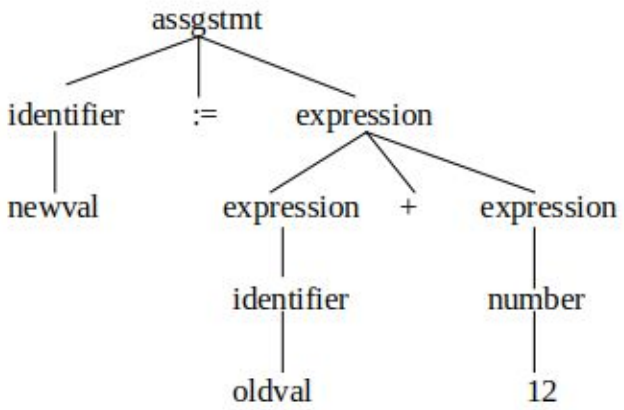
Syntax Analysis

for expression **newval := oldval + 12**

=> tokens:

newval identifier
:= assignment operator
oldval identifier
+ add operator
12 a number

Parse Tree



In a parse tree, all terminals are at leaves.

All inner nodes are non-terminals in a context free grammar.

Parsing Techniques

Phase-2

Depending on how the parse tree is created, there are different parsing techniques.
These parsing techniques are categorized into two groups:

1. **Top-Down Parsing,**
2. **Bottom-Up Parsing**

Top-Down Parsing:
Construction of the parse tree starts at the root, and proceeds towards the leaves.
Efficient top-down parsers can be easily constructed by hand.
Recursive Predictive Parsing, Non-Recursive Predictive Parsing (LL Parsing).

Bottom-Up Parsing:
Construction of the parse tree starts at the leaves, and proceeds towards the root.
Normally efficient bottom-up parsers are created with the help of some software tools.
Bottom-up parsing is also known as **shift-reduce parsing**.
Operator-Precedence Parsing – simple, restrictive, easy to implement
LR Parsing – much general form of shift-reduce parsing, LR, SLR, LALR

Phases of Compiler

Phase-3

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language.

For example: assignment of values is between compatible data types, and adding string to an integer.
Also, the semantic analyzer **keeps track of identifiers,** their types and expressions; whether identifiers are declared before use or not etc.

The semantic analyzer produces an **annotated syntax tree** as an output.

Phases of Compiler

Phase-3

A **semantic analyzer** checks the source program for semantic errors and collects the type information for the code generation.
Type-checking is an important part of semantic analyzer.
Normally semantic information cannot be represented by a context-free language used in syntax analyzers.
Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)

- the result is a syntax-directed translation,
- Attribute grammars

Ex:

```
newval := oldval + 12
```

The type of the ***identifier*** newval must match with type of the expression (oldval+12)

Phases of Compiler

Phase-4

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine.
It represents a program for some abstract machine. It is in between the high-level language and the machine language.

This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Phases of Compiler

Phase-4

Intermediate Code Generation

A compiler may produce an explicit intermediate codes representing the source program.
These intermediate codes are generally machine (architecture independent). But the level of intermediate codes is close to the level of machine codes.
Ex:

newval := oldval * fact + 1

id1 := id2 * id3 + 1

```
MULT     id2,id3,temp1
ADD    temp1,#1,temp2
MOV    temp2,,id1
```

Intermediates Codes (Quadruples)

Phases of Compiler

Phase-5

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that **removes unnecessary code** lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Phases of Compiler

Phase-5

Code Optimization

The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.

Intermediate Code Generated

MULT id2,id3,temp1
ADD temp1,#1,temp2
MOV temp2,,id1

Optimized Code
MULT id2,id3,temp1 ADD temp1,#1,id1

Phases of Compiler

Phase-6

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language.

The code generator translates the intermediate code into a sequence of (generally) relocatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Phases of Compiler

Phase-6

Code Generation

Produces the target language in a specific architecture.
The target program is normally is a relocatable object file containing the machine codes.

Optimized Code

```
MULT    id2,id3,temp1
ADD     temp1,#1,id1
```

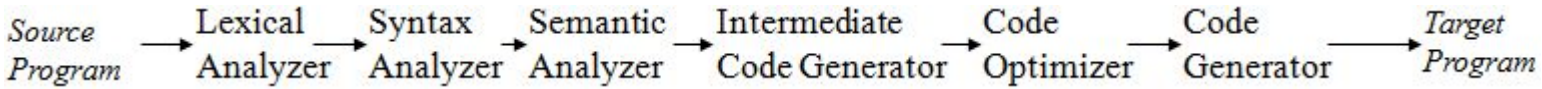
Code Generated

```
MOVE    id2,R1
MULT    id3,R1
ADD     #1,R1
MOVE    R1,id1
```

(assume that we have an architecture with instructions who's at least one of its operands is a machine register)

Phases of Compiler

Phase



C program compilation

C program Compilation steps

1. Preprocessing [Happens before compilation]
2. Compiling
3. Assembly
4. Linking

C program Compilation steps

Preprocessing [Happens before compilation] : for all Preprocessor Directives

Expanding Included Header files, it includes the header file into source code.

Expands macros, [Predefined Macros and User defined Macros]

Removal of non-codes means comments are removed from source code.

In gcc compiler you can check it by:
\> g++ -E example.c

C program Compilation steps

Compile

Converting the preprocessed file into an Assembly file.(dot s extension)

Normally not visible but you can view that file by:

\>g++ -S example.c

Assembly Language

Middle Level Programing Language considered to be very close to machine language.

It is hardware or CPU dependent language And also depends on the operating system.

Assembly codes are converted into Binary or Machine language by Assemblers.

More Reading:

https://en.wikipedia.org/wiki/Assembly_language

https://www.tutorialspoint.com/assembly_programming/assembly_tutorial.pdf

https://cs.lmu.edu/~ray/notes/x86assembly/

C program Compilation steps

Assembly

Converting the Assembly code into output code or binary/ machine code. Contains Zeros and Ones.

You can check it by:

\>g++ -c example.c

Reading Binary File

To read Binary file you can use online portal :

https://hexed.it/

Only upload your binary (dot o) file.

C program Compilation steps

Linking

Converting object code into executable code.
This process is **Operating system dependent** as on
Windows it generates dot exe (a.exe , default in gcc)
Ubuntu it generates dot out (a.out)

\>g++ example.c

To save the output file with other name:

Windows\>g++ -o example.c example.exe
Ubuntu\>g++ -o example.c example.out

Executable File

It depends on the hardware and Operating system.

Binary files and some system files are packed or linked together to create an executable file.

C program Compilation steps

Preprocessing	Compile	Assembly	Linking
It happens before compilation. 1. Header file codes are included in source file. 2. Macros expansion. 3. Removal of comments.	The preprocessed code is converted into Assembly code. With dot s extension.	Assembly code is converted into Binary code. [Object Code] in ubuntu File with dot o extension. In windows File with dot obj extension.	The final process depends on operating system. The object code is combined with other required files to make executable for the current operating system.

C program Compilation steps

In turbo C++

For source code example.c
When it compiles successfully 3 different new files were created.

1. example.BACK (Backup File)
2. example.obj (Object Code)
3. example.exe (the executable file)

By default all the files are stored in different locations.
For example : example.c and example.BACK are in Bin folder.