




# INTRODUCTION TO COMPILERS

Md Ferdous Bin Hafiz  
Lecturer  
East Delta University



# Mark Distribution

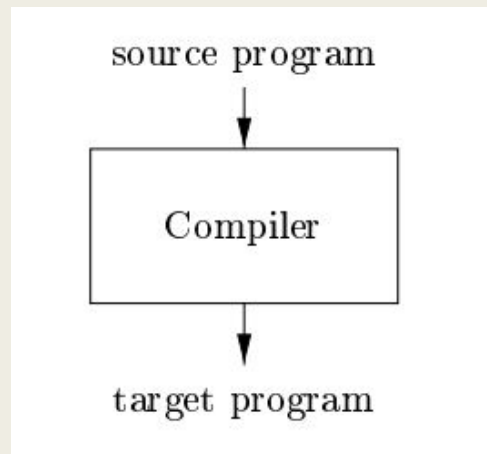
- Attendance: 10%
- Quiz/Assignments: 20%
- Mid Term Exam: 30%
- Final Exam: 40%

# Syllabus

- Textbook: “*Compilers: Principles, Techniques, and Tools*” by Aho, Sethi, and Ullman, 1st or 2nd edition
- Other material: Lex & Yacc, Doug Brown, John R. Levine, and Tony Mason
- Class lectures and Internet

# What is a compiler?

- 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**.
- Compiler works in 2 steps



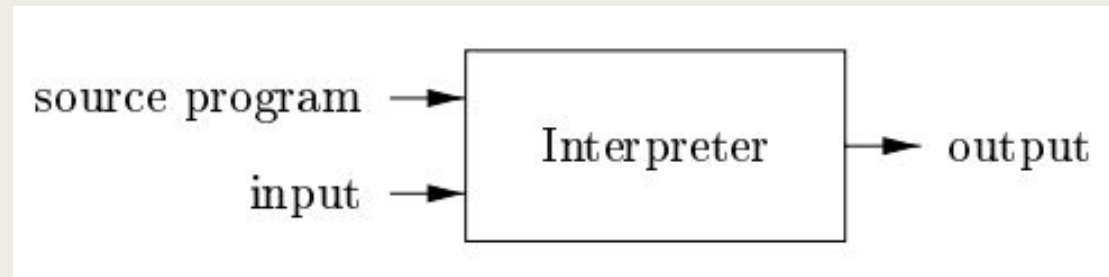
Translation Part [ eg C to Assembly/Obj Code]



Execution Part

# 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.



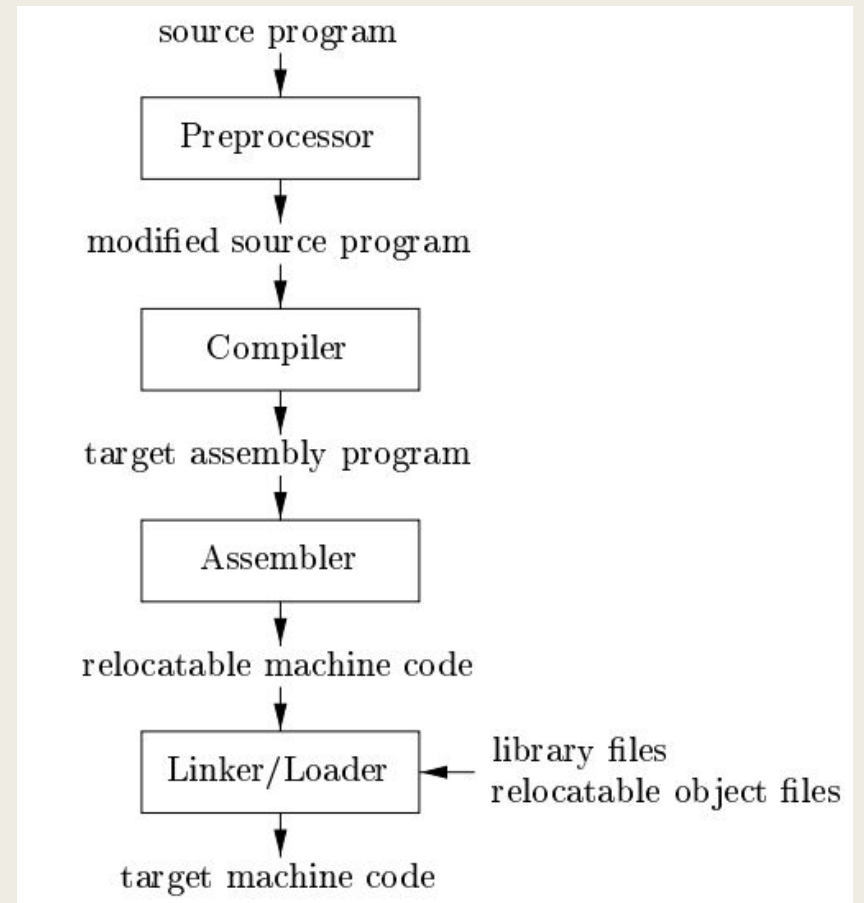
- The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

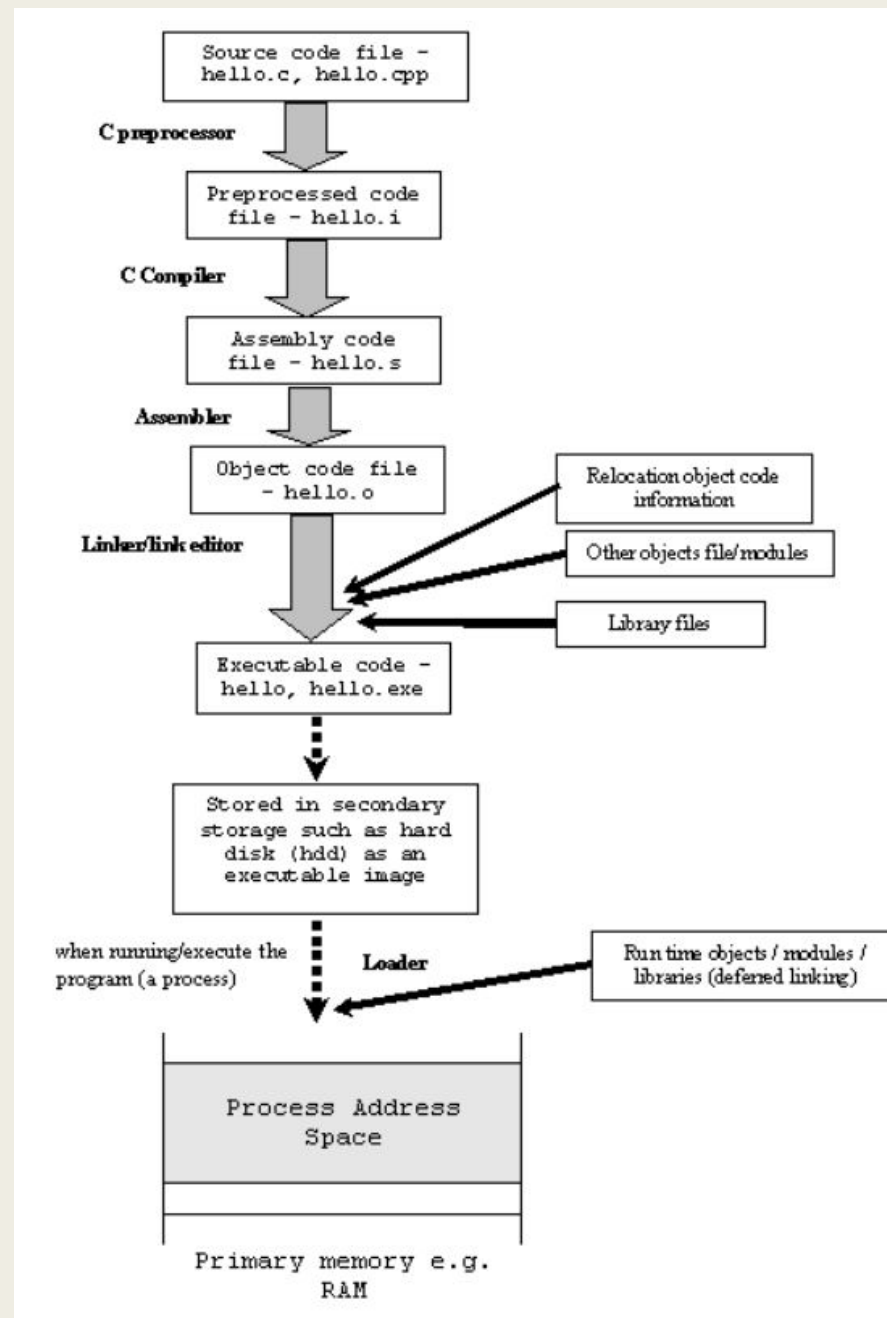
# Compiler v Interpreter

Compiler	Interpreter
Compiler scans the whole program and then translate into machine code	Interpreter scans one line at a time and translate into machine code
Intermediate object code is created	Intermediate object code is not created
It takes more memory space since object code is generated	It takes less memory
Compiler is faster	Interpreter is slower
Program doesn't need to be compiled everytime	Every time higher level program is converted into lower lever lever program
Errors are displayed after entire program is checked	Errors are displayed for every instruction interpreted

# Additional Components

- A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a preprocessor. The preprocessor may also expand shorthands, called macros, into source language statements.
- The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an assembler that produces relocatable machine code as its output.
- Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. The linker resolves external memory addresses, where the code in one file may refer to a location in another file. The loader then puts together all of the executable object files into memory for execution.







# Linking time

- Compiler+Assembler = Compile Time (time to translate HLL to binary code)
- Linker+Loader = Load Time

=====

- Why C Is Faster than Java?
- Answer: C has less execution Time
- Static Linker: Links before Execution (eg C)
- Dynamic Linker : Links during Execution (eg. Java)

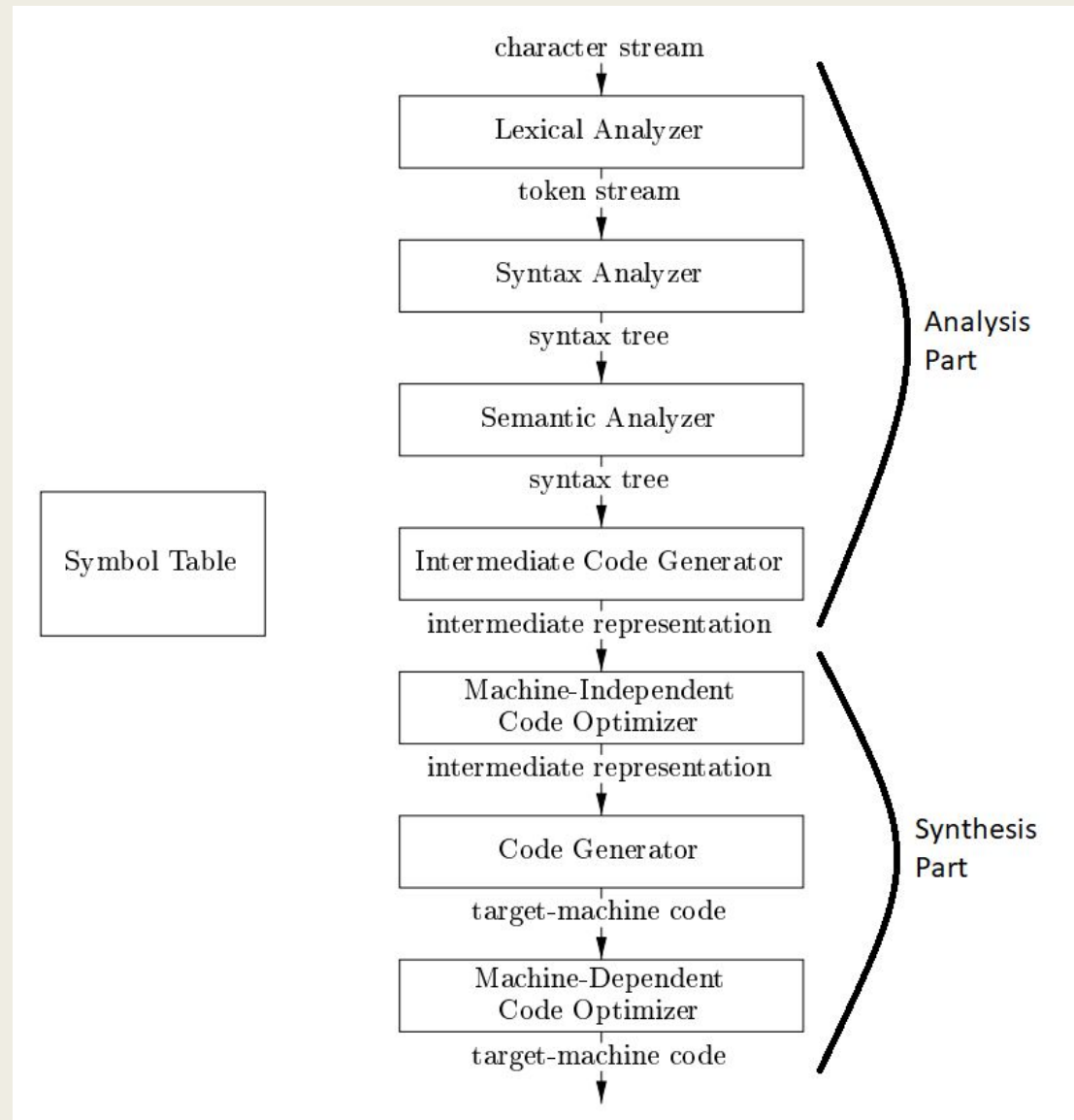
■ =====

- So static linker takes more time during translation as it does linking as well

# Structure of a Compiler

- It can be divided into two main parts: analysis and synthesis.
- If we examine the compilation process in more detail, we see that it operated as a sequence of phases, each of which transforms one representation of the source program to another
- The analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so that the user can take corrective action
- 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.
- If we go into further details, the compilation process can be defined as a sequence of phases.

# Structure of a Compiler



1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE

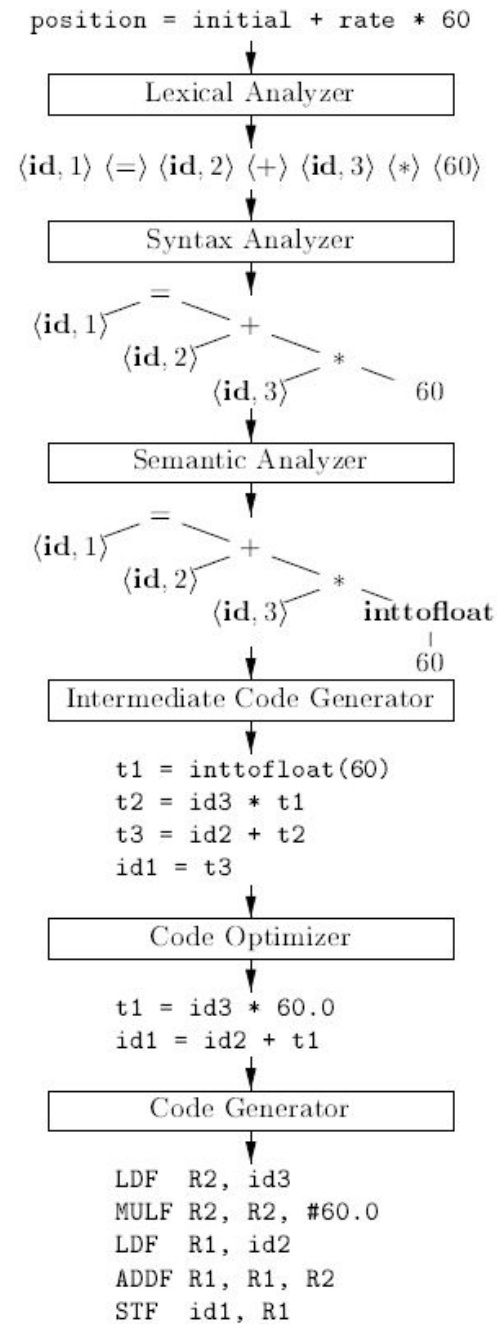


Figure 1.7: Translation of an assignment statement

# Lexical Analysis

- The first phase of a compiler is called **lexical analysis or scanning**. The lexical analyzer **reads the stream of characters** making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form

*⟨token-name, attribute-value⟩*

that it passes on to the subsequent phase, syntax analysis. In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation.

# Syntax Analysis

- The second phase of the compiler is syntax analysis or parsing. 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.
- The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program.

# 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.

# 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.
- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.



# 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.
- There is a great variation in the amount of code optimization different compilers perform. In those that do the most, the so-called “optimizing compilers,” a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.

# 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.

# Symbol Table

- An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.
- 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.