COMPILATION AND INTERPRETATION

Kwankamol Nongpong, Ph.D.
Department of Computer Science
Assumption University

What is this?

```
27bdffd0 afbf0014 0c1002a8 00000000 0c1002a8 afa2001c 8fa4001c
00401825 10820008 0064082a 10200003 00000000 10000002 00832023
00641823 1483fffa 0064082a 0c1002b2 00000000 8fbf0014 27bd0020
03e00008 00001025
```

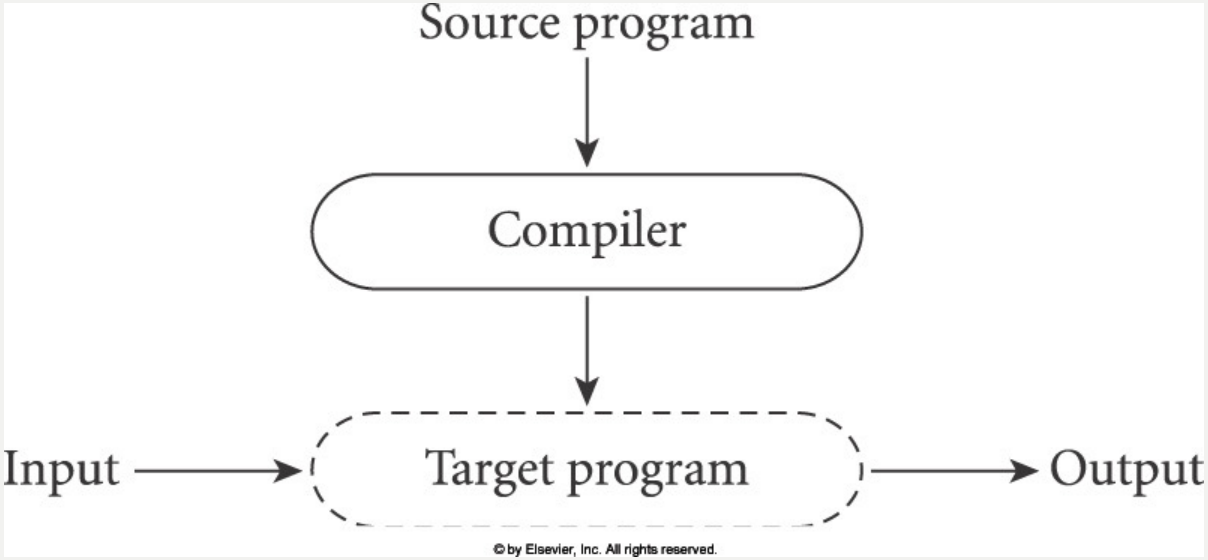
How about this?

```
    addiu sp,sp,-32
    sw     ra,20(sp)
    jal   getint
    nop
    jal   getint
    sw    v0,28(sp)
    lw    a0,28(sp)
    move  v1,v0
    beq   a0,v0,D
slt   at,v1,a0
```

```
A:  beq   at,zero,B
    nop
    b     C
    subu  a0,a0,v1
B:  subu  v1,v1,a0
C:  bne   a0,v1,A
    slt   at,v1,a0
D:  jal   putint
    nop
    lw    ra,20(sp)
    addiu sp,sp,32
    jr    ra
    move  v0,zero
```

Compilation (1)

- At the highest level of abstraction, the compilation and execution of a program in a high-level language look like:



Compilation (2)

- The compiler translates the high-level source program into a target program.
- Then, the user tells the operating system to run the target program.
- A target program is usually in machine language and is commonly known as object code.

Interpretation

- An alternative style of implementation for high-level language is known as interpretation.

- Unlike a compiler, an interpreter reads statements from a source program one at a time and executes them.

Benefits of Interpreters

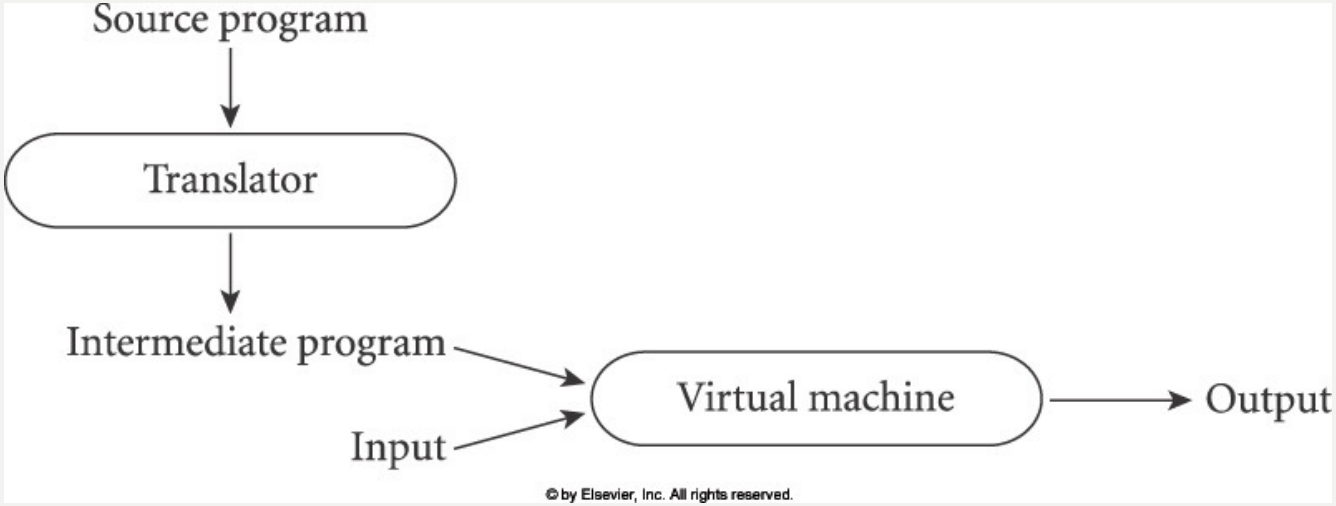
- Greater flexibility and diagnostics
- Allow program features (data types or sizes) to depend on the input.
- Lisp and Prolog programs can write new codes and execute them on-the-fly.
- Original Java implementation uses Java interpreter to execute Java byte code.

Benefits of Compilers

- Better performance.
- Many decisions are made only once, at the compile time, not at every run time.
- Errors can be detected (and fixed) early.

A Mixture of Both?

- While the conceptual difference between compilation and interpretation is clear, most implementations use both.



The Fuzzy Difference

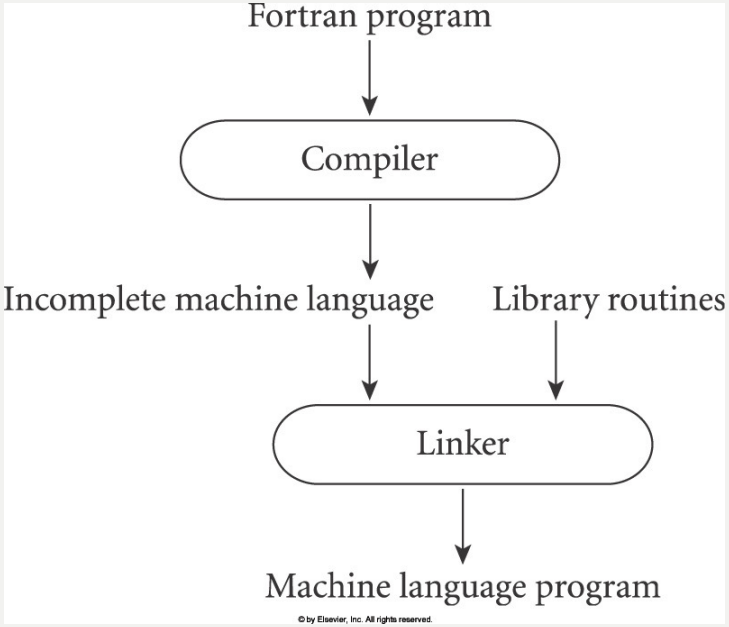
- A language is interpreted if the initial translation is simple.
- A language is compiled if the initial translation is complicated.
- It is possible for a complicated translator (compiler) to produce code that is executed by a complicated virtual machine (interpreter).

Preprocessor & Interpreter

- Most interpreted languages employ an initial translator (preprocessor) that produces an intermediate form that can be interpreted (and executed) more efficiently.
- Early implementations of Basic suggest that programmers remove comments in their program to increase its performance.

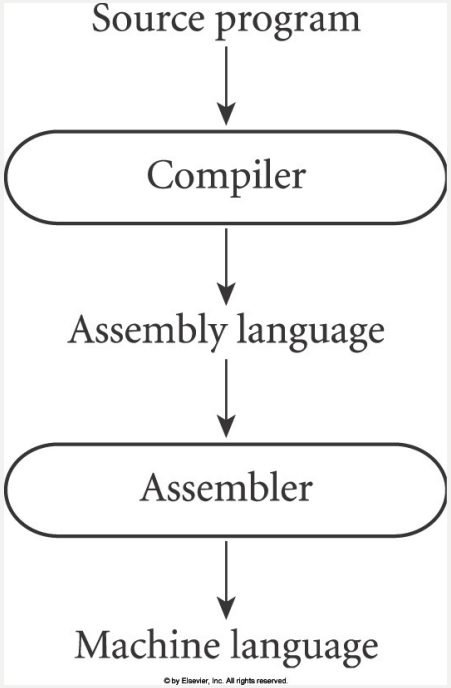
Compiler & Linker

- Fortran implementation comes close to pure compilation.
- A linker is used to connect an incomplete machine code with the library routines.



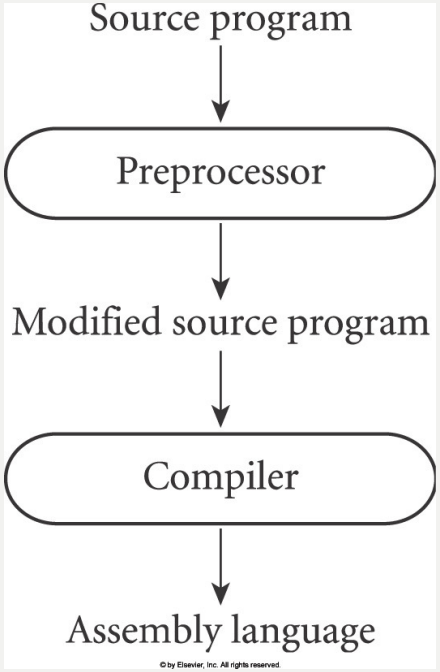
Compiler & Assembler

- Many compilers generate codes in assembly language rather than machine language.
- An assembler is used later to translate an assembly code to a machine code.



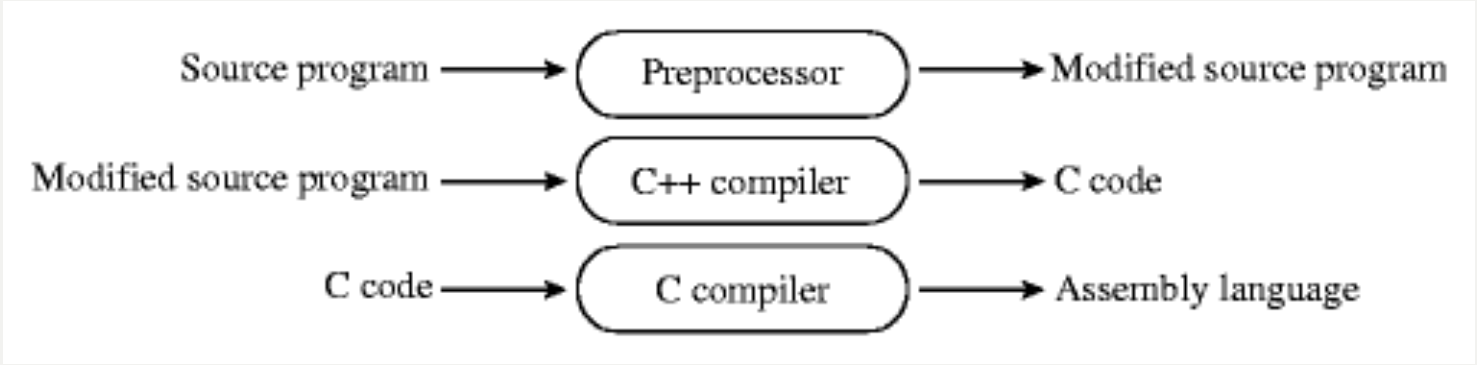
Preprocessor & Compiler

- Compilers for C begin with a preprocessor that removes comments and expands macros.
- It can also delete portions of the code to provide a conditional compilation facility.



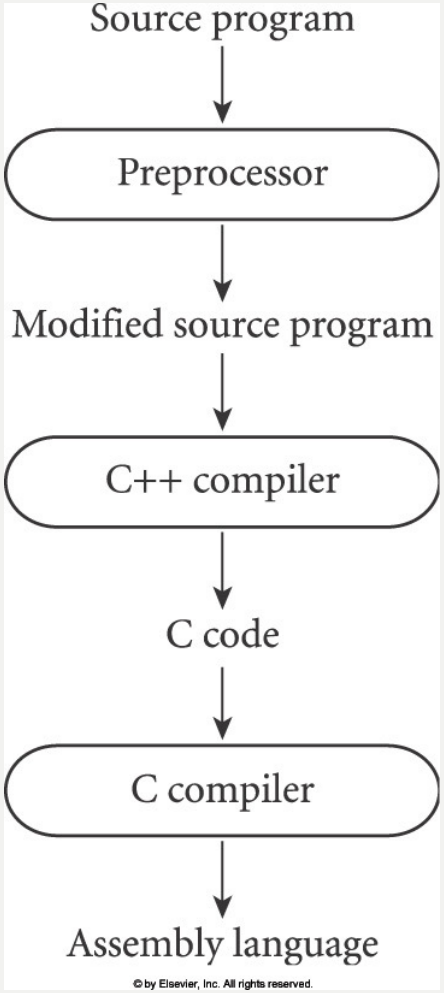
Compiler & Compiler

- The C++ compiler (AT&T) generates an intermediate program in C, instead of assembly language.



- Is C++ compiler actually a compiler?

Compiler & Compiler



Compiler and Interpreter

- Compilers for Lisp, Prolog, and SmallTalk permit a lot of late binding that are traditionally interpreted.
- These compilers work with an interpreter.
- The compiler does its best at compile time.
- The interpreter does the rest that the compiler cannot do at run time.

Just-In-Time (JIT) Compiler

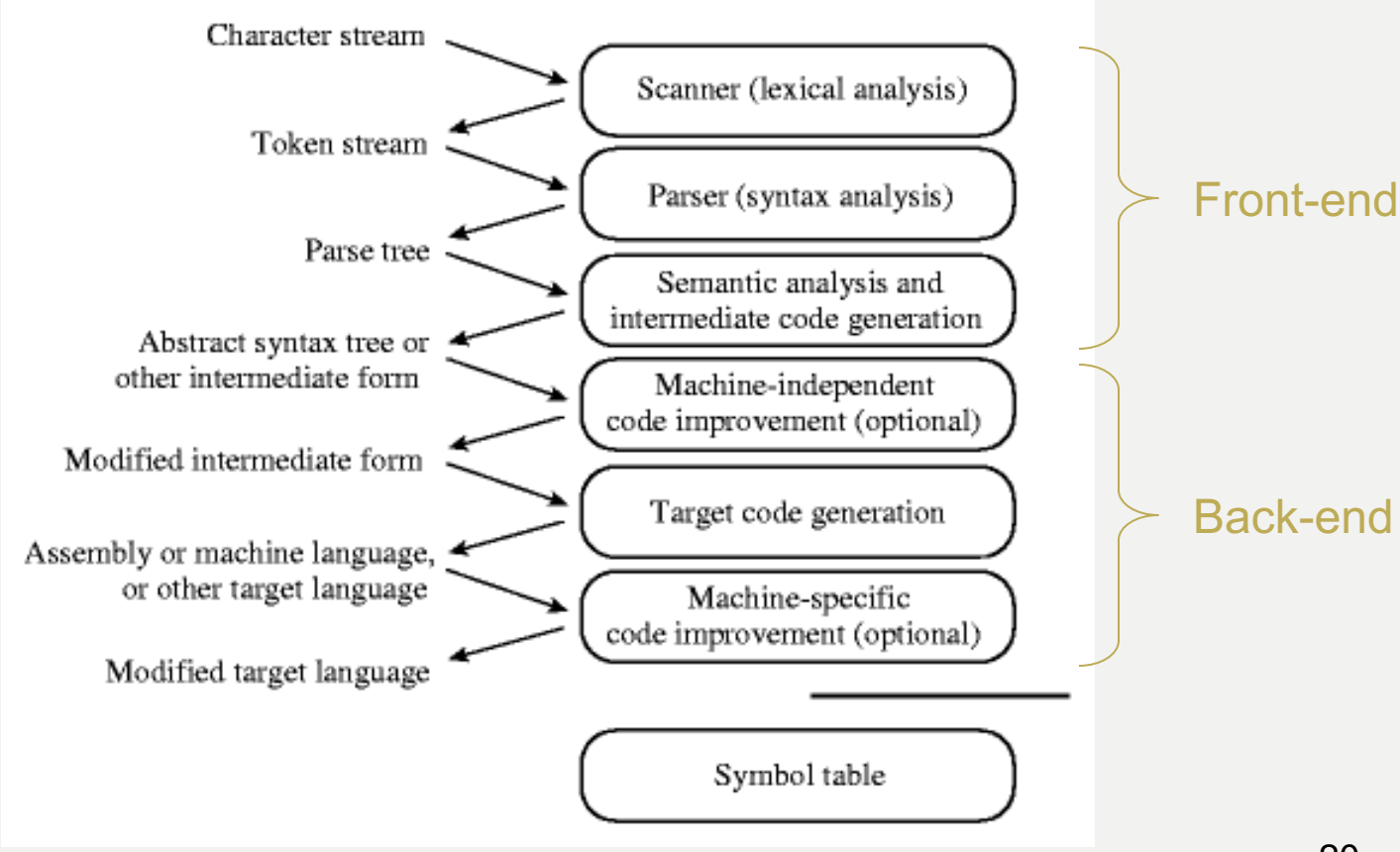
- Java language system defines a machine-independent intermediate form, known as Java byte code.
- Original implementation uses a byte-code interpreter to execute the code on JVM.
- Later implementation uses a just-in-time compiler to translate byte code into machine-specific language just before its execution.

Compilation

- A compiler does not necessarily translate from a high-level language into machine language.
- The term “compilation” applies when there is a translation from one nontrivial language to another (with full analysis of the meaning of the input).

Overview of Compilation

- Compilation proceeds through a series of well-defined phases.



Lexical Analysis

- A scanner is used to extract tokens from a source program.
- Tokens are smallest meaningful units in a program.
- Irrelevant characters such as comments and spaces are discarded.

Syntax Analysis

- A parser is used to organize tokens into a parse tree.
- A parse tree represents higher-level constructs in terms of their constituents.
- A context-free grammar (CFG) is a set of recursive rules that defines how tokens can be combined into a meaningful construct.

Semantic Analysis

- Semantic analysis is the discovery of meaning in a program.
- It recognizes when multiple occurrences of the same identifiers refer to the same program entity.
- It ensures that the type of a variable matches with the type of assigning expression.
- And much more…

Intermediate Code Generation

- A parse tree is known as a concrete syntax tree.
- After we know that a program is correct (syntactically), much of the information in the parse tree is irrelevant to further phases of compilation.
- An abstract syntax tree (AST) represents only useful information and annotates with extra information just discovered.

Target Code Generation

- This phase translates the intermediate form (*e.g.*, AST) into the target language.
- Generating code is usually not a difficult task but generating a *good* code is a different story.
- A code generator traverses the symbol table and assigns locations to variables.
- Then, it traverses the syntax tree to generate actual machine code.

Code Improvement

- Code improvement is often known as code optimization.
- It is an optional phase in compilation.
- Its goal is to transform an existing code into a new one that does the same thing but more efficient (*e.g.*, run faster, use less memory)

Errors in Compilation

- Lexical error
- Syntax error
- Static semantic error

Hello World in C#

```
namespace HelloWorld
{
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
                System.Console.WriteLine("Hello World");
        }
    }
}
```

Hello World in CIL

```
.namespace HelloWorld
{
  .class private auto ansi beforefieldinit Class1
       extends [mscorlib]System.Object
  {
    .method private hidebysig static void Main(string[] args) cil managed
    {
      .entrypoint
      .custom instance void [mscorlib]System.STAThreadAttribute::.ctor()
                = ( 01 00 00 00 )

       // Code size       11 (0xb)
       .maxstack  1

      IL_0000: ldstr   "Hello World"

      IL_0005: call    void [mscorlib]System.Console::WriteLine(string)

      IL_000a: ret
    } // end of method Class1::Main
```

Hello World in CIL (cont.)

```
    .method public hidebysig specialname rtspecialname
            instance void  .ctor() cil managed
    {
        // Code size       7 (0x7)
        .maxstack  1

      IL_0000: ldarg.0

      IL_0001: call  instance void [mscorlib]System.Object::.ctor()

      IL_0006: ret
    } // end of method Class1::.ctor

  } // end of class Class1

} // end of namespace HelloWorld
```